2024 DUBLIN

2 - 4 October, 2024 / Dublin, Ireland

BREAKING BOUNDARIES: INVESTIGATING VULNERABLE DRIVERS AND MITIGATING RISKS

Jiří Vinopal

Check Point, Czechia

jiriv@checkpoint.com

ABSTRACT

Have you ever wondered why there are so many vulnerable drivers and what might be causing them to be vulnerable? Do you want to understand why some drivers are prone to crossing security boundaries and how we can stop that?

The number of vulnerable drivers in the LOLDrivers project increases steadily every week, with several new vulnerabilities being discovered regularly. To address this issue, *Check Point Research* conducted an in-depth analysis focusing on the main causes of vulnerability.

This paper presents the findings of our research, which reveal that the majority of known vulnerable drivers share certain characteristics. Interestingly, these vulnerabilities are often not complex and can easily be addressed. Using the same methodology, we conducted a mass hunt for new drivers that may be vulnerable, uncovering thousands of potentially at-risk drivers. Additionally, we examine how drivers of well-known security products are attempting to mitigate abuse and provide a practical demonstration of how we were able to exploit chained vulnerabilities in one such product to bypass security measures and gain kernel privileges.

By demonstrating the practical vulnerability of a well-known security product, we underscore the fundamental idea that if a design flaw exists within the driver itself, it is only a matter of time and attacker ingenuity before security mechanisms are bypassed.

INTRODUCTION

Although the subject of vulnerable drivers has been around for a while, it has gained more attention from both vulnerability researchers and attackers in recent years. One potential reason for this is *Microsoft*'s effort to continuously evolve and improve *Windows* security to protect the kernel's boundary.

From the attackers' perspective, gaining kernel privileges is very often a crucial step towards owning the system. As it becomes harder to cross the security boundary (user-mode → kernel-mode) in a conventional way, driver exploitation has become their prioritized approach, both with traditional exploit primitives and abuse of legitimate driver functionalities.

There are a lot of reasons why attackers are abusing vulnerable *Windows* drivers. Exploiting drivers offers them interesting perspectives to reach certain capabilities usually not available from user-mode, such as:

- Rootkits – usually for hiding the malware on the compromised system for as long as possible and ensuring persistence.
- Minifilter drivers – intercepting I/O operations by registering pre/post operation callback routines. Malware mostly uses an 'active' minifilter to modify original I/O operations requests and results, but it can also be used passively to monitor the system.
- Elevation of privilege (EoP) – attackers can leverage the BYOVD (Bring Your Own Vulnerable Driver) technique to reach EoP, but they need to already have the privilege to load the vulnerable driver. They can also abuse known/unknown vulnerabilities in certain already installed/loaded drivers to cross the security boundary (user-mode → kernel-mode).
- Disabling/killing EDR – many techniques and projects have been introduced to abuse legitimate functionalities of vulnerable drivers to disable/kill EDR or other PP/PPL processes (e.g. via process termination, process suspending, thread suspending, closing all process objects, etc.).
- Loading unsigned malicious drivers, bypassing the driver signature enforcement.

Projects such as LOLDrivers [1] have significantly increased the popularity of vulnerable drivers among attackers. Even though the purpose of such projects is strictly focused on defensive measures, the existence of a huge, regularly updated database of vulnerable drivers also opens the space for offensive operations: there is nothing easier for attackers than to wait for the latest contributions and start the race with security companies to abuse the newly added vulnerable driver sooner than protections are properly applied.

The fact that there are so many known vulnerable drivers accumulating in one publicly accessible place should raise questions such as:

- Why are there so many vulnerable drivers, and what might be causing them to be vulnerable?
- Why are some drivers prone to crossing security boundaries?
- Do the vulnerable drivers have something in common, and if so can we address it for mitigation?

This paper presents our research findings, focusing on the vulnerable *Windows* drivers (WDM) [2, 3]. We reveal that the majority of known vulnerable drivers share certain characteristics, and that these vulnerabilities are often not complex and can easily be addressed. We also describe our hunt for new drivers that may be vulnerable, in which we uncovered thousands that were potentially at risk, and examine how the drivers of popular security products attempt to mitigate abuse, providing a demonstration of how we were able to exploit chained vulnerabilities in one such product to bypass security measures and gain kernel privileges.

BACKGROUND & KEY FINDINGS

In recent years, a lot of research has been conducted relating to vulnerable *Windows* drivers. Focusing only on the research that targeted vulnerable drivers in general (avoiding in-the-wild APT/malware-related abuse of vulnerable drivers), the majority has primarily been educational. Such works have mostly provided either some 101 steps for reverse engineering *Windows* drivers or a detailed analysis of an example vulnerable driver with the creation of a PoC.

Those that stand out, contributing to the community with some unique approach for mass hunting of vulnerable *Windows* drivers, have mostly focused on logical bugs via abusing drivers' functionalities in relation to physical memory mapping of low-level APIs (kernel functions such as `MmMapIoSpace`/`MmMapIoSpaceEx`), e.g. [4].

Despite thousands of initially detected *Windows* drivers that use these memory-mapped low-level APIs, only a handful of them survived further filtering, where the crucial moment was mainly in manual verification and finding out that only a privileged user (Administrator, System) can initiate the communication with those drivers. Because of that, even though the bug existed in those drivers, it could not be considered a vulnerability as there is no crossing of security boundary (Administrator → Kernel).

The crossing of security boundary [5] is the main aspect that plays a crucial role when it comes to a decision as to whether a certain bug can be submitted as a vulnerability. The table below summarizes the crossing of security boundaries and is simplified for our needs:

Initial	Elevated	Crossing security boundary
Non-privileged user	System	Yes
Non-privileged user	Administrator	Yes
Administrator	System	No
Service	Administrator	No
Service	System	Yes/no, it depends

There are rare cases where even an elevation Administrator → Kernel can be considered to be crossing a security boundary with an assigned vulnerability when it is actively being exploited as a zero-day in the wild (e.g. CVE-2024-21338) [6].

