Symantec by Broadcom

LIGHTS OUT

Defense Evasion in the Age of BYOVD

Symantec Threat Hunter Team

Symantec by Broadcom

TABLE OF CONTENTS

Introduction

As cyber defenses improve, attackers increase their effort to bypass those defenses in order for their attack to have any chance of succeeding. Antivirus engines, endpoint detection and response (EDR) software, and the protections built into Windows itself all stand in the way of the malware the attackers try to run, when they want to steal credentials, and when they want to exfiltrate or encrypt data. Defense Impairment the tactic, that “consists of techniques that degrade, disable, or undermine the effectiveness and trustworthiness of security controls and monitoring mechanisms” in the MITRE ATT&CK framework, has become one of the most consequential parts of many critical intrusions over the recent years. Especially in case of ransomware attacks, it is now a near-universal step in the attack chain.

The most significant development within defense evasion over the past three years has been wide adoption of techniques to undermine integrity of the Windows kernel. The most prominent technique is when the attackers abuse bugs or security issues in legitimate, validly signed kernel drivers. The attackers, after gaining admin privileges, drop such vulnerable drivers on the affected systems and instruct Windows to load them, so this technique is called "Bring Your Own Vulnerable Driver" (BYOVD). The attacker then sends a specially crafted command to the driver, abusing its vulnerability to perform an operation desired by the attacker. That operation is typically the termination of a security product process that no user-mode attacker should be able to touch otherwise. The Windows kernel fully trusts the third-party drivers once these are loaded into the kernel, such that any issue in third-party code can undermine integrity of the operating system. With a wide community of driver developers, we have become trapped in a situation, where hundreds of vulnerable drivers are already in circulation. New vulnerable drivers are found regularly and shared openly. BYOVD tooling is now routinely bundled into ransomware-as-a-service (RaaS) offerings alongside the encryptors themselves.

BYOVD is not the whole story. An administrator-level attacker can in some cases suspend or starve security processes directly. Other techniques abuse the shortcomings in the implementation of Protected Process Light to launch a protected attacker process that can interfere with protected defender processes.

As Microsoft and the security industry harden the user-mode surfaces that security software depends on, attackers continue to push deeper into privileged territory. The defenses that worked even a few years ago no longer cover the ground attackers are reaching for. New approaches are needed.

The Ransomware Attack Chain

DEFENSE EVASION IN DEPTH

Disabling the security product comes just before the payload

Without first disabling the AV or EDR product, modern ransomware risks being caught before encryption begins — and BYOVD has become the most common way to clear the path.

Protected Processes Light: Shielding Anti-Malware

Windows provides a dedicated mechanism, called Protected Process at Antimalware Light, to safeguard anti-malware user-mode components from interference. We briefly describe that protection mechanisms in this section, before documenting known attacks in the later parts of this paper.

The Protected Process (PP) model was introduced in Windows Vista to shield DRM-related media processes from interference. Windows 8.1 extended this model with the introduction of Protected Process Light (PPL), which broadened the protection primarily to guard anti-malware services. While PP and PPL share the same core principles, PPL uses a tiered signing scheme that allows a wider range of software vendors to participate without requiring Microsoft to sign every vendor's binaries directly.

PPL allows anti-malware user-mode services to be launched as a protected service. After the service is launched as protected, Windows uses code integrity to only allow trusted code to load into the protected service. Windows also protects these processes from code injection and other attacks from admin processes.

Of note is that, after the anti-malware service is launched as protected, other non-protected processes (and even admins) can't stop the service. When an anti-malware service is running as protected, it runs as a Protected Process at Antimalware Light protection level.

> ***"The protected process infrastructure only allows trusted, signed code to load and has built-in defense against code injection attacks."***

Once the anti-malware services have opted into the protected service mode, only Windows-signed code or code signed with the anti-malware vendor's certificates are allowed to load in that process. It stops code injection attacks and prevents unauthorized processes from writing into the virtual memory of the protected process.

PPL was introduced specifically to counter admin-level attacks on security software. AV and EDR services running as protected processes can only be loaded with trusted, Microsoft-approved code. Even an administrator account cannot stop or inject into a protected process using standard Windows APIs. This was a genuine improvement — it raised the bar significantly.

PPL operates across a hierarchy of protection levels, each associated with a different signing authority: WinSystem, WinTcb, Windows, LSA, Antimalware, CodeGen, Authenticode, and None, in descending order of trust. A process at a higher protection level can open one at a lower level with full access, but not vice versa. Anti-malware services run at the Antimalware Light level, meaning they are protected from processes running at lower or no protection level, but can themselves be opened by higher-privileged system processes. The Windows kernel enforces these access restrictions when any user-mode caller attempts to open a handle to a protected process or thread.

Both protected and non-protected processes, running with or without admin privileges, can also launch protected child processes.

Launching a protected child process at Antimalware Light protection level by a non-protected process may look like this:

1. Parent process calls CreateProcess() API with the following combination of parameters:
   - *lpApplicationName* pointing to path name of a trusted executable
   - *dwCreationFlags* value with flag *CREATE_PROTECTED_PROCESS* set
   - *lpStartupInfo->lpAttributeList* containing:
     - Attribute value *PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL*
     - *lpValue* pointing to a DWORD value of *PROTECTION_LEVEL_ANTIMALWARE_LIGHT*
2. Operating System enforces integrity of code to execute by the child process according to the policy for Antimalware Light protection level:
   - Only Windows signed code or code signed with vendor certificates can load when starting the executable and loading any libraries
   - The vendor certificates must be trusted via *MicrosoftElamCertificateInfo* resource, which should be linked into a corresponding Early Launch Anti-Malware (ELAM) driver

- The ELAM driver must be specially signed by Microsoft

3. The CreateProcess() API returns handles with limited access:
    - *lpProcessInformation->hProcess* with granted access: Terminate, *SuspendResume*, *QueryLimitedInformation*, *SetLimitedInformation*, Synchronize (instead of *AllAccess* when creating non-protected processes)
    - *lpProcessInformation->hThread* with granted access: Terminate, *SetLimitedInformation*, *QueryLimitedInformation*, Resume, Synchronize (instead of *AllAccess* when creating non-protected processes)

The Windows kernel also restricts what access is allowed when opening protected processes and their threads. These restrictions apply to user-mode requesters only and depend on requester process protection level and protection level of the target process.

PPL guards security processes against user-mode attacks, however, it does not tackle kernel-mode level attacks.

Attacks From Kernel Mode: BYOVD and other techniques

Key findings

- BYOVD is the most common way attackers shut down security software.
- The attacker brings a legitimate but flawed driver onto the machine and tricks the operating system into running it.
- Once running, the driver gives the attacker the same level of access as Windows itself — enough to switch off anti-virus and EDR software.
- A wide range of drivers can be abused this way, and ready-made tools that automate the attack are openly available.
- This technique now appears in a large share of ransomware attacks, and is even being built directly into the ransomware itself.

Overview

Modern security software is designed to spot and block malicious activity on a computer. For an attacker to succeed, they must first find a way to shut that software down, and BYOVD has become the dominant way of doing so on Windows.

Windows runs code at two levels of privilege. In user mode, where everyday applications such as browsers and email clients live, the operating system tightly limits what they can do. In kernel mode, where the operating system itself runs, code has few restrictions and can read or change almost anything on the machine, including the security software meant to protect it.

Reaching kernel mode is therefore one of the most valuable things an attacker can do, and the most reliable route is to bring along a vulnerable driver. A driver is a small piece of trusted, digitally signed code that vendors supply and the operating system loads to handle specialist tasks — and that signature is part of how Windows decides a file can be trusted.

