# Making Vulnerable Drivers Exploitable Without Hardware - The BYOVD Perspective

**Author**: Julian Horoszkiewicz, Atos Threat Research Center

## Table of contents

## 1 Introduction

This article provides a technical analysis of how many Windows kernel mode drivers can be interacted with from user mode without the hardware they were developed for. This work was motivated by driver-oriented vulnerability research and the need to evaluate the exploitability of individual findings, which frequently affect code whose reachability is hardware-gated. The methodology presented here should help anyone determine whether a particular Windows kernel mode driver vulnerability remains reachable — and thus potentially exploitable — even in the absence of the hardware the driver was developed for.

The reader is expected to have basic Windows driver knowledge, especially regarding device objects. The rest of this article is written with the assumption that the reader is already familiar with the concepts described in the introduction article: Anatomy of Access: Windows Device Objects from a Security Perspective.

Just like the introduction article, this resource is not focused on any specific bug class, but rather the attack surface and, to an extent, the Windows Plug and Play architecture.

All the tests demonstrated here were conducted on Windows 11 23H2 (winver 10.0.22631.3007).

## 2 The offensive value of kernel mode drivers

In addition to the obvious Local Privilege Escalation potential, vulnerable drivers are often abused in BYOVD attacks - a post-exploitation technique leveraged by attackers to disrupt system defenses such as EDR components.

Two main criteria determine whether a driver vulnerability is a strong candidate for BYOVD attacks:

1. Exploitation allows meaningful disruption of an otherwise tamper-resistant security component. Examples include kernel-level vulnerabilities granting arbitrary memory read/write access, arbitrary code execution, or arbitrary resource abuse (e.g., overwriting files, closing handles, or terminating processes).
2. Its exploitability is independent of rare system conditions, such as the presence of specific hardware.

Although BYOVD-style attacks have been well documented for years, with numerous public reports and research papers on the topic (e.g. https://www.ndss-symposium.org/wp-content/uploads/2026-s1491-paper.pdf, https://blackpointcyber.com/blog/qilin-ransomware-and-the-hidden-dangers-of-byovd/, https://www.sophos.com/en-us/blog/itll-be-back-attackers-still-abusing-terminator-tool-and-variants), none of them specifically examines the role of hardware-gating in driver vulnerability reachability.

## 3 Device object creation and maintenance - common patterns

The analysis provided in this resource is structured around device objects, because they are the most viable attack vector. However, the techniques demonstrated here practically impact driver code reachability from userland in general, not just via IRP.

The most common obstacles in attacking a driver via its device object are:

1. The device object is not created.
2. The driver's internal state does not allow to exercise the vulnerable behavior despite the device object being accessible.

Both scenarios are very common when dealing with a device driver deployed on a system without the corresponding physical hardware.

In the rest of the article I am often referring to device stacks and device nodes. I have covered device stacks quite broadly in my introduction article. While a device node and a device stack are not the

same thing, the terms are often used interchangeably, because every device node has exactly one device stack.

### 3.1 Unconditional creation upon driver load

Many drivers, especially non-PnP drivers, create their device objects either directly from within their DriverEntry function, or from some other function that gets invoked in the direct call chain originating from DriverEntry.

Multidev_WDM demo driver exemplifies this pattern. We can see the device creation invoked right away in DriverEntry:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    NTSTATUS        status;
    UNICODE_STRING  devName, linkName, sddl;

    UNREFERENCED_PARAMETER(RegistryPath);

    DbgPrint("[multidev] DriverEntry\n");

    DriverObject->MajorFunction[IRP_MJ_CREATE]         = DispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]          = DispatchClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDeviceControl;
    DriverObject->DriverUnload                         = DriverUnload;

    // --- Device 1: Admin-only ---
    RtlInitUnicodeString(&devName, ADMIN_DEV_NAME);
    RtlInitUnicodeString(&sddl, SDDL_ADMIN);

    status = IoCreateDeviceSecure(
        DriverObject,
        0,                          // DeviceExtensionSize
        &devName,
        FILE_DEVICE_UNKNOWN,
        FILE_DEVICE_SECURE_OPEN,
        FALSE,                      // Exclusive
        &sddl,
        //&MULTIDEV_GUID,
        NULL,
        &g_AdminDevice);
```

NO CONDITIONS, JUST CREATE THE DEVICE OBJECT

The driver also removes the device object by calling IoDeleteDevice, but that happens only when DriverUnload is called (when the driver is being unloaded):

```
static void Cleanup(void)
{
    UNICODE_STRING linkName;

    if (g_AdminLink) {
        RtlInitUnicodeString(&linkName, ADMIN_LINK_NAME);
        IoDeleteSymbolicLink(&linkName);
    }
    if (g_PublicLink) {
        RtlInitUnicodeString(&linkName, PUBLIC_LINK_NAME);
        IoDeleteSymbolicLink(&linkName);
    }
    if (g_AdminDevice)
        IoDeleteDevice(g_AdminDevice);
    if (g_PublicDevice)
        IoDeleteDevice(g_PublicDevice);
}

VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
    UNREFERENCED_PARAMETER(DriverObject);
    DbgPrint("[multidev] Unloading\n");
    Cleanup();
}
```

Drivers built this way can be interacted with after simple deployment consisting of just two steps:

1. Create the driver's service entry:

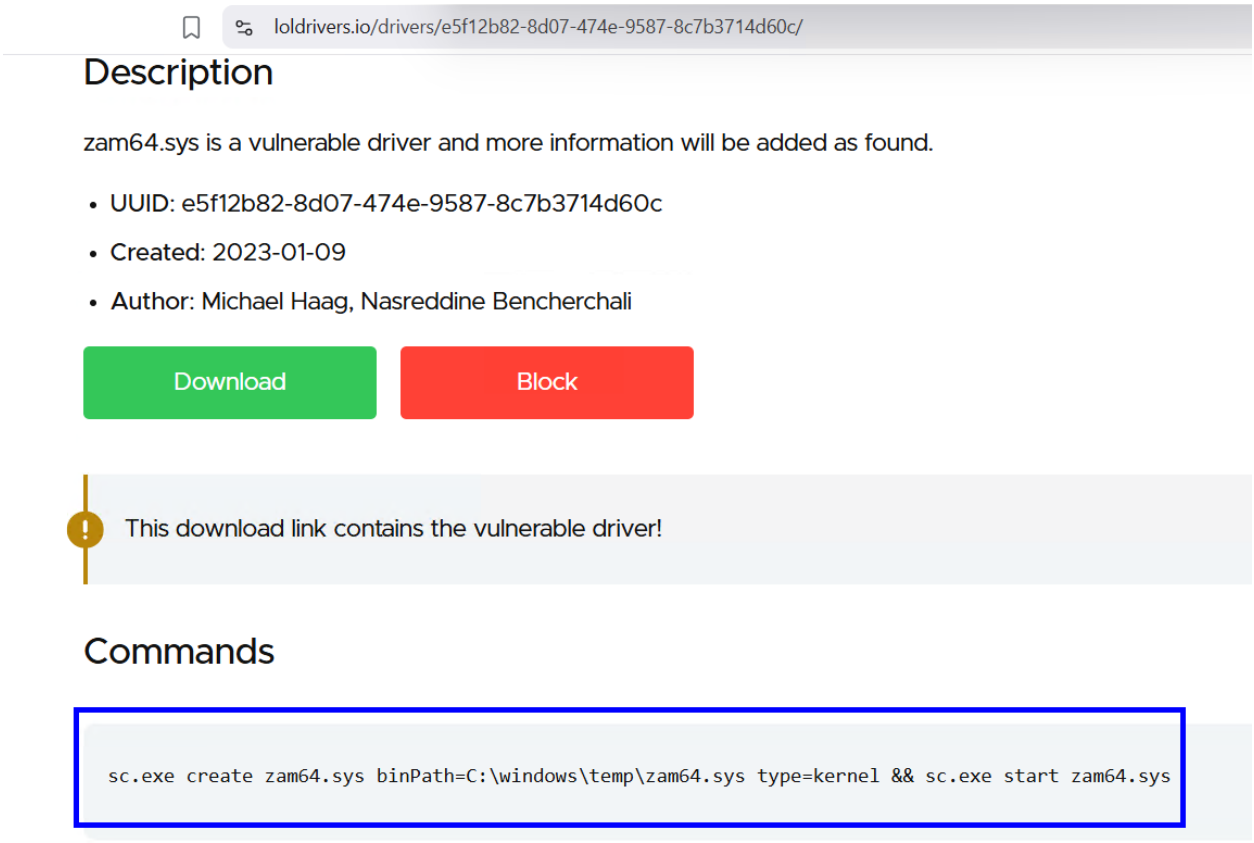
```
sc.exe create SampleDrv type= kernel start= demand binPath=
↪  System32\drivers\SampleDrv.sys
```

- Start the service (driver will load):

2.

```
sc.exe start SampleDrv
```

If we look at a randomly picked driver from https://loldrivers.io/, we will see that its deployment command matches this pattern:




But most device drivers do not fall into this category, as we will see in the following sections of this article.

### 3.2 Conditional device creation and maintenance

Oftentimes driver initialization routines perform additional checks. For example, kernel mode components of security software (EDR, anti-virus, monitoring, enhanced authentication etc.) tend to check for product-specific registry keys and entries, which are created and initialized during normal product deployment.

Actual device drivers (created to drive physical hardware) tend to only create their device objects in the presence of that hardware. Without it they either:

- do not attempt to create any device objects at all,
- they remove any device objects shortly after their creation, by calling IoDeleteDevice.

Let's focus on how that logic is implemented and evaluate whether and how it can be worked around, especially from the BYOVD perspective - by solely operating from userland (with no physical/hypervisor access).

By the way, the second scenario, in which a device object is first created and then deleted shortly after, creates a situation that could be considered a race condition, because there is a short time window in which the device object exists.

### 3.3 PnP-specific callbacks as the main location of PnP driver initialization logic

In PnP-compatible drivers (which makes most of device drivers), initialization logic extends beyond DriverEntry into the following PnP-specific routines: AddDevice and the IRP_MJ_PNP handler.

This section explores both of them, and explains why most PnP-compatible drivers need to be set up in a way that ensures these functions are called if we want to interact with the driver.

### 3.3.1 AddDevice

All PnP-compatible drivers must define this routine. It is responsible for creating functional device objects (FDO) and filter device objects (filter DO) for devices enumerated by the PnP manager. Which explains why AddDevice is where most of the initialization logic resides. That includes:

- creation of device objects (IoCreateDevice),
- initialization of various internal variables that are later required to reach the vulnerable code,
- I/O queue management in WDF (KMDF) drivers.

The MSDN page about managing I/O queues in WDF drivers says:

> Drivers typically call WdfIoQueueCreate from within an EvtDriverDeviceAdd callback function. The framework can begin delivering I/O requests to the driver after the driver's EvtDriverDeviceAdd callback function returns.

In the context of WDF (KMDF) drivers, AddDevice is referred to as EvtDriverDeviceAdd (different name, same application).

**AddDevice is not called from within the DriverEntry routine, which means it does not automatically execute upon driver load. Instead, the PnP manager invokes it only after it discovers a new device node and determines that this driver should either control the device directly or serve as a filter in the device stack.**

Let's look at some code. Note: all structure-specific offsets are for the 64-bit architecture.

Both in DriverEntry and in AddDevice, the first parameter the function receives is a pointer to the DRIVER_OBJECT structure. As we can read on the MSDN page, the structure is allocated by the I/O manager:

> The I/O manager allocates the DRIVER_OBJECT structure and passes it as an input parameter to a driver's DriverEntry, AddDevice, and optional Reinitialize routines and to its Unload routine, if any.

DRIVER_OBJECT contains pointers to the driver's dispatch routines, each at a specific offset (e.g. 0xe0 for IRP_MJ_DEVICE_CONTROL).

The pointer to AddDevice, however, is not stored directly in the DRIVER_OBJECT structure, but in the DRIVER_EXTENSION structure, accessed via DriverObject->DriverExtension->AddDevice. This fact is mentioned on the same MSDN page:

> Pointer to the driver extension. The only accessible member of the driver extension is DriverExtension->AddDevice, into which a driver's DriverEntry routine stores the driver's AddDevice routine.

So in the decompiler, the AddDevice assignment typically looks like:

```
// DriverObject->DriverExtension->AddDevice = SomeFunction;
*(*(param_1 + 0x30) + 8) = FUN_XXXXX;
```

So, a typical initialization sequence for driver dispatch routines and other standard callbacks we can usually find in a device driver's DriverEntry function looks like this (decompiled in Ghidra, comments added manually):

```
*(code **)(param_1 + 0x70) = FUN_00011a08;  // IRP_MJ_CREATE dispatch routine
*(code **)(param_1 + 0x80) = FUN_00011a08;  // IRP_MJ_CLOSE dispatch routine
*(code **)(param_1 + 0xe0) = FUN_00010614; // IRP_MJ_DEVICE_CONTROL dispatch routine
*(code **)(param_1 + 0xe8) = FUN_000104ac; // IRP_MJ_INTERNAL_DEVICE_CONTROL
*(code **)(param_1 + 0x148) = FUN_00011c70; // IRP_MJ_PNP dispatch routine
```

```
*(code **)(param_1 + 0x120) = FUN_00011bc8; // IRP_MJ_POWER dispatch routine
*(code **)(*(longlong *)(param_1 + 0x30) + 8) = FUN_00011ad4; // AddDevice
*(code **)(param_1 + 0x68) = FUN_00011b8c; // DriverUnload
```

So, AddDevice is defined in FUN_00011ad4 and upon driver load (DriverEntry execution) its pointer is written into DriverObject->DriverExtension->AddDevice, just as all dispatch routine pointers are written into their relevant offsets. But none of those functions have been invoked yet. For example, FUN_00010614 (IRP_MJ_DEVICE_CONTROL) will only execute once the driver receives an IRP with MajorFunction code = IRP_MJ_DEVICE_CONTROL (e.g. in response to DeviceIoControl call from userland). Likewise, AddDevice is not called by the driver itself, but rather by the PnP manager under specific circumstances.

Now, let's look into FUN_00011ad4 and see how a typical AddDevice implementation looks like:

```
undefined8 FUN_00011ad4(undefined8 param_1,undefined8 param_2)

{
  longlong lVar1;
  longlong lVar2;
  undefined8 uVar3;
  undefined8 uVar4;
  undefined8 uVar5;
  undefined8 uVar6;
  longlong local_res18 [2];

  local_res18[0] = 0;
  lVar1 = *(longlong *)(DAT_00011880 + 0x40);
  uVar3 = IoCreateDevice(param_1,0x100,0,0x22,0,0,local_res18);
  if (-1 < (int)uVar3) {
    lVar2 = *(longlong *)(local_res18[0] + 0x40);
    *(undefined1 *)(lVar2 + 5) = 0;
    *(undefined1 *)(lVar2 + 4) = 0;
    *(undefined8 *)(lVar2 + 0x18) = 0;
    *(undefined8 *)(lVar2 + 0x10) = param_2;
    *(longlong *)(lVar2 + 8) = local_res18[0];
    *(undefined4 *)(lVar2 + 0x20) = 0x10000004;
    ExInterlockedInsertHeadList(lVar1,lVar2 + 0x28,lVar1 + 0x18);
    LOCK();
    *(int *)(lVar1 + 0x10) = *(int *)(lVar1 + 0x10) + 1;
    UNLOCK();
    KeInitializeEvent(lVar2 + 0x50,1);
    *(undefined4 *)(lVar2 + 0x68) = 1;
    *(uint *)(local_res18[0] + 0x30) = *(uint *)(local_res18[0] + 0x30) & 0xfffffff7f;
    uVar3 = IoAttachDeviceToDeviceStack(local_res18[0],param_2);
    *(undefined8 *)(lVar2 + 0x18) = uVar3;
    uVar3 = 0;
    local_res18[0] = 0;
    RtlInitUnicodeString(&DAT_00011870,u_\Device\SampleDrv_00012270);
    uVar4 = IoCreateDevice(param_1,0x40,&DAT_00011870,0x22,0,0,local_res18);
    if (-1 < (int)uVar2) {
        RtlInitUnicodeString(&DAT_00011860,u_\DosDevices\SampleDrv_000122a0);
        uVar5 = IoCreateSymbolicLink(&DAT_00011860,&DAT_00011870);
        uVar6 = (ulonglong)uVar5;
        if ((int)uVar5 < 0) {
            IoDeleteDevice(*(undefined8 *)(param_1 + 8));
```

```
        }
      }
    }
    return uVar3;
}
```

As we can see, two separate device objects are created. First, we have the following call to IoCreateDevice, whose returned value is saved in uVar3:

```
uVar3 = IoCreateDevice(param_1,0x100,0,0x22,0,0,local_res18);
```

The first param - param_1 - is a pointer to the driver object.

The second parameter is the requested device extension size (0x100) for the newly created device. As the MSDN page says:

> The device extension is the most important data **structure associated with a device object**. Its internal **structure is driver-defined**, and it's typically used to:
>
> Maintain device state information. Provide storage for any kernel-defined objects or other system resources, such as spin locks, used by the driver. Hold any data the driver must have resident and in system space to carry out its I/O operations.

Device extension (**individual for every device object**) is not the same thing as driver extension (offset 0x30 in the DRIVER_OBJECT) mentioned earlier (where AddDevice pointer, if present, is stored at offset 0x8). I am emphasizing the difference, because both terms sound similar, which may create confusion. We will get back to the most common application of the device extension structure later in this section.

The third parameter is the device name - in this case empty (unnamed device object), which is typical for FDOs.

Looking further, after FDO creation we have a whole block of code, which only executes if device object creation was successful:

```
if (-1 < (int)uVar3)) {
```

Several instructions farther in that block we have a call to IoAttachDeviceToStack:

```
uVar3 = IoAttachDeviceToDeviceStack(local_res18[0],param_2);
```

In AddDevice callback param_2 holds a pointer to the PDO created by the relevant bus driver.

Since AddDevice is invoked by the PnP manager, both parameters - param_1 pointing at the DRIVER_OBJECT and param_2 pointing at the PDO (DEVICE_OBJECT) - are provided by the PnP manager.

So at this point we can clearly see that only if AddDevice is invoked will the driver create its FDO (and attach it to a device stack, making it accessible for IRP processing via handles opened on the PDO).

Most PnP drivers only create one device object (FDO) in their AddDevice, and attach that object to a device stack, on top of the PDO pointed by param_2.

This particular driver, however, also creates a CDO:

```
Var4 = IoCreateDevice(param_1,0x40,&DAT_00011870,0x22,0,0,local_res18);
```

Note that the third parameter is not 0 (which means a device name is provided). And there is no IoAttachDeviceToStack call on that device object. So the device object is named and standalone - typical CDO.

Both device objects are IRP entry points, and this driver will only create them when AddDevice is called.

This structure applies to all FDOs and filter DOs. In this particular driver we also have a CDO created in the AddDevice callback.

Additionally, AddDevice is where drivers initialize their custom internal structures, including the ones located in device extension structures. If we look back into the AddDevice function above, we have such an example right in the beginning of the conditional code block, starting with this line:

```
lVar2 = *(longlong *)(local_res18[0] + 0x40);
```

local_res18[0] holds a pointer to the device object created by the preceding IoCreateDevice call. In a DEVICE_OBJECT, 0x40 is the offset of the device extension structure. So lVar2 points at the device extension. Then, the next 7 instructions perform various initializations at arbitrary offsets of the device extension structure:

```
*(undefined1 *)(lVar2 + 5) = 0;
*(undefined1 *)(lVar2 + 4) = 0;
*(undefined8 *)(lVar2 + 0x18) = 0;
*(undefined8 *)(lVar2 + 0x10) = param_2;
*(longlong *)(lVar2 + 8) = local_res18[0];
*(undefined4 *)(lVar2 + 0x20) = 0x10000004;
ExInterlockedInsertHeadList(lVar1,lVar2 + 0x28,lVar1 + 0x18);
```

The contents of the device extension structure is how WDM drivers usually recognize (make distinction) between device objects used to deliver the current IRP. It makes sense - after all, the device extension is a structure inside the device object, not the driver object. So upon device object creation the driver may put different values into individual device extension fields, so later when a pointer to that device is received in param_1 by a dispatch routine, the routine can read those values and use them in `if` conditions. Oftentimes, vulnerable code in dispatch routines sits behind such conditional blocks, making vulnerable execution paths depend on the specific device object used to deliver the IRP.

Now it becomes clear why **having AddDevice called is crucial**:

1. It is required for the driver to initialize properly, which is oftentimes required for vulnerable code to become reachable from userland. This includes both:

a) Otherwise-inaccessible conditional code branches.

b) CDO creation (device object serving as entry point to the driver).

2. **More importantly, the purpose of AddDevice is to create a new PnP-compatible (unnamed FDO/FiDO) device object and attach it to the device stack** on top of the PDO provided by the PnPManager in the second argument (`[in] _DEVICE_OBJECT *PhysicalDeviceObject`). Which means that AddDevice is the function that connects the driver (via its FDO/FiDO) to a newly created device stack, allowing IRP travel.

For each driver, **multiple independent interaction (attack) vectors may exist**. Their activation depends on proper driver initialization and typically materializes in one of the following forms:

1. CDOs created from within the AddDevice routine. Most PnP-compatible drivers do not create CDOs, but some do.
2. FDOs and FiDOs created within AddDevice and attached on top of a newly created device stack. These devices can only be accessed via the stack.

### 3.3.2 IRP_MJ_PNP

IRP_MJ_PNP is a MajorFunction IRP code dedicated for PnP-related interactions. Each PnP-compatible driver must handle this code with a dedicated dispatch routine, often referred to as DispatchPnP.

As the above MSDN page reads:

Associated with the IRP_MJ_PNP function code are several minor I/O function codes (see Plug and Play Minor IRPs), some of which all drivers must handle and some of which can be optionally handled. The PnP manager uses these minor function codes to direct drivers to start, stop, and remove devices and to query drivers about their devices.

While these routines are not as critical as AddDevice, because they are not responsible for the creation of the PnP-type device object, they usually implement other usual steps of driver initialization logic, such as:

- initialization of global driver-internal variables,
- configuration file checks,
- device interface registration,
- hardware probing and validation.

It is worth keeping in mind that there is a difference in how WDM and WDF drivers structure those callbacks in their code. WDM drivers set a traditional IRP_MJ_PNP dispatch routine on the `DriverObject->MajorFunction` table. Any processing of PnP minor IRPs is handled in that routine. WDF (KMDF) drivers register PnP/power state-change callbacks via WdfDeviceInitSetPnpPowerEvent-Callbacks, which provides clear separation of functions dedicated for handling individual minor IRPs. These differences become relevant during static analysis and debugging, but they do not affect they way drivers are set up from userland to get those routines properly invoked.

### 3.4 Active hardware interaction and probing

Only a small fraction of driver code actually interacts with physical hardware.

The relevant direct and indirect interaction mechanisms include:

- legacy x86 port I/O (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-read_port_uchar, https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-write_port_uchar and related IN/OUT instruction wrappers),
- Memory-Mapped I/O (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmapiospace, https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-read_register_ulong and variants),
- PCI configuration space (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-halgetbusdatabyoffset),
- ACPI control methods (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/acpiioct/ni-acpiioct-ioctl_acpi_eval_method),
- Serial Peripheral Bus (https://learn.microsoft.com/en-us/windows-hardware/drivers/spb/spb-ioctls and related SPB I/O requests),
- GPIO (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/gpio/ni-gpio-ioctl_gpio_read_pins),
- DMA (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iogetdmaadapter),
- interrupts (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioconnectinterruptex),
- calls to other drivers via https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iofcalldriver.

When considering hardware-gated code and by extension hardware-gated vulnerabilities, it is crucial to understand the context. To illustrate this, let's consider three different examples, all involving the same mechanism - MMIO.

#### 3.4.1 Neutral hardware use

Fixed address 0xFEE00000, universally present:

```
// Local APIC - fixed at 0xFEE00000 on all x86 systems
base = MmMapIoSpace(0xFEE00000, PAGE_SIZE, MmNonCached);
version = READ_REGISTER_ULONG(base + 0x30);
MmUnmapIoSpace(base, PAGE_SIZE);
```

No hardware-gating, no security impact.

### 3.4.2 Vulnerable hardware use

In this scenario, we have an arbitrary physical memory write (vulnerable use of MmMapIoSpace, followed by WRITE_REGISTER_ULONG). It is unconditionally reachable — any system running the driver is exposed:

```
// Physical address and offset supplied by usermode via IOCTL
base = MmMapIoSpace(input->PhysicalAddress, input->Size, MmNonCached);
WRITE_REGISTER_ULONG(base + input->Offset, input->Value);
MmUnmapIoSpace(base, input->Size);
```

### 3.4.3 Hardware gating

And here we also have an arbitrary physical memory write, but an attacker can only reach it on machines where the hardware chip ID check passes. That's the hardware gate: the MmMapIoSpace on a non-existent BAR returns NULL or maps to nothing meaningful, and chipId won't match:

```
// BAR address obtained from PCI config space of a specific device
base = MmMapIoSpace(barAddress, BAR_SIZE, MmNonCached);
chipId = READ_REGISTER_ULONG(base + CHIP_ID_REGISTER);

if (chipId == 0x1234ABCD) {
    WRITE_REGISTER_ULONG(base + input->Offset, input->Value);
}
MmUnmapIoSpace(base, BAR_SIZE);
```

## 4 How driver deployment can be approached from the BYOVD perspective

In this section we are going to try to evaluate how much influence over proper driver initialization is possible by solely operating from userland (with administrative privileges), to reflect a typical BYOVD scenario.

So in this section we are not considering techniques involving:

- physical access,
- hypervisor level access allowing creation of virtualized hardware,
- non-standard/insecure system configurations, such as disabled driver signature enforcement,
- artificial alterations of execution flow using kernel mode debugger, or any other use of kernel mode debugger.

While the above techniques are all interesting and valuable for security research and testing, they are out of scope of this article.

### 4.1 Simple sc.exe deployment

This is the simplest, minimal step required to trigger driver load. We create a relevant service entry, then we trigger driver load by starting that service:

```
sc create SampleDrv type= kernel start= demand binPath= System32\drivers\SampleDrv.sys &&
↪  sc.exe start
```

Note, this deployment alone makes the driver execute its DriverEntry, but does not cover any PnP setup. In terms of named device creation, this setup approach is sufficient for drivers matching the pattern described in 3.1 Unconditional creation upon driver load.

Now, if we want to test if the driver created any named device objects, the easiest way not involving WinDBG usage is to:

1. Use NtObjectManager to list the \Devices directory and save that list.
2. Deploy and start the driver (sc create + sc start).
3. Use NtObjectManager again to list the \Devices directory and compare the result with the list obtained in step 1.
4. If a new device object was detected, try obtaining its SDDL.
5. Successful reading of SDDL proves it is possible to open a handle from userland, and only these devices are reported.

A Powershell implementation can be found here.

Let's see this script in action.

First, this is what we can expect to see for a driver that loads, but does not create any new devices:

```
PS C:\test> .\sc_deploy_detect.ps1 C:\runtime_service\IFM63X64.sys
Returning device list (193 elements).
[SC] CreateService SUCCESS

SERVICE_NAME: IFM63X64
        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              :
Returning device list (193 elements).
```

We can see that the driver successfully loaded, but the device list did not change after that.

Now, here is an example of a driver that does create a new device right away upon load:

```
PS C:\test> .\sc_deploy_detect.ps1 C:\runtime_service\KfeCo11x64.sys
Returning device list (193 elements).
[SC] CreateService SUCCESS

SERVICE_NAME: KfeCo11x64
        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              :
Returning device list (194 elements).
New device found for
 \Device\KfeCoDriver (symlink: ) O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)
```

This deployment and device detection approach is fast and practical for runtime discovery of drivers that create userland-accessible CDOs out of the box.

However, it is not sufficient for PnP device objects, which are far more common and thus constitute a much larger attack surface.

Also, keep in mind that many drivers deployed this way will fail to load due to missing dependencies. Those are usually satisfied when the deployment is conducted using the original installer and INF file.

## 4.2 Creating software-emulated devices with spoofed hardware ID

### 4.2.1 The idea

After digging a bit and learning more about the driver deployment process, I stumbled upon the test device functionality provided by devcon.exe, which provides the ability to create device nodes with arbitrary (spoofed) hardware IDs.

So it became clear to me these devices could be used to compensate for the missing hardware and get the AddDevice callback invoked.

Most device drivers come with INF files, which tie drivers to physical hardware by hardware IDs.

The easiest way to identify hardware ID (or IDs) matching a driver is by viewing its INF file. Hardware IDs are located in the Models sections, for example:

> [SampleDrv.NTamd64]
>
> %SampleDrv.DeviceDesc% = SampleDrv,**ACPI\SAMPLEDRV7853**

Here is a Python implementation extracting hardware IDs from INF files.

Once we have a matching hardware ID, instead of explicitly calling sc.exe, we deploy the driver as follows:

```
pnputil.exe /add-driver SampleDrv.inf /install
devcon.exe install SampleDrv.inf "ACPI\SAMPLEDRV7853"
```

First, we use pnputil to deploy the driver package into the Windows Driver Store.

Next, we use devcon to create a new software-emulated device node with an arbitrary hardware ID that matches one defined in the driver's INF file. This action triggers the PnP manager to detect the newly staged driver as the best match for the device.

As a result, the driver's AddDevice routine gets executed.

While pnputil.exe is present on every Windows system, devcon.exe is not, but it can be found in WDK.

The algorithm of detecting new named device objects in result of this deployment approach is the same, except for the deployment commands.

The devcon version of the deploy and detect Powershell script can be found here.

The output generated by this script looks the same as for the sc.exe version.

### 4.2.2 Initial test results

My preliminary experiments with this deployment approach resulted in **almost twice as many new device objects created** as compared to the simple sc.exe create, non-PnP deployment. Which clearly demonstrates that software-emulated device nodes with spoofed hardware IDs are a viable userland-only method of making (some) drivers reachable without their relevant hardware. I was able to find and confirm numerous driver vulnerabilities this way, including very good BYOVD candidates.

It is important to note that the algorithm used to detect new named device objects includes both CDOs as well as FDOs attached on top of the software-emulated PDO with an auto-generated name.

In the screenshot below, demonstrating a fragment of the aggregated result log, we can see one CDO and one PDO (with auto-generated name) created by the same driver, both with readable security descriptors:

```
snxppamd \Device\0000003d
snxpsamd \Device\0000003d
amdsfhkmdf \Device\00000032
amduart \Device\wdnisdrv  wdnisdrv
amduart \Device\0000003d
ibmcg \Device\0000003d
ibmcgft \Device\0000003d
ibmtp \Device\0000003d
ibmtpbs \Device\IBMTpBsDvc
O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1201bf;;;WD)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW) IBMTpBsDvcLnk
ibmtpbs \Device\0000003d
O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1201bf;;;WD)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
ibmtpbs \Device\KslD  KslD
ibmtpft \Device\00000032
iaisp64 \Device\wdnisdrv  wdnisdrv
iaisp64 \Device\0000003d
imx681 \Device\wdnisdrv  wdnisdrv
```

For visibility, the log file also includes newly discovered device objects whose SDDLs could not be obtained. Those make up the majority.

And here we can see 3 PDOs with auto-generated names, whose security descriptors are readable (the additional column is the GLOBAL?? symlink name, in this case automatically created with device interface registration):

```
ibtpci \Device\0000003d
ibtuart \Device\0000003e
mtkbtfilter \Device\0000003f
mtkbtfilterx \Device\00000040
BtFilter \Device\00000041
RtkBthLeVDsp \Device\00000042
RtkBtFilter2 \Device\00000043
RtkBtfilter \Device\00000045
GlPciSD \Device\00000047
bhtsddr \Device\00000048
RtsUer \Device\00000049
RtsPer \Device\0000004b
AmdAppCompat \Device\00000041 O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1201bf;;;WD)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
ROOT#SYSTEM#0003#{5010af27-9781-4cb3-810c-284fcfa3803a}
amdacpbus2 \Device\00000042
amdacpbus \Device\00000043
amdacpbtacx \Device\00000044
amd3dvcache \Device\0000003d O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1201bf;;;WD)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
ROOT#SYSTEM#0003#{24823a85-f283-4aab-8a5e-c2684121019c}
amdcamera \Device\0000003e
amdisp \Device\00000040 O:BAG:SYD:P(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1201bf;;;WD)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
ROOT#SYSTEM#0004#{db8e8dec-a69b-4ea7-ac85-e218fb057ffc}
sensorGC1029 \Device\00000041
sensorHM1092 \Device\00000042
sensorOV05C \Device\00000043
sensorOV08X \Device\00000044
sensorOV13B \Device\00000045
```

So, an obvious question arises: Why were the security descriptors of so many device objects created during this test not readable?

And secondly, what are we really doing when running ”devcon.exe install path_to.inf HWID”?

To answer these questions, let's have a closer look at the process of software-emulated device creation.

### 4.2.3 Creating software-emulated devices with SoftwareDevice and PnpManager

Keep in mind that creating a software-emulated device and telling Windows to use a specific driver to drive that device are two separate steps:

1. First, we create a software-emulated device with a spoofed hardware ID.

2. Then we invoke the driver installation/update process for that device using the original INF file (UpdateDriverForPlugAndPlayDevicesW), to eventually run the driver on the emulated device.

When it comes to the first step, the Windows kernel itself provides two similar mechanisms allowing creation of software-emulated devices with arbitrary hardware IDs:

1. The first method is provided by the PnPManager driver itself, and it can be performed by using Config Manager API/SetupAPI. This is how devcon.exe implements its software-emulated device creation.
2. The second one is provided by the SoftwareDevice driver, using Software Device API.

Both drivers are embedded in ntoskrnl.exe. In both cases we are creating PnP device nodes with arbitrary hardware IDs.

Let's have a closer look into this process.

#### 4.2.3.1 SetupAPI and PnpManager - process overview

Setting up a software-emulated device using SetupAPI requires the following sequence of API calls:

1. SetupDiCreateDeviceInfoList - create an empty device info set for our class.
2. SetupDiCreateDeviceInfoW - create device node.
3. SetupDiSetDeviceRegistryPropertyW - set the hardware ID on the devnode.
4. SetupDiCallClasInstaller - register the device with PnP.
5. UpdateDriverForPlugAndPlayDevicesW - force driver update for provided HWID, using provided INF file.

Calling SetupDiCallClassInstaller (step 4) triggers a sequence of operations on the kernel level, including a call to IoCreateDevice (PnpManager creating the new device object).

UpdateDriverForPlugAndPlayDevicesW requests the PnP manager to install a driver for that device. Before that happens, the device will show DOE_START_PENDING in its extension flags, when inspected with !devobj in WinDBG:

```
0: kd> !devobj \Device\0000003b
...
ExtensionFlags (0x00000810)  DOE_START_PENDING, DOE_DEFAULT_SD_PRESENT
...
```

Once the driver is bound to the device, the target driver's AddDevice will be invoked by PnpManager, passing a pointer to the PDO (owned by PnpManager) as the second argument. AddDevice is expected to create its FDO and attach it on top of the PDO using IoAttachDeviceToDeviceStack.

#### 4.2.3.2 SetupAPI and PnpManager - device node creation only

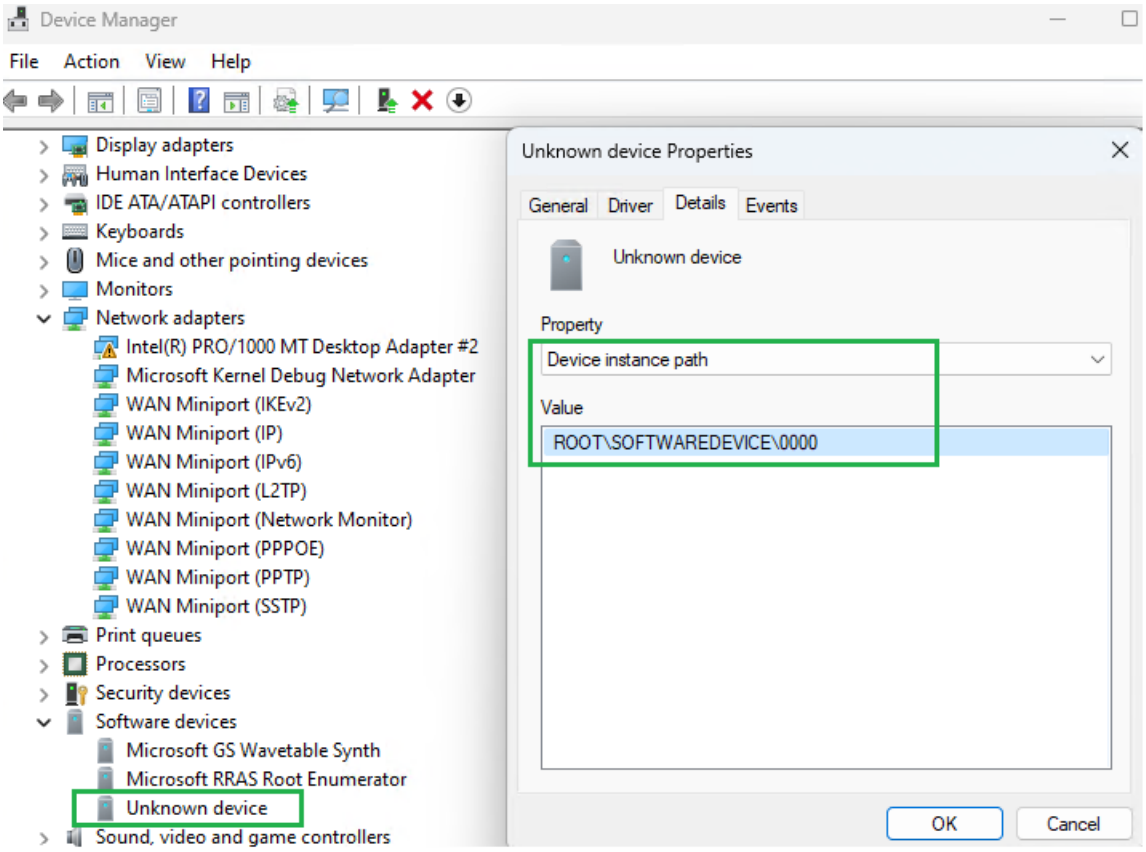
Let's use the following C implementation of steps 1-4, to only create a new device node with an arbitrary hardware ID, then inspect the device node in Device Manager and inspect its named device object in WinDBG.

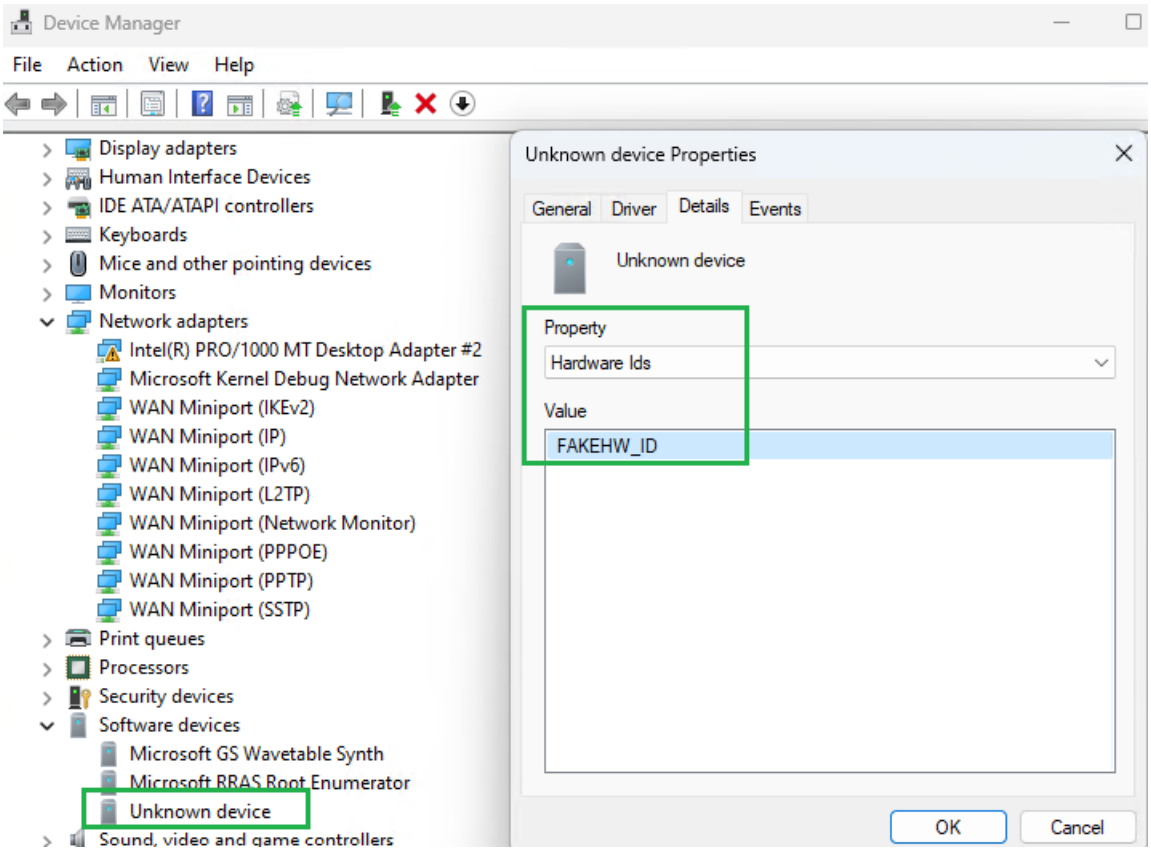
This way we can skip using an INF file entirely (for now) and examine the newly created named device object in its default state, without the PnP manager making any attempts to build a device stack on top of it.

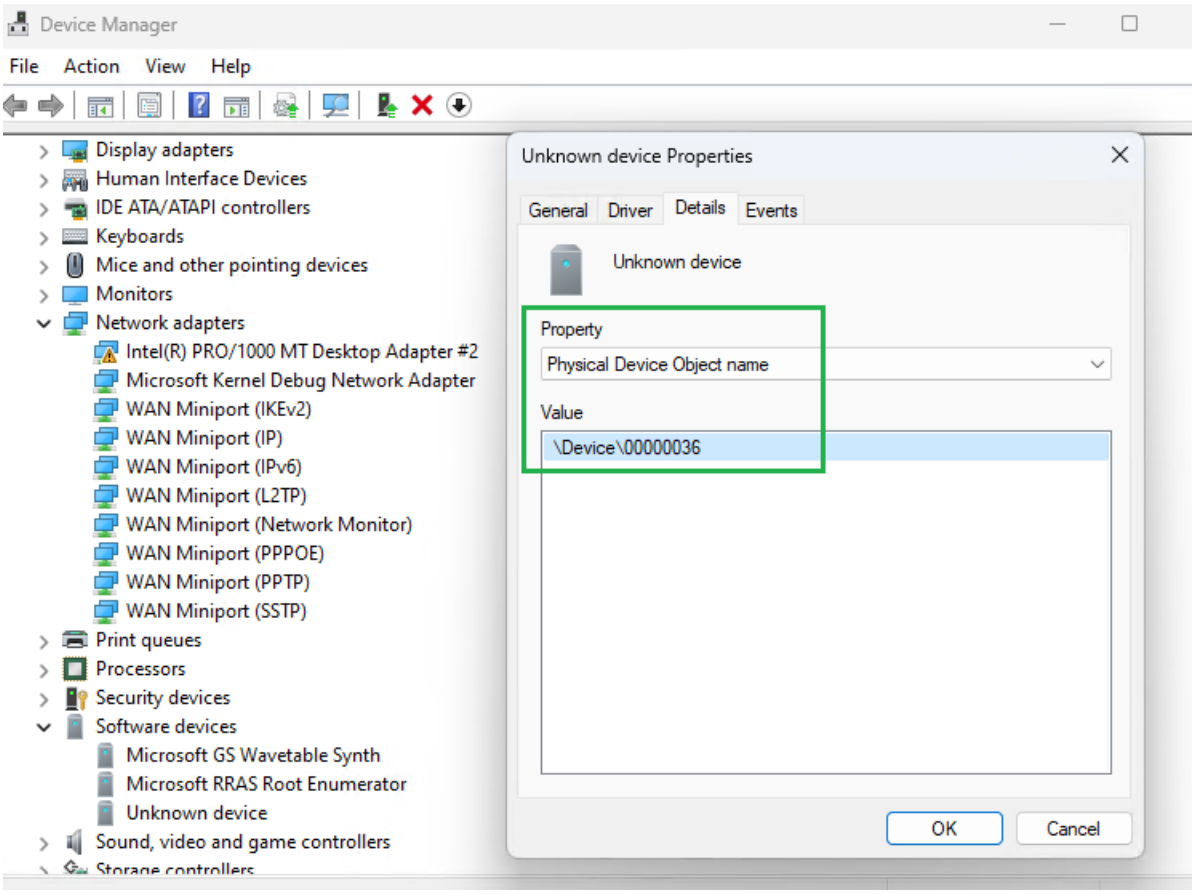
```
create_swdev_cm.exe FAKEHW_ID
Device node created successfully for hardware ID: FAKEHW_ID
```

We should be able to see the new device node (as "Unknown") in Software Devices in the Device Manager view.

We can manually select and view different device node properties, such as device instance path, hardware ID and even PDO name:


Device Manager
File Action View Help
Display adapters
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Intel(R) PRO/1000 MT Desktop Adapter #2
Microsoft Kernel Debug Network Adapter
WAN Miniport (IKEv2)
WAN Miniport (IP)
WAN Miniport (IPv6)
WAN Miniport (L2TP)
WAN Miniport (Network Monitor)
WAN Miniport (PPPOE)
WAN Miniport (PPTP)
WAN Miniport (SSTP)
Print queues
Processors
Security devices
Software devices
Microsoft GS Wavetable Synth
Microsoft RRAS Root Enumerator
Unknown device
Sound, video and game controllers
Unknown device Properties
General Driver Details Events
Unknown device
Property
Device instance path
Value
ROOT\SOFTWAREDEVICE\0000
OK
Cancel


Device Manager
File Action View Help
Display adapters
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Intel(R) PRO/1000 MT Desktop Adapter #2
Microsoft Kernel Debug Network Adapter
WAN Miniport (IKEv2)
WAN Miniport (IP)
WAN Miniport (IPv6)
WAN Miniport (L2TP)
WAN Miniport (Network Monitor)
WAN Miniport (PPPOE)
WAN Miniport (PPTP)
WAN Miniport (SSTP)
Print queues
Processors
Security devices
Software devices
Microsoft GS Wavetable Synth
Microsoft RRAS Root Enumerator
Unknown device
Sound, video and game controllers
Unknown device Properties
General Driver Details Events
Unknown device
Property
Hardware Ids
Value
FAKEHW_ID
OK
Cancel




Let's inspect the PDO name in WinDBG:

```
0: kd> !devobj \Device\00000036
Device object (ffff8207ddc03300) is for:
 00000036 \Driver\PnpManager DriverObject ffff8207d8aa3290
Current Irp 00000000 RefCount 0 Type 00000004 Flags 00001040
SecurityDescriptor ffffd408ceb1d260 DevExt ffff8207ddc03450 DevObjExt ffff8207ddc03458
 ↪  DevNode ffff8207deca0660
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000080)  FILE_AUTOGENERATED_DEVICE_NAME
Device queue is not busy.
```

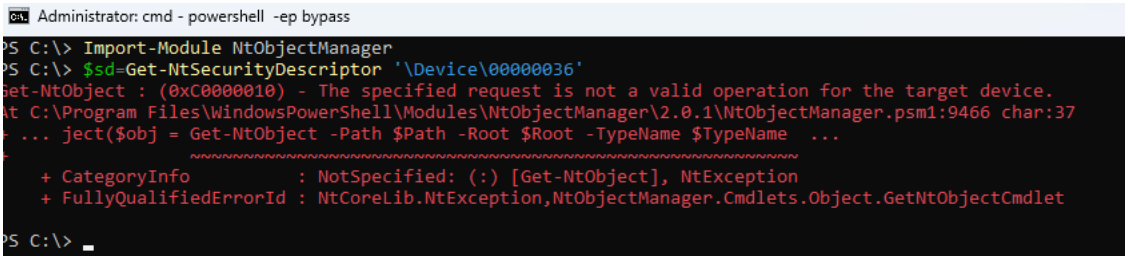
We can see that the driver owning the device object is `\Driver\PnpManager`, the device object has an auto-generated name and a default (permissive) security descriptor. Also note that the device object is NOT attached to any device stack here (there is no AttachedDevice etc.), so we can rule out a filter blocking access to it from above.

Examining the security descriptor in WinDBG confirms the default, permissive security descriptor:

```
0: kd> !sd ffffd408ceb1d260
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8814
            SE_DACL_PRESENT
            SE_SACL_PRESENT
            SE_SACL_AUTO_INHERITED
            SE_SELF_RELATIVE
```

```
->Owner    : S-1-5-32-544
->Group    : S-1-5-21-557163823-2925933541-2346282345-513
->Dacl     :
->Dacl     : ->AclRevision: 0x2
->Dacl     : ->Sbz1       : 0x0
->Dacl     : ->AclSize    : 0x5c
->Dacl     : ->AceCount   : 0x4
->Dacl     : ->Sbz2       : 0x0
->Dacl     : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[0]: ->AceFlags: 0x0
->Dacl     : ->Ace[0]: ->AceSize: 0x14
->Dacl     : ->Ace[0]: ->Mask : 0x001201bf
->Dacl     : ->Ace[0]: ->SID: S-1-1-0

->Dacl     : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[1]: ->AceFlags: 0x0
->Dacl     : ->Ace[1]: ->AceSize: 0x14
->Dacl     : ->Ace[1]: ->Mask : 0x001f01ff
->Dacl     : ->Ace[1]: ->SID: S-1-5-18

->Dacl     : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[2]: ->AceFlags: 0x0
->Dacl     : ->Ace[2]: ->AceSize: 0x18
->Dacl     : ->Ace[2]: ->Mask : 0x001f01ff
->Dacl     : ->Ace[2]: ->SID: S-1-5-32-544

->Dacl     : ->Ace[3]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl     : ->Ace[3]: ->AceFlags: 0x0
->Dacl     : ->Ace[3]: ->AceSize: 0x14
->Dacl     : ->Ace[3]: ->Mask : 0x001200a9
->Dacl     : ->Ace[3]: ->SID: S-1-5-12