Even though the decision regarding security boundaries can sometimes be confusing, when it comes to vulnerable drivers, the straightforward boundary between non-privileged users and the system is the one we should target and cross.

The main idea behind this research is to carry out mass hunting for vulnerable drivers in a slightly different way. First, we need to always be sure that we are crossing the security boundary, or in other words, that we can communicate with the driver as a non-privileged user. The DACL (Discretionary Access Control List), which is a part of SDDL (Security Descriptor Definition Language) applied on the driver's device, is the first thing that matters [7, 8].

Even when we find such drivers accessible from non-privileged users, it does not necessarily mean that they can be considered vulnerable. They need to be further filtered out by searching for certain capabilities of their legitimate functionalities that can potentially be abused to commit privileged operations, in the 'best' scenario leading to LPE (Local Privilege Escalation).

VULNERABLE DRIVERS – WHAT DO THEY HAVE IN COMMON?

When it comes to a publicly available database of known-to-be-vulnerable *Windows* drivers that can easily be processed, there is no better place than the LOLDrivers project. If we focus only on those drivers in the database that are *known to be vulnerable* (avoiding those that are known to be malicious) and further filter those that are 64-bit and signed (with the YARA rule provided below), we get 924 drivers that can serve as a starting point for our investigation.

```
import "pe"

rule signed_driver_64bit
{
    meta:
        description = "Detects 64-bit signed drivers"
        author = "Jiri Vinopal (jiriv)"
        date = "2024-06-09"
    condition:
        // Detect PE
        uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and
        // Detect 64-bit Windows drivers
        uint16(uint32(0x3C) + 0x5c) == 0x0001 and uint16(uint32(0x3C) + 0x18) == 0x020b and
```

```
        // Detect only signed drivers, not a real verification
        (pe.number_of_signatures > 0 and for all i in (0..pe.number_of_signatures -1):
            (pe.signatures[i].verified and not pe.signatures[i].subject contains "WDKTestCert"))
}
```

After initial analysis, we can quickly get to the main point of interest. Approximately 90% of those 924 drivers are accessible by non-privileged users (detected by a YARA rule similar to the one included in the 'Mass hunt for not-known-to-be-vulnerable drivers' section), and as such, they are prone to crossing the security boundary (non-privileged user → system), where any of their capabilities that can be abused to commit privileged operations make them vulnerable.

Focusing on those 90% of drivers prone to cross the security boundary (accessible by non-privileged users), we can immediately get an overview of typical design flaws that repeatedly occur in their code:

1. Creating devices with no DACL via the `IoCreateDevice` function. Unfortunately, the `IoCreateDevice` function does not allow DACL to be specified. As a result, the developers must define it either directly in the registry or via the configuration file (INF AddReg directive). If they fail to do so, any user can access the device.

```
snwprintf(&Dest, 0x1EuLL, L"\\Device\\%s", v5);
snwprintf(SourceString, 0x1EuLL, L"\\DosDevices\\%s", qword_1CDF0);
RtlInitUnicodeString(&DestinationString, &Dest);
RtlInitUnicodeString(&SymbolicLinkName, SourceString);
result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x9876u, 0, 1u, &DeviceObject);
dword_1CE3C = result;
if ( result >= 0 )
{
  dword_1CE3C = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
```

Figure 1: Example vulnerable driver – IoCreateDevice (no DACL).

2. Creating devices with a weak DACL using the `IoCreateDeviceSecure` function. The function `IoCreateDeviceSecure` allows DACL to be specified, and as such, it is considered more secure. Still, if a weak DACL is applied, the created device can be accessible by less privileged users.

```
result = IoCreateDeviceSecure(DriverObject, 0, &DeviceName, 0x22u, FILE_DEVICE_SECURE_OPEN, 0, &WeakSDDLString, 0LL, &DeviceObject);
if ( result >= 0 )
{
  result = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
  if ( result >= 0 )
  {
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)sub_29728;
    DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = (PDRIVER_DISPATCH)sub_29728;
```

Figure 2: Example vulnerable driver – IoCreateDeviceSecure (weak DACL).

3. Creating devices with a strong DACL using the `IoCreateDeviceSecure` function but without the `FILE_DEVICE_SECURE_OPEN` flag, which is part of the *device characteristics*. If the `DeviceCharacteristics` value is not ORed with 0x00000100 (`FILE_DEVICE_SECURE_OPEN`), the same security settings are not applied to the whole device's namespace. Every device has its own namespace, where names in the namespace are paths that begin with the device's name. For a device named `\Device\DeviceName`, its namespace consists of any name of the form `\Device\DeviceName\anyfile`. The lack of the `FILE_DEVICE_SECURE_OPEN` flag can be abused to obtain a full access handle to the device itself, even by a non-privileged user, because the strong DACL is not propagated to the namespace, e.g. opening a handle to `\Device\DeviceName\anyfile` will return a handle for the device itself `\Device\DeviceName`.

```
                text "UTF-16LE", 'D:P(A;;GA;;;SY)(A;;GA;;;BA)',0
; const UNICODE_STRING StrongSDDLString
StrongSDDLString UNICODE_STRING <36h, 38h, offset aDPAGaSyAGaBa>

result = IoCreateDeviceSecure(DriverObject, 0x10u, &DeviceName, 0x22u, 0, 0, &StrongSDDLString, 0LL, &DeviceObject);
if ( result < 0 || !DeviceObject )
{
                                                    DeviceCharacteristics: int
```

Figure 3: Example vulnerable driver – IoCreateDeviceSecure (strong DACL, no FILE_DEVICE_SECURE_OPEN).