Hundreds of drivers in circulation today have flaws that allow them to be misused. The attacker drops one onto the target machine; because it is signed, Windows loads it without complaint. A specially crafted command then exploits the flaw, and the operating system carries out actions on the attacker's behalf that no ordinary program could.

What the attacker does with these privileges varies. Most often they kill the processes that make up the antivirus or EDR product, leaving the computer defenceless. Subtler variants strip the agent of the rights it needs, leaving it running but unable to act, or tamper with the operating system's internal records so the product stops receiving notifications and goes effectively blind.

Ready-made BYOVD tools are openly traded and shared. Several are now standard parts of the ransomware toolkit, and some ransomware builds the BYOVD component directly into the payload.

User Mode vs. Kernel Mode

DEFENSE EVASION IN DEPTH

Why reaching the kernel is the prize

USER MODE

Everyday applications, admin tools and security tools

- Web browser
- Email
- Office apps
- Admin tools
- Security agent

Restricted — cannot reach hardware or another process's memory. Even administrator accounts run here.

PRIVILEGE BOUNDARY

KERNEL MODE

The operating-system core and device drivers

- NT kernel
- Device drivers
- Memory manager
- Scheduler

Unrestricted — can read and write anything, including security software.

ATTACKER'S GOAL

Code that reaches kernel mode bypasses almost every protection that ordinary applications — including security software — depend on.

SYMANTEC THREAT HUNTER TEAM

In depth

Kernel mode is the most privileged execution level of a CPU. User mode is where typical applications such as Chrome, Word, and others operate. They are restricted in what they have access to; they cannot access the hardware directly. However, kernel mode is where the OS kernel and most device drivers operate. At this level, the code has unrestricted access to the CPU and all physical memory.

If an attacker successfully executes code in kernel mode they can bypass almost all security software. They can disable firewalls and kill security processes. A malicious instruction issued in kernel mode could also potentially cause the impacted machine to crash. These are the reasons why gaining access to the kernel is so potentially important for malicious actors.

BYOVD is by far the most frequently used technique for defense impairment at the kernel level. Generally, attackers will attempt to deploy a signed vulnerable driver to machines on a target network, which they then exploit to elevate privileges and disable security software. In most cases, the vulnerable driver is deployed along with a malicious executable, which will use the driver to execute commands on its behalf.

These drivers are considered “vulnerable” as it should not be possible to leverage them in this way. A correctly written driver will contain safeguards to ensure they only respond to legitimate requests from authorized software. However, when these drivers fall into the wrong hands, they effectively become tools for privilege escalation.

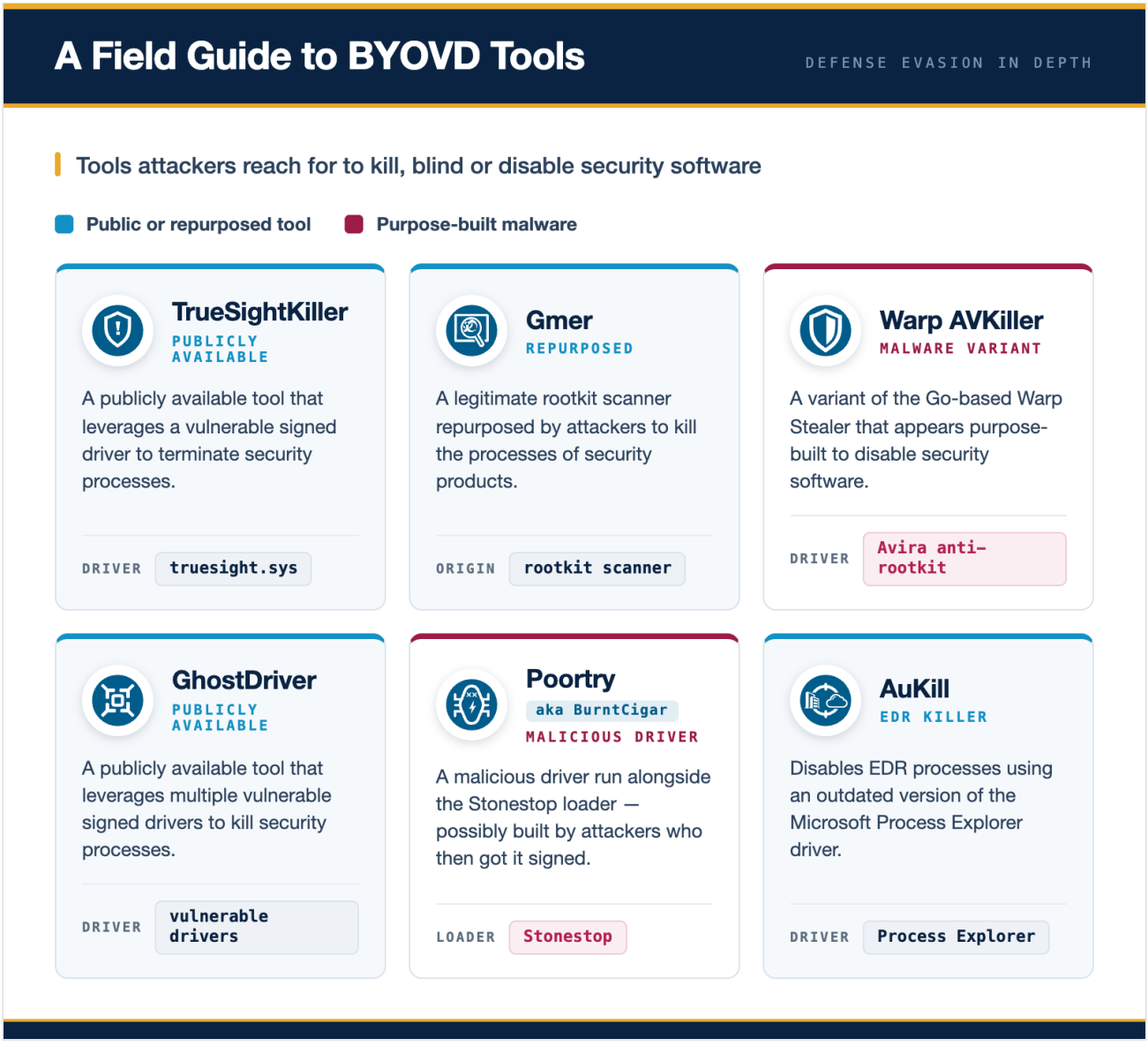
For BYOVD attacks, the attacker generally chooses a legitimate, third-party driver that is known to have a security flaw. These are often old versions of drivers for graphics cards, anti-cheat engines, or security software. The attacker then gains a foothold on the victim network, perhaps by exploiting a vulnerability or via a spear phishing email, they then install the vulnerable driver. Because the driver is digitally signed by a trusted company (like Dell, Microsoft, Intel or others), Windows will allow it to be loaded into the kernel. Once the driver is running in kernel mode, the attacker can exploit the known flaw in the driver to execute their own malicious code, most often to disable security software.

BYOVD is popular with attackers due to its effectiveness and reliance on legitimate, signed files, which are less likely to raise red flags. A wide range of drivers have been used in such attacks, with anti-rootkit drivers developed by security vendors being among the most commonly exploited. Popular BYOVD tools frequently used by attackers include:

- TrueSightKiller: A publicly available tool that leverages a vulnerable driver named truesight.sys.
- Gmer: A rootkit scanner that can be used to kill processes.
- Warp AVKiller: A variant of a Go-based information-stealing threat called Warp Stealer, which appears to be just used to bypass security software. It uses a vulnerable Avira anti-rootkit driver to disable security software.
- GhostDriver: A publicly available tool that leverages vulnerable drivers to kill processes.
- Poortry (aka BurntCigar): A malicious driver documented by Sophos that is frequently employed alongside a loader known as Stonestop. Unlike many drivers, Poortry may have been developed by attackers who then succeeded in getting it signed.
- AuKill: A tool documented by Sophos that uses an outdated version of the driver used by the Microsoft Process Explorer utility to disable EDR processes.