->Sacl     :
->Sacl     : ->AclRevision: 0x2
->Sacl     : ->Sbz1       : 0x0
->Sacl     : ->AclSize    : 0x1c
->Sacl     : ->AceCount   : 0x1
->Sacl     : ->Sbz2       : 0x0
->Sacl     : ->Ace[0]: ->AceType: SYSTEM_MANDATORY_LABEL_ACE_TYPE
->Sacl     : ->Ace[0]: ->AceFlags: 0x0
->Sacl     : ->Ace[0]: ->AceSize: 0x14
->Sacl     : ->Ace[0]: ->Mask : 0x00000001
->Sacl     : ->Ace[0]: ->SID: S-1-16-4096
```

But when we try to display the security descriptor with NtObjectManager, we will encounter the following error message:



The requested operation is not valid for the target device?

In the introduction article, in section 3.6.7 Filters as access control, I demonstrated a similar situation, only with Access denied. In that case the upper driver in the stack was blocking IRP_MJ_CREATE, so the IRP never even reached the named PDO down the stack (the one used to open the handle).

Since here we only have one device object instead of a device stack, it must be PnpManager itself blocking those requests.

Let's have a look at its dispatch routine table:

```
0: kd> !drvobj PnpManager 2
Driver object (ffff8207d8aa3290) is for:
 \Driver\PnpManager
...
Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[03] IRP_MJ_READ                        fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             fffff8053ff516b0
 ↪  nt!IopInvalidDeviceRequest
```

```
[14] IRP_MJ_QUERY_SECURITY              fffff8053ff516b0
↪  nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                fffff8053ff516b0
↪  nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       fffff8054015fa10        nt!IopPowerDispatch
[17] IRP_MJ_SYSTEM_CONTROL              fffff80540561f30
↪  nt!IopSystemControlDispatch
[18] IRP_MJ_DEVICE_CHANGE               fffff8053ff516b0
↪  nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 fffff8053ff516b0
↪  nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   fffff8053ff516b0
↪  nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         fffff805402c6940        nt!IopPnPDispatch
```

Aha! The dispatch routine values for most MajorFunction codes are set to nt!IopInvalidDeviceRequest.

Which means that the driver simply does not support them.

Without IRP_MJ_CREATE we cannot open a handle, even to read the security descriptor. In the case described in the introduction article (section 3.6.7 Filters as access control), the upper driver called IofCompleteRequest, with Irp->IoStatus.Status = STATUS_ACCESS_DENIED.

In this case, IRP_MJ_CREATE returns Irp->IoStatus.Status = STATUS_INVALID_DEVICE_REQUEST.

The reason this is happening is because the driver owning the PDO is simply not intended to be responsible for handling IRP_MJ_CREATE requests. In typical device stacks, the handling of IRP_MJ_CREATE should take place in the FDO and end there (with IofCompleteRequest), with IRP_MJ_CREATE never being passed down the stack.

**Which leads us to an important conclusion - if we are trying to open a handle to a device stack, at least one device in that stack must successfully handle our IRP_MJ_CREATE.**

We cannot open a handle to a device stack if neither of the following accepts IRP_MJ_CREATE:

1. Upper FiDO (if present).
2. FDO.
3. Lower FiDO (if present).
4. The PDO (if IRP ever reaches here). PDO is always the base of a device stack, so it's always present.

This is why we cannot open a handle to a bare (non-stack-attached) named PDO created by PnpManager.

A large portion of the failed deployment attempts observed in the aggegated log — where no security descriptors could be obtained for the newly created devices — was caused by the lack of IRP_MJ_CREATE support in the PnP Manager, combined with the absence of an upper-level driver in the device stack to handle that IRP.

Which is what happened when:

- UpdateDriverForPlugAndPlayDevicesW succeeded, but the target driver did not support IRP_MJ_CREATE either,
- UpdateDriverForPlugAndPlayDevicesW failed for any reason (.cat file missing, other dependancy referred in the INF file missing, or even the driver not loading).

#### 4.2.3.3 SetupAPI and PnpManager - complete and successful deployment

Now, for contrast, let's see how a full (steps 1-5) and successful deployment looks like, using the Powershell script.

We will use AwinicSmartKAmps.sys (I2C smart amplifier controller) driver as an example.

First, let's have a look at its INF file.

On line 32 we can find the hardware ID - ACPI\AWDZ8399.

It is also worth noting that on line 50 the "AddService" directive defines the driver's service name as AwinicChip.

This is how the driver object will be named, even though the .sys file itself is named AwinicSmartKAmps.sys (as visible on line 58):

AwinicSmartKAmps.inf

```
28  %COMPANY% = Awinic,NTamd64
29
30  ; Decorated model section take precedence over undecorated ones on XP and later.
31  [Awinic.NTamd64]
32  %AwinicChip.DeviceDesc% = AwinicChip,ACPI\AWDZ8399
33
34  [AwinicChip.NT]
35  CopyFiles = CopyConfigDir
36  CopyFiles = CopyDriverFile
37
38  [AwinicChip.NT.HW]
39  AddReg = AddConfigReg
40
41  [CopyDriverFile]
42  AwinicSmartKAmps.sys
43
44  [AddConfigReg]
45  ; Set to 1 to connect the first interrupt resource found, 0 to leave disconnected
46  HKR,Settings,"AutoPowerOn",0x00010001,0
47
48  ; Service installation
49  [AwinicChip.NT.Services]
50  AddService = AwinicChip,%SPSVCINST_ASSOCSERVICE%,InstallService
51
52  ; driver install sections
53  [InstallService]
54  DisplayName = %AwinicChip.SVCDESC%
55  ServiceType = 1                  ; SERVICE_KERNEL_DRIVER
56  StartType = 3                    ; SERVICE_DEMAND_START
57  ErrorControl = 1                 ; SERVICE_ERROR_NORMAL
58  ServiceBinary = %12%\AwinicSmartKAmps.sys
```

We run the deployment script:

```
Administrator: cmd - powershell  -ep bypass
PS C:\runtime_service> .\deploy_detect.ps1 AwinicSmartKAmps.inf "ACPI\AWDZ8399"
Returning device list (163 elements).
Microsoft PnP Utility

Adding driver package:  AwinicSmartKAmps.inf
Driver package added successfully.
Published Name:         oem7.inf

Total driver packages:  1
Added driver packages:  1
Device node created. Install is complete when drivers are installed...
Updating drivers for ACPI\AWDZ8399 from C:\runtime_service\AwinicSmartKAmps.inf.
Drivers installed successfully.
Returning device list (165 elements).
New device found for AwinicSmartKAmps.inf ACPI\AWDZ8399
 \Device\0000002d (symlink: ) O:BAG:S-1-5-21-557163823-2925933541-2346282345-513D:(A;;0x1201bf;;;WD)(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
New device found for AwinicSmartKAmps.inf ACPI\AWDZ8399
 \Device\0000002e (symlink: ) O:BAG:SYD:(A;;0x1201bf;;;WD)(A;;FA;;;SY)(A;;FA;;;BA)(A;;0x1200a9;;;RC)S:AI(ML;;NW;;;LW)
PS C:\runtime_service>
```

Interesting - two new named device objects were detected, and they are both userland-accessible (SDDLs could be retrieved)!

Let's inspect the driver object in WinDBG:

```
!drvobj AwinicChip 7
```

```
Driver object (ffffe18f33ff3e10) is for:
 \Driver\AwinicChip

Driver Extension List: (id , addr)
(fffff805394622e0 ffffe18f2ec1a950)
Device Object list:
ffffe18f32ce6de0

DriverEntry:   fffff80562ba0630 AwinicSmartKAmps
DriverStartIo: 00000000
DriverUnload:  fffff80562ba07c0 AwinicSmartKAmps
AddDevice:     fffff80539462090

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80539427ac0        +0xfffff80539427ac0
...
Device Object stacks:

!devstack ffffe18f32ce6de0 :
  !DevObj           !DrvObj            !DevExt           ObjectName
  ffffe18f34e51e00  \Driver\ksthunk    ffffe18f34e51f50  0000002e
> ffffe18f32ce6de0  \Driver\AwinicChip ffffe18f35cde310
  ffffe18f35b0db90  \Driver\PnpManager ffffe18f35b0dce0  0000002d
!DevNode ffffe18f32f32b20 :
  DeviceInst is "ROOT\MEDIA\0000"
  ServiceName is "AwinicChip"
```

We can see that our driver created one device object (ffffe18f32ce6de0), which was then attached into a device stack on top of `\Device\0000002d` (software-emulated PDO created by PnpManager), and additionally to that, the PnP manager also attached another (also named) device object on top of it - `\Device\0000002e` (owned by `\Driver\ksthunk`).

If we look at the beginning of the INF file, we'll notice this:

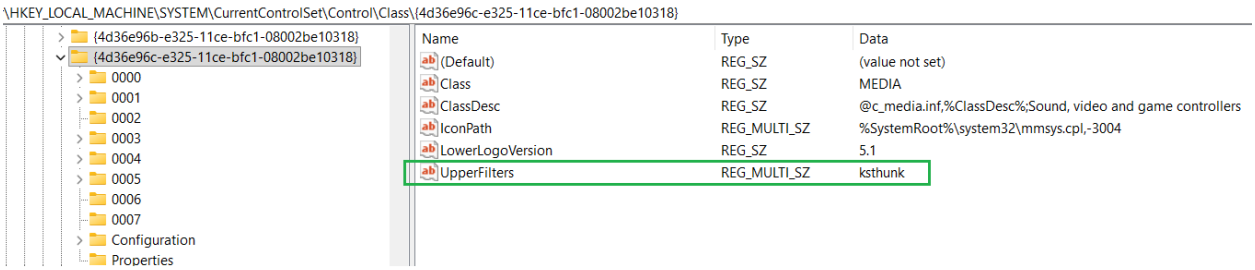
AwinicSmartKAmps.inf

```
1  [Version]
2  Signature = "$WINDOWS NT$"
3  Class = MEDIA
4  ClassGuid = {4d36e96c-e325-11ce-bfc1-08002be10318}
5  Provider = %COMPANY%
6  DriverVer = 03/12/2025,1.0.0.19
7  CatalogFile = AwinicSmartKAmps.cat
8  PnpLockDown = 1
```

The driver class is defined as Multimedia, using the well-known {4d36e96c-e325-11ce-bfc1-08002be10318} GUID.

ksthunk (Kernel Streaming) is registered as a class upper filter for the MEDIA device setup class.

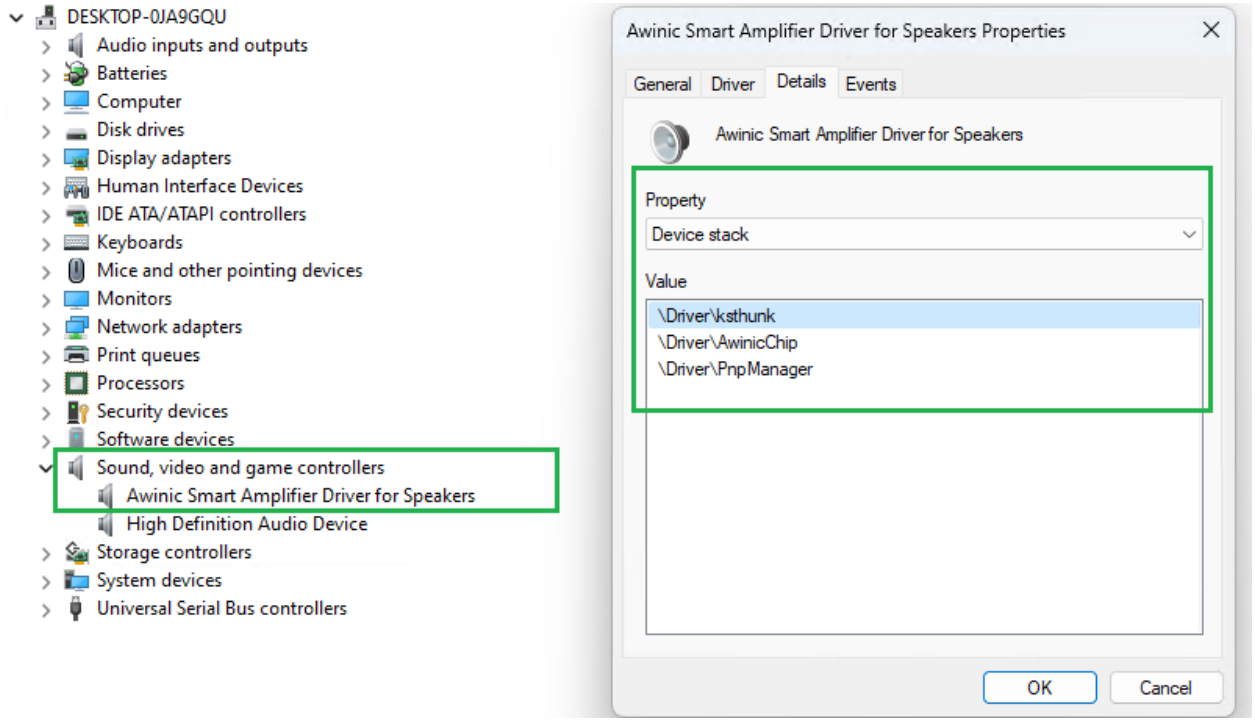
This can be confirmed by inspecting the UpperFilters REG_MULTI_SZ registry entry at `HKLM\SYSTEM\CurrentControlSet\`

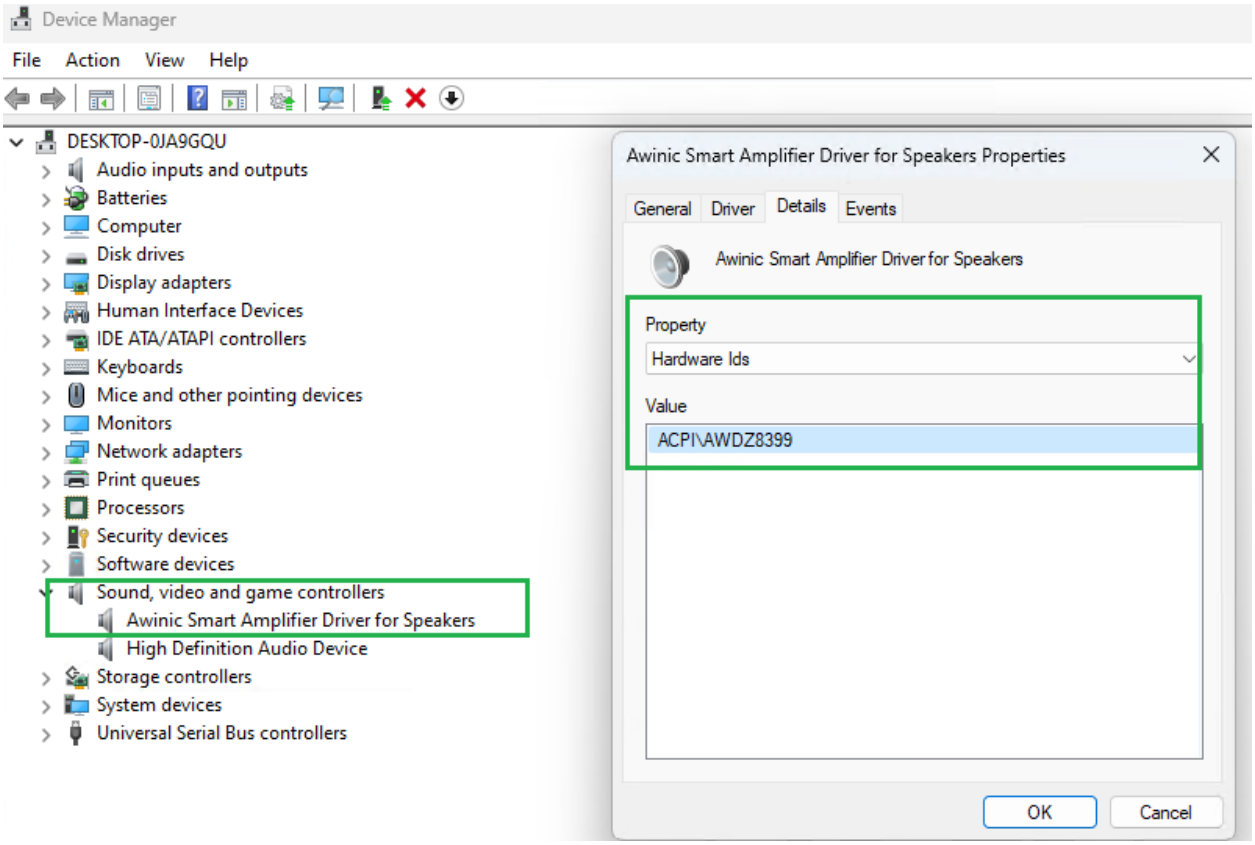



The PnP manager automatically attaches class upper filter device objects to every device in the class it is set up for. That's why `\Device\0000002e` owned by `\Driver\ksthunk` is present in the device stack on top of our driver's unnamed FDO.

We will revisit the UpperFilters mechanism later in this article.

Another consequence of the driver being installed as a Media device is how its device node is visible in the Device Manager GUI tool. It appears in the "Sound, video and game controllers" subtree:






Before we move on, while we already have the driver loaded, let's set up a couple of breakpoints:

```
0: kd> bp fffff80539462090 ".echo AddDevice called;g"
0: kd> bp fffff80539427ac0 ".echo IRP_MJ_CREATE called;g"
0: kd> g
```

We already know these addresses from the output of `!drvobj AwinicChip 7`.

Now, IRP_MJ_CREATE should hit whenever we attempt to open a handle to any device in the stack:

```
PS C:\runtime_service> Get-NtSecurityDescriptor '\Device\0000002d'

Owner                  DACL ACE Count SACL ACE Count Integrity Level
-----                  -------------- -------------- ---------------
BUILTIN\Administrators 4              1              Low


PS C:\runtime_service> Get-NtSecurityDescriptor '\Device\0000002e'

Owner                  DACL ACE Count SACL ACE Count Integrity Level
-----                  -------------- -------------- ---------------
BUILTIN\Administrators 4              1              Low
```

In the debugger output we should see:

```
IRP_MJ_CREATE called
IRP_MJ_CREATE called
```

And if we manually invoke the creation of another device node using the same hardware ID (by simply running `devcon.exe install AwinicSmartKAmps.inf "ACPI\AWDZ8399"` again), we should see the AddDevice breakpoint hitting as well:

```
AddDevice called
```

Keep in mind that AddDevice being invoked only means that we have managed to trick the PnP manager to call it. It does not neccessarily mean that AddDevice will successfully create a new device object and attach it to the the device stack - it may still fail internally due to additional unmet conditions.

From the practical perspective, the easiest way to confirm the success of this type of deployment, is reading the security descriptor of the software-emulated device. If that works, it means that:

- driver installation (UpdateDriverForPlugAndPlayDevicesW call) was successful,
- in the newly created device stack there is a driver that accepts IRP_MJ_CREATE.

Here is the full version the setup program (steps 1-5). Requires INF file.

#### 4.2.3.4 Software Device API

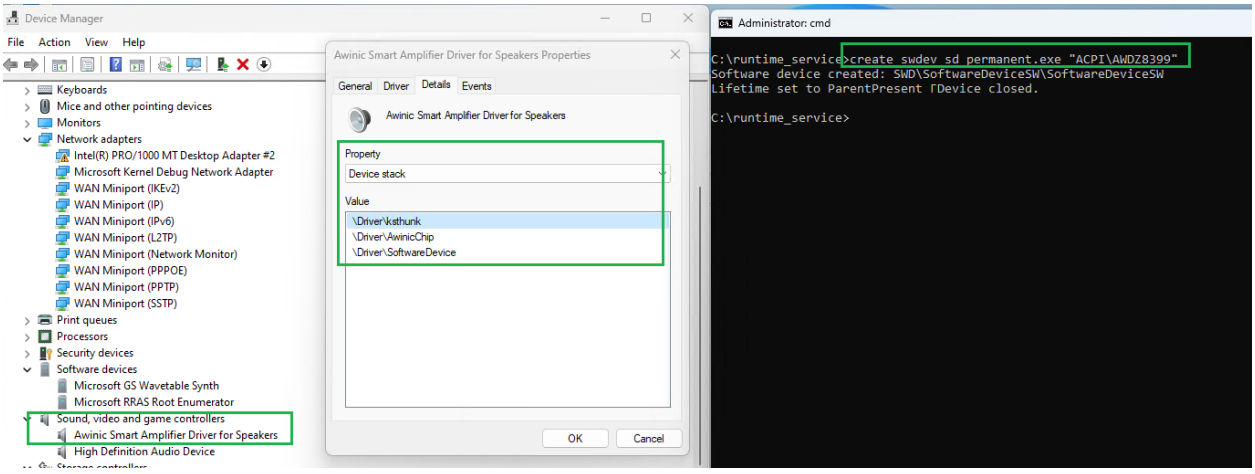
An alternative to the SetupAPI device creation approach is Software Device API.

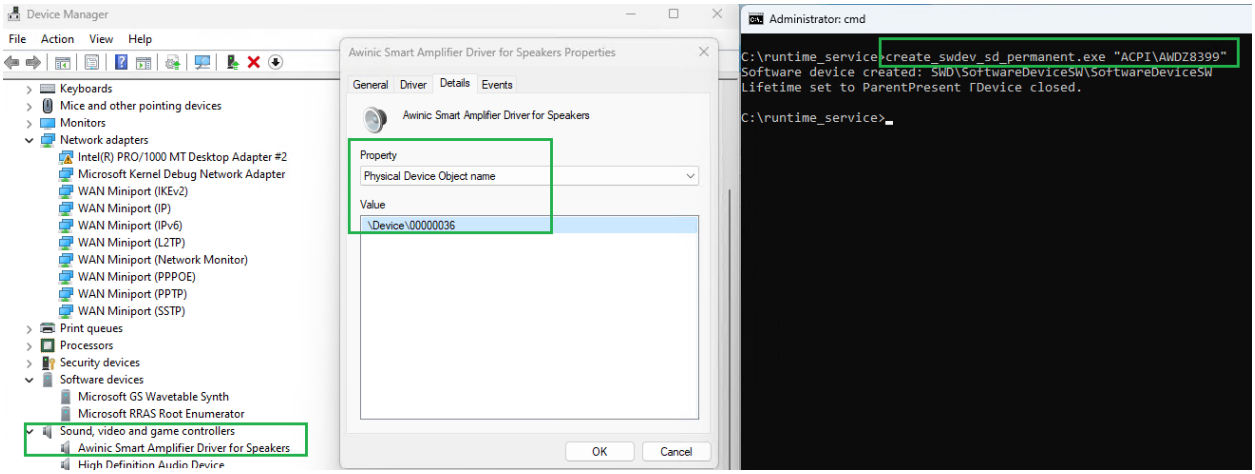
Creation of a software-emulated device with arbitrary hardware ID is simpler than with SetupAPI, as it boils down to just calling SwDeviceCreate.

A sample C implementation can be found here. It can be easily extended with UpdateDriverForPlugAndPlayDevicesW (requires INF file).

By default the device object gets removed when we close the `HSWDEVICE hSwDevice` handle (the handle populated by SwDeviceCreate). To prevent that, before closing the handle, the program calls `hr = SwDeviceSetLifetime(hSwDevice, SWDeviceLifetimeParentPresent);`.

Device objects created this way are owned by `\Driver\SoftwareDevice` (as a reminder, the ones created with SetupAPI are owned by `\Driver\PnpManager`).






A quick inspection of the device object is WinDBG:

```
0: kd> !devobj \Device\00000036
0: kd> Device object (ffffaa0a87a6de00) is for:
 00000036 \Driver\SoftwareDevice DriverObject ffffaa0a8273ce00
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00001040
SecurityDescriptor ffffce8086380820 DevExt ffffaa0a87a6df50 DevObjExt ffffaa0a87a6df60
 ↪  DevNode ffffaa0a86a1a8e0
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000180)  FILE_AUTOGENERATED_DEVICE_NAME, FILE_DEVICE_SECURE_OPEN
AttachedDevice (Upper) ffffaa0a88033de0 \Driver\AwinicChip
Device queue is not busy.
```

And just like `\Driver\PnpManager`, the `\Driver\SoftwareDevice` driver does not support IRP_MJ_CREATE:

```
0: kd> !drvobj SoftwareDevice 2

Driver object (ffffaa0a8273ce00) is for:
 \Driver\SoftwareDevice

DriverEntry:   fffff8074ad32f40 nt!PiSwPdoDriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     fffff8074a9e4400 nt!ArbPreprocessEntry

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8074a5516b0
 ↪  nt!IopInvalidDeviceRequest
```

### 4.3 Jumping device stacks

Making a PnP-compatible driver reachable with a software-emulated device is an example of **building and accessing a custom device stack**.

That **opens the way to even interact via IRP with drivers that were never intended for userland interaction** (e.g. not checking IRP's RequestorMode prior to further processing), because when deployed the original way they reside in device stacks where an upper driver prevents userland access (by denying or not supporting IRP_MJ_CREATE), just like in the example demonstrated in section 3.6.7 Filters as access control in the introduction article.

So, if we find a vulnerability in a driver that cannot be interacted with from userland in its original con-

figuration, that driver is not useful for Local Privilege Escalation. But for breaking the admin to kernel boundary it might. We just need to deploy it in a way that allows exploitability, by placing it in a device stack where the original upper filter is not present. We could call this malicious driver misconfiguration/deployment.

This approach is not only useful for testing, but also for making vulnerabilities reachable, and thus potentially exploitable during BYOVD attacks.

Let's see another variant of this.

### 4.3.1 Filter restacking

Once we understand that in order to access a device stack, one of its drivers must accept our IRP_MJ_CREATE, it becomes clear why the deployment scenario covered in section 4.3 could not succeed for filter drivers.

A typical filter driver is a pass-through IRP forwarder, which means it does not reject IRP_MJ_CREATE, but it does not accept it either. It does have the relevant dispatch routine, and that routine usually just forwards IRPs down the stack.

So, if we try to access a filter driver by putting it on top of a software-emulated PDO, we won't be able to open a handle.

That is because neither of the two drivers supports IRP_MJ_CREATE.

The filter driver forwards it down to the PDO, and the PDO rejects it via nt!IopInvalidDeviceRequest.

This is why for filter drivers we need a device stack with a typical FDO below (or just any driver that will accept IRP_MJ_CREATE from userland).

One way to build such a device stack is by abusing the Disk Drive class (GUID {4d36e967-e325-11ce-bfc1-08002be10318}).

The algorithm is as follows:

1. Deploy the filter driver with sc.exe create (no INF files involved).
2. Append the filer's service name to the UpperFilters registry key for the device Disk Drive device class in

`HKLM:\SYSTEM\CurrentControlSet\Control\Class\{4d36e967-e325-11ce-bfc1-08002be10318}`.

3. Mount a new hard drive from VHD, creating a new device node and PnP building a new device stack for it.
4. Filter becomes accessible once we open a handle on `\\.\PhysicalDrive0`.

A Powershell implementation can be found here.

This deployment scenario works for most filter drivers, regardless of the device class the filter is intended for. **This way I successfully ran (and in some cases exploited) filter drivers for various device clasess, such as disk, bluetooth, mouse, keyboard and audio.**

For instance, here we are loading gaming mouse filter driver on top of a disk stack:

```
C:\runtime_service>powershell -ep bypass
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\runtime_service> .\filter_setup_script.ps1 C:\runtime_service\GMLXDFltr.sys
03/31/2026 03:32:53 - [*] Setting up GMLXDFltr as a PnP upper disk filter...
03/31/2026 03:32:53 - [1] Checking driver binary location...
03/31/2026 03:32:53 - GMLXDFltr.sys not found in C:\Windows\System32\drivers\. Copying from C:\runtime_service\GMLXDFltr.sys
03/31/2026 03:32:53 - [2] Creating proper service entry (boot-start, kernel driver)...
[SC] CreateService SUCCESS
03/31/2026 03:32:53 - OK: Service entry created.
03/31/2026 03:32:53 - [3] Starting GMLXDFltr

SERVICE_NAME: GMLXDFltr
        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              :
03/31/2026 03:32:53 - [5] Registering as disk class upper filter...
03/31/2026 03:32:53 - OK: GMLXDFltr appended to UpperFilters.
03/31/2026 03:33:00 - Mounting empty.vhd using diskpart...

Microsoft DiskPart version 10.0.22621.1

Copyright (C) Microsoft Corporation.
On computer: DESKTOP-0JA9GQU

DiskPart successfully selected the virtual disk file.

  100 percent completed

DiskPart successfully attached the virtual disk file.
03/31/2026 03:33:02 - Restore: removing GMLXDFltr from UpperFilters (restore original value partmgr).
03/31/2026 03:33:02 - Done. To dispose of the stack, just detach the disk.
PS C:\runtime_service>
```

If we inspect the driver object in WinDBG, we will see the entire device stack, including GMLXDFltr with its unnamed FiDO on top of it:

```
0: kd> !drvobj GMLXDFltr
Driver object (ffffd20a9c4cde20) is for:
 \Driver\GMLXDFltr

Driver Extension List: (id , addr)

Device Object list:
ffffd20a9b8a3690  ffffd20a9753b910
0: kd> !devobj ffffd20a9b8a3690
Device object (ffffd20a9b8a3690) is for:
  \Driver\GMLXDFltr DriverObject ffffd20a9c4cde20
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00000000
SecurityDescriptor ffffe48d055db5a0 DevExt ffffd20a9b8a37e0 DevObjExt ffffd20a9b8a3868
ExtensionFlags (0000000000)
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
AttachedTo (Lower) ffffd20a9ba1f690 \Driver\partmgr
Device queue is not busy.
0: kd> !devstack ffffd20a9b8a3690
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffd20a9b8a3690  \Driver\GMLXDFltr  ffffd20a9b8a37e0
  ffffd20a9ba1f690  \Driver\partmgr    ffffd20a9ba1f7e0
  ffffd20a9f187060  \Driver\disk       ffffd20a9f1871b0  DR1
  ffffd20aa30e8050  \Driver\vhdmp      ffffd20aa30e81a0  00000038
!DevNode ffffd20a9b8998c0 :
  DeviceInst is "SCSI\Disk&Ven_Msft&Prod_Virtual_Disk\2&1f4adffe&0&000001"
```

```
  ServiceName is "disk"
```

Let's set a breakpoint on its IRP_MJ_CREATE dispatch routine. First, obtain the address:

```
0: kd> !drvobj GMLXDFltr 2
Driver object (ffffd20a9c4cde20) is for:
 \Driver\GMLXDFltr

DriverEntry:   fffff80085a52244 GMLXDFltr
DriverStartIo: 00000000
DriverUnload:  fffff80085a51b8c GMLXDFltr
AddDevice:     fffff80085a51ab4 GMLXDFltr

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80085a51a08        GMLXDFltr+0x1a08
```

Then set up a breakpoint, resume execution:

```
0: kd> bp GMLXDFltr+0x1a08 ".echo GMLXDFltr IRP_MJ_CREATE called;g"
0: kd> g
```

On the VM, resolve the device path and trigger IRP_MJ_CREATE by attempting to read the SDDL:

```
PS C:\> Get-NtSymbolicLinkTarget '\GLOBAL??\PhysicalDrive1'
\Device\Harddisk1\DR1
PS C:\> Get-NtSecurityDescriptor '\Device\Harddisk1\DR1'

Owner                  DACL ACE Count SACL ACE Count Integrity Level
-----                  -------------- -------------- ---------------
BUILTIN\Administrators 5              NONE           NONE
```

Debugger output:

```
0: kd> g
GMLXDFltr IRP_MJ_CREATE called
GMLXDFltr IRP_MJ_CREATE called
```

So, it is working! **We are accessing a gaming mouse filter driver via a handle opened on `\Device\Harddisk1\DR1`!**

On a side note, surprisingly, the breakpoint is activated twice, not once. Why?

To find out, let's start with confirming the identity of the userland process. For that we can attach the following script to run when the breakpoint hits, so it prints the image name of the current userland caller:

```
bp GMLXDFltr+0x1a08 ".echo GMLXDFltr IRP_MJ_CREATE ran;.scriptrun
 ↪  C:\\Users\\ewilded\\curr_image_name.js;g"
breakpoint 0 redefined
0: kd> g
```

And now, when `Get-NtSecurityDescriptor '\Device\Harddisk1\DR1'` is invoked again, we get:

```
GMLXDFltr IRP_MJ_CREATE ran
JavaScript script successfully loaded from
 ↪  'C:\Users\ewilded\WinDBG_scripts\curr_image_name.js'
Current Process Image Name: powershell.exe
GMLXDFltr IRP_MJ_CREATE ran
JavaScript script successfully loaded from
 ↪  'C:\Users\ewilded\WinDBG_scripts\curr_image_name.js'
Current Process Image Name: powershell.exe
```

So in both cases the caller is powershell.

Let's also print the DesiredAccess via IRP IO_STACK_LOCATION, to get more details:

```
1: kd> g
GMLXDFltr IRP_MJ_CREATE ran
Current Process Image Name: powershell.exe
GMLXDFltr+0x1a08:
fffff800`85a51a08 48895c2408      mov     qword ptr [rsp+8],rbx
2: kd> dt nt!_IRP @rdx Tail.Overlay.CurrentStackLocation
   +0x078 Tail                             :
      +0x000 Overlay                          :
         +0x040 CurrentStackLocation             : 0xffffd20a`9d4806f8
 ↪  _IO_STACK_LOCATION
2: kd> dt nt!_IO_STACK_LOCATION 0xffffd20a`9d4806f8 Parameters.Create.SecurityContext
   +0x008 Parameters                       :
      +0x000 Create                           :
         +0x000 SecurityContext                  : 0xfffff504`0ab565d0
 ↪  _IO_SECURITY_CONTEXT
2: kd> dt nt!_IO_SECURITY_CONTEXT 0xfffff504`0ab565d0 DesiredAccess
   +0x010 DesiredAccess : 0x20000
```

So during the first call of GMLXDFltr+0x1a08, the DesiredAccess is 0x20000. Let's resume execution and inspect the same property on the second breakpoint hit:

```
2: kd> g
GMLXDFltr IRP_MJ_CREATE ran
Current Process Image Name: powershell.exe
GMLXDFltr+0x1a08:
fffff800`85a51a08 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> dt nt!_IRP @rdx Tail.Overlay.CurrentStackLocation
   +0x078 Tail                             :
      +0x000 Overlay                          :
         +0x040 CurrentStackLocation             : 0xffffd20a`9dc4a808
 ↪  _IO_STACK_LOCATION
0: kd> dt nt!_IO_STACK_LOCATION 0xffffd20a`9dc4a808 Parameters.Create.SecurityContext
   +0x008 Parameters                       :
      +0x000 Create                           :
         +0x000 SecurityContext                  : (null)
```

So, in the first call, DesiredAccess is 0x20000. But in the second one the SecurityContext is NULL. Which strongly suggests that in this second call we are not dealing with IRP_MJ_CREATE at all.

Let's examine the call stacks when the breakpoint hits.

First:

```
GMLXDFltr+0x1a08:
fffff800`85a51a08 48895c2408      mov     qword ptr [rsp+8],rbx
3: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff504`0ab56448 fffff800`68eebef5     GMLXDFltr+0x1a08
01 fffff504`0ab56450 fffff800`692f753e     nt!IofCallDriver+0x55
02 fffff504`0ab56490 fffff800`692f2874     nt!IopParseDevice+0x8be
03 fffff504`0ab56660 fffff800`692f1222     nt!ObpLookupObjectName+0x1104
04 fffff504`0ab567f0 fffff800`692eecb1     nt!ObOpenObjectByNameEx+0x1f2
05 fffff504`0ab56920 fffff800`6934f438     nt!IopCreateFile+0x431
06 fffff504`0ab569e0 fffff800`6902bbe5     nt!NtOpenFile+0x58
```

```
07 fffff504`0ab56a70 00007ffe`e51af9d4     nt!KiSystemServiceCopyEnd+0x25
08 00000010`96acd9d8 00007ffe`7209aa32     0x00007ffe`e51af9d4
09 00000010`96acd9e0 00000000`00020000     0x00007ffe`7209aa32
0a 00000010`96acd9e8 00000010`96acda10     0x20000
0b 00000010`96acd9f0 00000010`96acdae0     0x00000010`96acda10
0c 00000010`96acd9f8 00000142`0a263498     0x00000010`96acdae0
0d 00000010`96acda00 00000000`00000005     0x00000142`0a263498
0e 00000010`96acda08 00000000`00000000     0x5
```

We can see nt!NtOpenFile in the Call Site column, which is expected, and confirms this is the handle-opening call.

Let's resume the execution and examine the second hit:

```
3: kd> g
GMLXDFltr IRP_MJ_CREATE ran
Current Process Image Name: powershell.exe
GMLXDFltr+0x1a08:
fffff800`85a51a08 48895c2408      mov     qword ptr [rsp+8],rbx
3: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff504`0ab56828 fffff800`68eebef5     GMLXDFltr+0x1a08
01 fffff504`0ab56830 fffff800`692f9bdc     nt!IofCallDriver+0x55
02 fffff504`0ab56870 fffff800`692f352e     nt!IopDeleteFile+0x13c
03 fffff504`0ab568f0 fffff800`68eec627     nt!ObpRemoveObjectRoutine+0x7e
04 fffff504`0ab56950 fffff800`693437d4     nt!ObfDereferenceObjectWithTag+0xc7
05 fffff504`0ab56990 fffff800`693403a9     nt!ObpCloseHandle+0x2a4
06 fffff504`0ab56ab0 fffff800`6902bbe5     nt!NtClose+0x39
07 fffff504`0ab56ae0 00007ffe`e51af554     nt!KiSystemServiceCopyEnd+0x25
08 00000010`96acd958 00007ffe`71e3db77     0x00007ffe`e51af554
09 00000010`96acd960 00007ffe`71e06bf0     0x00007ffe`71e3db77
0a 00000010`96acd968 00000000`00000d28     0x00007ffe`71e06bf0
0b 00000010`96acd970 00000142`1f783590     0xd28
0c 00000010`96acd978 00007ffe`d10a4bce     0x00000142`1f783590
0d 00000010`96acd980 00004f02`50afaf79     0x00007ffe`d10a4bce
0e 00000010`96acd988 00007ffe`d17b6370     0x00004f02`50afaf79
0f 00000010`96acd990 00000010`96acdb80     0x00007ffe`d17b6370
10 00000010`96acd998 00007ffe`71e06bf0     0x00000010`96acdb80
11 00000010`96acd9a0 00007ffe`71e06bf0     0x00007ffe`71e06bf0
12 00000010`96acd9a8 00000010`96acd960     0x00007ffe`71e06bf0
13 00000010`96acd9b0 00007ffe`71e3db77     0x00000010`96acd960
14 00000010`96acd9b8 00000010`96acda00     0x00007ffe`71e3db77
15 00000010`96acd9c0 00007ffe`71e06bf0     0x00000010`96acda00
16 00000010`96acd9c8 00000010`96acdae0     0x00007ffe`71e06bf0
17 00000010`96acd9d0 00000142`0a263798     0x00000010`96acdae0
18 00000010`96acd9d8 00000142`1f783590     0x00000142`0a263798
19 00000010`96acd9e0 00000142`0a2590f0     0x00000142`1f783590
1a 00000010`96acd9e8 00000000`00000000     0x00000142`0a2590f0
```

So, the second call is **not** IRP_MJ_CREATE. It is IRP_MJ_CLOSE (powershell closing the handle), and it comes from NtClose (visible in the Call Site colum).

The breakpoint hits twice, because GMLXDFltr uses the same dispatch routine for both MajorCodes.

A glance at the dispatch table confirms this. Both IRP_MJ_CREATE and IRP_MJ_CLOSE are set to GMLXDFltr+0x1a08:

```
0: kd> !drvobj GMLXDFltr 2
Driver object (ffffd20a9c4cde20) is for:
 \Driver\GMLXDFltr

DriverEntry:   fffff80085a52244 GMLXDFltr
DriverStartIo: 00000000
DriverUnload:  fffff80085a51b8c GMLXDFltr
AddDevice:     fffff80085a51ab4 GMLXDFltr

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80085a51a08        GMLXDFltr+0x1a08
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff80085a50388        GMLXDFltr+0x388
[02] IRP_MJ_CLOSE                       fffff80085a51a08        GMLXDFltr+0x1a08
```

### 4.3.2 Per-device and per-class filters

Global per-class device filters can be set up in the LowerFilters and UpperFilters entries in the relevant device class registry locations:

`HKLM:\SYSTEM\CurrentControlSet\Control\Class\{GUID}\`.

For example:

`HKLM:\SYSTEM\CurrentControlSet\Control\Class\{4d36e967-e325-11ce-bfc1-08002be10318}\UpperFilters`

for Disk Drives.

The list of well-known GUIDs representing device classes can be found here. As we can see, Storage Volumes, Disk Drives and Storage Disks constitute separate, although similar device classes.

UpperFilters and LowerFilters can also be set up for device nodes more selectively - per instance ID instead of device class.

These are located in `HKLM\SYSTEM\CurrentControlSet\Enum\<instance_ID>`. For example:

- `HKLM\SYSTEM\CurrentControlSet\Enum\SWD\HWID\0001` (SWD for devices created via the Software Device API),
- `HKLM\SYSTEM\CurrentControlSet\Enum\ROOT\HWID2\0000` (ROOT for root-enumerated devices created through SetupAPI).

This provides more flexibility in the ways filters can be run. We could use a software-emulated device with an FDO that supports IRP_MJ_CREATE only to run the filter on top of it, without creating a Disk Drive device node by mounting a disk.

## 4.4 Forced driver replacement

The real reason for using software-emulated devices as described in the previous sections is to get the driver properly initialized by tricking the PnP manager to call its AddDevice callback with a pointer to a PDO.

An alternative to that approach is using **one of the devices already existing in the system**.

So, can we simply force our driver to be installed and loaded for an existing piece of hardware, replacing the original one?

One reason against this approach is system stability.

If we force an incorrect driver on a piece of hardware, it is reasonable to assume that hardware will stop functioning correctly. To alleviate that risk we could select a device that is not critical and whose disruption is unlikely to be even noticed.

Another potential cause of critical disruption (resulting in system crash) is if the newly loaded driver makes a reference to the PDO's device extension at some specific offset that exists in the PDO the driver was designed to work with. So it is a matter of trial and error to find a suitable device node for this purpose.

Assuming that the target device is similar enough to the device the vulnerable driver was developed, or the driver is simple enough, it could work (depending on the driver).

So, what is stopping us from trying this?

During normal driver installation process, forcing an arbitrary driver on an arbitrary device node is problematic because of the hardware ID missmatch. PnP matches the hardware ID (reported by the bus driver owning the PDO) against all INF files in the local Windows driver repository.

PnP will not recognize a driver as correct if the device node's hardware ID does not match the hardware ID in the corresponding INF file.

The INF file is what ties a driver to a specific hardware ID.

### 4.4.1 The problem with INF files

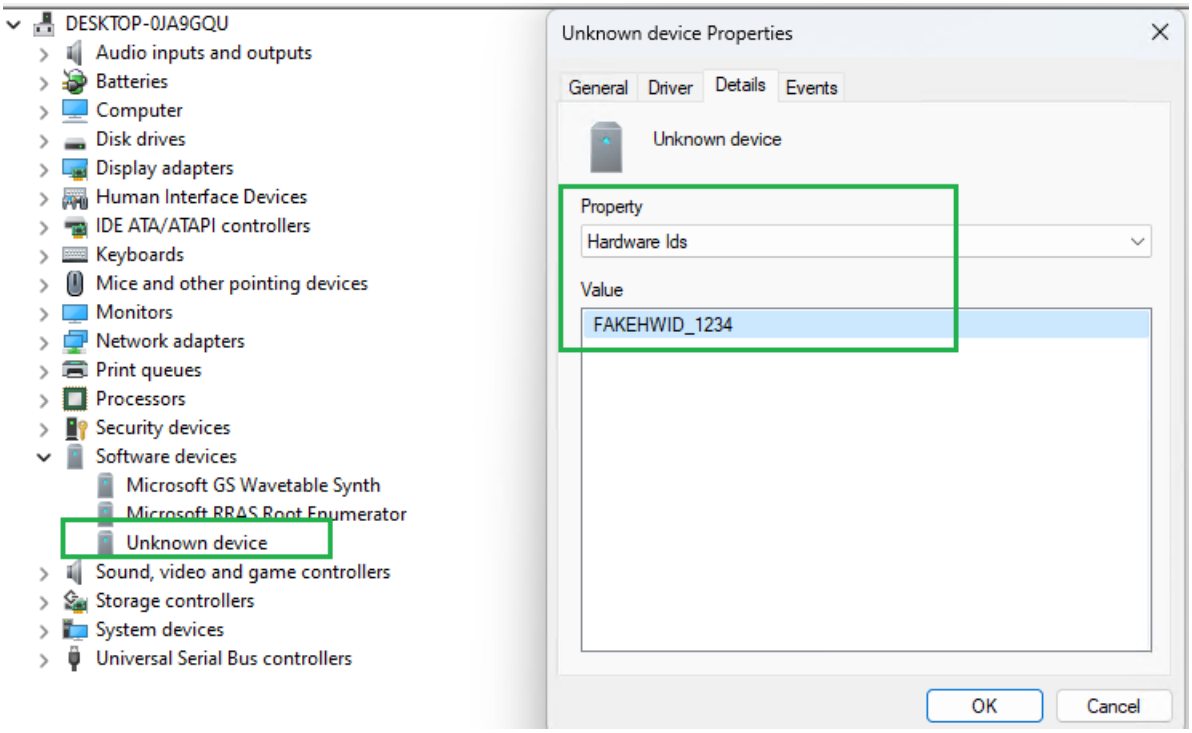
So, can we use a custom INF file, which simply ties FAKEHWID_1234 to AwinicSmartKAmps.sys instead?

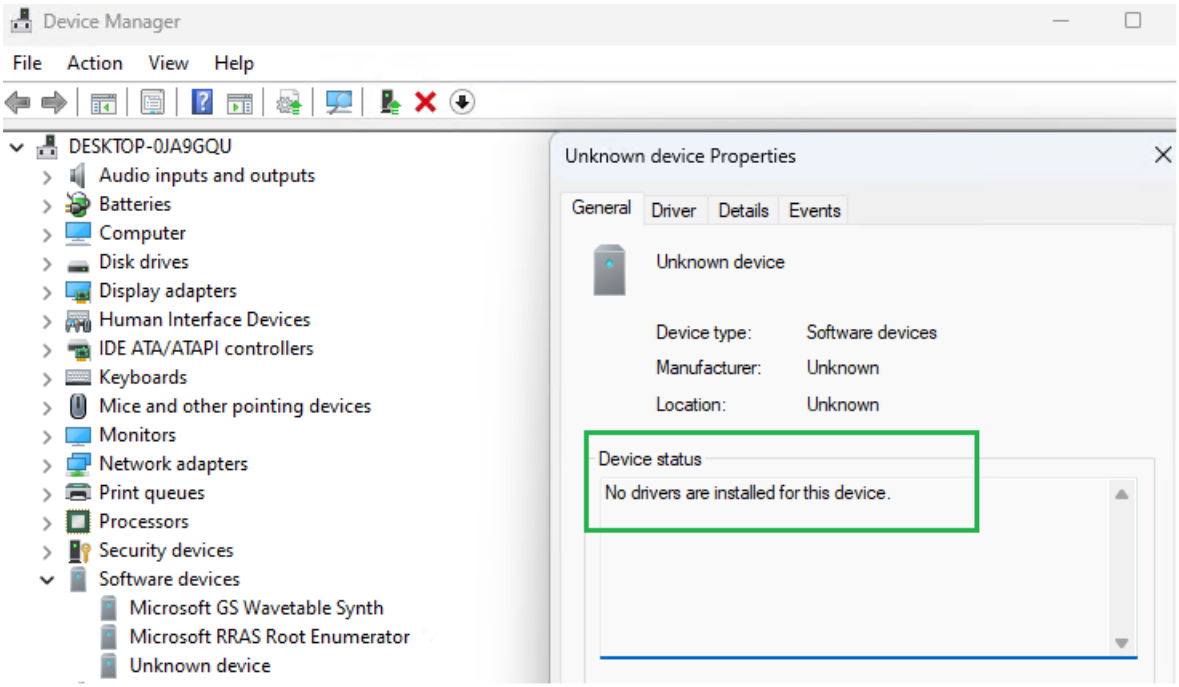
Let's see.

First, let's create a new device node with an arbitrary hardware ID, using create_swdev_cm.c:

```
create_swdev_cm.exe FAKEHWID_1234
Device node created successfully for hardware ID: FAKEHWID_1234
```

We can inspect it in Device Manager. It appears in the Software Devices group because of the class GUID hardcoded in create_swdev_cm.c, but that can be changed if needed. We can clearly see the arbitrary hardware ID and and that at this stage there is no driver installed for the device:






The PnP manager has not found any matching drivers, because FAKEHWID_1234 is not matched by any INF file from the driver repository. This is expected.

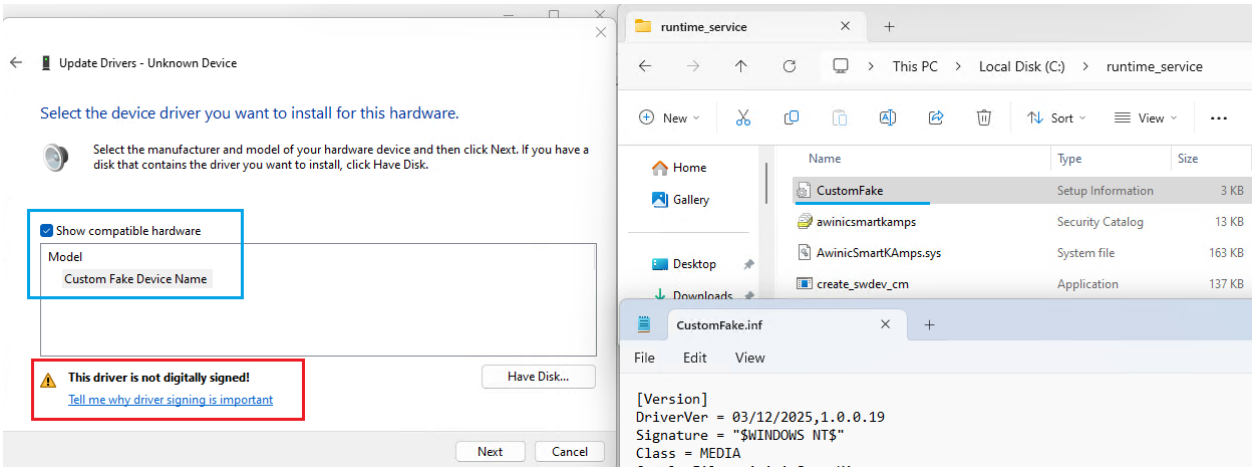
Now, let's see what happens when we try to update the driver using a minimum custom INF file, tying FAKEHWID_1234 to AwinicSmartKAmps.sys:

CustomFake.inf

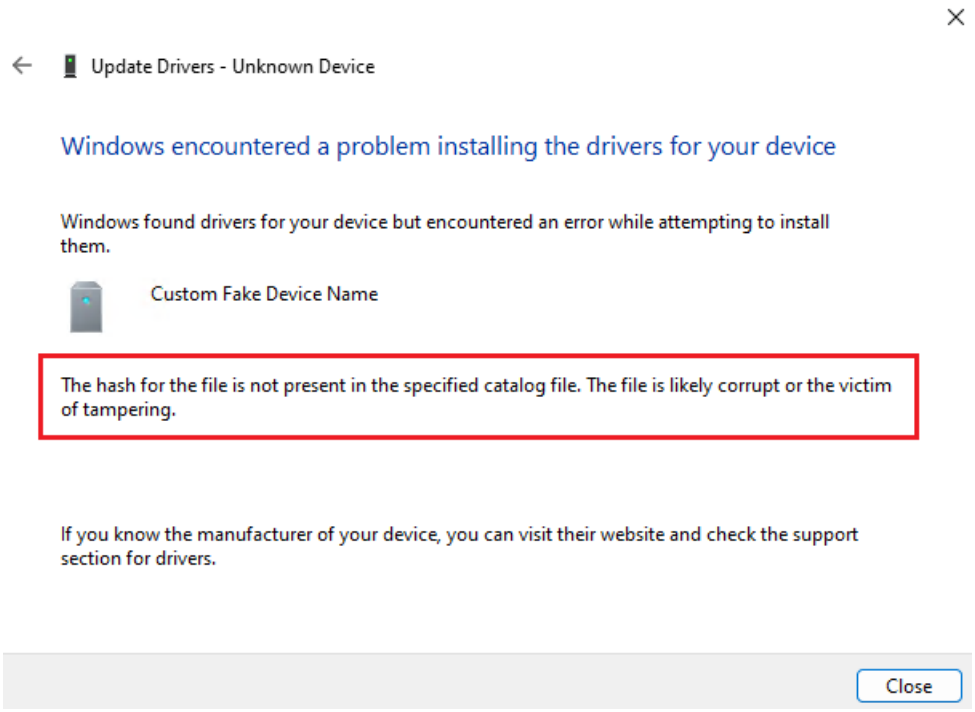
File Edit View

```
[Version]
DriverVer = 03/12/2025,1.0.0.19
Signature = "$WINDOWS NT$"
Class = MEDIA
CatalogFile = AwinicSmartKAmps.cat
ClassGuid = {4d36e96c-e325-11ce-bfc1-08002be10318}
Provider = %COMPANY%
[SourceDisksNames]
1 = %DISK_ID_1%,,,""
[SourceDisksFiles]
AwinicSmartKAmps.sys = 1,,
[DestinationDirs]
DefaultDestDir = 12
[Manufacturer]
%COMPANY% = FakeCompany,NTamd64
[FakeCompany.NTamd64]
%CustomFake.DeviceDesc% = CustomFake,FAKEHWID_1234
[CustomFake.NT]
CopyFiles = CopyDriverFile
[CustomFake.NT.HW]
AddReg = AddConfigReg
[CopyDriverFile]
AwinicSmartKAmps.sys
[AddConfigReg]
HKR,Settings,"AutoPowerOn",0x00010001,0
[CustomFake.NT.Services]
AddService = CustomFake,%SPSVCINST_ASSOCSERVICE%,InstallService
[InstallService]
ErrorControl = 1
DisplayName = %CustomFake.SVCDESC%
StartType = 3
ServiceType = 1
ServiceBinary = %12%\AwinicSmartKAmps.sys
[Strings]
DISK_ID_1 = "Custom Fake Device Install Disk"
CustomFake.DeviceDesc = "Custom Fake Device Name"
CustomFake.SVCDESC = "Custom Fake Device Service Name"
DRIVER_VERSION = 1.0.0.19
DRIVER_VERSION_DATE = 03/12/2025
SPSVCINST_ASSOCSERVICE = 0x00000002
COMPANY = "FakeCompany"
```

If we right-click on the "Unknown device" and invoke "Update driver", we can manually point the directory with the INF file after choosing: "Browse my computer" -> "Let me pick from a list of available drivers on my computer" -> "Have disk", we'll see a warning "This driver is not digitally signed!":

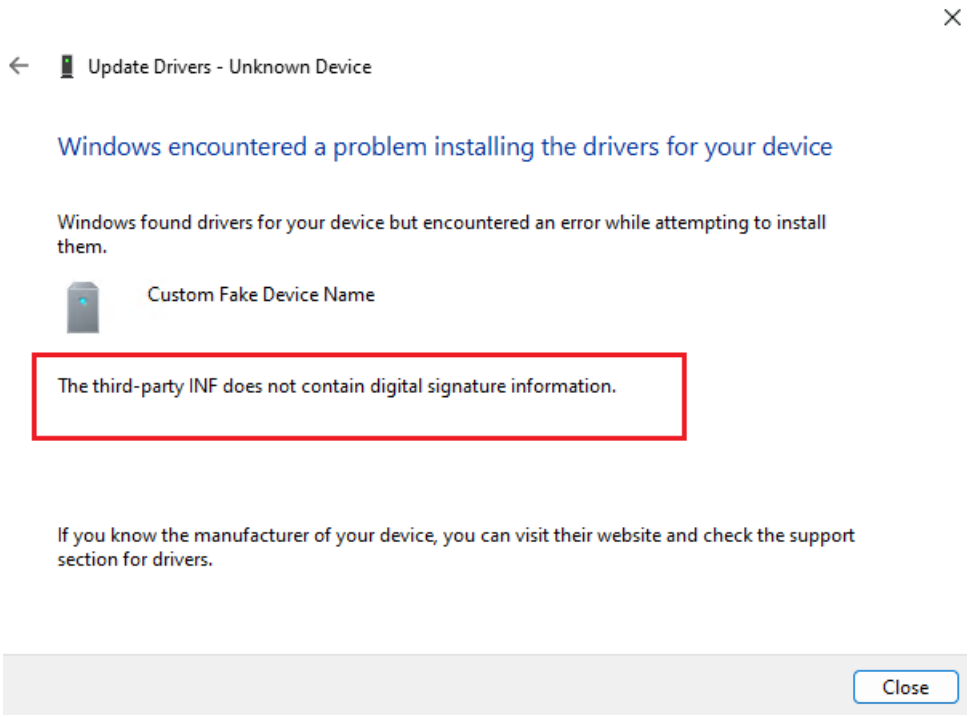



If we ignore the warning and click "Next", we'll see this:




This is because files in a driver package, including the INF file, are protected by digital signatures defined in the catalog file (.cat).

If we remove the `CatalogFile = AwinicSmartKAmps.cat` line entirely and try again, we will end up with the same outcome and a slightly different error message:




This logic is implemented behind UpdateDriverForPlugAndPlayDevicesW - if we try to do this programmatically instead of using GUI, we will reach the same outcome, with UpdateDriverForPlugAndPlayDevicesW returning relevant error codes.

### 4.4.2 Bypassing the INF file mechanism

If we dig deeper into the PnP architecture, we will discover that what **really** ties a driver to a device node is dedicated registry structures.

Depending on whether we are creating a new device, or forcing our driver on an existing one, we can either directly create new registry structures or modify the existing ones, to make them reflect the state normally attained with a successful installation, effectively skipping the entire INF mechanism.

It boils down to:

1. Deploying the driver with sc.exe create, the standard way.
2. Choosing the target device (for this test we will create one using SetupAPI).
3. Creating the relevant registry structures in the following locations, tying the driver's service name to the device:
   - `SYSTEM\\CurrentControlSet\\Enum\\<DEVICE_INSTANCE_ID>`,
   - `SYSTEM\\CurrentControlSet\\Control\\Class\\<GUID>\\<INSTANCE_INDEX>`.
4. Restarting the device (triggers PnP to load the driver).

The device instance ID is the full path that uniquely identifies a device in the system and is the actual identifier of a device node.

It makes sense when we think about it. The hardware ID only identifies the make and model, and using it as the device node identifier would prevent the OS from supporting multiple devices with the same hardware ID connected to the system at the same time.

Device instance ID has the following form: `<Enumerator>\<DeviceID>\<InstanceID>`, for example: `PCI\VEN_8086&DEV_A370&SUBSYS_02348086&REV_10\3&11583659&0&F5`.

Instance ID is just the last segment — the part that disambiguates multiple devices with the same enumerator and device ID. In the example above, that's `3&11583659&0&F5`. It identifies which physical/logical device this is among devices sharing the same enumerator and device ID.

In our case the device instance ID will be `ROOT\SOFTWAREDEVICE\0000`, whereas:

- ROOT is the enumerator name,
- SOFTWAREDEVICE is the device ID (the value comes from uppercase conversion of the string passed as the second argument to SetupDiCreateDeviceInfoW),
- 0000 is the instance ID.

The instance index, on the other hand, is a zero-based, four-digit sequential number that identifies a specific device instance within a device setup class. Therefore, if no other device instances exist for the given device ID at the time of creation, the instance index will be 0000.

Let's deploy the same driver as earlier using this approach.

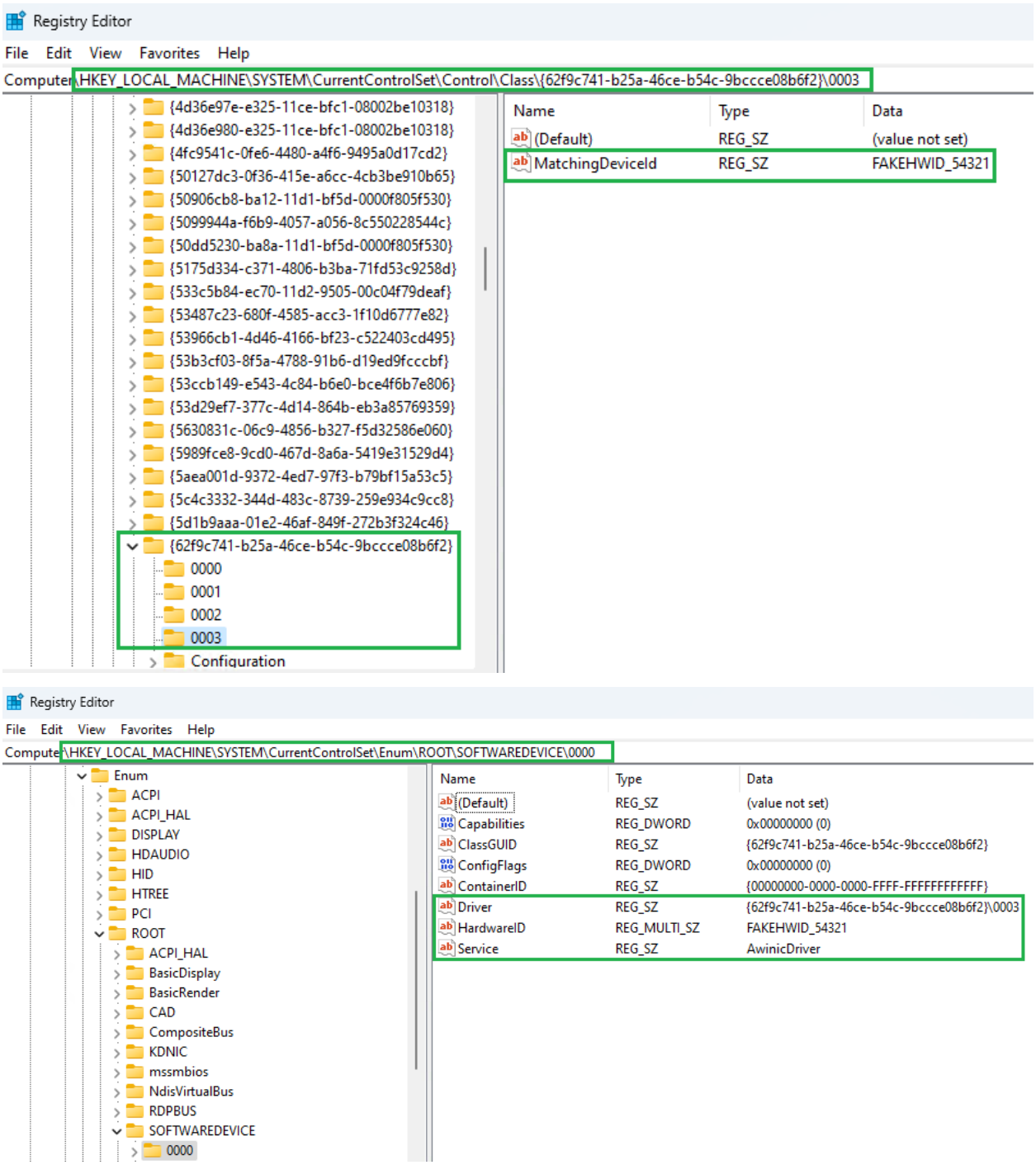
First, we create the service the usual `sc.exe create` way:

```
sc.exe create AwinicDriver binPath= "C:\\runtime_service\\AwinicSmartKAmps.sys" type=
 ↪  kernel start= demand
[SC] CreateService SUCCESS

sc.exe start AwinicDriver

SERVICE_NAME: AwinicDriver
        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
```

Then, we run the installer (full source code of create_sd_bind_driver.c can be found here):

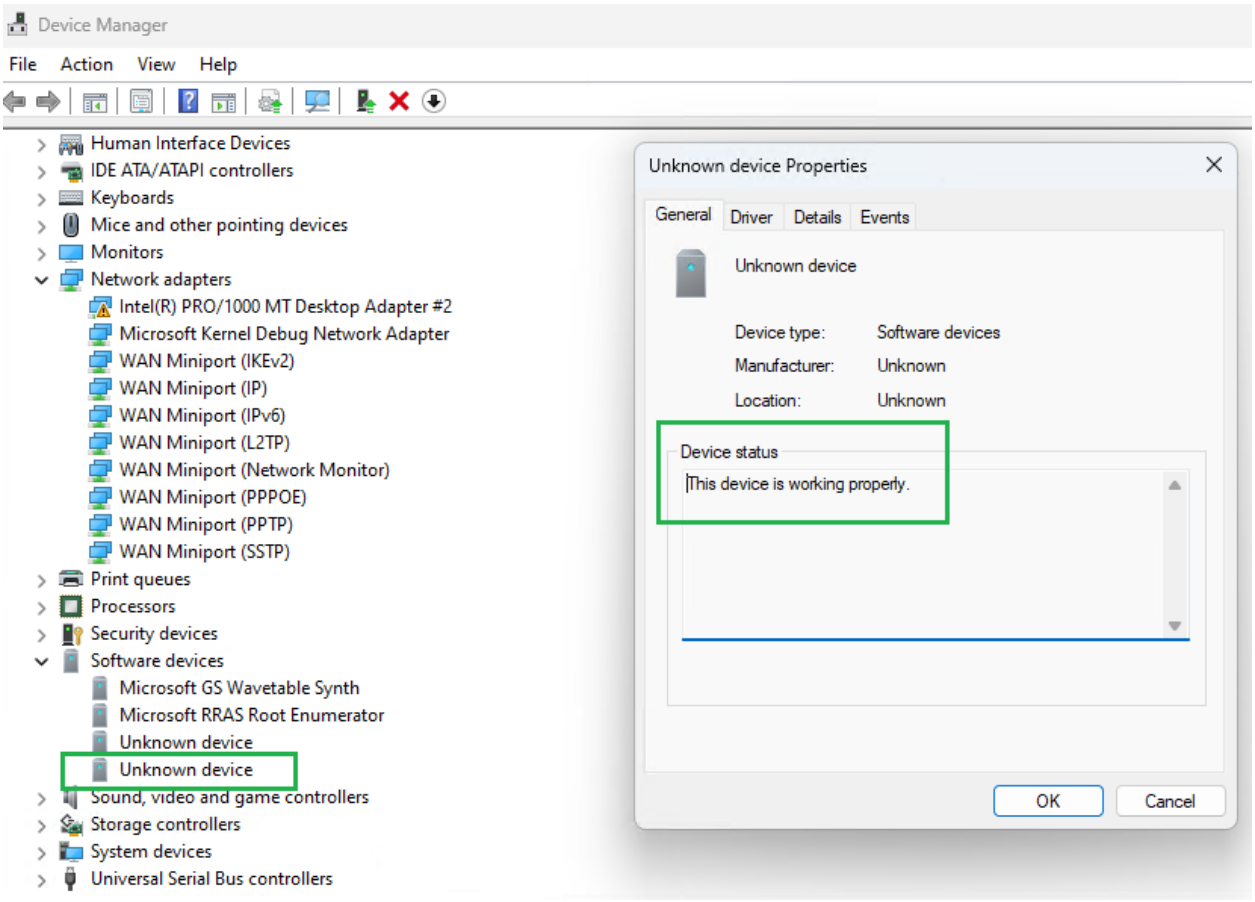
```
create_sd_bind_driver.exe FAKEHWID_54321 AwinicDriver
 ↪  {62f9c741-b25a-46ce-b54c-9bccce08b6f2}
Device node created: ROOT\SOFTWAREDEVICE\0000 (DevInst=2)
Service bound: AwinicDriver
Class key created:
 ↪  SYSTEM\CurrentControlSet\Control\Class\{62f9c741-b25a-46ce-b54c-9bccce08b6f2}\0003
Driver value set: {62f9c741-b25a-46ce-b54c-9bccce08b6f2}\0003
ConfigFlags set: 0
Restarting device (disable/enable cycle)...
Device enabled. AddDevice should have been called.
```

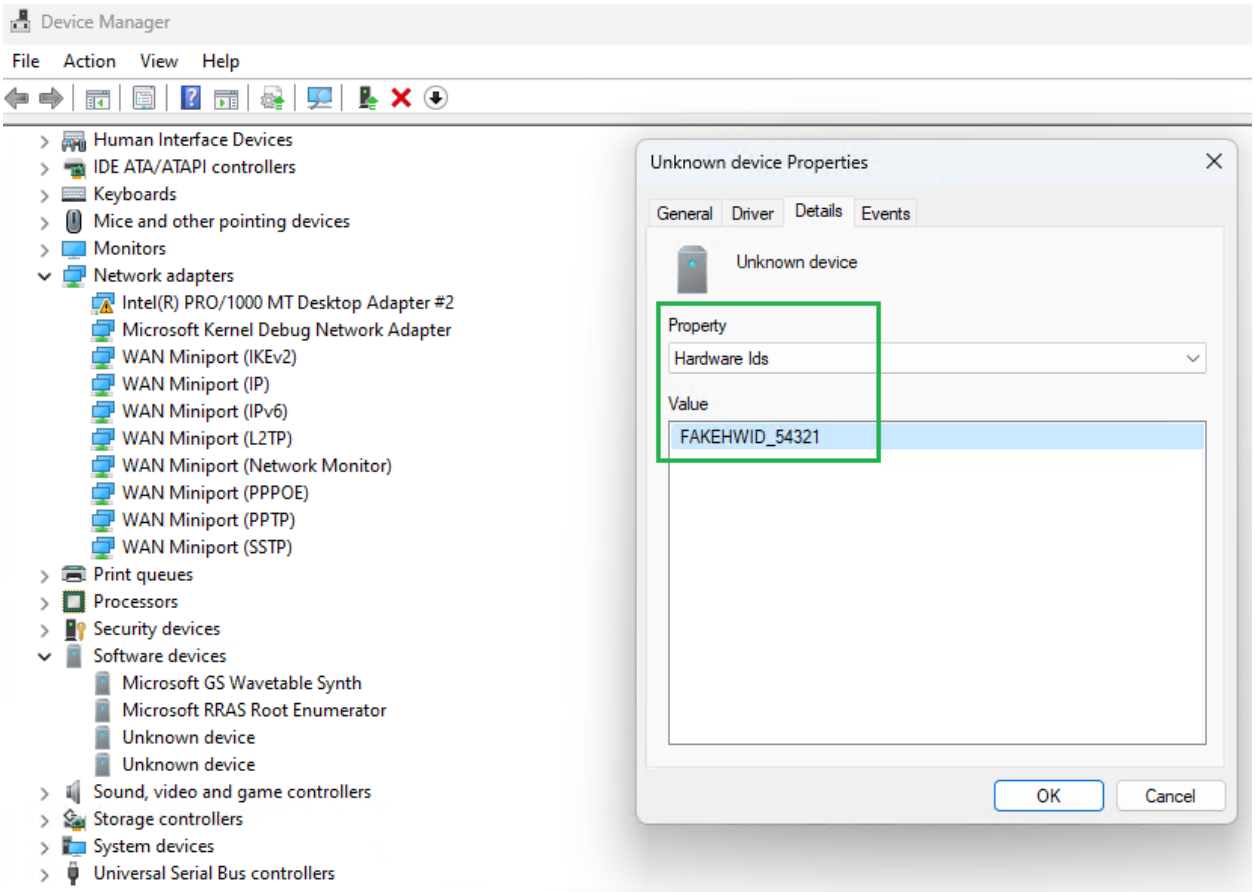
Here are the relevant registry structures after running create_sd_bind_driver.exe:

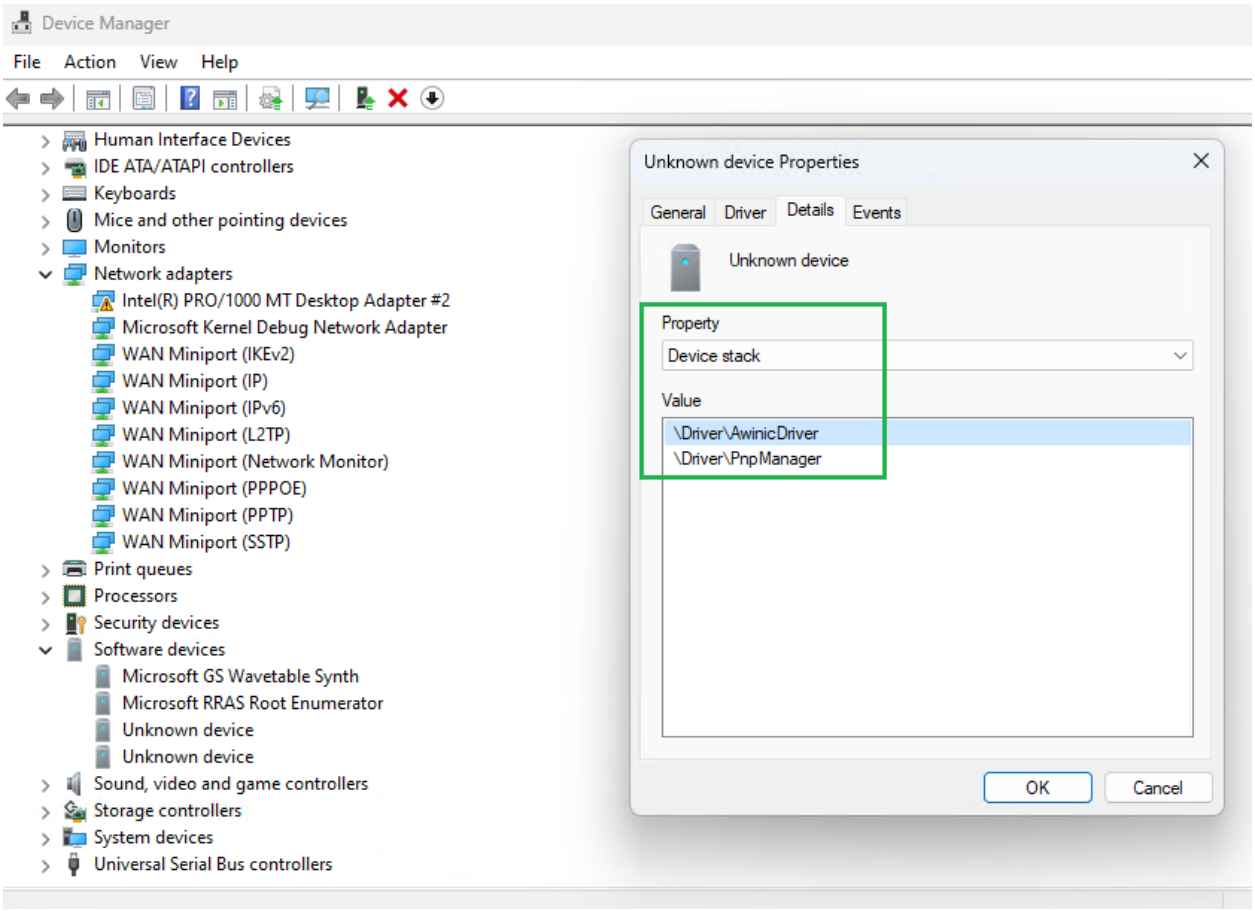



By inspecting the device again in the Device Manager we can clearly see that:

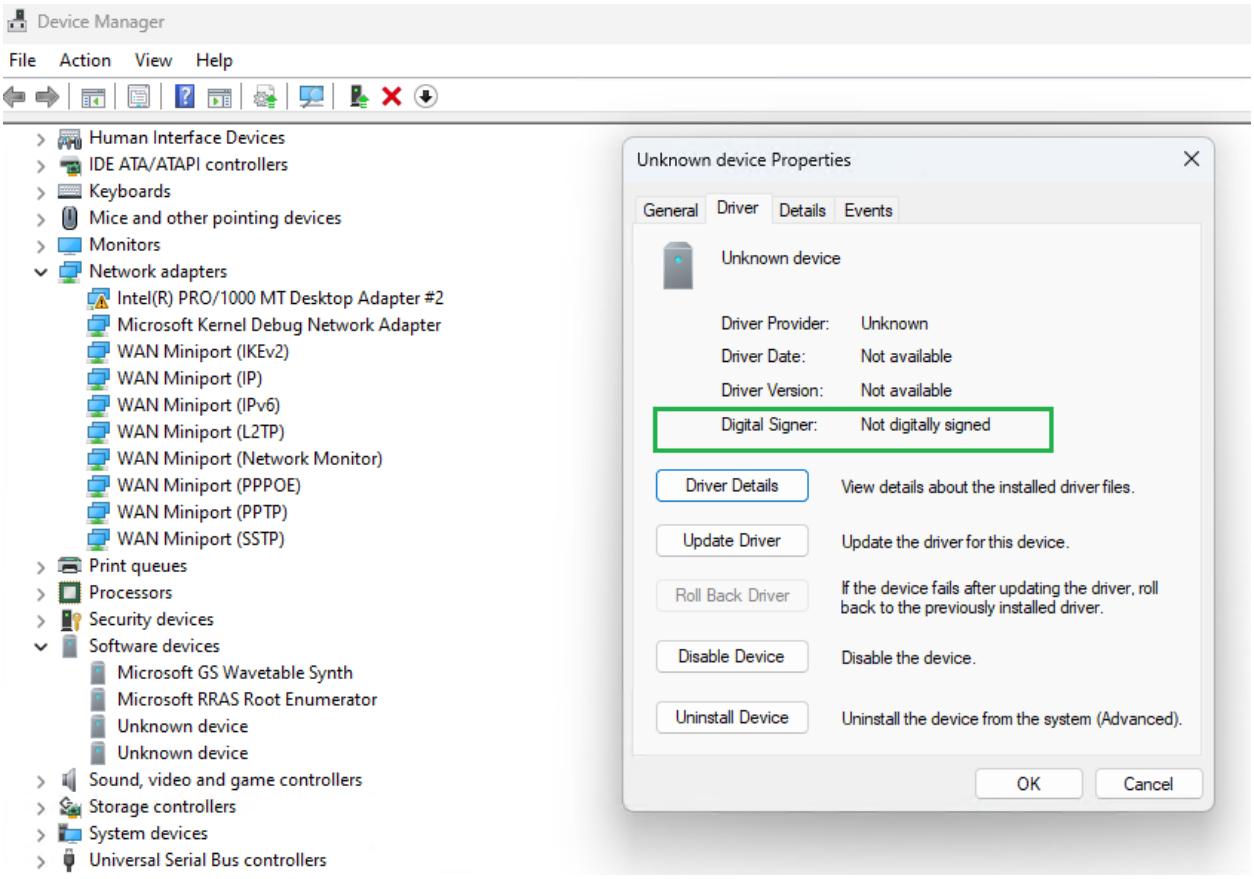
- the device was created,
- the driver got installed for it,
- the device stack got created.


Device Manager
File Action View Help
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Intel(R) PRO/1000 MT Desktop Adapter #2
Microsoft Kernel Debug Network Adapter
WAN Miniport (IKEv2)
WAN Miniport (IP)
WAN Miniport (IPv6)
WAN Miniport (L2TP)
WAN Miniport (Network Monitor)
WAN Miniport (PPPOE)
WAN Miniport (PPTP)
WAN Miniport (SSTP)
Print queues
Processors
Security devices
Software devices
Microsoft GS Wavetable Synth
Microsoft RRAS Root Enumerator
Unknown device
Unknown device
Sound, video and game controllers
Storage controllers
System devices
Universal Serial Bus controllers
Unknown device Properties
General Driver Details Events
Unknown device
Device type: Software devices
Manufacturer: Unknown
Location: Unknown
Device status
This device is working properly.
OK
Cancel


Device Manager
File Action View Help
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Intel(R) PRO/1000 MT Desktop Adapter #2
Microsoft Kernel Debug Network Adapter
WAN Miniport (IKEv2)
WAN Miniport (IP)
WAN Miniport (IPv6)
WAN Miniport (L2TP)
WAN Miniport (Network Monitor)
WAN Miniport (PPPOE)
WAN Miniport (PPTP)
WAN Miniport (SSTP)
Print queues
Processors
Security devices
Software devices
Microsoft GS Wavetable Synth
Microsoft RRAS Root Enumerator
Unknown device
Unknown device
Sound, video and game controllers
Storage controllers
System devices
Universal Serial Bus controllers
Unknown device Properties
General Driver Details Events
Unknown device
Property
Hardware Ids
Value
FAKEHWID_54321
OK
Cancel

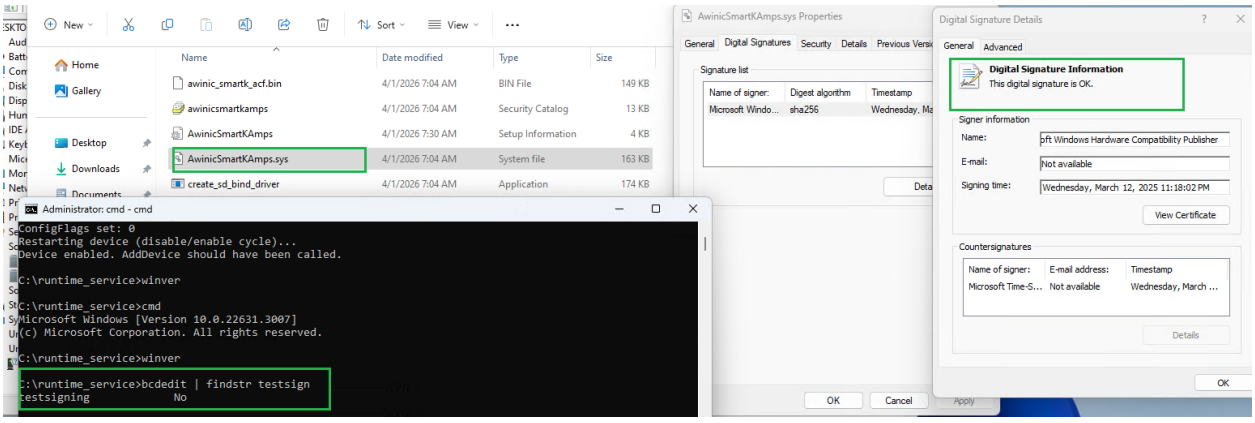



If we look into the Driver tab, we will notice that the driver is "not digitally signed", which can be a bit misleading:




The reason we are seeing this is because there is no INF file and no CAT file tied to this driver in the registry.

The .sys file itself is digitally signed, and of course test signing was not enabled at the time of testing:




This demonstrates that INF files and their signatures are not a security boundary, but rather a feature meant to prevent INF file abuse (e.g. threat actors backdooring INF files by adding malicious directives, which would then execute with administrative privileges during installation).

That makes sense - after all, we did not need INF files to append drivers to the UpperFilters/LowerFilters keys. And we don't really need them to tie a driver to a piece of hardware.

The same forced installation approach can be performed for already existing devices. We deploy the driver with sc.exe, modify the registry to tie it as the driver for that device, then restart the device.

Alternatively, we could even replace the relevant .sys file and restart the device, however that would not work if we are not able to unload the original driver before replacement (because the kernel will not load another module with the same name).

### 4.5 Working around active hardware probing

Passing various hardware probing checks mentioned in section 3.4 Active hardware interaction and probing without the corresponding physical hardware is for the most part possible, but requires either custom kernel mode code (drivers dedicated explicitly for this purpose, especially bus and filter drivers) or hypervisor access.

None of these requirements are satisfied in a typical BYOVD scenario. A thorough analysis of hardware interaction mechanisms and the feasibility of spoofing them from user-mode alone is beyond the scope of this article; however, my preliminary assessment suggests that no purely userland technique can generically bypass hardware-gated code paths without loading additional kernel mode components.

## 5. Final thoughts

While critical vulnerabilities are still quite common in Windows kernel mode drivers, not all drivers with relevant vulnerabilities existing in their code have practical value from the BYOVD perspective due to conditional (usually hardware-dependant) reachability of vulnerable code paths.

As I have demonstrated, many of those cases can be worked around by influencing the internal driver state from user-mode by creating software-emulated device nodes with spoofed hardware IDs or forced replacement of drivers for existing hardware.

Understanding these caveats can help better assess risks associated with specific device driver vulnerabilities.

It is reasonable to assume that at some point in the near future the range of BYOVD-viable drivers will start shrinking due to AI-accelerated vulnerability research and Microsoft removing trust for cross-signed drivers, just to name a few reasons.

Once that starts happening, it may give threat actors the incentive to start taking advantage of vulnerable drivers whose exploitability only seemingly depends on the presence of the corresponding physical hardware, while in practice can be worked around by solely operating from userland. Therefore, it is worth for defenders to pay attention to the forensic footprint associated with the techniques demonstrated in this article.