As we revealed above, the crucial condition when the driver can be considered vulnerable begins with its ability to cross security boundaries, rather than the bug itself. In other words, hunting for non-privileged user-accessible drivers can be used as an interesting starting point to mass hunt for new, not-known-to-be-vulnerable drivers. In general, we must initially focus only on those *Windows* drivers that are either not explicit enough about the DACL (no/weak DACL) or on those that are using a strong DACL with a combination of not presented `FILE_DEVICE_SECURE_OPEN` device characteristics flag.

Even though some products using drivers without a directly applied DACL (in-code) can set the DACL during installation, in the appropriate registry, or via the INF configuration file, these drivers can still be abused using the BYOVD technique.

Furthermore, non-privileged user-accessible drivers cannot be considered as vulnerable in general, rather only those that also combine this design flaw with a demonstrable abuse of their capabilities to reach some privileged operations.

MASS HUNT FOR NOT-KNOWN-TO-BE-VULNERABLE DRIVERS

As we described in the previous section, the crucial condition when the driver can be considered vulnerable begins not with the bug itself but with its ability to cross security boundaries. Because of that, we start our mass hunting for new potentially vulnerable drivers by putting together the common design flaws that lead to non-privileged user-accessible drivers. We created a general methodology describing the main steps we followed during the hunting process.

General methodology

1. The initial creation of the YARA rule to find new potentially vulnerable drivers (non-restricted access, non-privileged users can communicate with).
2. Enriching the YARA rule with common driver capabilities (usage of certain kernel functions) that can be abused to reach some privileged operation.
3. Further improving the created YARA rule and using it with *VT Retrohunt*.
4. Filtering valid-signed 64-bit drivers.
5. De-duplication of found drivers.
6. Reverse engineering and verifying the driver vulnerability (there can be a variety of impacts, but we should primarily target the EoP).
7. PoC creation for the found vulnerability.
8. Description of the vulnerability and reporting to vendor.

Initially, we put together a *YARA* rule and used the *VirusTotal Retrohunt* service to mass hunt for 64-bit signed drivers accessible by non-privileged users. This YARA rule targeted those drivers that are either not using the DACL (e.g. direct usage of `IoCreateDevice`) or using a weak one (e.g. usage of `IoCreateDeviceSecure` with a weak DACL). Note that using this YARA rule to hunt for new potentially vulnerable drivers, we avoid the detection of those drivers that are already a part of the LOLDrivers project.

We enriched the YARA rule with a list of common driver capabilities (usage of certain kernel APIs, e.g. `ZwOpenProcess`, `ZwOpenThread`, `ZwOpenProcessTokenEx`, etc.) that can be abused to reach some privileged operation, potentially leading to LPE and other vulnerabilities. This served us to be more strict about the detected drivers regarding their potential abuse, focusing only on the rich-featured ones.

```
import "pe"
import "hash"

rule susp_risk_vuln_driver
{
    meta:
        description = "Detects new potentially vulnerable (at-risk) 64-bit signed drivers
with easy-to-abuse capabilities, loldrivers excluded"
        author = "Jiri Vinopal (jiriv)"
    strings:
        $IoCreateDevice = "IoCreateDevice" ascii wide
        $IoCreateDeviceSecure = "IoCreateDeviceSecure" ascii wide
        $api_capa_a1 = "ObCloseHandle" ascii wide
        $api_capa_a2 = "ZwTerminateProcess" ascii wide
        $api_capa_a3 = "ZwSuspendthread" ascii wide
        $api_capa_a4 = "ZwOpenProcess" ascii wide
        $api_capa_a5 = "ZwOpenThread" ascii wide
        $api_capa_a6 = "ZwOpenProcessTokenEx" ascii wide
        $api_capa_a7 = "ZwAdjustPrivilegesToken" ascii wide
        $api_capa_a8 = "ZwDeleteFile" ascii wide
        $api_capa_b1 = "ZwCreateFile" ascii wide
```

```
        $api_capa_b2 = "IoCreateFile" ascii wide
        $api_capa_b3 = "ZwOpenSymbolicLinkObject" ascii wide
        $api_capa_b4 = "ZwDeleteKey" ascii wide
        $api_capa_b5 = "MmSystemRangeStart" ascii wide
        $api_capa_b6 = "ProbeForRead" ascii wide
        $api_capa_b7 = "ProbeForWrite" ascii wide
        $api_capa_b8 = "MmMapIoSpace" ascii wide
        $api_capa_b9= "ZwMapViewOfSection" ascii wide
        $api_capa_b10 = "IoAllocateMdl" ascii wide
        $dacl1 = "(A;;GRGW;;;WD)" ascii wide
        $dacl2 = "(A;;GWGR;;;WD)" ascii wide
        $dacl3 = "(A;;GA;;;WD)" ascii wide
        $dacl4 = "(A;;GRGW;;;BU)" ascii wide
        $dacl5 = "(A;;GWGR;;;BU)" ascii wide
        $dacl6 = "(A;;GA;;;BU)" ascii wide
        $dacl7 = "(A;;GRGW;;;AU)" ascii wide
        $dacl8 = "(A;;GWGR;;;AU)" ascii wide
        $dacl9 = "(A;;GA;;;AU)" ascii wide
    condition:
        // Detect PE
        uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and
        // Detect 64-bit Windows drivers
        uint16(uint32(0x3C) + 0x5c) == 0x0001 and uint16(uint32(0x3C) + 0x18) == 0x020b and
        (($IoCreateDevice and not $IoCreateDeviceSecure) or ($IoCreateDeviceSecure and any
of ($dacl*))) and
        (2 of ($api_capa_a*) or 6 of ($api_capa_b*)) and
        (pe.number_of_signatures > 0 and for all i in (0..pe.number_of_signatures -1):
            (pe.signatures[i].verified and not pe.signatures[i].subject contains
"WDKTestCert")) and
        not (
            // Exclude all LOLDrivers "https://github.com/magicsword-io/LOLDrivers/tree/
main/drivers"
            hash.md5(0, filesize) == "003dc41d148ec3286dc7df404ba3f2aa" or
            hash.md5(0, filesize) == "0067c788e1cb174f008c325ebde56c22" or
            ...
            )
}
```

This YARA rule was further improved to cover scenarios such as using `IoCreateDeviceSecure` with strong DACL but without the `FILE_DEVICE_SECURE_OPEN` flag, which is part of the device characteristics (the same security settings are not applied to the whole device's namespace) [11].