While attackers most often use pre-existing vulnerable drivers with known flaws in BYOVD attacks, they can also create their own vulnerable drivers and use them, provided they are able to trick Microsoft into signing them. Poortry (aka BurntCigar) is believed to be one such driver making it the exception rather than the norm. The

challenges of getting a “fake” driver certified is likely one of the reasons why it is not a common choice for attackers.



Reynolds: BYOVD component embedded in ransomware payload

One unusual BYOVD technique we observed in early 2026 was in a Reynolds ransomware campaign where the ransomware contained a BYOVD defense evasion component embedded within the ransomware payload itself. While bundling a defense evasion component within the ransomware itself isn't entirely novel, it is quite unusual and not what we typically see ransomware actors doing today. It was previously seen in a Ryuk ransomware attack in 2020, as well as an attack in which a little-known ransomware called Obscura was deployed in 2025.

The ransomware payload drops a vulnerable NsecSoft NSecKrnl driver and tries to create an NSecKrnl service. This driver is then exploited to attempt to kill processes.

The NSecKrnl driver is a Windows kernel-mode driver with a known critical security vulnerability (CVE-2025-68947), due to its failure to verify if a caller has sufficient permissions before executing commands. This allows a local, authenticated attacker to terminate processes owned by other users, including SYSTEM and Protected Processes, by issuing crafted Input/Output Control (IOCTL) requests to the driver.

The impairment of defenses, usually by attempting to disable antivirus (AV) or endpoint detection and response (EDR) software, remains a key part of ransomware attacks in 2026.

Terminating security software processes

During the course of incident response, we have seen countless examples of vulnerable drivers being used to disable protected processes.

One such abused kernel driver observed by our researchers was signed by "Webroot Inc" and "Microsoft Windows Hardware Compatibility Publisher" and appears to originate from the "Webroot SecureAnywhere" anti-malware software.

The driver implements a proprietary API that, among others, accepts the following requests from user-mode clients:

- Terminate arbitrary process

  The request parameter contains process_id to terminate.
- Delete arbitrary file

  The request parameter contains filename to delete.

The anti-malware driver implements the following safeguards:

- Admin privileges are required to access the API, but non-protected processes with admin privileges are allowed to terminate protected processes.
- Request parameters are encrypted using a simple encryption method.

A malicious actor was able to reverse-engineer that proprietary API. They then implemented their own malicious client that abused t driver in order to terminate security software processes. The access control implemented by the abused driver is weaker than the Windows API. When using the standard user-space Windows API, non-protected processes - including admin processes – are unable to tamper with the protected processes. However,

the proprietary API implemented by the abused driver allows any non-protected processes with admin privileges to terminate a protected process.

Specifically, the malicious client program obtains a list of running processes by calling the user-mode Windows API. For each running process, it then computes the hash of the process name and checks the hash against a hardcoded list. On match, the malicious client obtains the filename of the executing binary and then calls the driver's proprietary APIs to terminate the process and delete the executable file.

For example, when executed as a non-protected process with admin privileges, that malicious client is able to terminate Windows Defender protected process "MsMpEng.exe" and then delete the corresponding "MsMpEng.exe" file from disk.

Stripping processes of OS resources and privileges

Attackers could also use a compromised driver to strip processes of operating system resources or protection. Using this technique, they could strip a protected process of its protected status, allowing the attackers to kill or inject malicious code into the process. They could also strip privileges from, for example, security software, so the software appears to be monitoring the system, but if it tries to stop or block malicious activity it is unable to do so. Malicious actors could also disable "callbacks" - meaning that new malicious processes could be started but monitoring tools won't know because the callback function has been disabled. Attackers can also install fake callbacks that help protect the malware and allow it to operate. Attackers might also attempt to use this technique to make their activity quieter, for example by stripping permissions from a legitimate process into which it has injected malicious code to reduce the chances of the process's behavior being flagged as suspicious.

Process stripping is often used legitimately to improve defenses to uphold the principle of least privilege by deliberately stripping an application's ability to access certain parts of the operating system or critical services, so that one app's compromise cannot bring down the whole system. However, malicious actors have been able to use this technique for their own advantage by exploiting vulnerable drivers to allow them to strip functionality from processes when they are carrying out malicious attacks.

In an example of a driver being used to strip processes of OS resources, we observed an abused kernel driver signed by "Microsoft Windows Hardware Compatibility Publisher," which originated from "Process Explorer" by Sysinternals.

That driver implemented a proprietary API that, among others, accepted the following requests from user-mode clients:

- Open arbitrary process for GENERIC_ALL access

  The request parameter contains the process_id to open.

  The response contains a handle returned by ZwOpenProcess() API call.

- Close arbitrary handle

  The request parameter contains the process_id of the handle owner, the memory address of the object associated with the handle, and the handle itself.

Analyzed malicious clients appear to reuse code from an open-source Github project called Backstab. This original code includes functionality to close all handles that belong to a specified process. The client obtains a

table of all handles on the affected computer by calling NtQuerySystemInformation() API with the undocumented SystemInformationClass parameter value 0x10 (often referred as SystemHandleInformation).

For each record in the table, the client then checks if it refers to the given process.
On match, the client obtains the memory address of the object associated with the handle (calling NtQuerySystemInformation() API with the same SystemInformationClass parameter) and then calls the vulnerable driver's proprietary API to get it to close the process associated with the handle. The attacked programs are unlikely to have any built-in logic to recover once certain handles are unexpectedly closed and usually chose to terminate upon such errors.

Another malicious tool also built upon Backstab , by adding more attack techniques.

One of the added techniques was to open a targeted process for GENERIC_ALL access by calling the driver's proprietary API and using the returned handle to tamper with the process.

That tampering is performed from user-mode and includes:

- Attempting to suspend the process by calling NtSuspendProcess() API
- Attempting to terminate the process by calling NtTerminateProcess() API
- Attempting to inject a thread at RtlExitUserProcess() API
- Attempting to inject a thread at NtTerminateProcess() API
- Attempting to strip the targeted process of memory by calling NtUnmapViewOfSection() API, NtFreeVirtualMemory() API and NtProtectVirtualMemory() API

In another example, we saw a malicious kernel driver being used to stop processes. This driver was signed by "WDKTestCert User,133205899322917322." The driver attempts to set the integrity level of the primary process token as "untrusted mandatory" for any processes that match a name check.

Whenever the driver is loaded it starts a kernel thread to execute all the steps described below:

- Obtains a table of all running processes on the affected computer by calling NtQuerySystemInformation() API with SystemInformationClass parameter value SystemProcessInformation.
- For each record in the table, it then checks if a hash of the process name is one of the targeted values.
- If there's a match, the driver obtains EPROCESS structure representing the matching process_id by calling PsLookupProcessByProcessId() API.
- Gets the handle of the process token by calling PsReferencePrimaryToken() API followed by ObOpenObjectByPointer() API.
- The driver calls ZwSetInformationToken() API with TokenInformationClass parameter value TokenIntegrityLevel and TokenInformation parameter representing Untrusted Mandatory Level.

