Abusing Windows Drivers For Privilege Escalation

Hello everyone,

This post is a theoretical and practical walkthrough of one of the most abused vectors for privilege escalation and EDR bypass on Windows: vulnerable kernel drivers. This technique is widely used by APT groups and security researchers. Let's see some statistics to make it more interesting.

BYOVD Discovery Statistics

Known Vulnerable Drivers:

- 700+ in LOLDrivers project
- 924 known vulnerable 64-bit signed drivers (Check Point, June 2024)
- 90% accessible by non-privileged users

The Unknown:

- 5,589 potentially vulnerable drivers found via dangerous API import analysis (EURECOM)
- 22,500 at-risk drivers flagged in 1-year VirusTotal retrohunt (Check Point)
- 7 new weaponizable drivers discovered through automated analysis - none were on any blocklist (NDSS 2026)

Blocklist Lag:

- Average 6+ months from public disclosure to Microsoft blocklist update
- Many drivers never get added

How Driver Abuse Works

A driver is a piece of software that runs in kernel mode and performs specific low-level tasks. Because drivers execute with kernel-level privileges, a vulnerable driver can often be abused to gain arbitrary kernel read/write or other powerful primitives.

Once we find an abusable driver that is accessible from a low-privileged user, we can often:

- Escalate privileges to `NT AUTHORITY\SYSTEM`
- Bypass EDR or security controls that rely on kernel integrity
- Manipulate kernel memory, process tokens, or critical structures

The first step is to find such a driver that:

1. Is installed on the target system
2. Is accessible by a regular (non-admin) user
3. Exposes dangerous functionality through IOCTL handlers

Accordingly, I divided the blog into four steps:

1. Driver Enumeration
2. Reversing and Analyzing the Driver
3. Identifying the Correct IOCTLs
4. Exploit Writing

Step 1: Driver Enumeration

We start from a regular user context and look for user-accessible drivers.

You can do this manually with built-in commands:

```
sc query type= kernel > drivers.txt
start drivers.txt
```

From there, you would manually inspect:

- Driver paths
- Signatures
- Vendors

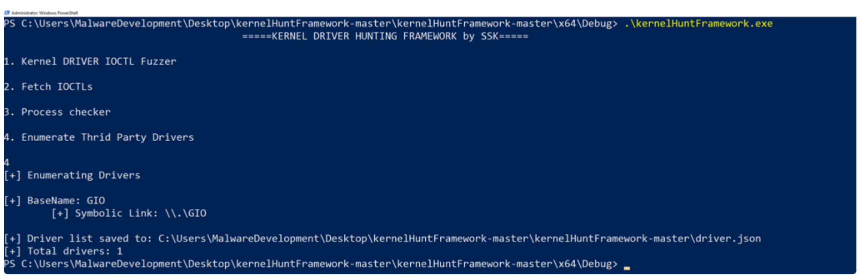
The goal is to find Microsoft-trusted third-party drivers from vendors like ASUS, NVIDIA, Dell, and similar, where vulnerabilities are commonly found and sometimes persist for a long time.

To increase efficiency, we can automate this with tooling. For example, using the enumeration feature in KernelHuntFramework will:

> **Note:** KernelHuntFramework should be run on Windows 10, not Windows 11, due to `EnumDeviceDrivers` / `NtQuerySystemInformation` restrictions in newer builds. See [OS Version Notes](#) for details.

- Enumerate installed kernel drivers
- Identify drivers that are user-accessible
- Produce a JSON file for further analysis

From the output, we identify a vulnerable driver **GIO.sys** as a candidate for exploitation.



Step 2: Reversing and Analyzing the Driver

Next, we reverse and analyze the target driver.