The detected drivers were further processed to eliminate those whose signatures caused verification errors (using *Sigcheck* and *SignTool* [12, 13]) or explicitly set a strong DACL in the configuration file (*INF AddReg directive*) [14] – despite the fact they can still be used in BYOVD scenarios. Duplicates of the same driver (different versions) were filtered out with mass processing of the PE version information and using the *imphash* comparison [15].

Our primary tool for this was the *Sigcheck* utility as it enables the mass processing of the detected drivers, providing all information needed for further post-processing (signature verification + revocation status, imphash, PE version info) in CSV file format.

```
.\sigcheck64.exe -c -h -nobanner -w output.csv .\drivers\
```

Still, the *SignTool* utility is helpful for further signature verification by using the x64 kernel-mode driver signing policy.

```
.\signtool.exe verify /kp .\drivers\driver.sys
```

The resulting CSV output with all the necessary information was visualized and post-processed using the *Timeline Explorer* tool [18], which helps to quickly filter, sort, and group information as needed.

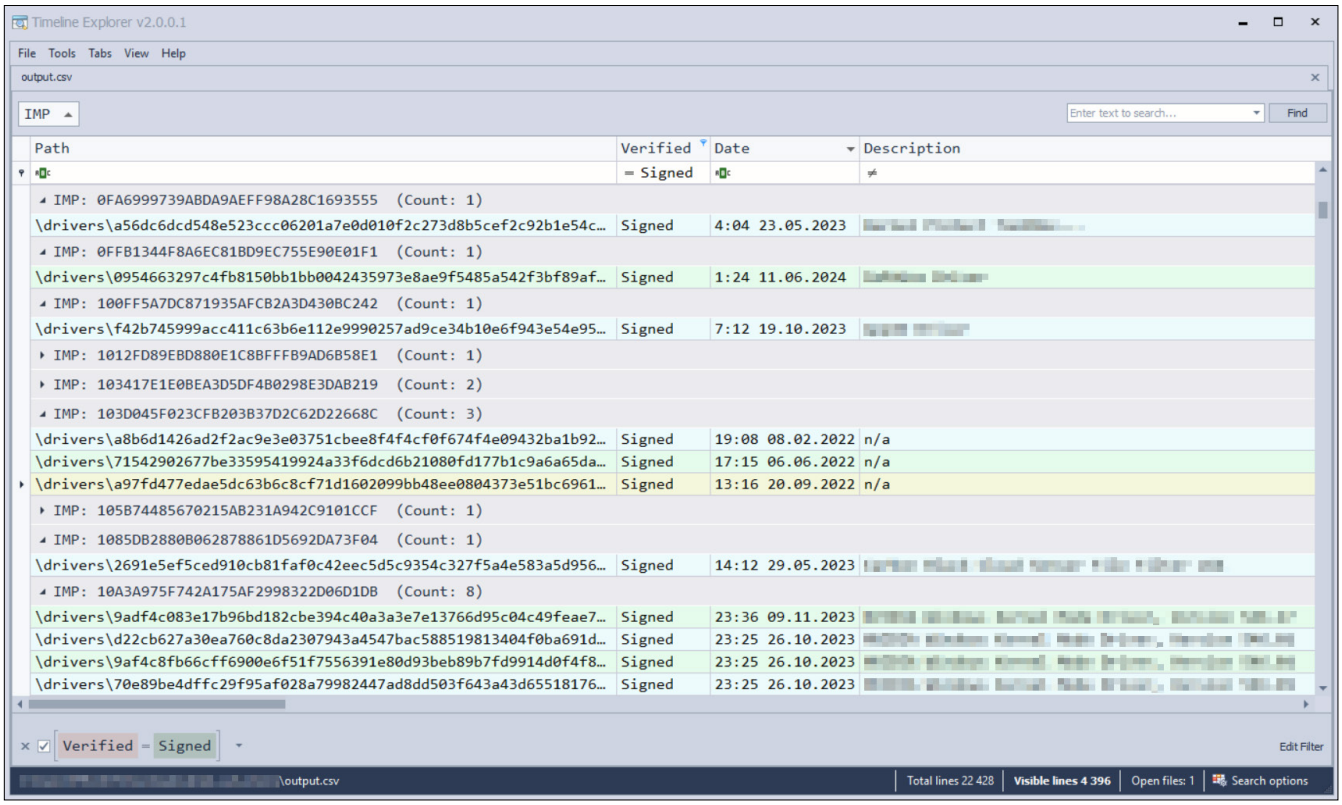

Figure 4: Using the Timeline Explorer tool for post-processing detected drivers' information.

Hunting results

The above-described YARA rule was first used against the up-to-date LOLDrivers database. As of June 2024, this database contains more than 1,800 *Windows* drivers (both known-malicious and known-vulnerable drivers), of which 924 are 64-bit, signed drivers belonging to the known-to-be-vulnerable group. Using the YARA rule, we were able to confirm that 90% of those 924 drivers are accessible by non-privileged users, and approximately 20% passed through the more strict rule, filtering only those containing easy-to-abuse capabilities (kernel functions).

This relatively strict YARA rule (detecting non-privileged user accessible, 64-bit, signed drivers containing easy-to-abuse capabilities, avoiding LOLDrivers) was used with *VT Retrohunt* over a one-year period from June 2023 to June 2024, resulting in initial detection of about 22.5k *Windows* at-risk drivers.