The affected processes include Windows Defender protected process "MsMpEng.exe." All the above actions are performed autonomously. However, because Microsoft is aware the driver is vulnerable, the driver signature is untrusted and the Windows kernel does not load it under default configuration. To overcome this, a dropper carries a vulnerable driver in addition to the malicious driver discussed earlier. The dropper starts the vulnerable driver and exploits it in order to load the malicious driver into the Windows kernel.

Tampering with kernel data structures and callbacks

Killing or disabling the user-mode security processes is the most frequent outcome of a successful BYOVD attack, but kernel-mode access also lets attackers tamper directly with the operating system mechanisms that security software relies on to monitor machine activity and remediate threats. In this scenario the agent itself keeps running, but the kernel is no longer feeding it telemetry for it to function correctly or has been induced to give the malware special protection. Strictly speaking these techniques target a different attack surface from the user-mode AV/EDR processes covered in the rest of this chapter, but they share the same BYOVD entry point and so are included here for completeness. The two main variants observed in the wild are removing callbacks from kernel notification lists – which is the more common – and installing attacker-controlled callbacks that act as a protective shield around the malware.

Security software register kernel-mode callbacks with the operating system in order to be notified of, and where appropriate to block, sensitive events on the machine such as process creation, image loads, registry modifications, and handle requests against protected processes. If an attacker is able to take the relevant notification arrays from kernel mode and clear out the entries belonging to a security software, the software will continue to run normally but will effectively become blind: it will stop receiving the events it depends on and will therefore stop detecting and blocking the attacker's subsequent activity.

To do this an attacker needs two things: the ability to read and write kernel virtual memory, and the address inside the kernel of the relevant data structures.

For the latter, attackers commonly retrieve the memory address of the targeted kernel module by calling NtQuerySystemInformation() with the SystemInformationClass parameter SystemModuleInformation, which returns a list of loaded kernel modules and their base addresses. From there it is straightforward to compute the offsets of the notification arrays inside that module.

For the former, the kernel read/write primitive, the most common route is via BYOVD. The attacker abuses a vulnerable signed driver that exposes raw memory read/write primitives to user-mode clients, and uses those primitives to read out the notification arrays and zero out the callback pointers belonging to security drivers. The publicly available tool RealBlindingEDR illustrates this approach, performing data-only attacks to directly read and write Windows kernel data structures via a vulnerable third-party driver in order to remove the kernel callbacks registered by EDR software.

A variant of this technique uses physical memory access rather than virtual memory access. Some vulnerable drivers expose a primitive that allows mapping arbitrary physical memory ranges into the calling process. Public tooling such as physmeme demonstrates this by enumerating physical memory ranges reported under the registry key HARDWARE\RESOURCEMAP\System Resources\Physical Memory and then using a vulnerable driver to map those ranges. A real-world example documented by Kaspersky involves abuse of the legitimate ThrottleStop driver: the attacker calls NtQuerySystemInformation() with the undocumented SystemSuperfetchInformation class to obtain virtual-to-physical mappings, then uses the ThrottleStop driver to read and write the corresponding physical memory pages and tamper with kernel data structures from there.

The same primitives can be used in reverse not to remove the security software's callbacks but to install new ones controlled by the attacker. The attacker writes a callback function into kernel memory and registers it with the relevant notification list, so that the malware is consulted by the kernel whenever certain sensitive events occur. This can be used to give the malware a chance to terminate or interfere with security tooling that tries to inspect or remove it, or simply to keep the attacker's components hidden. This technique is less commonly observed in the wild than callback removal, but it is the natural counterpart to it. Once an attacker has reliable kernel read/write primitives, both operations are within reach.

A particularly compact form of kernel data structure tampering is to flip the Protection field of the EPROCESS structure that the kernel uses to represent each process on the system. By updating that field directly in kernel memory an attacker can either strip a target process – for example a Protected Process at Antimalware Light protection level such as MsMpEng.exe – of its protection so that ordinary user-mode tooling can subsequently terminate or inject into it, or alternatively elevate their own attacker process into a higher protection level so that it acquires the right to tamper with security software. The Mimikatz !processprotect command, backed by the Mimikatz driver, illustrates the technique publicly: !processprotect /process:MsMpEng.exe /remove demonstrates the strip-protection variant, and !processprotect /process:[ATTACKER_PROCESS] the attacker-elevation variant. As with the callback-removal techniques, in real-world incidents the kernel read/write primitive needed to update the Protection field in place is usually obtained via BYOVD rather than via a bespoke malicious driver such as Mimikatz's.

The techniques described above all rely on kernel read/write primitives obtained from a vulnerable third-party driver. A higher-end variant abuses primitives that arise from flaws in Windows itself: an attacker running as a Protected Process at the WinTcb-Light protection level could reportedly tamper with kernel data structures to

gain access to \Device\PhysicalMemory, from which they can in turn modify any kernel structure they choose. Public proof-of-concept work on this class of attack includes GodFault and EDRSandblast-GodFault, which illustrate the technique but are not currently associated with widespread in-the-wild use.

Other Defense Evasion Tactics

Key findings

- Attackers with administrator rights can in some cases pause, rather than terminate, a security product, leaving it running but unable to respond.
- Windows treats certain processes as more trusted than others, and attackers who reach those higher trust levels can shut down security software that ordinary administrators cannot.
- Attackers can cut a security product off from the vendor's cloud, blocking the lookups and telemetry it relies on and leaving it apparently healthy but unable to do its job.
- A number of publicly available research tools and techniques further illustrate how many ways security software can be neutralized without using a kernel-mode driver.

Overview

Although BYOVD dominates the picture, it is not the only way to neutralize security software; some techniques involve no flawed driver at all. This chapter looks at the alternatives, which fall into three broad groups.

The first group exploits the rules Windows uses to manage its most sensitive processes. Security software usually runs as a protected process that ordinary administrators cannot interfere with — but while an administrator cannot end such a process, they can pause it. For an attacker, a paused security product is as good as a stopped one: its scans and cloud checks stop, and restart logic never triggers because the product has not crashed.

The second group exploits the fact that some processes are more trusted than others. Windows arranges its sensitive processes into a hierarchy of trust levels, where a higher process can interfere with a lower one, and security software usually sits in the middle. If an attacker takes over or impersonates a higher-trust process — often a Windows system service — the kernel treats its request to terminate or tamper with the security software as coming from a more trusted source, and allows it.

The third group ignores the local machine altogether. Modern security solutions are not just programs that run on the computer. They also rely heavily on services hosted by the vendor's own cloud, where reputation checks, file-prevalence lookups, and cross-endpoint correlation actually happen. If an attacker can quietly block the security product's access to the vendor's cloud, large parts of its detection logic may stop working. The product looks healthy, but its decisions are made with much less information. Windows already provides a legitimate set of tools for managing network traffic, that attackers with administrator rights can abuse to cut a security product off from its back-end.

In addition to these three groups, a range of publicly available research tools and techniques show that the surface for non-driver attacks is wider than it might first appear, and remains an active area of research.

In depth

In this section, we will take a closer look at how these other evasion techniques work in more detail.

Non-protected process with administrator privileges suspending protected processes

Normally the rules of PPL mean that a protected process cannot be interfered with by a non-protected user. However, attackers have discovered a side door. They cannot terminate a protected process from a non-protected administrator account, but they can suspend it. In many cases, suspending a process is enough to allow an attack to succeed.

The Windows kernel restricts what access a user-mode caller can request when opening a protected process or its threads. These restrictions sit on top of the usual Mandatory Integrity Control and Access Control List checks, and they are validated against a policy associated with the RtlProtectedAccess data structure in the kernel.