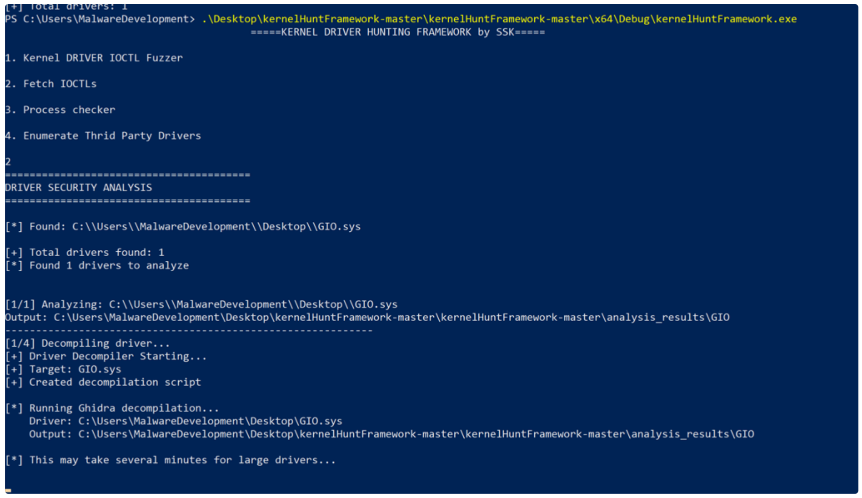
The manual approach would be:

1. Identify the `DriverEntry` function
2. Locate the `DriverObject->MajorFunction` (dispatcher) table
3. Enumerate and analyze each IOCTL handler
4. Understand what each IOCTL does and whether it can be abused

This is time-consuming, so we use `KernelHuntFramework.exe` to automate a large part of this analysis.

The tool:

- Decompiles the driver using Ghidra's headless analyzer
- Produces a dump of relevant functions
- Generates an HTML report describing potentially dangerous functions and IOCTLs based on their capabilities and severity



From the HTML report, we see that this driver exposes multiple functions that can be abused. The framework flagged several dangerous primitives, including:

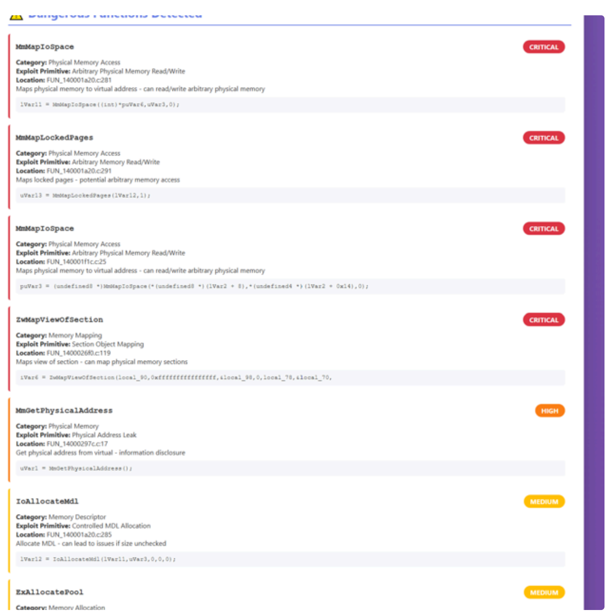
- **ZwMapViewOfSection** – Map kernel memory into user space
- **ZwUnmapViewOfSection** – Unmap previously mapped memory
- **Additional abusable functions** – The report identified other potentially dangerous IOCTLs that could be chained or used for alternative exploitation paths (for example, primitives that allow physical memory access, MSR read/write, or I/O port access, depending on what the driver exposes)

For this blog, we focus on `ZwMapViewOfSection` as our primary primitive.

`ZwMapViewOfSection` can be abused to:

- Map kernel memory into user space
- Leak sensitive information such as the System process token
- Use our kernel-level primitive to swap the current process token with the System token

This forms the core of the privilege escalation path.



Step 3: Identifying the Correct IOCTLs

The next step is to identify the exact IOCTL codes that trigger the dangerous behavior.