Further post-processing of those 22.5k drivers resulted in 4.4k of them passing through signature verification, where approximately 1.9k of them survived the de-duplication process (via PE version information and *imphash* comparison).

Even if the driver is initially detected by the created YARA rule and passes through all the processing, it does not necessarily mean that the driver is vulnerable. Only if we can abuse the driver's capabilities to perform some privileged operation can it be considered vulnerable. For this, manual verification (reversing with *IDA* [16] or a similar disassembler) and the creation of a PoC are always needed. Despite the time-consuming procedure, which is barely possible to perform reliably on 1.9k drivers (and out of the scope of this research), we found the verification dangerously easy just by going through the first few dozen of them, resulting in newly discovered vulnerable drivers that were responsibly reported. Because of the overwhelming number of potential at-risk drivers, we decided to share our findings to provide some mitigation, still demonstrating one such example that underlines our results.

PRACTICAL DEMONSTRATION

One of the newly discovered vulnerable drivers, which we responsibly reported, was an anti-rootkit module used by *Dr.Web* products [19]. The reported vulnerability was patched, and the public disclosure tracked under the BDU:2024-02836 was assigned a high severity 8.8 base score (CVSS 3.0) [20]. The assigning of an industry-standard CVE identifier to this vulnerability is currently in progress (June 2024).

The vulnerable component of *Dr.Web* products is a 64-bit, valid-signed *Windows* kernel device driver. Unfortunately, this driver has no name and description, but it is usually dropped to disk with a name such as 'dwt-6088-1976-26975aba.sys' or 'dwt-2444-2348-9cc4e5df.sys'.

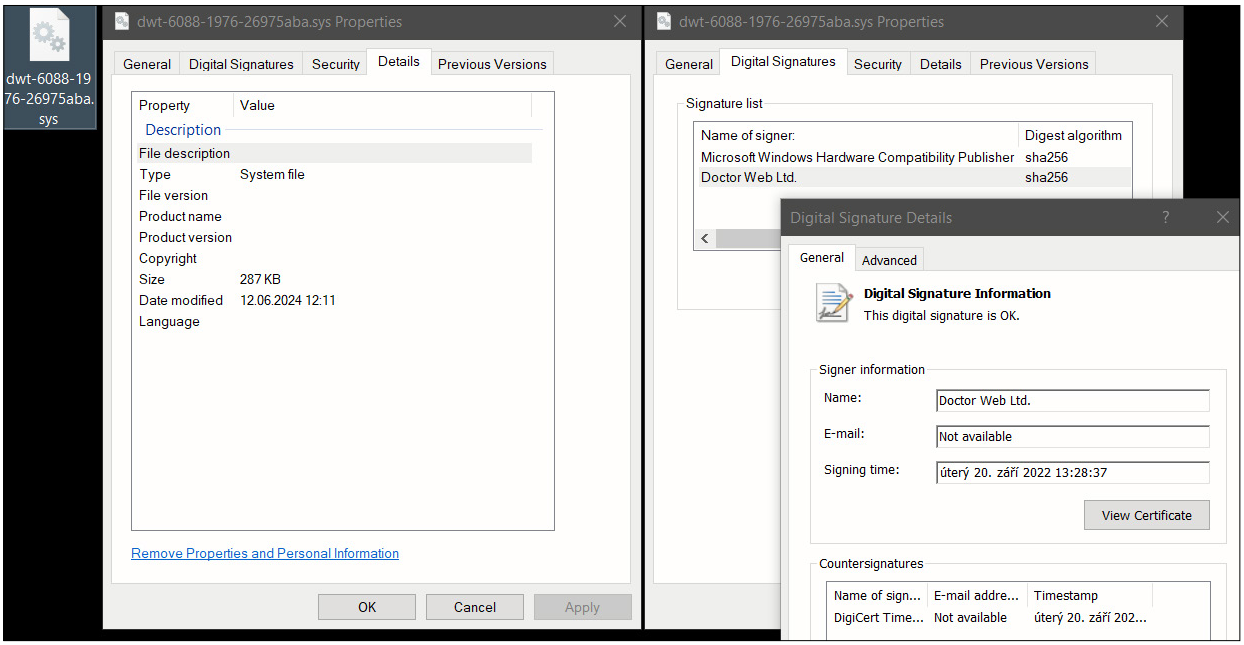

Figure 5: Vulnerable component of Dr.Web products.

The original 'pdb' path still reveals the name 'dwshield_x64.pdb' (dwshield.sys).

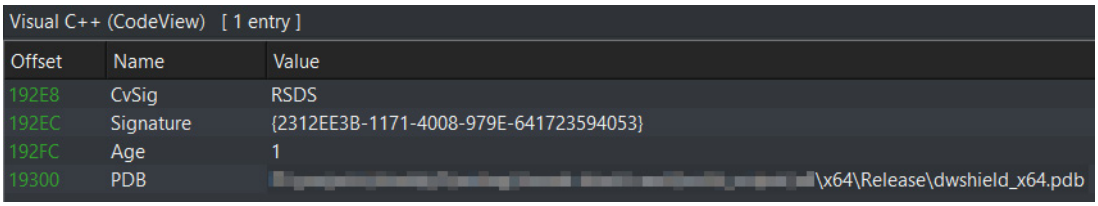

Figure 6: The original 'pdb' path revealing the name 'dwshield_x64.pdb'.