Under that policy, certain access rights are restricted depending on the protected signer of the target. There are three exceptions. The check is skipped when the requester is not user-mode, when the target's protected type is PsProtectedTypeNone, or when the requester's protected type is greater than or equal to the target's and the requester's protected signer overrides the target's.

In practice, this means that when an unprotected user-mode process tries to open an Antimalware Light protected process, every access right outside the set { SuspendResume, QueryLimitedInformation, SetLimitedInformation, Synchronize } is always denied. The same applies to threads inside the target process, with a similar but slightly different limited set. PROCESS_TERMINATE is always denied. PROCESS_SUSPEND_RESUME, however, is not.

A public proof of concept by 0gtweet using Sysinternals' pssuspend.exe demonstrates the technique against MsMpEng, the protected user-mode component of Windows Defender . The practical effect depends on how the rest of the AV stack reacts when the user-mode component goes unresponsive. If the kernel-mode driver asks the user-mode part to scan a file, and the user-mode part fails to respond within its internal timeout, perhaps two seconds, the kernel side may simply give up and let the file through.

There is a similar effect at the service-management layer. A process can be protected, but the service control manager that manages it is not. An attacker may therefore be able to send a service pause or service stop command to the unprotected service wrapper of a protected service. The result is much the same as attaching a debugger or calling pssuspend: the protected process stops responding.

Suspending can sometimes be more effective than killing. The process still appears to be running, so users see no obvious failure, and many security software automatically restarts when its main process is terminated. If the process is only suspended, the restart logic does not trigger.

The kernel can, separately, allow an attacker to suspend an unprotected process through a different user-mode route. If a target process lacks Protected Process (PPL) defenses, an attacker possessing SeDebugPrivilege can call the DebugActiveProcess API against it. When a debugger attaches to a target process, the Windows kernel automatically suspends all threads in the target so the debugger can inspect them. The attacker simply attaches a debugger and then does nothing, paralyzing the process belonging to the security software and continuing with their malicious activity unhindered.

Across all of these variants the underlying point is the same. Non-protected processes cannot kill a protected process, but they can suspend one. By the time the machine reboots and the security software restarts, the damage is already done.

Protected processes with higher protection levels killing or tampering with protected processes

The Windows security model relies on trust hierarchies. A process can only access or terminate another process if it has an equal or higher protection level.

In Windows, not all protected processes are created equal. There is a ladder of trust, processes that are on a higher rung you can manipulate, tamper with, and even terminate processes on equal and lower rungs. From highest to lowest, a simplified view of the relevant levels looks like this:

- WinSystem (highest): internal Windows kernel-level tasks.
- WinTcb (Trusted Computing Base): high-trust system processes such as lsass.exe.
- Antimalware: where security software such as Windows Defender's MsMpEng usually sit.
- Authenticators, Windows, and others: lower-level system tasks.

The real picture is a directed graph of dominance relationships rather than a clean pyramid, but for the purposes of understanding attacks against security products the simplified hierarchy is enough.

When a process wants to kill another process, it generally calls an API like OpenProcess with the PROCESS_TERMINATE right. The Windows kernel then looks at the protection level of both the source (the process trying to kill) and the target (the process to be killed). If the source sits at a higher protected level than the target, the kernel grants permission, and the process can be killed. If the source sits below the target, the kernel returns an Access Denied error. This holds true even if the user is an administrator on the machine.

Given its high level on the trust ladder, attackers will often try to compromise a WinTcb-level process. Since WinTcb is higher than Antimalware, a compromised WinTcb process can tell the kernel to shut down security software, and the kernel will allow it. According to its own rules, the higher-trust process is allowed to manage the lower-trust one.

While killing a process is an effective way of stopping it, it is also a noisy action. For this reason, attackers often prefer tampering. Tampering may involve memory injection, where attackers write malicious code into the memory space of a protected process, or thread hijacking, where they take over a task the protected process is currently performing.

Again, the kernel enforces the hierarchy. An Antimalware-level process cannot read the memory of a WinTcb-level process. This means security software that has been tampered with could not be used to steal passwords from the higher-protection lsass.exe process, for example.

A common way attackers level up to a protection level high enough to kill another protected process is through token impersonation. The attacker finds a system service that runs at a high protection level, such as a printer or network manager, hijacks it, and uses it to perform malicious activity.

Attacking AV/EDR network services

A modern AV or EDR product is not just a collection of local processes. It also depends on network services operated by the vendor. Typical services provided in the cloud include reputation lookups, file-prevalence checks, and cross-endpoint correlation in EDR software. If an attacker can prevent the local agent from accessing those services, the agent loses a significant part of its decision-making capability without it being killed, suspended, or tampered with.

The most common way to do this is to block the relevant network access from the host itself. Windows includes the Windows Filtering Platform (WFP), a set of APIs that legitimate firewall and network-monitoring software use to inspect and filter network traffic. An administrator can use WFP to add filter rules that drop outbound traffic from specific processes. An attacker with administrative privileges can use exactly the same APIs to add rules that drop the AV or EDR agent's outbound traffic. The agent stays running and looks healthy, but its connections back to the vendor's cloud services fails. Reputation queries return no answer. EDR telemetry never reaches the analysis pipeline. Detections that depend on cloud lookups silently degrade.

One open-source implementation of this technique is EDRSilencer, which programmatically inserts WFP filters that block outbound traffic from a hardcoded list of common EDR and AV process executables.

Network-layer evasion is interesting because, unlike most of the techniques discussed elsewhere in this paper, it does not rely on a vulnerable driver or on tampering with kernel data structures. The attacker only needs

administrative privileges and the willingness to run a small user-mode tool. From a defender's point of view the visibility cost can be significant. Any detection logic that depends on cloud reputation, on EDR cross-endpoint correlation, or on the agent successfully reporting telemetry is rendered partially or wholly ineffective. The local agent appears to be running normally, which can make the failure mode harder to spot than an outright crash.

Other techniques and tools

Several additional techniques have been described by external researchers over recent years. Some have been observed in the wild only rarely, but they round out the picture of what is possible against PPL-protected security software.

In a 2022 Black Hat USA presentation, researcher Matt Graeber explored how attackers can use Early Launch Anti-Malware (ELAM) and PPL to disable security software. Graeber's primary finding was that many ELAM drivers contain overly permissive allowlists, with some drivers allowing any binary signed by a broad certificate, such as a general Microsoft code-signing certificate, to run as a protected process.

In his presentation, Graeber demonstrated how attackers might take advantage of this to disable security software, even when it is running as protected. Attackers could use a legitimate, signed executable like

MSBuild.exe that is covered by an overly permissive ELAM allowlist, and then register a legitimate but overly permissive ELAM driver on the system. They could then create a service for the chosen executable and set it to launch as Antimalware Light, allowing the attacker to run their own code with PPL protections. Once running at the Antimalware Light level, the attacker's process can kill other protected security processes, because they are now at an equal or higher trust level.

Another way to bypass PPL is PPLdump, a turnkey user-mode tool that exploits a Windows vulnerability to achieve arbitrary PPL code execution and dump any PPL process. This can be used to dump lsass.exe and enable lateral movement. PPLdump is open source, making it easy to alter the payload to perform other privileged actions, such as disabling security software. Microsoft considers PPL a defense-in-depth measure, not a formal security boundary, so these bugs do not usually qualify for patches. The vulnerability underlying and predating PPLdump was publicly disclosed in 2018, but Microsoft did not patch it until 2022, over a year after PPLdump's 2021 release.