We can do this in two ways:

- Directly from the HTML report generated by KernelHuntFramework (IOCTL codes section)
- Manually in IDA or another disassembler

Since we already know which function is interesting (`ZwMapViewOfSection`) and its location from the report, we can:

1. Load the driver into IDA
2. Navigate to the function that corresponds to the IOCTL handler (for example, a function whose name or suffix matches what is reported, such as one ending with `f0`)
3. Synchronize the pseudocode view with the graph view
4. Cross-reference the function to find where it is used in the dispatch routine

From this, we determine:

- The IOCTL for `ZwMapViewOfSection` is: `0xC3502004`
- The IOCTL for `ZwUnmapViewOfSection` is: `0xC3502008`

These codes are what we will use in our exploit to map and unmap kernel memory from a user process.

Verifying Our Current Privileges

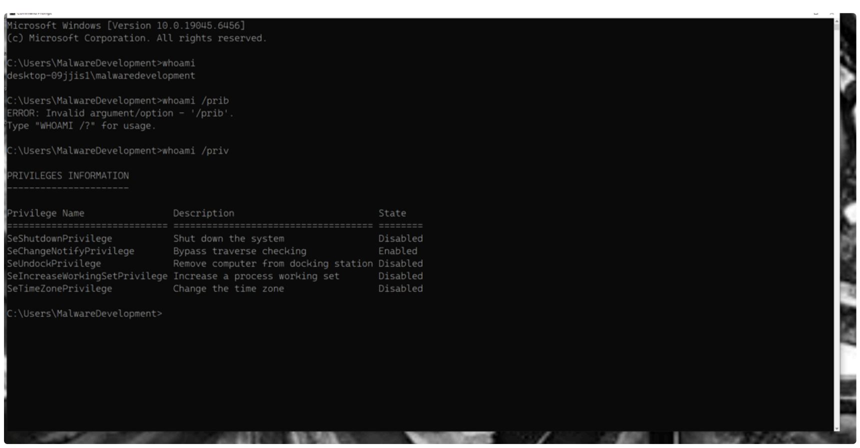
Before exploitation, we verify that we are indeed running as a low-privileged user. At this point, we should see that the current context is a regular user account, not SYSTEM.

This confirms that any later escalation is the result of abusing the vulnerable driver, not prior privileges.

From this, we determine:

- The IOCTL for `ZwMapViewOfSection` is: `0xC3502004`
- The IOCTL for `ZwUnmapViewOfSection` is: `0xC3502008`

These codes are what we will use in our exploit to map and unmap kernel memory from a user process.



Step 4: Exploit Development

Writing a full exploit from scratch is out of scope for this blog. Instead, the approach taken here is (runnable exploit will be attached):

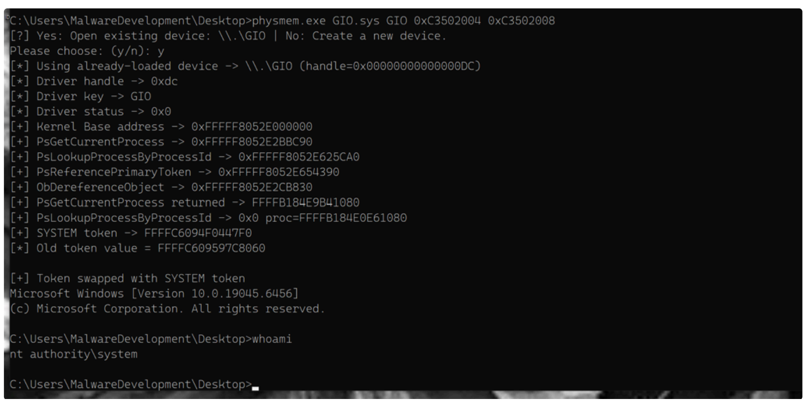
- Reuse an existing Windows kernel exploit template (based on physmem)

- Adapt it to this specific driver and its IOCTLs
- Update structures, IOCTL codes, and offsets where necessary

Exploit Strategy:

This exploit uses a syscall hooking technique rather than manual EPROCESS traversal. The driver's physical memory map/unmap primitives allow us to:

1. **Establish physical memory read/write** using the vulnerable IOCTLs (`0xC3502004` for map, `0xC3502008` for unmap)
2. **Locate a syscall in physical memory** by scanning physical memory ranges for the byte signature of an infrequently used syscall (`NtShutdownSystem`)
3. **Patch the syscall** with a JMP instruction pointing to our target kernel function, call it from usermode, then restore the original bytes - effectively giving us the ability to invoke any exported kernel API
4. **Steal the SYSTEM token** by calling kernel APIs directly:
   - `PsGetCurrentProcess` → get our process's EPROCESS
   - `PsLookupProcessByProcessId(4)` → get SYSTEM's EPROCESS
   - `PsReferencePrimaryToken` → get SYSTEM token
   - Write SYSTEM token to our process's Token field
5. **Spawn elevated shell** and restore the original token on exit for clean operation



Why Syscall Hooking?

Approach	Description
Manual EPROCESS walk	Data-only, but requires leaking addresses and traversing kernel structures yourself
Syscall hook	More flexible - call any kernel API (`ZwTerminateProcess` , `ZwOpenProcess` , etc.) directly from usermode

Build-Specific Considerations:

The `Token` offset within `EPROCESS` varies by Windows build. For Windows 10/11 builds 19041–22631, the offset is typically `0x4b8`. Verify using WinDbg:

```
dt nt!_EPROCESS Token
```

Once the exploit runs successfully, we obtain a shell with `NT AUTHORITY\SYSTEM` privileges.

Post-Exploitation and EDR Bypass

Once the exploit runs successfully:

- We obtain a shell or process with `NT AUTHORITY\SYSTEM` privileges
- Any EDR or security mechanism that relies on trusting kernel-mode operations can now be bypassed or tampered with, depending on what the driver allows

Examples of what may be possible, depending on the additional functions the driver exposes:

- Disabling or patching kernel callbacks used by EDR
- Manipulating process protections or handle tables
- Reading or tampering with other protected kernel data structures

This is where the other abusable functions identified in the HTML report come into play. Even if `ZwMapViewOfSection` is your primary primitive for token stealing, primitives such as:

- Physical memory read/write
- MSR read/write
- I/O port access

can enable alternative exploitation chains or more powerful post-exploitation actions (for example, tampering with hypervisor-related state or low-level platform configuration).

OS Version Notes and Framework Usage

> **Note:** Finding previously unknown vulnerable drivers remains the most effective BYOVD approach, as no blocklist can protect against what isn't documented yet.

Framework Setup:

Run KernelHuntFramework with Ghidra which should be present in the path specified below:

```
C:\ghidra_11.0_PUBLIC
```

Windows 10:

This exploit chain (used in the demonstration) is reliable on Windows 10, where VBS/HVCI is disabled by default. Traditional token-stealing shellcode works without issue.

Windows 11:

Windows 11 enables VBS/HVCI by default, which introduces several complications:

Change	Impact
HVCI enabled by default	Blocks unsigned shellcode execution in kernel

MBEC (Mode-Based Execution Control)	Prevents U/S PTE bit flip bypass
kCFG (Kernel Control Flow Guard)	Makes ROP gadget chaining harder
Stricter API access	`EnumDeviceDrivers` / `NtQuerySystemInformation` leaks patched in some builds
EPROCESS offsets changed	`Token`, `ActiveProcessLinks`, `UniqueProcessId` offsets vary per build

Check if VBS/HVCI is active:

```
Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard | Select-Object VirtualizationBasedSecurityStatus, SecurityServicesRunning
```

- `VirtualizationBasedSecurityStatus = 2` → VBS running
- `SecurityServicesRunning contains 2` → HVCI active

On the Vulnerable Driver Blocklist:

Microsoft maintains a blocklist of known vulnerable drivers integrated with HVCI. However, this only blocks known drivers. Discovering new vulnerable drivers bypasses this protection entirely - the driver is legitimately signed and not yet blocklisted

Links

- LOLDrivers
- Microsoft Vulnerable Driver Blocklist
- HVCI Documentation
- Kernel Hunt Framework
- Exploit and driver

physmem.exe
27 Nov 2025, 01:29 AM

GIO.sys
27 Nov 2025, 01:33 AM