We have found several vulnerable versions of this driver:

MD5 hash	SHA-256 hash
4cf84abc9e2d9a85b42c98a6b91bb011	a97fd477edae5dc63b6c8cf71d1602099bb48ee0804373e51bc6961fb0db6d5b
c142d4ce995b37e43e4ff76b6920fc5d	c452ae27e934c0a411a840dc8e824ccaeaf22fdfadf9f3072c1c162203a3fc2d
20a385e458b520a7a3decd6157f80c75	ca671b88f6476caa1b55cc4c6d1aef5fef5c546a17fff5b01d5d5a1c53516650
adef75aefdfc84f36fd349c5c2ccda26	a8b6d1426ad2f2ac9e3e03751cbee8f4f4cf0f674f4e09432ba1b92c36d80e4d
e44ab7b12eabc03dad15a882bb1dd8e2	5fb9b947026afab01076f35d9626e996b108af3fe76e0d0dd61eb8177a3d4075
7db0a75f8d6b7b53418a6652234ff595	71542902677be33595419924a33f6dcd6b21080fd177b1c9a6a65dab59ed93cb

Among the affected products that were confirmed to be exploitable are *Dr.Web Security Space* [21], *Dr.Web KATANA* [22] and *Dr.Web CureIt* [23], where in all cases, the vulnerability in the driver component leads to local privilege escalation (LPE), arbitrary read/write kernel/user mode access and arbitrary process termination.

Vulnerability description

The vulnerable driver does not explicitly specify the DACL (non-privileged user access allowed) for the created device (using the `IoCreateDevice` function) but implements different protection mechanisms to restrict access to the driver's device. All these protections can be bypassed.

The driver creates a device with an auto-generated name but also sets up a symbolic link (using `IoCreateSymbolicLink`) with a randomly generated name (different on each driver load) that can be used to obtain a handle to the device from user-mode. As the device's symlink name is always quite unique and rare (a 16-character hexadecimal string, e.g. `46ed8954975a9788`), it can be brute-forced by enumerating all symlink names (via `QueryDosDeviceA`) and finding the desired one.

The driver implements a digital signature check of the process that tries to obtain a handle for the driver's device. To bypass this protection, we need to impersonate some of the *Dr.Web* components that are valid-signed and allowed to

communicate with the driver. Usually, code-injection techniques can be used to achieve this, but as our target is a security product that successfully blocks the majority of them, we need to come up with a different solution.

Digging deeper, we were able to find DLL side-loading vulnerabilities in a few *Dr.Web* components that are allowed to communicate with the driver (valid-signed) and used them to bypass the digital signature protection mechanism:

Dr.Web component	SHA-256 hash	Side-loaded DLL
dwservice.exe	6e60fdcabdfd74274a7e2da62315fba484ef8c587bafbb3c39cdeb741a39b79c	wldp.dll
spideragent.exe	ba2a0cba80bb02e6a4fa7a5dca6045804e54d14839ef33af1168a053014719c5	uxtheme.dll

This way, we were able to bypass the driver's access restrictions and proceed with different IOCTLs that led to LPE, arbitrary RW kernel/user mode access, and arbitrary process termination.

IOCTL	Functionality
0x22E076	Arbitrary kernel/user mode memory read
0x22E078	Arbitrary kernel/user mode memory write
0x22E044	Obtaining arbitrary full access process token handle
0x22E024	Obtaining arbitrary full access process handle
0x22E034	Obtaining arbitrary full access thread handle
0x22A02C	Arbitrary process termination

The main product, *Dr.Web Security Space*, is a full-featured antivirus and implements another protection to restrict access to the vulnerable driver.

First of all, the installed components are different and no longer vulnerable to DLL side-loading. Still, we can bypass the digital signature check by deploying a different vulnerable version of `dwservice.exe` (copied from the *Dr.Web KATANA* product) and using it to side-load the `wldp.dll` (or a combination of `spideragent.exe` + `uxtheme.dll`).

Another protection mechanism added by *Dr.Web Security Space* is 'module caching'. DLLs that *Dr.Web* components can load are verified if they are signed by trusted authorities and added to the 'module cache'. The *Dr.Web* filter driver monitors changes to these files in the cache. However, we were able to bypass this protection by finding a bug in the filter driver logic that monitors the changes to files already in the module cache and, as a result, skipping the verification.

By running the *Dr.Web* component vulnerable to DLL side-loading (`dwservice.exe`) to side-load an original valid-signed DLL (`wldp.dll`) from the user-accessible location, the DLL is successfully verified and put into the module cache. Deleting the original signed DLL and moving our custom implanted DLL (with the same name, `wldp.dll`) to the same location will not result in a new file creation or data changes that the *Dr.Web* filter driver actively monitors (note, it is important to move the custom implanted DLL and not copy it because of different system behaviour – only the NTFS metadata change). The second execution of the vulnerable component side-loads our custom implanted DLL, skipping the verification and, with that, bypassing the protection.

By chaining all of the above-mentioned vulnerabilities, we were able to reach the LPE even in *Dr.Web Security Space*.