PPLFault is a proof-of-concept exploit that demonstrates a method to bypass Windows PPL by exploiting a specific class of vulnerability known as False File Immutability (FFI). The tool allows an administrative user to inject arbitrary code into a PPL process such as services.exe or lsass.exe, effectively elevating privileges from admin to kernel-level access, since the attacker can use the compromised PPL process to load unsigned drivers or disable security software. It relies on a race-condition vulnerability where the Windows kernel and memory manager incorrectly assume a file is immutable once opened. PPLFault exploits this by using Server Message Block (SMB) to modify the contents of a DLL after the Windows Code Integrity (CI) module has already validated its signature, but before it is fully loaded into memory. This could allow an attacker to dump memory from lsass.exe and disable security software. Microsoft released mitigations for this specific technique in Windows 11 Build 25941 by improving the code integrity subsystem to better handle file immutability during the DLL-loading process. However, the broader class of FFI vulnerabilities continues to be a subject of active research.

Kernel Driver Utility (KDU) is an open-source tool designed to give users a simplified way to manipulate the Windows kernel without requiring a local debugger or extensive setup. It primarily facilitates BYOVD attacks. KDU can load unsigned drivers into the kernel by bypassing Driver Signature Enforcement (DSE), which it achieves by mapping the driver directly to kernel memory using shellcode rather than through the standard Windows loader. Once that capability is in hand, an attacker using KDU can downgrade protected processes, kill security software, and more.

Kernel Hardening and How Attackers Bypass It

Key findings

- Microsoft added a series of protective features to the Windows kernel over the past decade to make it harder for attackers to do damage even if they reach this most privileged part of the system.
- Each of these features has a known way around it, but Microsoft does not treat attacks at this level of access as standard security vulnerabilities.
- The features are designed to stop attackers from running their own code, redirecting how the operating system works, or rewriting the memory map.
- The attacks most commonly used against security software do none of these things. They simply change information already in the system.
- As a result, kernel hardening is a worthwhile safeguard but cannot be relied on to stop the attacks that matter most in real-world defense evasion.

Overview

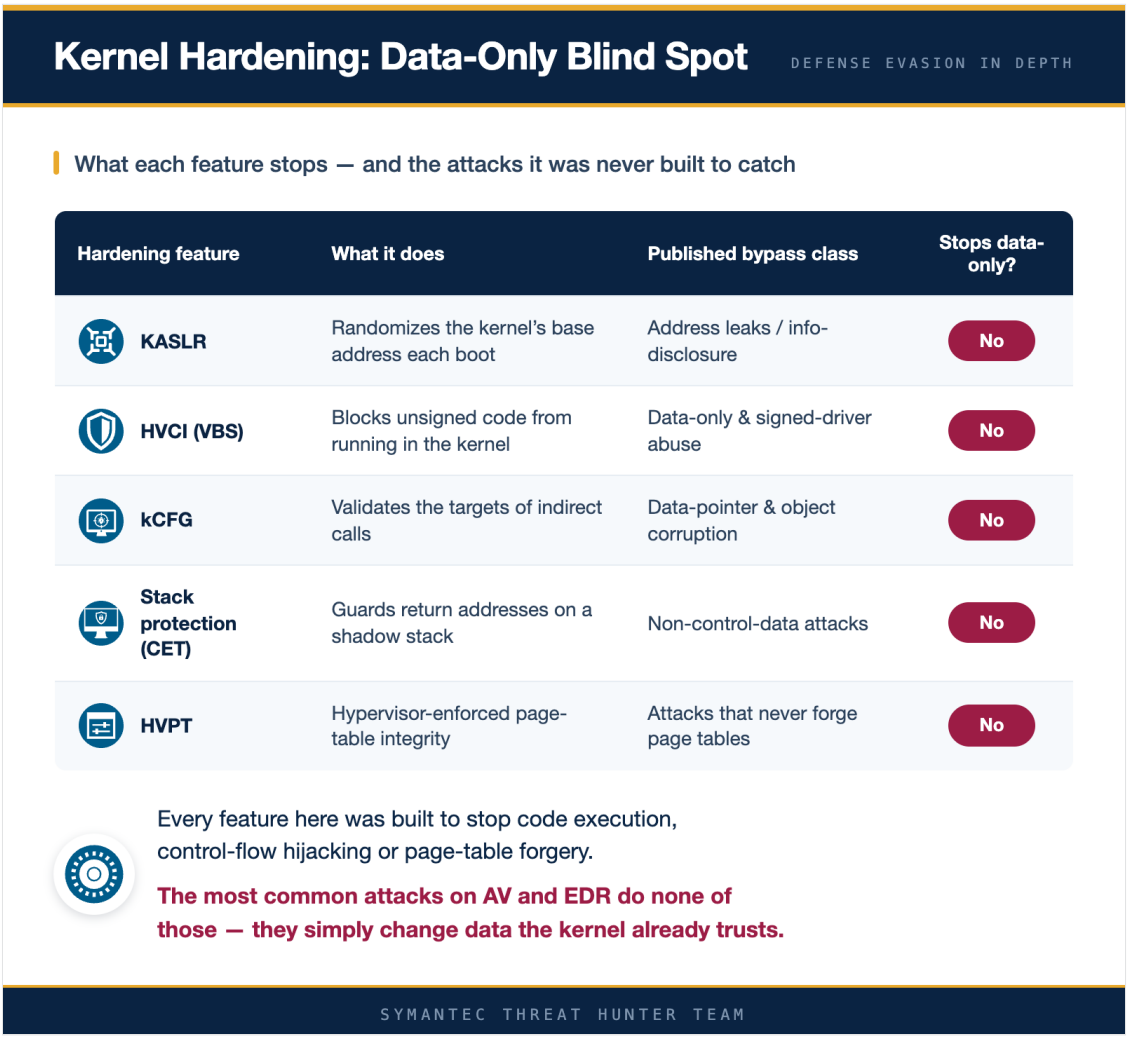
Once an attacker has reached the kernel, the most privileged part of Windows, they have access to almost everything on the machine. Microsoft has, over the past decade or so, added a series of protective features to make that access less useful to them. This chapter looks at those features, and at why they have a limited effect on the attacks that matter most.

Microsoft itself is explicit that attacks launched by an attacker who is already an administrator and has reached the kernel are not treated as standard security vulnerabilities. The features in this chapter exist only to make life harder for the attacker, not to provide a guaranteed barrier, and each has a publicly known bypass.

The features themselves block different categories of behaviour: finding what the attacker is looking for, installing or running their own code in the kernel, redirecting how the operating system flows from one instruction to the next (return-oriented programming), or reaching particularly sensitive data held in a separate, hidden layer of the operating system. Each closes off a category of attack that would otherwise be available.

The trouble is that the attacks most commonly used against security software fall into none of these categories. The attacker is not running new code, redirecting the operating system, or rewriting the memory map — they are simply changing information the operating system already holds: the flag that says which processes are protected, the entries that tell the kernel which security software wants to be notified of events, or the privileges of a running agent. None of these are blocked by the protections in this chapter, because none require the behaviours those protections were designed to stop.

The practical takeaway is that kernel hardening is worth deploying, because each feature closes off some attacks that would otherwise be possible, but it cannot be relied on as the only defense against the attackers seen in the wild.



Kernel Hardening: Data-Only Blind Spot

DEFENSE EVASION IN DEPTH

What each feature stops — and the attacks it was never built to catch

Hardening feature	What it does	Published bypass class	Stops data-only?
KASLR	Randomizes the kernel's base address each boot	Address leaks / info-disclosure	No
HVCI (VBS)	Blocks unsigned code from running in the kernel	Data-only & signed-driver abuse	No
kCFG	Validates the targets of indirect calls	Data-pointer & object corruption	No
Stack protection (CET)	Guards return addresses on a shadow stack	Non-control-data attacks	No
HVPT	Hypervisor-enforced page-table integrity	Attacks that never forge page tables	No

Every feature here was built to stop code execution, control-flow hijacking or page-table forgery.

The most common attacks on AV and EDR do none of those — they simply change data the kernel already trusts.

SYMANTEC THREAT HUNTER TEAM

In depth

Once an attacker is running in the Windows kernel, the operating system is not entirely defenseless. Microsoft has, over the past decade or so, layered a set of kernel hardening features on top of the kernel itself. These features are designed to make exploitation harder. They randomize the kernel's load address, restrict where executable code can come from, and constrain how indirect function calls and returns can be made. In principle, they should be enough to slow down or stop an attacker who has gained kernel access through a BYOVD attack or other techniques already seen.

However, in practice, kernel hardening is best understood as a set of mitigations rather than as a hard security boundary. It is worth noting how Microsoft views this whole category of issue. Microsoft's published servicing criteria for Windows are explicit that, for the NT kernel, administrator-to-kernel is not a security boundary. By extension, an administrator who tampers with a PPL is not, by default, treated as exploiting a vulnerability. The techniques described above are therefore best understood as bypasses of mitigations rather than as exploitation of formal security boundaries. Microsoft's Defender team does sometimes patch them outside of the standard Microsoft Security Response Center (MSRC) process, but without CVEs, without bug-bounty acknowledgement, and on a schedule that is not synchronized with MSRC. The fragmentation between MSRC and the Defender team is part of the reason this category of issue persists in the wild.

The result is that most of the hardening features described in this chapter exist to make the attacker's life more difficult, not to make exploitation impossible. Each one has a published bypass, and once an attacker has reliable read and write primitives in the kernel, the bypasses tend to look very similar: data-only attacks that route around the protection rather than confront it head-on.

Kernel Address Space Layout Randomization (KASLR)

KASLR is the most basic hardening feature. The Windows kernel and its modules load at a different base address on every boot. With this feature in place, an attacker who knows where a particular target structure was on their own development machine has no way to know know where it will be on the target machine. If the attacker has a write primitive but cannot work out where to write, they cannot reliably tamper with the required kernel data structures. If they write to the wrong place, the machine is likely to crash.

KASLR makes it highly unlikely that two machines have the kernel laid out at exactly the same address. Against an attacker who has only a limited write primitive and no way to learn the kernel's layout, KASLR can be a meaningful obstacle.

The bypass, however, is well understood. Any process running with SeDebugPrivilege can call NtQuerySystemInformation with the SystemModuleInformation class to retrieve a list of every kernel module loaded on the machine, along with its base address. Administrators routinely have SeDebugPrivilege, and on some Windows versions and configurations even non-administrator users can obtain it. Once the attacker has the base address of the kernel module they care about, they can compute the address of any structure inside it, and KASLR provides no further protection.

Public tooling such as EDRSandblast and RealBlindingEDR use this approach. Once the attacker has administrator level access, KASLR is not a meaningful obstacle.

Virtualization-Based Security and Hypervisor-Protected Code Integrity

Virtualization-Based Security (VBS) and Hypervisor-Protected Code Integrity (HVCI) are a more substantial mitigation. On supported hardware, VBS introduces a secure kernel that runs alongside the traditional NT kernel under a hypervisor. The traditional kernel becomes, in effect, a client of the secure kernel for certain operations. Some sensitive data is moved out of the traditional kernel altogether, where an attacker running with admin or even kernel privileges in the traditional kernel cannot reach it.

The clearest example is Credential Guard. Before VBS, LSASS held credential material in its own address space, where tools such as Mimikatz could read and decrypt it once they had administrator access. With Credential Guard enabled, the credential material is managed by the secure kernel and is no longer reachable from the traditional kernel. A Mimikatz-style attack that walks the LSASS address space simply finds nothing useful. From Microsoft's point of view this is one of the few admin-to-kernel scenarios that genuinely is treated as a security boundary, because the secure kernel is considered separate from the traditional kernel.

HVCI applies a separate invariant to the traditional kernel itself. Pages that are executable cannot also be writable. Even if an attacker has obtained physical memory write access, for example via the GodFault primitive described previously, they cannot directly patch kernel code. Any attempt to write to a page that holds executable instructions is rejected. This forces the attacker to confine themselves to data tampering rather than code patching.

In principle, HVCI is a strong mitigation. In reality, it does not stop the attacks that matter most against security software, because most of those attacks are data-only to begin with. Removing entries from the kernel's notification-callback arrays, flipping the Protection field of an EPROCESS structure, unhooking a callback list, or unmapping memory inside a security process are all writes against data, not against code. HVCI does not restrict any of them. As long as the attacker only needs to read and write data structures, HVCI is essentially transparent to them.

There are also more advanced techniques designed specifically to defeat HVCI in cases where the attacker does want to call kernel code. KernelForge is a public research project that arranges stack frames in such a way that the Windows kernel itself executes the desired sequence of API calls on the attacker's behalf. Loading an unsigned driver, which would normally be blocked by code-integrity enforcement, is one of the operations the technique can support. KernelForge turns HVCI from a hard barrier into just another obstacle that can be worked around, but the bar is meaningfully higher than for a simple data-only attack.

Kernel Control Flow Guard and Intel CET

Kernel Control Flow Guard (kCFG) is Microsoft's mitigation for control-flow hijacking attacks in the kernel. The classic example of such an attack is return-oriented programming (ROP), where the attacker chains together short fragments of existing code, called gadgets, by manipulating return addresses or function pointers. kCFG inserts checks at indirect call sites, so that the kernel can only call into a known set of valid targets. An attacker who has loaded a function pointer into a register and is trying to call into the middle of a subroutine will be stopped.

However, kCFG is not a complete defense against ROP. The Windows kernel contains many megabytes of code, and the set of valid call targets is large enough that an attacker with enough time can still find a usable chain. It does, however, narrow the gadget set and make exploitation slower and more fragile.

Intel's Control-flow Enforcement Technology (CET), and in particular its Indirect Branch Tracking feature, is meaningfully stronger than kCFG. It uses hardware support to enforce that indirect branches can only land at instructions explicitly marked as valid targets. As far as we are aware, however, Indirect Branch Tracking is not currently built into the Windows kernel, even on CPUs that support it. Until Microsoft chooses to implement it, kCFG remains the relevant kernel mitigation.

As with the other hardening features, both kCFG and CET only matter when the attacker is trying to redirect control flow in the kernel. Data-only attacks bypass them entirely. Public research illustrates the gap: a 2024 HN Security writeup describes how an arbitrary pointer dereference in Windows 11 can be turned into an arbitrary read and write primitive without ever needing to bypass kCFG, simply by working only with data.

Kernel-mode Hardware-enforced Stack Protection

Kernel-mode Hardware-enforced Stack Protection complements kCFG by protecting the other end of the call sequence. Where kCFG focuses on indirect call instructions, stack protection focuses on returns. The mitigation maintains a separate shadow stack for return addresses, in hardware where available. When a function returns, the return address it pops off the regular stack is checked against the corresponding entry on the shadow stack. If they do not match, the return is blocked. This makes return-oriented programming significantly harder, because the attacker can no longer simply overwrite a return address on the stack to redirect control flow.

The bypass story is, by now, familiar. Data-only attacks ignore stack protection entirely, because they do not redirect control flow. For attacks that do need to redirect control flow, more advanced exploitation techniques are available.

One example is keyjumper, a public research technique that demonstrates a way of getting around shadow-stack enforcement under specific circumstances. Some attackers reach for keyjumper when they need a bypass; others prefer simpler routes when the situation allows. Either way, the same pattern applies: stack protection makes life harder, but with enough control over registers or memory, attackers can usually still reach their goal.

Hypervisor-Enforced Paging Translation

Hypervisor-Enforced Paging Translation (HVPT) is a more recent mitigation that uses the hypervisor to validate page table modifications in the traditional kernel. The goal is to prevent an attacker who has obtained kernel write primitives from forging or tampering with page table entries to remap memory in ways that defeat other mitigations.

In the context of attacks against security software, HVPT is unlikely to make a meaningful difference. Most of the practical attacks we see in the wild do not need to forge page- able entries. They tamper with EPROCESS protection bytes, kernel notification arrays, or process tokens, and HVPT does not restrict those operations. The mitigation is welcome, but it sits some distance from the most common bypass paths.

The common thread: data-only attacks

Looking across the hardening features as a group, a clear pattern emerges. Each one is designed to make some specific category of exploitation harder. KASLR makes it harder to find targets. HVCI makes it harder to patch code. kCFG and CET make it harder to redirect indirect calls. Stack protection makes it harder to redirect returns. HVPT makes it harder to forge page tables.

What they share is that they all assume the attacker needs to do one of these things in order to succeed. Against an attacker whose goal is to disable security software, that assumption is mostly wrong. The decisive operations against security software, removing kernel callbacks, flipping protection bytes on EPROCESS structures, stripping privileges from process tokens, and unmapping memory inside protected processes, are all achieved through writes against data structures. None of them require executing attacker code in the kernel. None of them require redirecting control flow. None of them require forging page tables.

Once an attacker has reliable read and write primitives in the kernel, typically obtained from a vulnerable signed driver, kernel hardening offers limited protection against these classes of attacks. Hardening can still slow down or stop the kinds of attack that need code execution, control-flow hijacking, or page-table manipulation. For the bulk of in-the-wild defense evasion, attackers simply work with the data they need to change and route around the hardening entirely.

This is not an argument against kernel hardening. Each of these mitigations closes off a class of attack that would otherwise be available, and in combination they limit the choices an attacker has. It is, however, an argument for not relying on kernel hardening alone. Defenders should not assume that an attacker who has reached kernel mode will be stopped by the kernel's own protections. In the face of these obstacles, they will simply switch to a data-only approach instead.

Protecting Against BYOVD

While multiple defense evasion techniques exist, BYOVD continues to dominate the picture. The natural question for the defender is what can help to protect against it. Two defensive approaches dominate today's response to BYOVD, and both have some limitations.

The first is Microsoft's Vulnerable Driver Blocklist. When a vulnerable driver is reported, Microsoft works with the publisher to address the issue and adds the driver to a blocklist so that up-to-date Windows installations refuse to load it. This is genuinely useful, but it has two problems. The first is latency. There is a lag of days, more often weeks, from the moment a driver is identified as vulnerable to the moment the blocklist update reaches enterprise endpoints. During that window the driver is freely usable in attacks. The second problem is volume. Hundreds of known vulnerable drivers are already in circulation, and only a subset is blocklisted at any given time. When defenders block one driver, attackers simply switch to the next.

The second approach is signature-based detection by security software. When researchers analyze a particular BYOVD tool, they calculate a hash and write a signature for the file. The signature flags any future appearance of that exact tool. While this works, the effectiveness time window is short. Many BYOVD tools begin life as open-source proof-of-concept projects on public code-sharing platforms. When the original project is flagged, attackers re-implement it within days, often in a different language. We often see the same underlying technique reimplemented in C, C++, C#, Rust, and Go in rapid succession. Each language change defeats the existing hash signature. Cryptors and packers compound the problem by mutating the binary even when the underlying logic is unchanged.

BYOVD tools are now frequently bundled inside ransomware-as-a-service packages, which gives affiliates a steady supply of fresh variants with operational support behind them.

The problem with these types of defenses is that they are purely reactive in nature. Each new variant has to be observed in the wild before it can be defended against, and by the time the defenders have reacted, the attackers have moved on.

Proactive, not reactive defenses

An approach that genuinely shifts the balance is to monitor what files do when they interact with kernel drivers, not simply which drivers are loaded. Whether a BYOVD tool is a year-old C++ project or a fresh Rust port written last week, the operational pattern is largely the same. A vulnerable driver is dropped to disk. A service is created and started to load it. The malicious component then enumerates running processes, identifies the targets it cares about, and sends a specific input/output control (IOCTL) command to the driver instructing it to take a privileged action against those targets.

Watching for that pattern, rather than for the specific driver or the specific binary that loaded it, gives defenders a vantage point that is independent of the surface details. Symantec and Carbon Black endpoint products now include behavioral monitoring of these driver interactions. The detection logic flags anomalous IOCTL traffic against kernel drivers, including process termination requests directed at security software, handle stripping that would crash a security process, and callback removal that would blind the EDR.

The combined detection logic operates at two layers:

- At the more specific layer, we maintain a catalog of known vulnerable drivers along with the IOCTL codes those drivers accept for sensitive operations. When a driver in the catalog is observed receiving a sensitive IOCTL, the request is examined. If the target is a process known to be security-relevant, the activity is blocked. The catalog is broad, and it is extended as new vulnerable drivers are identified and analyzed.
- At a more general layer, we recognize that a request to terminate a running security software process is anomalous regardless of which driver is being asked to perform it. By tracking the process identifiers of security software on the endpoint, the system can detect attempts to direct an IOCTL against one of those identifiers, even when the driver being abused is one that has not previously been seen. This generic layer is what catches an unknown vulnerable driver the first time it appears in the wild, before it can be added to any catalog or blocklist.

Why behavioral monitoring catches what static defenses miss

Three properties make behavioral monitoring meaningfully different from the defensive approaches it complements.

- It is driver-agnostic. The generic layer of the detection flags sensitive IOCTLs directed against security software process identifiers no matter which driver is involved. Attackers cannot defeat this protection simply by switching to a less-known vulnerable driver, because the detection does not depend on identifying the driver in advance.
- It is independent of Microsoft's release cadence. The Vulnerable Driver Blocklist updates intermittently, typically with major Windows releases. Behavioral protection ships through the security software's update channel, which enables us to react rapidly to changing threats. A new vulnerable driver that emerges can be covered by behavioral protection within hours rather than weeks or months.
- It carries a low false-positive risk. Legitimate kernel drivers have predictable behavior profiles. They are built to serve a specific product, they accept a known set of commands from a known set of clients, and in normal use they do not terminate security software on demand. Blocking the malicious behavior pattern leaves legitimate driver use unaffected.

The net effect is that the time advantage attackers have enjoyed since BYOVD became popular begins to narrow. A new vulnerable driver, a new loader written in a new language, a new packer applied to an old loader: none of these on their own defeat behavioral protection, because none of them change the operational pattern that behavioral protection watches for.

The wider lesson

BYOVD is the dominant in-the-wild defense evasion technique today, but it is part of a longer trend. As Microsoft and the security software industry have hardened the user-mode and kernel-mode interfaces, attackers have moved to abuse the privileged code that the operating system itself trusts. The same shift is visible in protected process tampering, in network silencing of EDR cloud services, and in the gradual industrialization of all these techniques inside ransomware-as-a-service offerings.

The defensive response cannot be limited to writing signatures for each new variant or waiting for each new vulnerable driver to be blocklisted. The signature-and-blocklist approach buys time, but the time it buys is shrinking. What is needed in addition is sustained attention on the behaviors an attacker has to perform regardless of which tool they reach for. Only Symantec and Carbon Black Endpoint products provide comprehensive protection at a behavioral level.