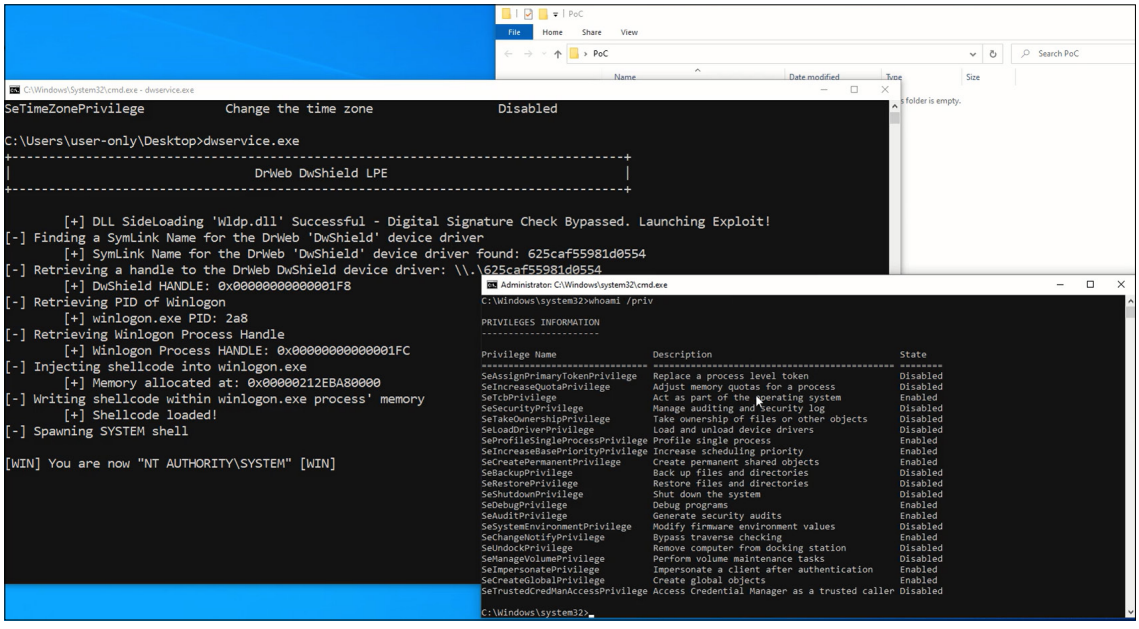

Figure 7: PoC: LPE in the Dr.Web Security Space product.

The core of the vulnerability in the *Dr.Web* anti-rootkit driver is not as complex as the implemented above-mentioned protection mechanisms and can easily be addressed. Using the more secure kernel function `IoCreateDeviceSecure` allows explicit setting of a strong DACL that can be used to restrict the device access to only high-privileged accounts (Administrator, System). Specifying the `FILE_DEVICE_SECURE_OPEN` device characteristic ensures the propagation of DACL to the whole device namespace. The last step is the repairing of the bug in the *Dr.Web* filter driver logic responsible for the reliability of the module caching protection.

By demonstrating the practical vulnerability of a well-known security product, we underscore the fundamental idea that if a design flaw exists within the driver itself, it is only a matter of time and attacker ingenuity before security mechanisms are bypassed.

MITIGATION & REMEDIATION

Following all that has been described in previous sections, it is obvious that crossing the security boundary is the main aspect when it comes to deciding whether a certain driver's bug can be considered a vulnerability. The combination of a non-privileged user-accessible driver with a demonstrable abuse of its capabilities to commit some privileged operations makes the driver vulnerable.

Whenever access to the driver is allowed, even for non-privileged users, the developers must be absolutely sure that none of its functionalities that can be used to reach some privileged operation are exposed to them. This can be very problematic in a lot of cases as already the idea behind the need to have a kernel module comes from the requirement of certain functionality that can be reached only from the kernel. Once this functionality is exposed to other user-mode components of certain products, it can also be abused by others in a way that is certainly out of the developer's initial intention.

It can be relatively easy to track the driver's user-mode accessible functionalities from its first release, but it gets harder as time passes and new versions of the driver are released, enriching the driver with other capabilities and functionalities.

As we already proved, in many cases, the accessibility restriction is lost in time and, sooner or later, some of the latest added user-mode-exposed functionality is abused.

To address this issue, there are a few relatively simple steps that can be taken to remediate the resulting vulnerability:

1. Always ensure that none of the driver's functionalities that allow privileged operations are exposed to a non-privileged user.
2. Use the 'secure' version of the function to create the driver's device `IoCreateDeviceSecure` instead of the 'non-secure' `IoCreateDevice` and combine it with a strong DACL (SDDL) to restrict the access only to high-privileged accounts (Administrator, System), e.g. `D:P(A;;GA;;;SY)(A;;GA;;;BA)`. Usually, whenever developers follow best practices, certain product components that are allowed to communicate with the driver are already running under the System service, so there is no need to expose the accessibility to a non-privileged user.
3. The previous step can be replaced by defining the DACL either directly in the registry or via the configuration file (*INF AddReg directive*).
4. Whenever a certain driver's device is being created, and it is possible to do so, be explicit about the device characteristics and set the `FILE_DEVICE_SECURE_OPEN` flag. This ensures the propagation of DACL to the whole device namespace.

Going through some of the newly detected 1.9k drivers accessible by non-privileged users, we quickly noticed that those that usually belong to well-known security products attempt to mitigate potential abuse using custom protection mechanisms. We examined the common techniques used by such products, which, in most cases, served primarily as an added accessibility restriction.

Even though the implemented custom protections complicate the driver's exploitation of the fundamental design flaw in its accessibility, in general, it is merely an obstacle that can be overcome with some ingenuity.

The table below shows some of the most common mitigation techniques usually implemented by security products, together with their example bypasses.

Security product mitigation	**Example bypass**
Digital signature check of the process's main module	Code injection techniques, DLL side-loading
IOCTL registration, a specific data structure used as a client-registration	Reverse engineering the data structure
PID/TID registration	Reverse engineering the registration logic
First-only registration	Race condition
Randomly generated device/symbolic link name	Brute-force

Security product mitigation	Example bypass
IOCTL code sorting via required access permissions (default, only a developer design)	Abusing read-only permission to commit write operation
High-privilege check	-
Encoding of the IOCTL codes and transferred data	Reverse engineering the encoding logic

As the implementation of custom protections that should mitigate the potential abuse of a driver's accessibility appears to be less effective than it should be, we are left with remediation. Unfortunately, even though we remediate the driver's vulnerability (crossing the security boundary) with its access restriction, this driver ('not vulnerable') still features some of the kernel-restricted capabilities that can possibly be abused using the BYOVD technique.

The other problematic area is what actually happens with a reported vulnerable driver. In the best scenario, the vulnerability is patched (typically just by setting the DACL, rejecting non-privileged users' requests), the certificate is revoked (not so often), and a new version of the driver is released. But as *Microsoft Windows* allows the loading of kernel drivers with signatures whose certificates are expired or revoked, there is no real obstacle to prevent attackers from continuing to abuse the reported vulnerable driver.

One promising protection available since the *Windows 11* 2022 update is the *Microsoft* vulnerable driver blocklist [24]. The vulnerable drivers are blocked by default using Hypervisor-Protected Code Integrity (HVCI) [25]. However, this approach is only effective if the vulnerable driver is known in advance and part of the blocklist. Note that the blocklist is typically updated 1-2 times per year.

Furthermore, not all drivers identified as vulnerable in the LOLDrivers project are included in *Microsoft*'s vulnerable driver blocklist. Once a vulnerable driver is publicly disclosed, it is very likely to be added quickly to the LOLDrivers database. Even if *Microsoft* decides to include such a driver in its blocklist, attackers have a window of at least six months to exploit it before the blocklist is updated.

It appears that a more comprehensive solution is needed to protect against vulnerable drivers and their exploitation. Preventing the loading of drivers signed with revoked or expired certificates and using *Microsoft's* vulnerable driver blocklist would be far more effective than relying solely on the blocklist.

Such a solution is unlikely to be implemented soon. Therefore, we can expect that threat actors will continue to exploit both known and yet-to-be-discovered vulnerable drivers.

CONCLUSION

This paper presented the findings of our research, focusing on the vulnerable *Windows* drivers. We revealed that the majority of known vulnerable drivers share some of the most common design flaws, resulting in non-restricted access, even for the non-privileged user. The crucial condition when the driver can be considered vulnerable begins not with the bug itself but with its ability to cross security boundaries. In other words, the combination of a non-privileged user-accessible driver with a demonstrable abuse of its capabilities to commit some privileged operations is what makes the driver vulnerable. These vulnerabilities are often not complex and can easily be addressed just by properly restricting the access to the driver's device (setting a strong DACL) and propagating the same restriction to the whole device namespace.

Using the same methodology, we put together the common design flaws that lead to non-privileged user-accessible drivers and created a YARA rule to conduct a mass hunt for new drivers that may be vulnerable, uncovering thousands of potentially at-risk drivers.

Additionally, we examined how some of the newly detected at-risk drivers that belong to well-known security products attempt to mitigate abuse and provided some simple common bypasses that underline the non-effectiveness of the implemented mitigations.

By demonstrating the practical vulnerability of a well-known security product, we underscored the fundamental idea that if a design flaw exists within the driver itself, it is only a matter of time and attacker ingenuity before security mechanisms are bypassed.

A more comprehensive solution is needed to protect against vulnerable drivers and their exploitation. Preventing the loading of drivers signed with revoked or expired certificates and using *Microsoft*'s vulnerable driver blocklist would be far more effective than relying solely on the blocklist. Such a solution is unlikely to be implemented soon. Therefore, we can expect that threat actors will continue to exploit both known and yet-to-be-discovered vulnerable drivers, and we should keep monitoring this issue.

REFERENCES

[1] LOLDrivers. https://github.com/magicsword-io/LOLDrivers.

[2] Microsoft. WDM Drivers. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-wdm.

[3] Microsoft. WDM Drivers. https://learn.microsoft.com/en-us/windows-hardware/drivers/wdf/differences-between-wdm-and-kmdf.

[4] Haruyama, T. Hunting Vulnerable Kernel Drivers. VMware. 31 October 2023. https://blogs.vmware.com/security/2023/10/hunting-vulnerable-kernel-drivers.html.

[5] Microsoft. Security Boundaries. https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria.

[6] Vojtěšek, J. Lazarus and the FudModule Rootkit: Beyond BYOVD with an Admin-to-Kernel Zero-Day. Avast. 28 February 2024. https://decoded.avast.io/janvojtesek/lazarus-and-the-fudmodule-rootkit-beyond-byovd-with-an-admin-to-kernel-zero-day/.

[7] Microsoft. SDDL for Device Objects. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/sddl-for-device-objects.

[8] Microsoft. Applying Security Descriptors on the Device Object. https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/applying-security-descriptors-on-the-device-object.

[9] YARA. https://github.com/VirusTotal/yara.

[10] VirusTotal Retrohunt. https://virustotal.readme.io/docs/retrohunt.

[11] Microsoft. Device characteristics. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/specifying-device-characteristics.

[12] Microsoft. SignTool. https://learn.microsoft.com/en-us/windows/win32/seccrypto/signtool.

[13] Microsoft. Sigcheck. https://learn.microsoft.com/en-us/sysinternals/downloads/sigcheck.

[14] Microsoft. INF AddReg directive. https://learn.microsoft.com/en-us/windows-hardware/drivers/install/inf-addreg-directive.

[15] Mandiant. Tracking Malware with Import Hashing. 23 January 2014. https://cloud.google.com/blog/topics/threat-intelligence/tracking-malware-import-hashing/.

[16] IDA. https://hex-rays.com/.

[17] Shimony, E. Finding Bugs in Windows Drivers, Part 1 – WDM. Cyber Ark. 24 May 2022. https://www.cyberark.com/resources/threat-research-blog/finding-bugs-in-windows-drivers-part-1-wdm.

[18] Timeline Explorer. https://ericzimmerman.github.io/#!index.md.

[19] Dr.Web. https://www.drweb.com/.

[20] Dr.Web BDU:2024-02836. https://bdu.fstec.ru/vul/2024-02836.

[21] Dr.Web Security Space. https://products.drweb.com/win/security_space/?lng=en.

[22] Dr.Web KATANA. https://products.drweb.com/home/katana/?lng=en.

[23] Dr.Web CureIt. https://free.drweb.com/cureit/?lng=en.

[24] Microsoft.Microsoft vulnerable driver blocklist. https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/microsoft-recommended-driver-block-rules.

[25] Microsoft. Hypervisor-Protected Code Integrity (HVCI). https://learn.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard.