# Exploiting Reversing (ER) series:
# Article 02 | Windows kernel drivers – part 02

***(a step-by-step vulnerability research series on Win, macOS, hypervisors and browsers)***

**by Alexandre Borges**
**release date: JAN/03/2024 | rev: A.1**

## 0. Quote

***"If you asked me, would I do it again, do I think it's worth it? Yeah, I think it's worth it."***
*(Jeffrey Wigand | "The Insider" movie - 1999)*

## 1. Introduction

Welcome to the second article of **Exploiting Reversing (ER) series**, **a step-by-step vulnerability research series on Windows, macOS, hypervisors and browsers**, where we will review concepts, architecture and practical steps related to vulnerability research. My last articles are listed below:

- **ERS_01:** https://exploitreversing.files.wordpress.com/2024/05/exploit_reversing_01-1.pdf
- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_4:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_5:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_6:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_7:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/

This is an **introductory, step-by-step article**. I chose as a starting point the patch diffing topic to introduce new concepts and ideas, and our objective is to collect and understand facts related to an already existing vulnerability and also to establish a line of comprehension about how the process of patch diffing and respective analysis and reverse engineering works.

In summary, and as I mentioned in my previous article, we will be studying binaries and non-binaries, and this investigation can be composed for the following phases over this text and next ones:

a. getting a better understanding through well-known approaches such as reverse engineering (static or dynamic analysis), auditing (if it is the case) and, who knows, even fuzzing.
b. If it is necessary, explaining fundamental topics to provide a better context for the comprehension of the binary and involved context.
c. applying well-known techniques to get additional information about the target file.
d. eventually, trying to understand the vulnerability and how to trigger it.

Honestly, I would like to keep articles not so long, and try to focus on specific topics by articles, and due to the nature of my real job, I cannot comment on anything about exploitation for now.

## 2. Acknowledgments

I could not write this series (Exploiting and Reversing Series) and the MAS (Malware Analysis Series) without receiving the decisive help from **Ilfak Guilfanov (@ilfak)**, from **Hex-Rays SA (@HexRaysSA)**, because I didn't have an own IDA Pro license, and he kindly provided everything I needed to write this series about reversing and vulnerabilities, and other one that are coming. However, his help did not stop in 2021, and he and **Hex-Rays** have continuously helped until the present moment by providing immediate support for everything I need to keep these public projects. Additionally, **Ilfak** is always truly kind replying to me every single time that I send a message to him. This section, about acknowledgments, can be translated to one word: gratitude. Personally, all messages from **Ilfak** and **Hex-Rays** expressing their trust and praises on my previous articles are one of most motivation to keep writing as well readers who send me even a single message thanking me. Once again: **thank you for everything, Ilfak.**

I have chosen a quote to start each article to subtly show my thinking about life and information security in general, sometimes mirroring the present days and all challenges that have forced me to make a deep reflection over. At the end of day, we should invest in the work that we really love doing, no matter our age, because life is short, and the ahead day is our future. Enjoy the journey!

## 3. Environment Setup

This article demands a lab setup using the following environment:

- **Two virtual machines with Windows 11.** You can download a **virtual machine for VMware, Hyper-V, VirtualBox or Parallels from Microsoft** on: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/. If you already have a valid license for Windows 11, so you can download the **ISO file** from: https://www.microsoft.com/software-download/windows11

- **IDA Pro or IDA Home version (@HexRaysSA):** https://hex-rays.com/ida-pro/ . Of course, readers might use other reverse engineering tools, but I will be using IDA Pro and its decompiler in this article.

- As plugins for **IDA Pro**, install two products to analyzing binary patches:

  - **BinDiff:** https://github.com/google/bindiff/releases/tag/v8
  - **Diaphora:** https://github.com/joxeankoret/diaphora

- Personally, my recommendation is that you install **Windows SDK + Visual Studio + Windows Development Kit (WDK)** in both virtual machines (actually, I also have the same configuration in my host machine):

- **Visual Studio:** https://visualstudio.microsoft.com/downloads/
- **Windows SDK:** https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/
- **Windows Development Kit (WDK):** https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

- For investigating Windows Internals' details or kernel drivers, it is necessary to debug them to be able to understand the flow of information and involved context. Assumes that we have two virtual machines, or one virtual machine is the host and the other one is the target, and execute:

  **On target:**

  - *bcdedit /set {default} DEBUG YES*
  - *bcdedit /dbgsettings net hostip:<host ip> port:50100 key:1.2.3.4*
  - *bcdedit /dbgsettings*
  - *shutdown /r /t 0*

  **On host:**

  - *windbg -k net:port=50100,key=1.2.3.4*
  - *Make sure that symbols are configured:*
    - *File → Symbol File Path: srv*c:\symbols*https://msdl.microsoft.com/download/symbols*
    - *set _NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols (personally, I prefer setting it at Advanced Windows Setting → Environment Variables and creating the _NT_SYMBOL_PATH as explained above)*
  - *Debug → Break*

# 4. References

There is a brief list of references about this SMB vulnerability and even other ones, which are always useful for getting further information and understanding different point of views:

- https://blog.zecops.com/research/smbleedingghost-writeup-chaining-smbleed-cve-2020-1206-with-smbghost/
- https://www.coresecurity.com/core-labs/articles/ms15-083-microsoft-windows-smb-memory-corruption-vulnerability
- https://threatprotect.qualys.com/2020/06/10/microsoft-windows-smbv3-smbleed-vulnerability-cve-2020-1206/

# 5. Gathering initial information

We will be **starting** the analysis the **CVE-2022-35804**, which is described *as "SMB Client and Server Remote Code Execution Vulnerability"* and, Microsoft provides a short list of facts about it:

- **CVE:** 2022-35804
- **Description:** SMB Client and Server Remote Code Execution Vulnerability
- **CVSS:** 3.1 8.8 / 7.7
- **Date Release:** Aug 9, 2022
- **Exploitability:** Exploitation More Likely
- **Workaround:** Install updates or disable compression:

    - **(unauthenticated attackers)** *Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Parameters" DisableCompression -Type DWORD -Value 1 -Force*

    - **(authenticated attackers)** *Set-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters" DisableCompression -Type DWORD -Value 1 -Force*

Links related to this vulnerability follows:

- **Microsoft:** https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-35804
- **Mitre:** https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-35804
- **Rapid7:** https://www.rapid7.com/db/vulnerabilities/msft-cve-2022-35804/

As exposed above, we have a series of clues about what we should look for and the time range. The next key point is to find what are the affected executables, DLLs, or drivers, and perform a binary diffing on it. However, one of the key problems here is really learning what we are looking for (executable, DLL or even a driver) because the notification from Microsoft does not tell us anything about the involved binary or binaries.

Thus, before proceeding and searching for anything vaguely related to SMB inside the patches, I did a short investigation about Microsoft SMB and collected support documentation:

- **SMB Management API:** https://learn.microsoft.com/en-us/previous-versions/windows/desktop/smb/smb-management-api-portal

- **MS-SMB: Server Message Block (SMB) Protocol:** https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/f210069c-7086-4dc2-885e-861d837df688

- **Overview of file sharing using the SMB 3 protocol in Windows Server:** https://learn.microsoft.com/en-us/windows-server/storage/file-server/file-server-smb-overview

SMB is organized in client and server components like other services, and its **SMB client side** is composed by binaries that are hold under **%windir%\system32\Drivers folder**, such as:

- **mrxsmb.sys**
- **mrxsmb10.sys**
- **mrxsmb20.sys**
- **mup.sys**

- **rdbss.sys**
- **smbdirect.sys**

On the other side, the **SMB Server** is composed by binaries that make part of %**windir%\system32\Drivers folder**, such as:

- **srvnet.sys**
- **srv.sys**
- **srv2.sys**
- **smbdirect.sys**
- **srvsvc.dll** (*under %windir%\system32*)

No doubt, SMB is a complex application (or session) level protocol, and issues could come up from the protocol itself and by its integration with TPC/IP. Of course, we are not discussing on the previous SMB v.1 implementation, which should be disabled in any system due its multiple security problems.

Additionally, **SMB v3 works with AES-256**, which provides us with a more secure encrypting scheme when moving any file over a network, and not to mention its support to **S2D (Storage Spaced Direct)**, **QUIC** and other protocols and services.

From these references above and after googling about it, there are other files that potentially we can examine for recent changes, and which might be included within any of the released patches by Microsoft:

- **srvsvc.dll:** file system driver.
- **srv.sys, srv2.sys and srvnet.sys:** drivers associated with SMB server.
- **srvcli.dll:** associated with the SMB client to make communication to the SMB server.
- **mrxsmb.sys, mrxsmb10.sys and mrxsmb20.sys:** drivers related to network redirectors drivers.

Not so surprisingly, some of these files were constantly updated in my host until the end of 2022 (I checked in **C:\Windows\System32** and **C:\Windows\System32\drivers** directories). As the date of the patch CVE was **Aug 09, 2022,** and the patch had already been released, so a good first shot is trying to search for something at that date.

On Windows, the **CIFS (Common Internet File System)** is accountable for determining the network transport protocol used over the communication, and it is related to **SMB (Server Message Block)** protocol. Additionally, the **SMB Server**, which is the **LAN Manager Server**, is responsible for providing file server service.

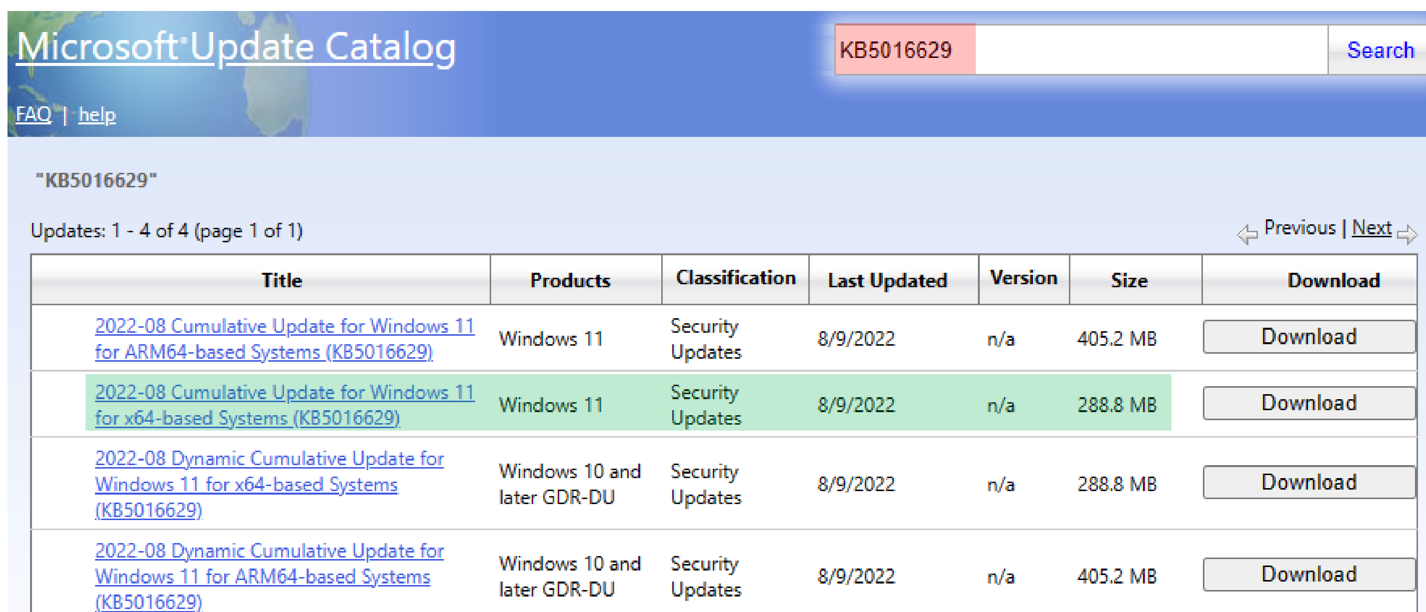
## 6. Investigating patches

Readers should always collect every single piece of information related to the target, including references to public exploits and good write-ups, so a concise list of suggested websites follow below:

- https://www.cvedetails.com/vulnerability-list/
- https://msrc.microsoft.com/update-guide/
- https://cvexploits.io/
- https://www.zerodayinitiative.com/blog

- https://www.catalog.update.microsoft.com/Home.aspx

At the bottom of the Microsoft's page that reports the vulnerability (https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-35804), there is a link to download associated patches from **Microsoft Update Catalog** (https://catalog.update.microsoft.com/Search.aspx?q=KB5016629).




| Title | Products | Classification | Last Updated | Version | Size | Download |
|---|---|---|---|---|---|---|
| 2022-08 Cumulative Update for Windows 11 for ARM64-based Systems (KB5016629) | Windows 11 | Security Updates | 8/9/2022 | n/a | 405.2 MB | Download |
| 2022-08 Cumulative Update for Windows 11 for x64-based Systems (KB5016629) | Windows 11 | Security Updates | 8/9/2022 | n/a | 288.8 MB | Download |
| 2022-08 Dynamic Cumulative Update for Windows 11 for x64-based Systems (KB5016629) | Windows 10 and later GDR-DU | Security Updates | 8/9/2022 | n/a | 288.8 MB | Download |
| 2022-08 Dynamic Cumulative Update for Windows 11 for ARM64-based Systems (KB5016629) | Windows 10 and later GDR-DU | Security Updates | 8/9/2022 | n/a | 405.2 MB | Download |

**[Figure 01] Content of the extracted MSU file**

I have downloaded the **"Cumulative Updated for Windows 11 for x64-based Systems (KB5016629)"**, which is a **Microsoft Standalone Update (.msu)** and, according to this **KB (Knowledge Base)**, it replaces the following patches (truncated list):

- 2021-10 Cumulative Update for Windows 11 for x64-based Systems (KB5006674)
- 2022-07 Cumulative Update for Windows 11 for x64-based Systems (KB5015814)
- 2021-11 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5007262)
- 2021-10 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5006746)
- 2021-11 Cumulative Update for Windows 11 for x64-based Systems (KB5007215)
- 2022-04 Cumulative Update for Windows 11 for x64-based Systems (KB5012592)
- 2022-02 Cumulative Update for Windows 11 for x64-based Systems (KB5010386)
- 2022-03 Cumulative Update for Windows 11 for x64-based Systems (KB5011493)
- 2022-03 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5011563)
- 2022-02 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5010414)
- 2022-07 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5015882)
- 2021-12 Cumulative Update for Windows 11 for x64-based Systems (KB5008215)
- 2022-05 Cumulative Update for Windows 11 for x64-based Systems (KB5013943)
- 2022-06 Cumulative Update Preview for Windows 11 for x64-based Systems (KB5014668)
- 2022-06 Cumulative Update for Windows 11 for x64-based Systems (KB5014697)

Of course, at the time I am drafting this article, other quite a lengthy list of patches has already **replaced this own KB5016629**, but it does not matter because this article aims to explain to you the mechanism and approaches.

A **.msu** file usually contains a few **.cab (Cabinet) files**, and these **.cab files** contain files that participate in the patching process. Therefore, we must extract files from inside of the **.msu file** and this task can be performed by executing the following steps that provide a structured hierarch of directories (folders):

- **mkdir PATCHES_MS**
- **cd PATCHES_MS**
- **mkdir 2022_08**
- **cd 2022_08**
- **mkdir patch**
- **mkdir all**
- **expand.exe -F:* "windows10.0-kb5016629-x64_5c835cd538774e6191bb98343231c095c7918a72.msu" .\all**

So far, it is everything OK, and we have the following content in the **PATCHES_MS** folder:

```
    Directory: C:\Users\Administrator\Desktop\PATCHES_MS\2022_08\all


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          8/4/2022   4:16 PM        4643240 DesktopDeployment.cab
-a----          8/4/2022   1:54 PM        3714685 DesktopDeployment_X86.cab
-a----          8/5/2022   4:37 AM          21898 onepackage.AggregatedMetadata.cab
-a----          8/5/2022   4:33 AM       15002811 SSU-22000.826-x64.cab
-a----          8/5/2022   4:33 AM       76614512 Windows10.0-KB5016629-x64.cab
-a----          8/5/2022   4:33 AM      208548072 Windows10.0-KB5016629-x64.psf
-a----          8/5/2022   4:44 AM         931154 wsusscan.cab
```

**[Figure 02] Content of the extracted MSU file**

As I had mentioned previously, readers need to pay attention that there are several **.cab files**, but one of files has **.psf extension**, which means it is a **Patch Storage File (PSF)**.

This **PSF file** in the output above is exactly the biggest one and, excepting its extension, there is another file exactly with the same name (**Windows10-0-KB5016629-x64**). In previous versions of Windows, Microsoft used only **.cab files** for patching and, as readers will see, its extraction is easy (not necessarily fast). However, this scenario brings a key point to readers: from time to time, Microsoft changes the patch structure and respective organization, and it is always recommended to read all released documents.

For the **Windows 11**, the current patch scheme used by Microsoft is a pair of files with the same name, although different extensions (**.cab** and **.psf**), which also demands a different method to extract the necessary information. For example, to this example, we have the following:

- **Windows10.0-KB5016629-x64.cab:** in general, this **.cab file** contains only metadata. After extracting it, we will find substantial list of files with extensions like **.cat (security catalog), .mum (hold metadata about the referred package)** and **.manifest (manifests).**

- **Windows10.0-KB5016629-x64.psf:** this is the file that we are interested in because it can contain full patch files and delta patches, and the latter is composed of **forward differential (f)** and **reverse differential (r) files**. Of course, it is always much better to have full patch files, but even in case where we do not have it, the differential files can provide us with useful information.

If we were only unpacking a **.cab file** from the previous patch scheme used by Windows, we would find a list of full patch files, differential files, and metadata files, and all of them within the **.cab file**. In the current patch scheme used by Windows 11, **we will find only metadata files in the provided .cab file**.

We can manually extract/deflate the provided **.cab file** (*Windows10.0-KB5016629-x64.cab*) that is inside:

- **expand.exe -F:* ".\all\Windows10.0-KB5016629-x64.cab" .\patches**

Extracting patches takes time, so be patient while the command is not finished. After having finished, the command extracted 42924 files!

Afterwards, create the following folders within the patches directory: **catalogs**, **manifests,** and **mums**. Finally, move files according to their respective extensions.

The simple sequence of commands to perform the described operation follow below:

- **cd patches**
- **mkdir catalogs**
- **mkdir manifests**
- **mkdir mums**
- **mv *.cat catalogs**
- **mv *.manifest manifests**
- **mv *.mum mums**

Readers will have all files inside their respective folders, but they contain only metadata.

To perform the same task, we have the opportunity to use a scripted named **PatchExtract.ps1** (https://gist.githubusercontent.com/wumb0/306f97dc8376c6f53b9f9865f60b4fb5/raw/c93a0c3162b8c4289c96bf23b334d37f7e420150/PatchExtract.ps1), which does the same job of extracting and sorting all files in their respective directories that are created by the script. As an important note, **PatchExtract.ps1** was authored by **Greg Linares (@Laughing_Mantis)**.

Running **PatchExtract.ps1** against the **Windows10.0-KB5016629-x64.cab** file is accomplished by executing the following command on PowerShell prompt:

- **.\PatchExtract.ps1 .\Windows10.0-KB5016629-x64.cab .\auto**

The scripts will offer itself to create the directory (auto), so type "**Y**". Once again, the entire procedure takes time and as unpacked patches are usually big, it is suggested to reserve a reasonable space of the file system. After it finishes, we will find the following structure:

```
    Directory: C:\Users\Administrador\Desktop\PATCHES_MS\2022_08\all\auto


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         2/19/2023   6:38 PM                JUNK
d-----         2/19/2023   6:30 PM                MSIL
d-----         2/19/2023   6:35 PM                PATCH
d-----         2/19/2023   6:30 PM                WOW64
d-----         2/19/2023   6:30 PM                x64
d-----         2/19/2023   6:30 PM                x86
```

**[Figure 03] Content of the extracted MSU fil using PatchExtract.ps1.**

The output above is clear, and **PatchExtract.ps1 script** sorts all files in their appropriate directory, which provides us with a good indication about where to look for what we need.

Returning to our scenario, we also have **the .psf file**, but **PatchExtract.ps1 script** is not able to open and extract it, unfortunately. To handle this situation, we will use another very interesting script named **PSFExtractor**, which can be found and downloaded (both x86 and x64 versions) from https://github.com/Secant1006/PSFExtractor or even https://github.com/Secant1006/PSFExtractor/releases/tag/v3.07.

To use this script, we must **put both .cab and .psf file in the same directory** and ensure that both files have the same name, excepting its extension. To make the procedure simple, I copied the extracted file into the same "all" folder and executed the following command on **PowerShell**:

- **.\PSFExtractor.exe .\Windows10.0-KB5016629-x64.cab**

The script has worked very well, as expected, but it does not sort files and folders to an appropriate directory. Thus, we must do it manually as shown below (by the way, from this point onward I will be using commands from **Cygwin** on Windows if it necessary: https://www.cygwin.com/):

- **mkdir manifests**
- **mv *.manifest manifests/**
- **mkdir mums**
- **mv *.mum mums/**
- **mkdir catalogs**
- **mv *.cat catalogs**
- **mkdir x64**
- **mv amd64_* x64/**
- **mkdir msilfolder**
- **mv msil_* msilfolder/**
- **mkdir wow64folder**
- **mv wow64* wow64folder**
- **mkdir x86folder**
- **mv x86_* x86folder**

Certainly, we could write a simple script to accomplish this time-consuming task, but I have been trying to show a step-by-step procedure about how this can be done. At end, readers will have something like:

```
02/19/2023  01:09 PM    <DIR>          .
02/18/2023  02:23 PM    <DIR>          ..
02/19/2023  11:36 AM    <DIR>          catalogs
08/04/2022  10:36 PM        13,612,858 express.psf.cix.xml
08/04/2022  10:29 PM         3,787,872 historycix.cab
02/18/2023  03:09 PM    <DIR>          manifests
02/19/2023  01:03 PM    <DIR>          msilfolder
02/18/2023  03:18 PM    <DIR>          mums
02/19/2023  01:07 PM    <DIR>          wow64folder
02/19/2023  02:38 PM    <DIR>          x64
02/19/2023  01:09 PM    <DIR>          x86folder
               2 File(s)     17,400,730 bytes
```

**[Figure 04] Sorting files to different folders**

Once we have done it, change **to the x64 directory** and list anything related to **SMB**, but exclude eventual resources, and we will have:

```
C:\Users\Administrator\Desktop\PATCHES_MS\2022_08\all\Windows10.0-KB5016629-x64\x64>ls -d *smb* | grep
-v resources
amd64_microsoft-hyper-v-vstack-vsmb_31bf3856ad364e35_10.0.22000.41_none_115191547f919d8f
amd64_microsoft-windows-smb10-minirdr_31bf3856ad364e35_10.0.22000.778_none_83dbba7fd0bd72a4
amd64_microsoft-windows-smb20-minirdr_31bf3856ad364e35_10.0.22000.778_none_861224920f14a615
amd64_microsoft-windows-smbhelperclasses_31bf3856ad364e35_10.0.22000.653_none_e584f0f65b6592bd
amd64_microsoft-windows-smbminirdr_31bf3856ad364e35_10.0.22000.856_none_7adc3d55d65dc6d9
amd64_microsoft-windows-smbserver-apis_31bf3856ad364e35_10.0.22000.778_none_cb5f016008df339e
amd64_microsoft-windows-smbserver-common_31bf3856ad364e35_10.0.22000.856_none_ff20fe5495676e6d
amd64_microsoft-windows-smbserver-netapi_31bf3856ad364e35_10.0.22000.613_none_3c197e63d5ae2d5a
amd64_microsoft-windows-smbserver-powershell_31bf3856ad364e35_10.0.22000.778_none_5842f2e6467b7a0e
amd64_microsoft-windows-smbserver-v1_31bf3856ad364e35_10.0.22000.778_none_d52e98517bc25cf4
amd64_microsoft-windows-smbserver-v2_31bf3856ad364e35_10.0.22000.832_none_d549083f7baf6e3e
amd64_microsoft-windows-smbserver_31bf3856ad364e35_10.0.22000.795_none_f4137739703d2e8d
amd64_microsoft-windows-smbwitnessservice-apis_31bf3856ad364e35_10.0.22000.653_none_9f634bef6f2cc85b
```

**[Figure 05] Listing SMB related directories**

We confirmed that reported issues related to SMB server. If readers inspect these folders, they will find only **forward differentials** (inside **f folders**). However, examining inside them, a series of differential files with extensions such as .dll, .sys and other ones, will come up:

```
C:\Users\Administrator\Desktop\PATCHES_MS\2022_08\all\Windows10.0-KB5016629-x64\x64>ls -d *smb* | grep
-v resources | xargs ls -lR | grep "dll\|sys"
-rwxrwx---+ 1 Administrators None 3720 Aug  4  2022 vmsmb.dll
-rwxrwx---+ 1 Administrators None 1998 Aug  4  2022 vmusrv.dll
-rwxrwx---+ 1 Administrators None 4611 Aug  4  2022 mrxsmb10.sys
-rwxrwx---+ 1 Administrators None 27157 Aug  4  2022 mrxsmb20.sys
-rwxrwx---+ 1 Administrators None 228 Aug  4  2022 smbhelperclass.dll
-rwxrwx---+ 1 Administrators None 48097 Aug  4  2022 mrxsmb.sys
-rwxrwx---+ 1 Administrators None 19264 Aug  4  2022 smbwmiv2.dll
-rwxrwx---+ 1 Administrators None 23207 Aug  4  2022 srvnet.sys
-rwxrwx---+ 1 Administrators None 5765 Aug  4  2022 srvcli.dll
-rwxrwx---+ 1 Administrators None 6170 Aug  4  2022 srv.sys
-rwxrwx---+ 1 Administrators None 29854 Aug  4  2022 srv2.sys
-rwxrwx---+ 1 Administrators None 3820 Aug  4  2022 srvsvc.dll
-rwxrwx---+ 1 Administrators None  879 Aug  4  2022 sscore.dll
-rwxrwx---+ 1 Administrators None 160 Aug  4  2022 witnesswmiv2provider.dll
```

**[Figure 06] Searching for files potentially involved.**

That is great! We made a small progress and found the following files:

- **mrxsmb10.sys:** network kernel-mode driver, as well known as Microsoft Server Message Block redirector, which is responsible for providing network redirector functions. Number 10 is a reference to SMB version 1.0.
- **mrxsmb20.sys:** network kernel-mode driver, also known as Microsoft Server Message Block redirector, which is responsible for providing network redirector functions. Number 20 also is a reference to SMB version 2.0.
- **mrxsmb.sys:** network kernel-mode driver, as well known as Microsoft Server Message Block redirector, which is responsible for providing network redirector functions. Eventually, it is related to the current version of SMB (version 3.0).
- **smbhelperclass.dll:** as the name indicates, it is an auxiliary SMB file.
- **smbwmiv2.dll:** this DLL, as well known as WMIv2 Provider for SMB File Server/Client, is related to SMB APIs.
- **srvcli.dll:** this DLL, which is known as Server Service Client DLL, and it is related to Net API.
- **srv.sys:** this driver is associated to SMB server version 1.

- **srv2.sys:** this driver is associated to the SMB server version 2.
- This driver is associated with the SMB server and, according to description, it is a common component.
- **srvsvc.dll:** this DLL is also related to SMB server.
- **sscore.dll:** this DLL, as known as System Restore Core Library, is related to SMB server.

Right now, I am not considering other types and formats of files such as **ps1xml, psm1** and **cdxml** related to PowerShell interaction with SMB Server, but they also exist within the patch:

```
amd64_microsoft-windows-smbserver-powershell_31bf3856ad364e35_10.0.22000.778_none_5842f2e6467b7a0e
/f:
total 23
-rwxrwx---+ 1 Administrators None 901 May 11  2022 smb.format.ps1xml
-rwxrwx---+ 1 Administrators None  52 May 11  2022 smb.types.ps1xml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbbandwidthlimit.cdxml
-rwxrwx---+ 1 Administrators None 336 May 11  2022 smbclientconfiguration.cdxml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbclientnetworkinterface.cdxml
-rwxrwx---+ 1 Administrators None  50 May 11  2022 smbcomponent.cdxml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbconnection.cdxml
-rwxrwx---+ 1 Administrators None  51 Jun 10  2021 smbglobalmapping.cdxml
-rwxrwx---+ 1 Administrators None  52 May 11  2022 smbmapping.cdxml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbmultichannelconnection.cdxml
-rwxrwx---+ 1 Administrators None  51 Jun 10  2021 smbmultichannelconstraint.cdxml
-rwxrwx---+ 1 Administrators None  51 Jun 10  2021 smbopenfile.cdxml
-rwxrwx---+ 1 Administrators None  52 May 11  2022 smbscriptmodule.psm1
-rwxrwx---+ 1 Administrators None  52 May 11  2022 smbservercertificatemapping.cdxml
-rwxrwx---+ 1 Administrators None 386 May 11  2022 smbserverconfiguration.cdxml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbservernetworkinterface.cdxml
-rwxrwx---+ 1 Administrators None  51 Jun 10  2021 smbsession.cdxml
-rwxrwx---+ 1 Administrators None  52 May 11  2022 smbshare.cdxml
-rwxrwx---+ 1 Administrators None  50 Jun 10  2021 smbshare.format.helper.psm1
-rwxrwx---+ 1 Administrators None  82 May 11  2022 smbshare.psd1
```

**[Figure 07] Searching for files potentially involved.**

Proceeding with our analysis, we should remember that we do not have the binary itself, but only a set of forward differential patches that must be applied on the last delivered file with the same name and as expected, there is also scripts to automatize our task.

If readers want to get a better comprehension about differential patches, I suggest to read this document written by Microsoft: https://learn.microsoft.com/en-us/windows/deployment/update/psfxwhitepaper.

We have the names of the drivers and DLLs being patched and related to the SMB server flaw, so the next step is choosing a starting point, and we will be picking up **srv2.sys** that is a first potential candidate. However, there will certainly be other files such as **srv.sys** and **srvcli.dll** might be chosen too.

At this point, the next step would be search for previous **Microsoft Accumulative Patches** until finding one that have the full version of the same files being investigated, applying the differential patch, and getting the updated version, and finally diffing them. Nonetheless, there is a better and reliable way to get the same result, which is much handier and saves us from a time-consuming and error-prone task.

We will be using a fantastic service named **Winbindex** (https://winbindex.m417z.com/), which provides us with the last versions of Windows binaries and associated information to each file. Thus, searching for **srv2.sys** on **Winbindex**, we receive the following results:

srv2.sys - Winbindex

Smb 2.0 Server driver

Show 10 entries Search:

| SHA256 | Windows | Update | File arch | File version | File size | Extra | Download |
|---|---|---|---|---|---|---|---|
| f6a0b3... | Windows 10 1507 | KB5022858 | x64 | ??? | 655.5 KB | Show | Download |
| 855ee5... | Windows 10 1607 | KB5022838 | x64 | 10.0.14393.5717 | 696 KB | Show | Download |
| 58651d... | Windows 11 22H2 | KB5022360 (+1) | x64 | 10.0.22621.1194 | 844 KB | Show | Download |
| 33f607... | Windows 11 21H2 | KB5019274 (+2) | x64 | 10.0.22000.1516 | 824 KB | Show | Download |
| 14ad15... | Windows 10 1607 | KB5021235 (+1) | x64 | 10.0.14393.5582 | 696.5 KB | Show | Download |
| 711a7a... | Windows 10 1607 | KB5019964 (+1) | x64 | 10.0.14393.5501 | 696.5 KB | Show | Download |
| 6a5f62... | Windows 11 21H2 | KB5018483 (+4) | x64 | 10.0.22000.1165 | 828 KB | Show | Download |
| a3f8bc... | Windows 11 22H2 | KB5017389 (+7) | x64 | 10.0.22621.608 | 848 KB | Show | Download |
| b04a4c... | Windows 11 22H2 | KB5019311 | x64 | 10.0.22621.457 | 848 KB | Show | Download |
| bec211... | Windows 11 21H2 | KB5017383 (+2) | x64 | 10.0.22000.1042 | 832 KB | Show | Download |
| SHA256 | Windows | Update | File arch | File version | File size | Extra | Download |

Showing 1 to 10 of 128 entries Previous 1 2 3 4 5 ... 13 Next

**[Figure 08] Searching for srv2.sys on Winbindex.**

There is a considerable list of versions of the file (**srv2.sys**) being searched for and, initially, it could be difficult finding the correct one, but the filter mechanisms help us a lot and we already have existing information to refine our search as the **KB** (**KB5016629**) and **Date Release** (Aug 9, 2022).

Show 10 entries Search:

| SHA256 | Windows | 2022-08-... | File arch | File version | File size | Extra | Download |
|---|---|---|---|---|---|---|---|
| d76708... | Windows 11 21H2 | KB5015882 (+1) | x64 | 10.0.22000.832 | 832 KB | Show | Download |
| SHA256 | Windows | Update | File arch | File version | File size | Extra | Download |

Showing 1 to 1 of 1 entries (filtered from 128 total entries) Previous 1 Next

**[Figure 09] Searching for srv2.sys on Winbindex with filters.**

Curiously, there is an extensive list of updates to these drivers after **Aug 09, 2022** (at the time I am drafting this article, there are six new versions, at least), which we can investigate in a future article.

Even most important for us, there is a previous version of this file, which KB is **KB5015882** and was released **on July 21, 2022**. Actually, this driver has suffered multiple updates throughout the year, and we cannot determine precisely when the vulnerability has been introduced without performing a digression.

To demonstrate the next four versions of this driver and the previous one:

| SHA256 | Windows 1... | Update | File arch | File version | File size | Extra | Download |
|---|---|---|---|---|---|---|---|
| 33f607... | Windows 11 21H2 | KB5019274 (+2) | x64 | 10.0.22000.1516 | 824 KB | Show | Download |
| 6a5f62... | Windows 11 21H2 | KB5018483 (+4) | x64 | 10.0.22000.1165 | 828 KB | Show | Download |
| bec211... | Windows 11 21H2 | KB5017383 (+2) | x64 | 10.0.22000.1042 | 832 KB | Show | Download |
| 080406... | Windows 11 21H2 | KB5016691 (+1) | x64 | 10.0.22000.918 | 832 KB | Show | Download |
| d76708... | Windows 11 21H2 | KB5015882 (+1) | x64 | 10.0.22000.832 | 832 KB | Show | Download |
| 62b71c... | Windows 11 21H2 | KB5014668 (+1) | x64 | 10.0.22000.778 | 832 KB | Show | Download |

**[Figure 10] Next four updated versions of this driver, and one released one month before our target.**

Additional information about the driver released on **August 09, 2022** (our target) follows:

- **Name:** srv2.sys
- **SHA256:** d767085d244cd6bf0ea4b9070b99f054dcfe8714c03e75c892c1ecd6c122df2e
- **File version:** 10.0.22000.832
- **Date:** AUG 09, 2022
- **KB:** KB5016629
- **Win. Version:** Windows 11 21H2

The driver can be easily downloaded by clicking on the **Download** button, and it is saved as **.blob**, so it is enough to rename it to anything you want to. Examining a bit more, the previous version of the same driver (**srv2.sys**), from **July 12, 2022** (as shown in the **Figure 10** above) has the following information:

- **Name:** srv2.sys
- **Hash:** 1eba7518043a96d20c6f66f5e138cbce6e5d1a592bf3426737327b63c2f87e96
- **File Version:** 10.0.22000.778
- **Date:** JULY 12, 2022
- **KB:** KB5015814
- **Win. Version:** Windows 11 21H2

We must download them to compare both driver's version using **BinDiff** tool and **Diaphora** and, eventually, to try to understand all fixes that have been applied to the driver. It is quite important to highlight that we are starting with the **srv2.sys driver** because it is only an attempt to get some grasp of the issue. At this point, we do not even know whether the flaw is present in this driver itself, but it is still a good exercise.

## 7. Binary Diffing using BinDiff tool

Binary diffing is an excellent approach to understanding fixes and vulnerabilities, and also finding new vulnerabilities at the same fixed binary. As we already have two first candidates to diffing, it could be interesting to look at the code and examine changes from one version to another. First, we will try to use **BinDiff (**https://zynamics.com/software.html**)**, which is an oustanding tool and plugin for IDA Pro. To use **BinDiff**, execute:

- Open the first binary in IDA Pro (for example, **srv2_sys_AUG_2022.blob**, which is the name given to identify the **srv2.sys** from AUG)
- Once IDA has finished analyzing the code, you can close the database.
- Open the second binary using IDA Pro (**srv2_sys_JUL_2022.blob**).
- Once the IDA Pro has analyzed the code, you can close the database.
- Open the first saved database in IDA Pro (from **srv2_sys_AUG_2022.blob** binary)
- Go to **File → BinDiff** and pick up the second saved database (from **srv2_sys_JUL_2022.blob** binary)

The first results from the comparison between databases can be found on **Matched Functions**, sort functions by **Similarity** and we will get the following results:

| Similarity | Confidence | Change | EA Primary | Name Primary | EA Secondary | Name Secondary |
|---|---|---|---|---|---|---|
| 0.98 | 0.99 | GI----- | 00000001C004E3F0 | Smb2ValidateWrite | 00000001C004E3F0 | Smb2ValidateWrite |
| 0.87 | 0.98 | GI-JE-C | 00000001C005D6F0 | Smb2QueryFileNormalizedName | 00000001C005D6F0 | Smb2QueryFileNormalizedName |
| 1.00 | 0.99 | ------- | 00000001C0001008 | RfsHashTableEnumerate | 00000001C0001008 | RfsHashTableEnumerate |
| 1.00 | 0.99 | ------- | 00000001C00010AC | Smb2LogNetNameChange_i | 00000001C00010AC | Smb2LogNetNameChange_i |
| 1.00 | 0.99 | ------- | 00000001C0001140 | Srv2UpdateNetnameTable | 00000001C0001140 | Srv2UpdateNetnameTable |
| 1.00 | 0.99 | ------- | 00000001C000132C | Smb2DereferenceNetname | 00000001C000132C | Smb2DereferenceNetname |
| 1.00 | 0.99 | ------- | 00000001C000135C | RfsHashTableLookup | 00000001C000135C | RfsHashTableLookup |
| 1.00 | 0.99 | ------- | 00000001C0001440 | RfsHashBucketReleaseLockShared | 00000001C0001440 | RfsHashBucketReleaseLockShared |
| 1.00 | 0.99 | ------- | 00000001C0001468 | RfsHashBucketAcquireLockShared | 00000001C0001468 | RfsHashBucketAcquireLockShared |
| 1.00 | 0.99 | ------- | 00000001C0001498 | RfsHashGenerateKey | 00000001C0001498 | RfsHashGenerateKey |
| 1.00 | 0.99 | ------- | 00000001C00014D0 | Smb2NetnameEqualKey | 00000001C00014D0 | Smb2NetnameEqualKey |
| 1.00 | 0.99 | ------- | 00000001C0001540 | RfsHashTableInsertEx | 00000001C0001540 | RfsHashTableInsertEx |
| 1.00 | 0.99 | ------- | 00000001C0001770 | RfsHashTableRemove | 00000001C0001770 | RfsHashTableRemove |
| 1.00 | 0.99 | ------- | 00000001C0001910 | Srv2DereferenceEndpoint | 00000001C0001910 | Srv2DereferenceEndpoint |
| 1.00 | 0.99 | ------- | 00000001C0001934 | Srv2RemoveConnectionFromEndpointList | 00000001C0001934 | Srv2RemoveConnectionFromEndpoin... |
| 1.00 | 0.99 | ------- | 00000001C0001A58 | RfsTable64Cleanup | 00000001C0001A58 | RfsTable64Cleanup |

**[Figure 11] Binary Diffing of two different versions of srv2.sys**

After having run **BinDiff**, readers will see few columns:

- **Similarity:** how similar two matches functions are.
- **Confidence:** indicates the confidence of the Similarity score.
- **Change:** this letter highlights differences between matched functions, such as:
  - **G (Graph):** there is structural difference between changes in functions.
  - **I (Instruction):** either number of instructions or one mnemonic (at least) has changed.
  - **J (Jump):** indicates a branch inversion.
  - **E (Entrypoint):** the basic blocks of the entry point have not been matched.
  - **L (Loop):** the number of loops has changed.
  - **C (Call):** one of the call targets has not been matched, at least.

- **EA Primary:** the effective address of the function in the current IDA database.
- **Name Primary:** the name of the function in the current IDA database.
- **EA Secondary:** the effective address of the function in the second loaded IDA database.

- **Name Secondary:** the name of the function in the second loaded IDA database.

There are other columns, but these ones are the most important right now. According to our results, we have the following scenario:

- **Smb2QueryFileNormalizedName:** GI-JE-C
- **Smb2ValidateWrite:** GI----

These functions are our starting point to perform an investigation, but we should underscore that we are only focused on **srv2.sys** and, eventually, we will need to expand the scope of our analysis later.

As readers can notice, both functions have few changes between versions, but there are a lengthy number of cases that only a handful of instructions can represent a fix or, analyzed from the opposite angle, they might be the cause of vulnerability.

For example, readers might remember about **CVE-2020-1350**, as known as **SIGRed**, which was a vulnerability in the **dns.exe file**:

```
00000001400AED2A    mov     rbp, rax
00000001400AED2D    test    rax, rax
00000001400AED30    jz      short loc_1400AEDAF
00000001400AED32    sub     rdi, rax
00000001400AED35    cmp     rdi, 0FFFFh
00000001400AED3C    ja      short loc_1400AEDAF
00000001400AED3E    movzx   eax, [rsp+168h+var_138]
00000001400AED43    add     ax, 14h
00000001400AED47    cmp     ax, 12h
00000001400AED4B    jb      short loc_1400AEDAF
00000001400AED4D    lea     ecx, [rax+rdi]
00000001400AED50    cmp     cx, ax
00000001400AED53    jb      short loc_1400AEDAF
00000001400AED55    xor     edx, edx
00000001400AED57    xor     r8d, r8d
00000001400AED5A    call    RR_AllocateEx
00000001400AED5F    mov     rsi, rax
00000001400AED62    test    rax, rax
00000001400AED65    jz      short loc_1400AEDAF
```

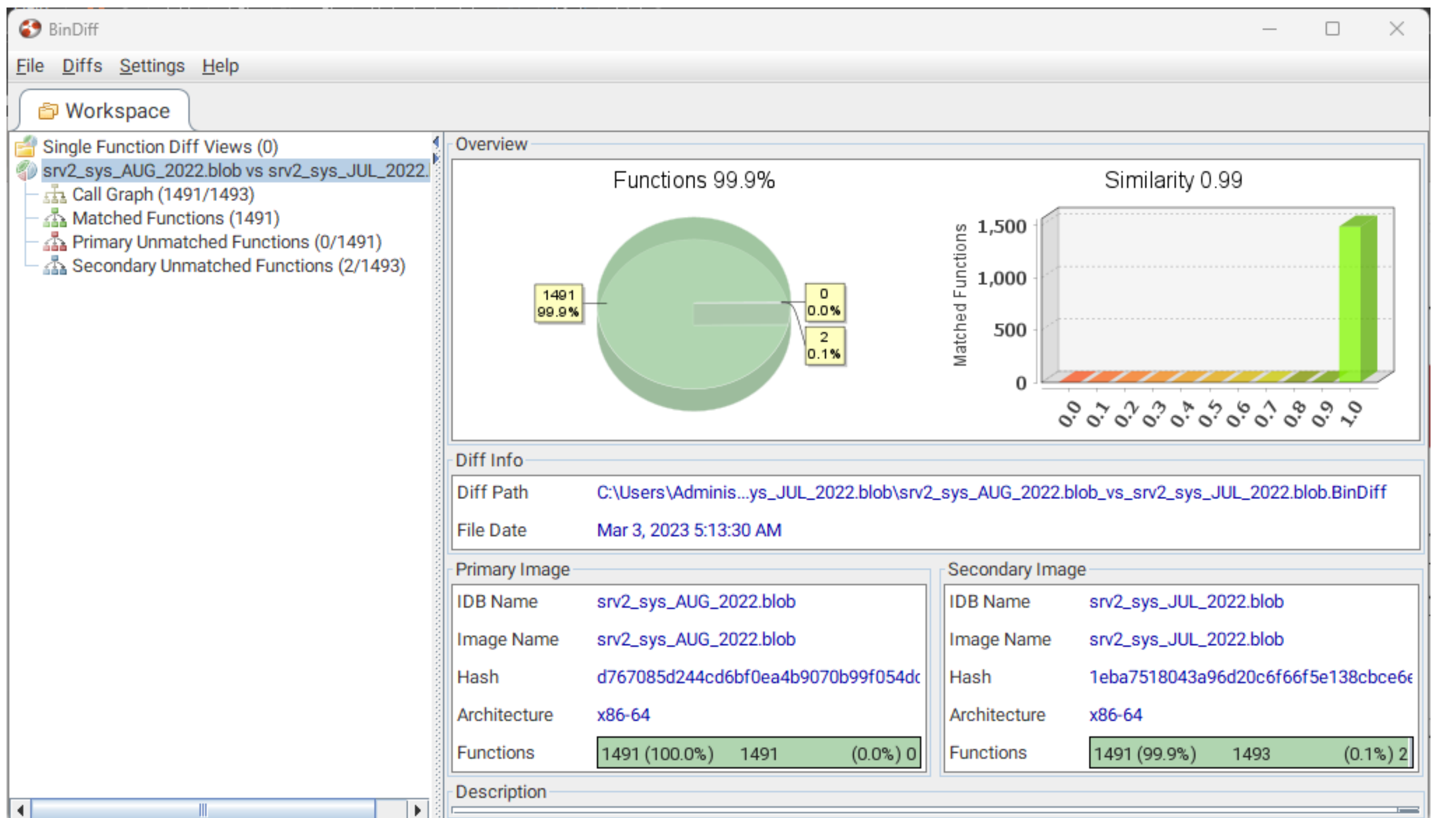
(CVE-2020-1350) The disassembly shows two highlighted instructions that were introduced by Microsoft into SigWireRead routine. The vulnerability was caused by an integer overflow that would be explored over a memcpy routine, so causing a heap buffer overflow too.

**[Figure 12] Fix introduced by Microsoft to fix CVE-2020-1350 at past.**

There is a series of blogs and articles commenting about this vulnerability, which was critical, "*wormable*" and treated as a RCE (remote code execution) issue. If readers are interested in reading about such old and important vulnerability, a brief list of websites might be useful:

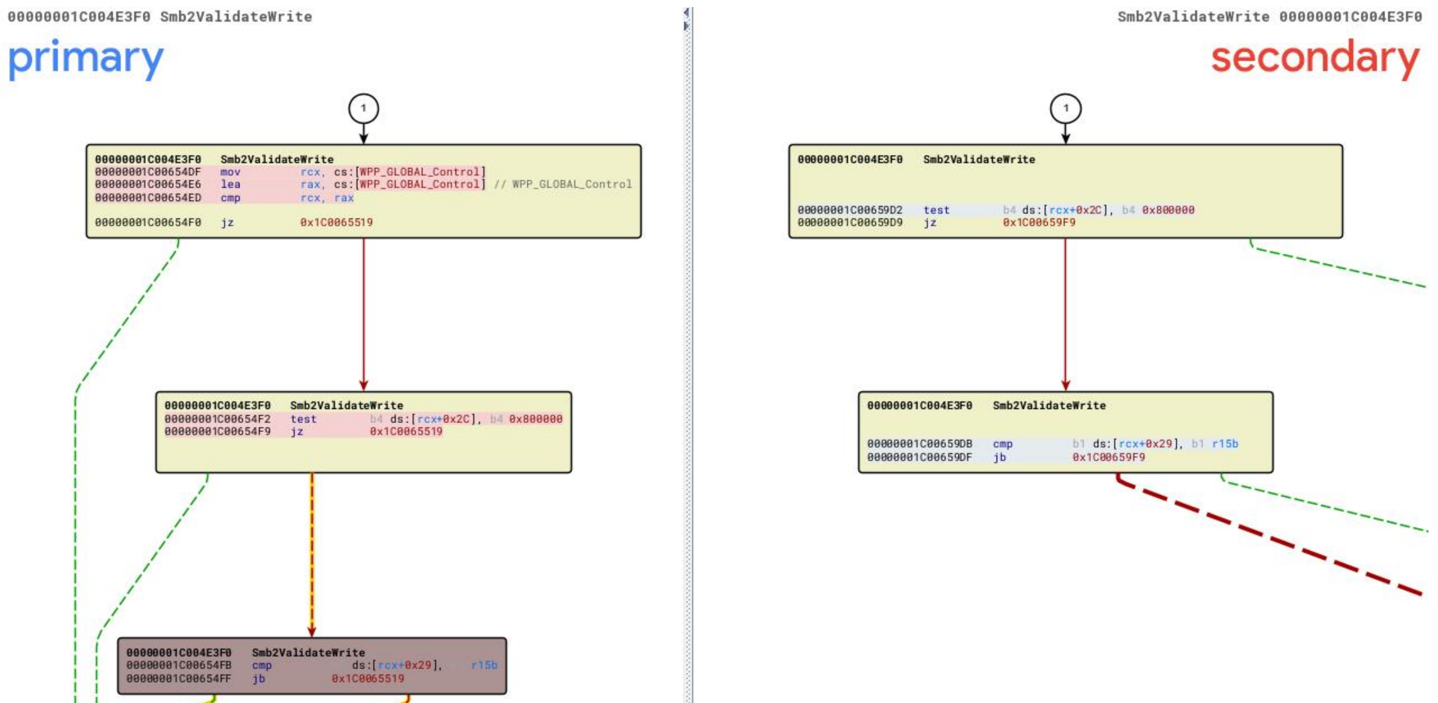
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-1350
- https://www.rapid7.com/blog/post/2020/07/14/windows-dns-server-remote-code-execution-vulnerability-cve-2020-1350-what-you-need-to-know/
- https://support.microsoft.com/en-gb/topic/kb4569509-guidance-for-dns-server-vulnerability-cve-2020-1350-6bdf3ae7-1961-2d25-7244-cce61b056569

Returning to our preliminary investigation, save the result of the comparison by going to **Edit → Plugins → BinDiff** and choose **Save Results**. Afterwards, open **BinDiff application (out of IDA Pro)** and load the saved results. The output will be something like:




**[Figure 13] BinDiff application**

Checking the graph related **Smb2ValidateWrite routine** from both binaries, we have:




**[Figure 14] BinDiff: highlighting differences**

There are important and interesting differences between two binaries on the **Smb2ValidateWrite routine** and we will return to study them in next sections. Before proceeding, I would like to show **Diaphora**.

## 8. Binary Diffing using Diaphora tool

**Diaphora** (https://github.com/joxeankoret/diaphora) is an great diffing tool supported by all recent versions of IDA Pro (including the current version 8.3), and offers a series of useful features to analyze binary diffing, and one of them is exactly the possibility to work on the **IDA pseudo-code**. According to its GitHub, a small list of capabilities of Diaphora are:

- diffing assembler and pseudo-code-based heuristics.
- diffing pseudo-code and microcode support
- parallel diffing
- similarity ratio calculation
- batch automation

We can clone or download its last version (https://github.com/joxeankoret/diaphora/releases/tag/3.1.):

- **git clone https://github.com/joxeankoret/diaphora**

After cloning it, you have the following directory structure:

```
C:\github\diaphora>dir
 Volume in drive C is Windows
 Volume Serial Number is D45E-7379

 Directory of C:\github\diaphora

09/19/2023  03:10 PM    <DIR>          .
09/10/2023  03:07 PM    <DIR>          ..
01/31/2023  10:12 PM                 7 .gitignore
09/19/2023  03:10 PM    <DIR>          codecut
09/19/2023  03:10 PM    <DIR>          db_support
09/19/2023  03:10 PM           112,521 diaphora.py
09/19/2023  03:10 PM             7,212 diaphora_config.py
09/19/2023  03:10 PM            40,053 diaphora_heuristics.py
09/19/2023  03:10 PM           122,104 diaphora_ida.py
09/19/2023  03:10 PM               997 diaphora_import.py
09/19/2023  03:10 PM               985 diaphora_load.py
01/31/2023  10:12 PM             1,053 diaphora_load_and_import.py
09/19/2023  03:10 PM    <DIR>          doc
01/31/2023  10:12 PM    <DIR>          hooks
09/19/2023  03:10 PM    <DIR>          jkutils
01/31/2023  10:12 PM            35,184 LICENSE
01/31/2023  10:12 PM    <DIR>          others
01/31/2023  10:12 PM    <DIR>          pygments
09/19/2023  03:10 PM             5,378 README.md
09/19/2023  03:10 PM    <DIR>          scripts
09/19/2023  03:10 PM    <DIR>          tester
              10 File(s)        325,494 bytes
              11 Dir(s)  347,103,657,984 bytes free
```

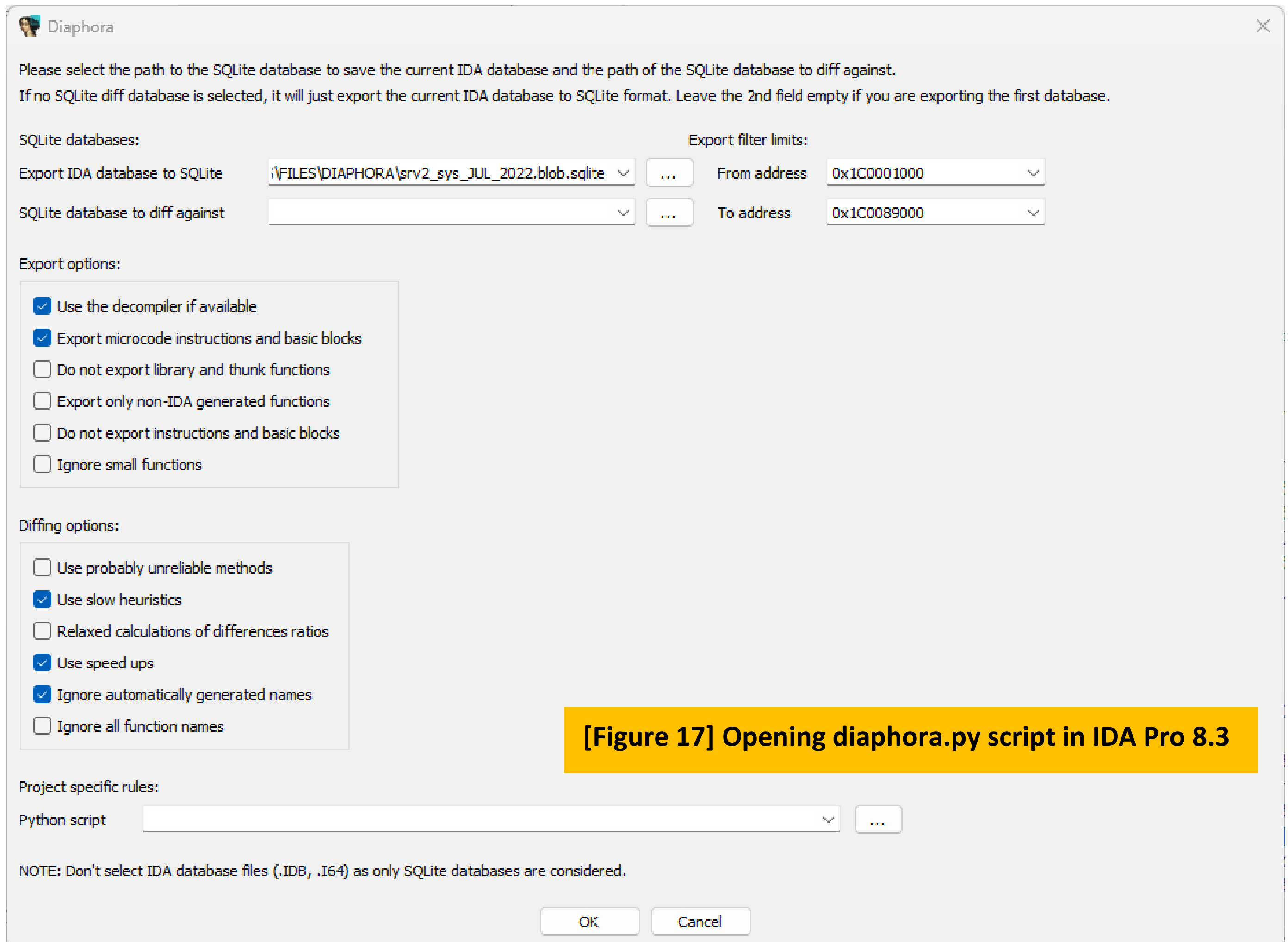
**[Figure 15] Diaphora: directory organization**

Differently from **BinDiff**, **Diaphora** must be launched and used through command line. Only to make the procedure cleaner, I have copied both **srv2.sys files** and its respective **.idb files** (from IDA Pro) to a separate folder (in my case, ***C:\Users\Administrador\Desktop\EXPLOITING_REVERSING\FILES\DIAPHORA***) and my initial structure is:

```
C:\Users\Administrador\Desktop\EXPLOITING_REVERSING\FILES\DIAPHORA>dir
 Volume in drive C is Windows
 Volume Serial Number is D45E-7379

 Directory of C:\Users\Administrador\Desktop\EXPLOITING_REVERSING\FILES\DIAPHORA

09/19/2023  03:42 PM    <DIR>          .
09/19/2023  03:39 PM    <DIR>          ..
02/23/2023  06:34 PM           851,968 srv2_sys_AUG_2022.blob
03/09/2023  03:30 AM        31,277,123 srv2_sys_AUG_2022.blob.i64
02/23/2023  05:48 PM           851,968 srv2_sys_JUL_2022.blob
02/28/2023  02:56 PM        30,490,691 srv2_sys_JUL_2022.blob.i64
               4 File(s)     63,471,750 bytes
               2 Dir(s)  347,012,419,584 bytes free

C:\Users\Administrador\Desktop\EXPLOITING_REVERSING\FILES\DIAPHORA>file *
srv2_sys_AUG_2022.blob:     PE32+ executable (native) x86-64, for MS Windows, 11 sections
srv2_sys_AUG_2022.blob.i64: data
srv2_sys_JUL_2022.blob:     PE32+ executable (native) x86-64, for MS Windows, 11 sections
srv2_sys_JUL_2022.blob.i64: data
```

**[Figure 16] Separate folder containing binaries and respective IDA Pro databases (.idb files)**

We have two **.idb files**: **srv2_sys_AUG_2022.blob.i64**, which is associated with **srv2.sys** file from AUG 2022, and the second one (**srv2_sys_JUL_2022.blob.i64**), which is associated with **srv2.sys** file from JUL 2022. Open the older one, go **to File | Script File (ALT+F7)** and open **diaphora.py file**, as shown below:



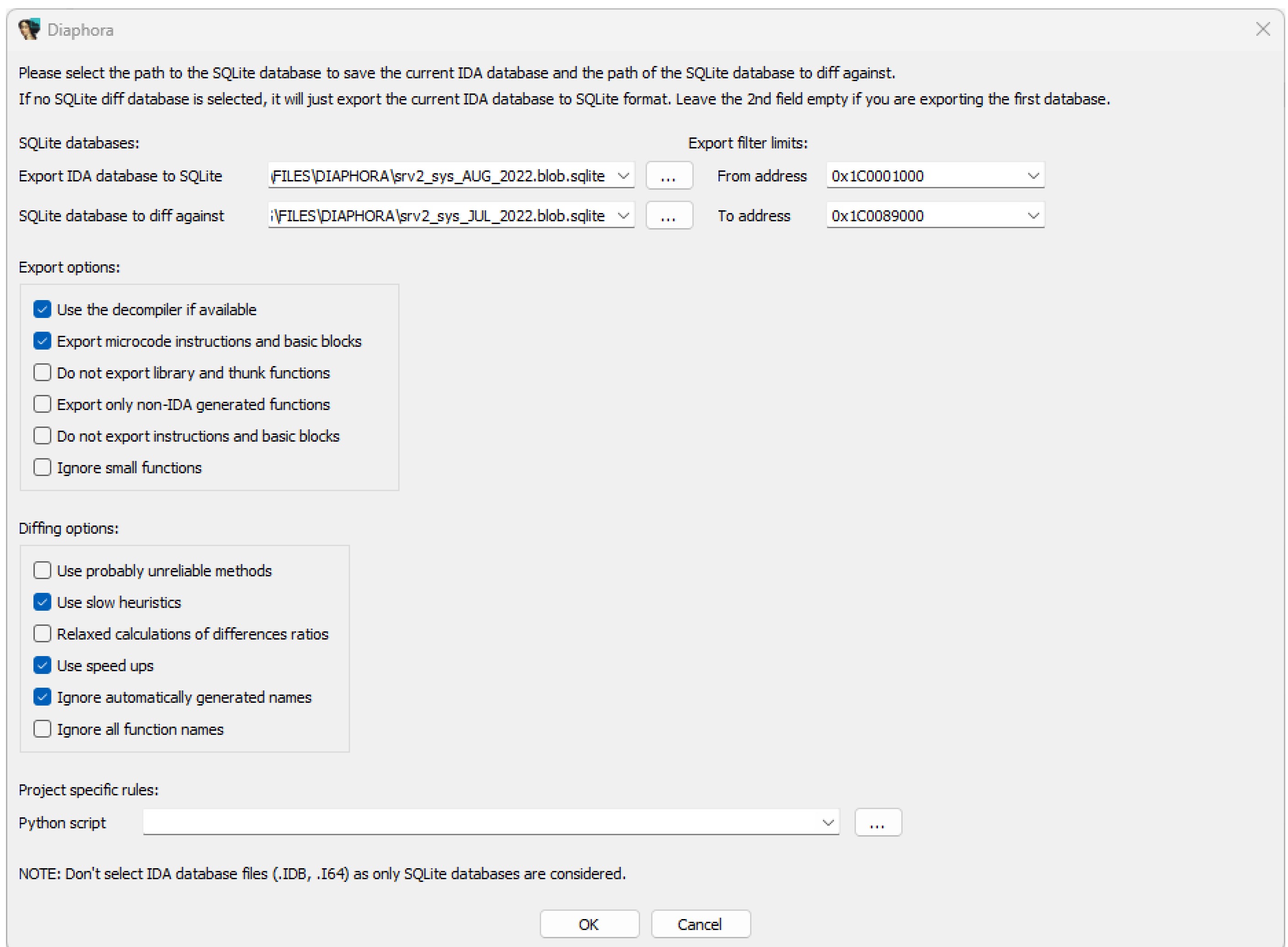

**[Figure 17] Opening diaphora.py script in IDA Pro 8.3**

Right now, the following options matter for us:

- **Export IDA database to SQLite:** it indicates the folder where Diaphora will store IDA database information as an exported file in SQLite. I kept the same folder of IDA Pro database.
- **Use the decompiler if available:** this option is one of notable features of Diaphora, and as readers have the decompiler, certainly you want to use it.
- **Use slow heuristics:** this option produces better results, but it might be slow with large databases.

Click on the **OK** button and wait for Diaphora while it exports **.idb** information into an **SQLite database**. In my case, it took over five minutes to finish the export activity. Close the current **.idb file** and save it.

Open the newer .idb file (**srv2_sys_AUG_2022.blob.i64**) and repeat the same steps by going to **File | Script File (ALT+F7)** and open **diaphora.py file**, as shown below:




**[Figure 18] Opening diaphora.py script in IDA Pro 8.3 for the second .idb database**

This time **Diaphora** will compare the JULY's database to AUGUST's database, which is specified in the field named **SQLite database to diff against**. In the **Output windows** two pieces of information came up: it took over 05 minutes to generate performing the diffing process and there was a matching of 99.81% of the binary functions, which are only five partial matches, one interesting match and two unmatched functions.

Any closed tab can be reopened by pressing **F3** or even going to **Edit | Plugins | Diaphora – Show results**.

The result produced by **Diaphora** is shown below, where **primary is the newer version** of the binary (AUG/2022) and **secondary is the older version** (JULY/2022):

Interesting matches

| Line | Address | Name |
|---|---|---|
| 00000 | 1c005d6f0 | Smb2QueryFileNormalizedName |

Best matches

| Line | Address | Name |
|---|---|---|
| 00000 | 1c0001104 | Smb2ReferenceShare |
| 00001 | 1c0001120 | Smb2ReferenceSession |
| 00002 | 1c0001130 | Srv2ReferenceConnection |
| 00003 | 1c000132c | Smb2DereferenceNetname |
| 00004 | 1c000135c | RfsHashTableLookup |
| 00005 | 1c0001498 | RfsHashGenerateKey |
| 00006 | 1c00014b8 | Srv2ReferenceInstance |
| 00007 | 1c00014c4 | RfsHashTableEntryCount |
| 00008 | 1c0001500 | Smb2GetAuthenticatedConnection... |
| 00009 | 1c0001510 | Smb2GetHonorServerCompressionA... |
| 00010 | 1c0001520 | Smb2GetHonorServerSigningAlgOrde |
| 00011 | 1c0001530 | Smb2GetHonorServerCipherOrder |
| 00012 | 1c00018e0 | Smb2ReferenceWorkItemFromAsyn... |
| 00013 | 1c0001900 | Smb2ReferenceNetname |
| 00014 | 1c0001934 | Srv2RemoveConnectionFromEndpoi... |
| 00015 | 1c0001a58 | RfsTable64Cleanup |
| 00016 | 1c0001bec | Srv2GlobalConnectionListRemove |
| 00017 | 1c0001fa0 | Srv2NegotiateHandler |
| 00018 | 1c0001fcc | Smb2NegotiateHandler |
| 00019 | 1c0002810 | Smb2FindDialectRecord |
| 00020 | 1c0002838 | RtlUShortAdd |
| 00021 | 1c0002c14 | Smb2CancelQueuedLeaseBreak |
| 00022 | 1c0002f84 | Srv2CancelWorkItem |
| 00023 | 1c0003550 | Srv2DereferenceCompoundWorkIter |
| 00024 | 1c00039e8 | Smb2FreeResponseBufferForAsync... |
| 00025 | 1c0003fd0 | Smb2LeaseSendComplete |
| 00026 | 1c0003fe8 | Smb2LeaseAcquireSpinLockFromOp... |
| 00027 | 1c00042a0 | Srv2PostAfterReceive |
| 00028 | 1c00043b0 | Srv2PostOnCompletionAndClearCa... |
| 00029 | 1c0004480 | Srv2ProcSendComplete |
| 00030 | 1c0004f60 | RfspThreadPoolNodeQueueWorkIter |
| 00031 | 1c0005140 | Smb2VerifyChannelSequence |
| 00032 | 1c00051a4 | Srv2IsEncryptionRequiredForInstan... |
| 00033 | 1c00057f0 | RfsHashTableDereference |
| 00034 | 1c0005814 | Smb2LeaseFindFreeOplockParams |
| 00035 | 1c000697c | Smb2LeaseDecrementOpensInProg... |
| 00036 | 1c00074dc | Smb2GetFileHandle |
| 00037 | 1c0007590 | Smb2ReferenceSessionAndTreeCon... |
| 00038 | 1c00084b0 | Smb2DereferenceChannel |

Partial matches

| Line | Address | Name |
|---|---|---|
| 00000 | 1c004d420 | Smb2ValidateGetFileInfoParameters |
| 00001 | 1c0050150 | Smb2ExecuteQueryInfo |
| 00002 | 1c00535c0 | Smb2ContinueQueryInfo |
| 00003 | 1c004d850 | Smb2ValidateQueryInfo |
| 00004 | 1c005d6f0 | Smb2QueryFileNormalizedName |

Unmatched in secondary

| Line | Address | Name |
|---|---|---|
| 00000 | 1c000e38c | Smb2ValidateVolumeObjectsMatch_Servicing |
| 00001 | 1c000e470 | Feature_Servicing_SMBNullCheck_3803337 |

**[Figure 19] Results of diffing produced by Diaphora.**

According to the image above, there is an interesting match:

- **Smb2QueryFileNormalizedName** | ratio: 0.9678% | 58 blocks (previously were 63) | description: Potential size check added.

We can also see that there are five partial matches:

- **Smb2ValidateGetFileInfoParameters** | ratio: 0.9995%
- **Smb2ExecuteQueryInfo** | ratio: 0.9991%
- **Smb2ContinueQueryInfo** | ratio: 0.9985%
- **Smb2ValidateQueryInfo** | ratio: 0.9930%
- **Smb2QueryFileNormalizedName** | ratio: 0.9781% | 58 Blocks (previously were 63)

There are two unmatched functions in the secondary binary (not available in the newer version):

- **Smb2ValidadeVolumeObjectsMatch_Servicing**
- **Feature_Servicing_SMBNullCheck_38033381__private_isEnabled.**

Finally, **Diaphora** has found **1063 functions** that have **a perfect match** and **there were no changes** between the two **srv2.sys** binaries (**JULY** and **AUGUST/2022**).

The report generated by **Diaphora** is very appropriate because it provides us with a clear status of functions and their respective differences, but it also provides further possibility as we will learn soon.

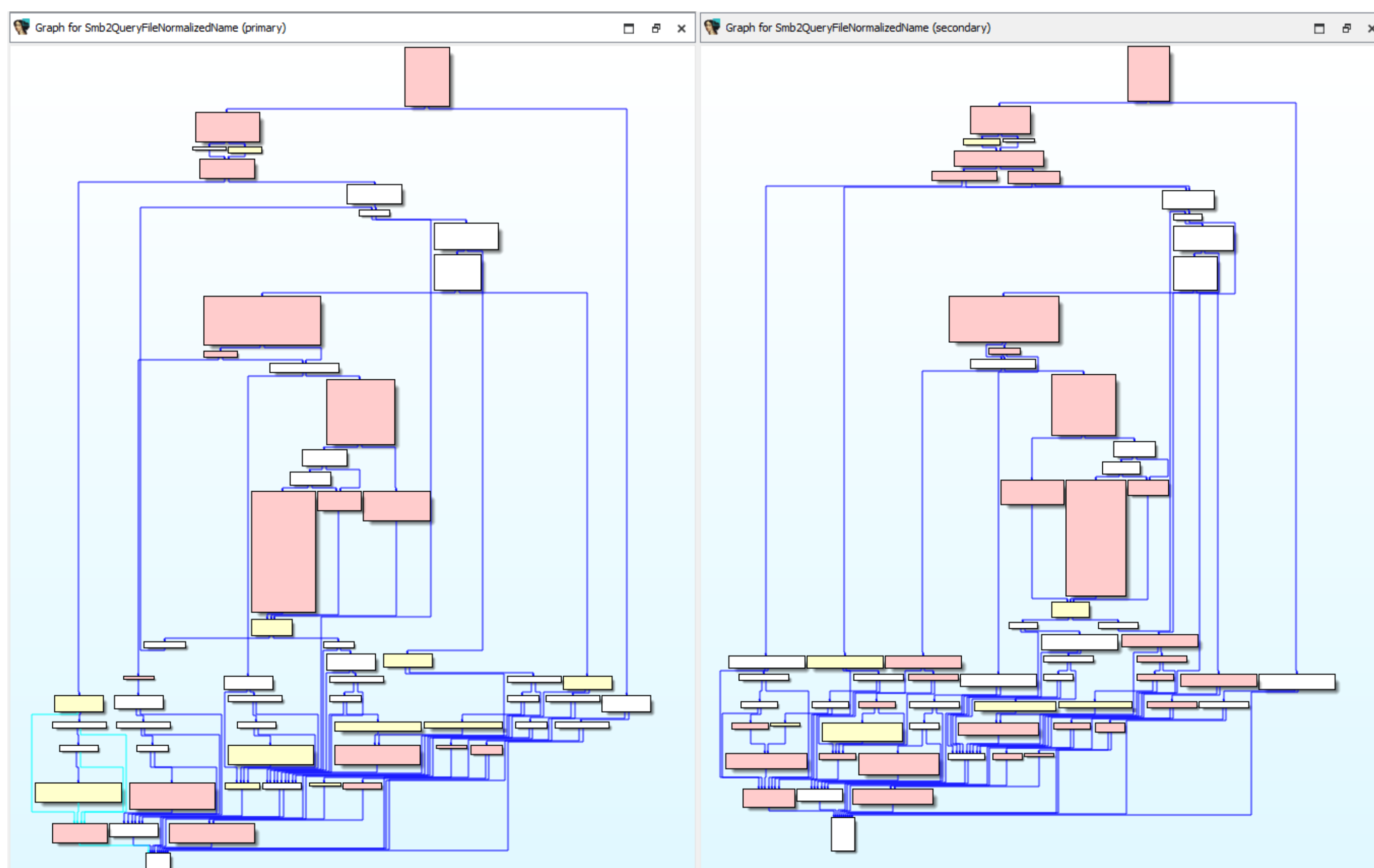
As readers can see, there is a series of slight details, but for now they are only based on the little information that we have retrieved above. It would be unnecessary to mention that every single piece of information is relevant through the process of exploitation:

- Our natural first target is **Smb2QueryFileNormalizedName**, which supposedly has received a **size checking code**, which might be caused by a **buffer overflow|underflow** or even an **integer overflow| underflow**. Of course, there are other potential reasons, and we do not know anything about it because we have not analyzed the function yet and, so far, it is just a speculation.

- There are **five partial matches**, and one of them (**Smb2QueryFileNormalizedName**) indicates a change for the number of blocks, but this kind of change itself does not mean that something useful happened (it could occur a merge, for example).

- Two functions have been removed in the newer version (AUG/2022). There are multiple potential reasons: the code has been incorporated by other functions, these functions had security issues, functionalities have been eliminated, and so on.

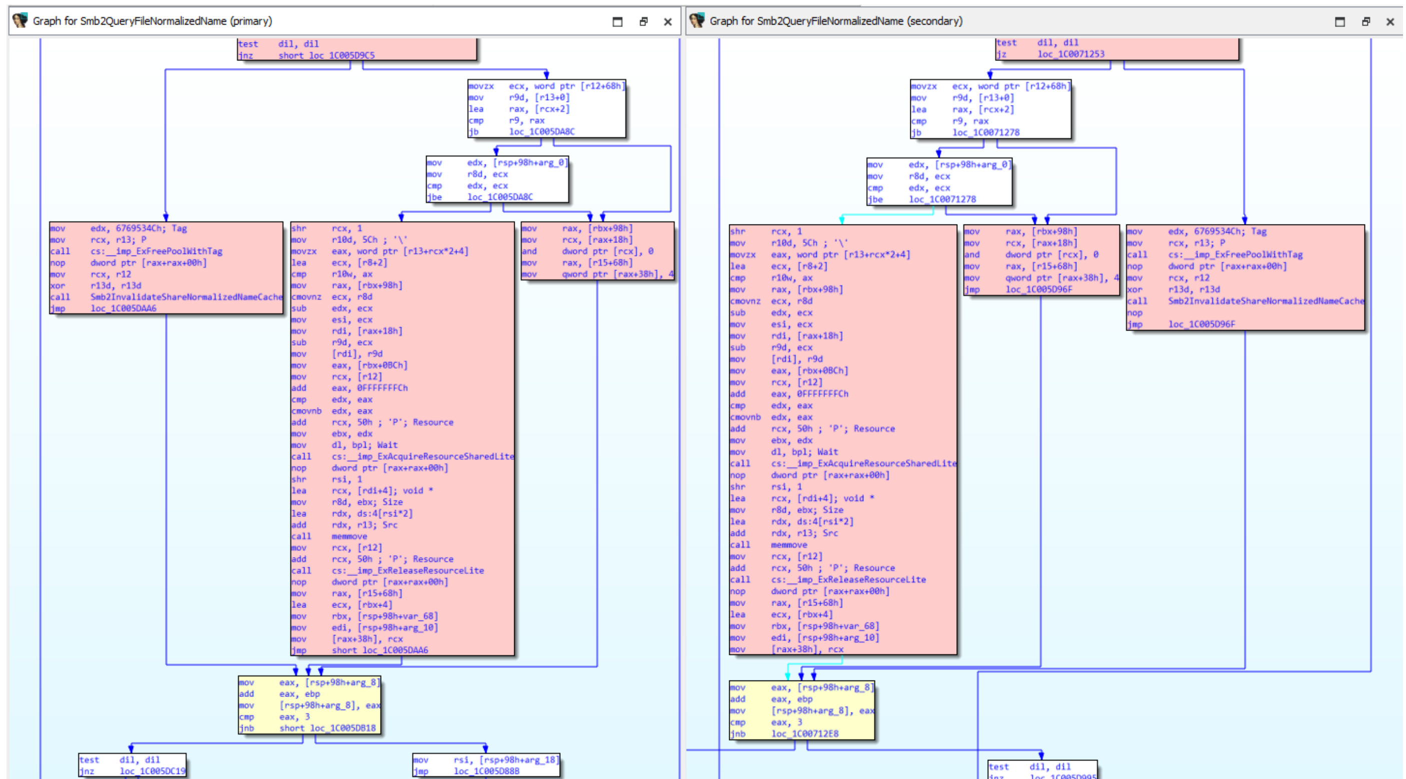
**Diaphora** provides us with multiple good options beyond the comparison summary presented previously. For example, taking as example the **Smb2QueryFileNormalizedName function**, right-click it and choose **Diff assembly in a graph** and both graphs from AUGUST and JULY versions of the binary will be showed containing necessary highlights to differences between them:




**[Figure 20] Results of diffing assembly produced by Diaphora for a target function.**

This overview already helps us to notice areas that suffered changes, which the yellow one's present minor or medium changes, and the red blocks offer an indication of the introduction of substantial changes or an addition of new code (there is not an equivalent when compared with the other version).

Readers can zoom the graph easily on IDA Pro to see the assembly code, as shown on the next page:




**[Figure 21] Graph showing differences for the same function between two different versions of a binary.**

We can examine differences between assemblies by **right-clicking | Diff Assembly**, as shown below:

```
08    nop      dword ptr [rax+rax+00h]
09    test     dil, dil
10    jz       loc_1C0071253
11loc_1c005d8aa:
12    movzx    ecx, word ptr [r12+68h]
13    mov      r9d, [r13+0]
14    lea      rax, [rcx+2]
15    cmp      r9, rax
16    jb       loc_1C0071278
17loc_1c005d8c1:
18    mov      edx, [rsp+98h+arg_0]
19    mov      r8d, ecx
20    cmp      edx, ecx
21    jbe      loc_1C0071278
22loc_1c005d8d3:
23    shr      rcx, 1
24    mov      r10d, 5Ch ; '\'
25    movzx    eax, word ptr [r13+rcx*2+4]
26    lea      ecx, [r8+2]
27    cmp      r10w, ax
28    mov      rax, [rbx+98h]
29    cmovnz   ecx, r8d
30    sub      edx, ecx
31    mov      esi, ecx
32    mov      rdi, [rax+18h]
33    sub      r9d, ecx
34    mov      [rdi], r9d
35    mov      eax, [rbx+0BCh]
36    mov      rcx, [r12]
37    add      eax, 0FFFFFFFCh
38    cmp      edx, eax
39    cmovnb   edx, eax
40    add      rcx, 50h ; 'P'; Resource
41    mov      ebx, edx
```

```
183    nop      dword ptr [rax+rax+00h]
184    test     dil, dil
185    jnz      short loc_1C005D9C5
186loc_1c005d9a1:
187    mov      edx, 6769534Ch; Tag
188    mov      rcx, r13; P
189    call     cs:__imp_ExFreePoolWithTag
190    nop      dword ptr [rax+rax+00h]
191    mov      rcx, r12
192    xor      r13d, r13d
193    call     Smb2InvalidateShareNormalizedNameCache
194    jmp      loc_1C005DAA6
195loc_1c005d9c5:
196    movzx    ecx, word ptr [r12+68h]
197    mov      r9d, [r13+0]
198    lea      rax, [rcx+2]
199    cmp      r9, rax
200    jb       loc_1C005DA8C
201loc_1c005d9dc:
202    mov      edx, [rsp+98h+arg_0]
203    mov      r8d, ecx
204    cmp      edx, ecx
205    jbe      loc_1C005DA8C
206loc_1c005d9ee:
207    shr      rcx, 1
208    mov      r10d, 5Ch ; '\'
209    movzx    eax, word ptr [r13+rcx*2+4]
210    lea      ecx, [r8+2]
211    cmp      r10w, ax
212    mov      rax, [rbx+98h]
213    cmovnz   ecx, r8d
214    sub      edx, ecx
215    mov      esi, ecx
216    mov      rdi, [rax+18h]
217    sub      r9d, ecx
218    mov      [rdi], r9d
219    mov      eax, [rbx+0BCh]
220    mov      rcx, [r12]
221    add      eax, 0FFFFFFFCh
222    cmp      edx, eax
223    cmovnb   edx, eax
224    add      rcx, 50h ; 'P'; Resource
225    mov      ebx, edx
```

**[Figure 22] Assembly differences for a given function.**

**Diaphora** offers another interesting feature that shows and highlight the code differences throughout pseudo-code by **right-clicking | Diff pseudo-code**, as shown below:

```
1  __int64 __fastcall Smb2QueryFileNormalizedName(__int64 a1)
2  {
3    __int64 v1; // rbx
4    unsigned int v2; // r14d
5    unsigned int *v3; // r13
6    __int64 v5; // r12
7    __int64 v6; // rsi
8    bool v7; // di
9    ULONG v8; // esi
10   unsigned int *Pool2; // rax
11   __int64 v10; // rcx
12   bool v11; // al
13   __int64 v12; // rcx
14   unsigned __int64 v13; // rcx
15   unsigned __int64 v14; // r9
16   __int16 v15; // ax
17   unsigned int v16; // ecx
18   unsigned int v17; // edx
19   unsigned __int64 v18; // rsi
20   _DWORD *v19; // rdi
21   unsigned int v20; // ebx
22   __int64 v21; // rcx
23   PDEVICE_OBJECT v23; // rcx
24   unsigned __int16 v24; // dx
25   PDEVICE_OBJECT v25; // rcx
26   unsigned __int16 v26; // dx
27   __int64 FileInformationClass; // [rsp+20h] [rbp-78h]
28   __int64 v28; // [rsp+30h] [rbp-68h]
29   struct _IO_STATUS_BLOCK IoStatusBlock; // [rsp+38h] [rbp-60h] BYREF
30   unsigned int v30; // [rsp+A0h] [rbp+8h] BYREF
31   int v31; // [rsp+A8h] [rbp+10h]
32   BOOL v32; // [rsp+B0h] [rbp+18h]
33   __int64 v33; // [rsp+B8h] [rbp+20h]
34
35   v1 = *(_QWORD *)(a1 + 504);
36   v2 = 0;
37   v3 = 0i64;
38   v28 = v1;
39   v5 = *(_QWORD *)(*(_QWORD *)(v1 + 64) + 112i64);
40   if ( *(_DWORD *)(v1 + 188) < 4u )
```

```
1  __int64 __fastcall Smb2QueryFileNormalizedName(__int64 a1)
2  {
3    __int64 v1; // rbx
4    unsigned int v2; // r14d
5    unsigned int *v3; // r13
6    __int64 v5; // r12
7    PDEVICE_OBJECT v6; // rcx
8    unsigned __int16 v7; // dx
9    __int64 v8; // rsi
10   PDEVICE_OBJECT v9; // rcx
11   unsigned __int16 v10; // dx
12   bool v11; // di
13   ULONG v12; // esi
14   unsigned int *Pool2; // rax
15   __int64 v14; // rcx
16   bool v15; // al
17   __int64 v16; // rcx
18   unsigned __int64 v17; // rcx
19   unsigned __int64 v18; // r9
20   __int16 v19; // ax
21   unsigned int v20; // ecx
22   unsigned int v21; // edx
23   unsigned __int64 v22; // rsi
24   _DWORD *v23; // rdi
25   unsigned int v24; // ebx
26   __int64 v25; // rcx
27   PDEVICE_OBJECT v26; // rcx
28   unsigned __int16 v27; // dx
29   __int64 FileInformationClass; // [rsp+20h] [rbp-78h]
30   __int64 v30; // [rsp+30h] [rbp-68h]
31   struct _IO_STATUS_BLOCK IoStatusBlock; // [rsp+38h] [rbp-60h] BYREF
32   unsigned int v32; // [rsp+A0h] [rbp+8h] BYREF
33   int v33; // [rsp+A8h] [rbp+10h]
34   BOOL v34; // [rsp+B0h] [rbp+18h]
35   __int64 v35; // [rsp+B8h] [rbp+20h]
36
37   v1 = *(_QWORD *)(a1 + 504);
38   v2 = 0;
39   v3 = 0i64;
40   v30 = v1;
41   v5 = *(_QWORD *)(*(_QWORD *)(v1 + 64) + 112i64);
42   if ( *(_DWORD *)(v1 + 188) < 4u )
```

**[Figure 23] Pseudo code differences for a given function.**

Do not forget to save the report: **Edit → Plugins → Diaphora – Save Results**. Later, we are going to have the opportunity to load and show the results, which can be done through the same menu path with **Diaphora – Load Results** and **Diaphora – Show Results**, respectively. There is an excellent **advantage in comparing side by side two pseudo codes** already highlighted to quickly spot potential critical points. Personally, I usually open **three IDA Pro instances with the following setup**:

1. The first instance contains the new code and its comparison with the old code (as shown above), using **Diaphora** (preferred) and/or **BinDiff**. This instance will be my draft and guideline.

2. The **other two instances** contain the newer and older versions, respectively.

Eventually, we will be able to detect vulnerabilities only by analyzing the code statically, but in diverse situations, it will not be possible, and we need to use WinDbg to support and clear our questions and doubts. That is not a big deal, but it demands further setup and skills that will be useful over the journey.

Do not forget that sometimes finding vulnerabilities can seem easy, but it is not. And, no doubt, **writing a functional and stable exploit is usually hundreds of times more difficult and harder than it.**

## 9. General code notes

So far, we have commented about how to use both **BinDiff** and **Diaphora**, but that is only one stop along the way, and the next step is to spot and review the first found differences in the code.

Based on **BinDiff** output, we have two main functions to focus right now:

- **Smb2QueryFileNormalizedName**
- **Smb2ValidateWrite**

There is a series of locations within the **Smb2QueryFileNormalizedName** function (87 % of similarity) that present substantial differences when compared to its previous version, and readers can see them by right on **Smb2QueryFileNormalizedName** function (inside **Matched Functions tab** when using **BinDiff**) and pick up **View Flow Graph**, where on the left side (primary function – the newer one in this case) and the right side, secondary, is the old (and vulnerable) code. The technique is to analyze the vulnerable code and compare with the new one to under for understanding what has been changed. It might seem obvious, but it is not because you must follow conditionals, make hypothesis, and understand what the consequences of each logical decision will be. For example, in eventual opportunities, an already existing instruction changes of place, from one address to another one, for preventing a logical issue as, for example, freeing a memory address twice or copying a freed memory location.

Additionally, another function named **Smb2ValidateWrite** presented 98% similarity, and using the same approach, readers will see the following location marked one address marked in red and other ones in yellow.

There are quite a few instructions that we can examine in both functions, but there are also innumerous pending questions that should be considered before proceeding. Except to the brief description offered by Microsoft ***"SMB Client and Server Remote Code Execution Vulnerability"***, we do not have anything else as starting point and, of course, first doubts come up:

- Has the vulnerability been found only in the server component or both (client and server)?
- Did Microsoft only add patches for such vulnerabilities, or did they also insert any code that is no related to the patch as improvement for performance?
- What do **Smb2QueryFileNormalizedName** and **Smb2ValidateWrite** do?
- Why did Microsoft remove **Smb2ValidateVolumeObjectsMatch_Servicing** and **Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled** functions (readers should check the **Secondary Unmatched tab** on **BinDiff**)? Have their functionalities migrated to the new code?

Certainly, we will need the maximum number of sources to search for prototype of functions, native APIs, structures, and any kind of help, so a brief list of suggestions follows below:

- **Virgilius project:** https://www.vergiliusproject.com/
- **Phnt:** https://github.com/winsiderss/phnt
- **NtDoc:** https://ntdoc.m417z.com/
- **ReactOS:** https://github.com/reactos/reactos
- **Windows SDK:** C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\km (but not only).

Of course, any further and available resources found through Google is welcome.

# 10. Review of kernel driver concepts and considerations about vulnerabilities

In the previous article from this series (https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/) I offered a long explanation about **kernel drivers, mini-filter drivers** and even **Windows Filtering Platform (WFP).** However, I want to perform a brief list of reviews and observations and, eventually, a small amount of them might be new for readers:

- The **DriverEntry function** is the entry point for kernel drivers, and such function is called at **IRQL == PASSIVEL_LEVEL (zero)**.

- A driver is represented by a driver object (**DRIVER_OBJECT**), but the entire communication with clients is performed through a device object (**DEVICE_OBJECT**), which is created by invoking **IoCreateDevice** function (actually, clients will interact with device objects through their handles). There can be one or more associated with the **driver objects**.

- The client will open a driver's device object by calling **CreateFile** function and providing symbolic link argument created by invoking **IoCreateSymbolicLink** function.

- The communication between client and driver is based on requests and responses, and such requests are represented (wrapped) by an **IRP (I/O Request Packet)**.

- Drivers manipulate IRPs, which also have an or more associated structure (depending on the number of layers in the device stack) whose type is **IO_STACK_LOCATION**. Later, **IoGetCurrentIrpStackLocation** function will get the I/O stack location (**IO_STACK_LOCATION**) and will parse it (from the current layer) to get information for IRP request.

- The number of I/O stack locations (**IO_STACK_LOCATION**) is related to the number of device objects associated with this driver (represented by a **DRIVER_OBJECT**) over the driver stack.

- The communication between drivers across the driver stack is performed by passing the request (IRP) down to the next driver in the stack. Each driver in the stack has the following options:

    - Pass the request down to the next driver: the driver is not interested in processing this request, so it forwards the IRP to the next driver by calling **IoSkipCurrentIrpStackLocation function**, which modifies the pointer in the array of **IO_STACK_LOCATION** for that the next driver receives the same **IO_STACK_LOCATION** that the current driver received, and **IoCallDriver function** that sends the IRP to the driver associated with the provided device object.

    - Process the IRP request without replicating it to next layers and, once such processing has finished, it calls **IoCompleteRequest function**, which returns the IRP to the I/O manager.

    - A composition of the two options above.

- Kernel drivers should have an unload routine, which is accessed through the driver object: **DriverObject → DriverUnload = UnloadRoutine**.

- To get a pointer to a given object through a valid handle, kernel drivers use the **ObReferenceObjectByHandle** function, which is a reference object function (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obreferenceobjectbyhandle).

- Kernel drivers can use three different memory pools: **Paged Pools, Non-Paged Pools and NonPagedPoolNx** (recommended). The most used functions to allocate these various kinds of memory pools are **ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePool2** and **ExAllocatePool3**.

- A **DRIVER_OBJECT** represents a kernel driver, which contains a special member named **MajorFunction**. This member holds a pointer to an array of function pointers, indexed by indexes starting with **IRP_MJ_** prefix, and that specifies all operations the driver supports such as **IRP_MJ_CREATE** (it is usually implemented as a dispatch routine, which will be invoked through **NtCreateFile function** later, because the client needs to have a handle to establish communication with the device), **IRP_MJ_WRITE, IRP_MJ_READ** and mainly **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL**.

- Furthermore, we should remember that driver dispatch routines (**DRIVER_DISPATCH**), which are also named as callbacks, follow the following prototype: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-driver_dispatch. Any non-used array element is set with a pointer to the **IoInvalidDeviceRequest** routine. Therefore, readers will see something like:

    - **DriverObject → MajorFunction[IRP_MJ_CREATE]= MyCreateDispatchRoutine**
    - **DriverObject → MajorFunction[IRP_MJ_WRITE]= MyWriteDispatchRoutine**
    - **DriverObject → MajorFunction[IRP_MJ_READ]= MyReadDispatchRoutine**
    - **DriverObject → MajorFunction[IRP_MJ_DEVICE_CONTROL]= MyDeviceControlDispatchRoutine**
    - **DriverObject → MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL]= MyInternalDeviceControlDispatchRoutine**

- Clients (mostly from user mode) send and receive data from drivers through buffers, which are usually involved with **IRP_MJ_DEVICE_CONTROL, IRP_MJ_WRITE** and **IRP_MJ_READ** dispatch routines operations (as shown above). When managing **read** and **write operation (IRP_MJ_WRITE** and **IRP_MJ_READ)**, there are different transfer methods to perform the data transmission using buffers, which are reviewed in the next two bullets.

- **Buffered I/O (DeviceObject → Flags |= DO_BUFFER_IO):** this method is supervised by the I/O manager, which allocates a buffer from non-paged pool. As the memory address is allocated in the non-paged pool by invoking any of mentioned functions above such as **ExAllocatePool, ExAllocatePoolWithTag or ExAllocatePool2 function** then the address does not change, and it is

valid to any thread context. Additionally, it is not paged (without needing to lock down the physical page). The data is returned to the client through a **special kernel APC**, which is executed right before the scheduled thread is executed. If readers remember about **APC injection** (even using **user-mode APC**), the principle is the same. Check about types of APC on: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-apcs. As expected, **Buffered I/O** is great for small buffers being used by devices such as mouse, keyboard, video, and serial, but it is inefficient for large ones.

- **Direct I/O (DeviceObject → Flags |= DO_DIRECT_IO):** the approach followed by this method is to allow access to the user buffer, but eliminating copy operations, and it is appropriate to manage large amount of data, besides improving the performance. To make this technique possible, the I/O Manager creates an **MDL (Memory Descriptor List)**, which describes the buffer. As expected by readers, the **I/O Manager** needs to check whether the memory address is valid and, to prevent any paging out, so such memory buffer is locked, and both operations are accomplished by invoking **MmProbeAndLockPages function and taking as argument exactly a pointer to MDL recently created**. Therefore, the MDL works as a second mapping (or even an abstraction, and the first mapping address comes from the original buffer allocated by the requesting thread/process) for the "real address" and presents an advantage of being located in the kernel-side, so it is valid and the same for any arbitrary thread. To get the address of the buffer (the original or first mapping address), **MmGetMdlVirtualAddress or MmGetMdlVirtualAddressSafe function** must be called.

- When we are referring to device control operations (**IRP_MJ_DEVICE_CONTROL** / **IRP_MJ_INTERNAL_DEVICE_CONTROL**), which the key API is **DeviceIoControl( )**, there are distinguished methods to access the buffer. However, the chosen access method is intrinsically associated with the **DeviceIoControl** function, as shown below:

```
BOOL DeviceIoControl(
  HANDLE       hDevice,
  DWORD        dwIoControlCode,
  LPVOID       lpInBuffer,
  DWORD        nInBufferSize,
  LPVOID       lpOutBuffer,
  DWORD        nOutBufferSize,
  LPDWORD      lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

**[Figure 24] DeviceIoControl prototype**

- As readers can notice from the function's prototype above, a few parameters are quite interesting:

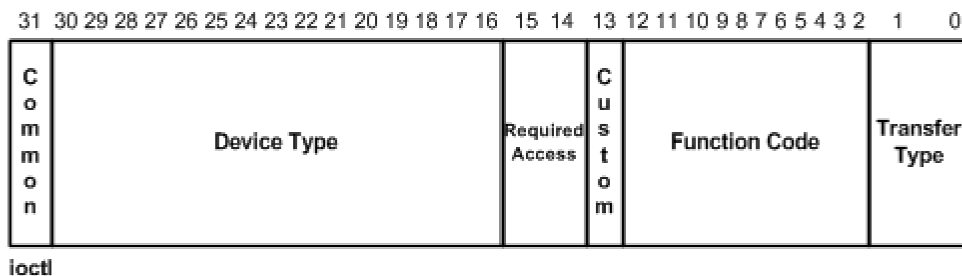
    - **hDevice:** it is a handle to the device, and this handle is retrieved through **CreateFile** function.

    - **dwIoControlCode**: it is the control code for the operation, which is calculated by the **CTL_CODE macro** and given by the following prototype (**C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\km\wdm.h):**

```
6816
6817 #define CTL_CODE( DeviceType, Function, Method, Access ) (                 \
6818     ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
6819 )
6820
```

**[Figure 25] CTL_CODE macro definition**

- The visual representation of the **IOCTL code** is the following:




**[Figure 26] CTL_CODE macro scheme (credits: Microsoft)**

- The **CTL_CODE macro**, as shown, also has parameters that demands a concise explanation:

    - **DeviceType:** this parameter is a constant that defines the device type. Possible values are described on: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/specifying-device-types.

    - **Access:** this parameter determines the type of access required by the client for opening the target device, which works as an access control to use or not the device as planned by the client. A list of potential values is:

        - **FILE_ANY_ACCESS**
        - **FILE_READ_DATA**
        - **FILE_WRITE_DATA**
        - A **combination** of **FILE_READ_DATA** and **FILE_WRITE_DATA**.

    - **Function:** this argument identifies the function to be executed by the driver, and values equal and above of 0x800 can be used by vendors.

    - **Method:** this argument identifies how data will be passed between the client and driver:

        - **METHOD_BUFFERED:** this value determines that the transfer method is **Buffered I/O** (it has already been explained).

        - **METHOD_IN_DIRECT** or **METHOD_OUT_DIRECT:** this value determines that the transfer method is **Direct I/O** for input or output buffer, respectively.

        - **METHOD_NEITHER**: this value specifies that the I/O manager does not provide any help for data transferring and is also not managing it. Thus, the system does not

provide any system buffer or even an MDL, and the driver itself is responsible for handling the transference of data between the client and the driver.

As we are interested in analyzing the driver from a different point of view, we need to focus on finding possible key functions/routines and areas that could offer security problems. Thus, other points could be considered:

- The usual order of actions inside **DriverEntry routine** is:

  - Build the **Device Name** and **Symbolic Link strings** using functions like **RtlInitUnicodeString.**

  - Create a device using **IoCreateDevice** or **IoCreateDeviceSecure** functions. One of parameters of **IoCreateDevice function**, *DeviceCharacteristics*, could being interesting if it doesn't contain the **FILE_DEVICE_SECURE_OPEN value**, which is usually specified for most drivers, enforces that security checks (ACLs) are enabled and verified for file open requests and guarantees that the same security settings are applied for requests to the device. In other words, the I/O manager would not apply the correct ACL while attending the request. For this reason, the recommendation is to use **IoCreateDeviceSecure** or **WdmlibIoCreateDeviceSecure** functions because it applies security settings, block non-privileged users from open a handle to and, in addition, it deletes the object when is it is no longer necessary. However, even if the driver is not using **IoCreateDeviceSecure function**, it is sometimes a bit more difficult to open the device without having additional information that is not even expected. On the other hand, as offensive readers already know, enumerating device objects and trying to open them is key :)

  - Create a **symbolic link** for making the device's access available for the client. Unfortunately, not all drivers have such "friendly name", and they could have been generated automatically (**FILE_AUTOGENERATED_DEVICE_NAME** flag present in **DeviceCharacteristics** parameter), which is the fact that we see "names" that are a sequence of hexadecimals. Readers can see such hexadecimal names by opening **WinObj tool**, from **Sysinternals** suite, and going to **Device branch**.

  - Filling the dispatch table of the driver object (**DEVICE_OBJECT**).

- As expected, the main point of a driver communication is the **DeviceIoControl function**, which provides an interface to communication between the application (client) and the kernel driver, and this communication is controlled (and defined) by a **IOCTL code** (**dwIoControlCode parameter).** There is a series of resources such as plugins, scripts, and websites to decode such **IOCTL codes**:

  - **DriverBuddyReloaded:** https://github.com/VoidSec/DriverBuddyReloaded
  - **Windows Driver Plugin (original):** https://github.com/FSecureLABS/win_driver_plugin
  - **Windows Driver Plugin (fork 1):** https://github.com/alexander-pick/win_driver_plugin
  - **Windows Driver Plugin (fork 2):** https://github.com/tacbliw/win_driver_plugin
  - **ioctl.py:** https://github.com/h0mbre/ioctl.py
  - **OSR Online IOCTL Decoder:** https://www.osronline.com/article.cfm%5earticle=229.htm

- Once the **IOCTL code** is decoded, we will **have DeviceType, Access, Function and Method**. No doubt, the **Access** and **Method** are the most important parameters for us because we need them to interact with the device and, eventually, explore the data transfer, which might be interesting. If the driver specifies **METHOD_BUFFERED** as access method then basically the buffer (limited by its size) is copied to the kernel and this action prevents a late changing, so it is more secure, even though is not guarantee of having a secure code as, for example, a simple out-of-boundary write operation means writing in a non-initialized memory, which might be translated in an eventual compromising. If the driver specifies **METHOD_NEITHER**, the **I/O Manager** is not managing data transfer, and attackers can change buffers' properties such as its length or even its memory allocation. Finally, if the driver specifies **METHOD_IN_DIRECT** or **METHOD_OUT_DIRECT**, the **I/O Manager** is managing the buffer allocation and checking such buffer is accessible for reading or writing according to value indicated in the **Access parameter**.

- In terms of code, the following fields are relevant for us (readers should check the previous article for IRP fields):

  - **METHOD_BUFFERED: IRP→AssociatedIrp. SystemBuffer** (input and output buffers). In terms of buffer size, **IRP→Parameters.DeviceIoControl.InputBufferLength** field (from **IO_STACK_LOCATION**) is used for input buffer and **IRP→Parameters.DeviceIoControl.OutputBufferLength** field (from **IO_STACK_LOCATION**) is used for output buffer. **Both length fields should be checked before reading and writing operations to prevent out-of-bonds read and write vulnerabilities.**

  - **METHOD_IN_DIRECT and METHOD_OUT_DIRECT: IRP→AssociatedIrp. SystemBuffer** (input buffer) and **IRP→MdlAddress** (output buffer). The buffer size is represented by **IRP→Parameters.DeviceIoControl.InputBufferLength** and **IRP→Parameters.DeviceIoControl.OutputBufferLength** fields for input and output buffer (described by an MDL), respectively. Same advice about length fields as mentioned above.

  - **METHOD_NEITHER: IRP→Parameters.DeviceIoControl.Type3InputBuffer** field from **IO_STACK_LOCATION structure** (input buffer) and **IRP→UserBuffer** field (output buffer). The buffer size is provided by **IRP→Parameters.DeviceIoControl.InputBufferLength** and **IRP→Parameters.DeviceIoControl.OutputBufferLength** fields (from **IO_STACK_LOCATION structure**) for input and output buffers, respectively. Same advice about length fields.

- As readers can notice, there is a concise list of critical points:

  - The provided data cannot be of a larger size than supported by the buffer.
  - The buffer's address must point to a valid address.
  - The best scenario for us is when the drivers use **METHOD_NEITHER**.
  - Buffer overflow would be the simpler attack in this case.
  - If there is a buffer overflow, then a crash is most expected.

- A generic guideline for writing an exploit is not so complicated:

- Get a handle the device object exposed by the target driver.
- Open the retrieved handle (**CreateFile function**).
- Allocate a buffer.
- Call the **DeviceIoControl function** by providing necessary information such as handle to device object, **IOCTL code** (it regards the fact whether access and method access), input and output buffers, and their respective sizes.
- Write a shellcode (eventually, the content comes from a file, so there is another **CreateFile function** call) into the target buffer.
- The shellcode can perform token stealing (data-writing attack) or change any other structure.

- Readers have already heard about **arbitrary write vulnerability** and **arbitrary read vulnerability**. The latter one might be interesting to leak valuable information from memory. Nonetheless, arbitrary write vulnerability is usually more attractive because allows an attacker to write a shellcode in a controlled address to later dereference it, so executing the code.

- Unfortunately, different methods of attack depend on operating system protections, and on Windows there are a lot of them, as for example, **Memory Integrity** (as known as **Hypervisor Protected Memory Integrity – HVCI**), which restricts memory allocations that could be used to compromise the system and, in practical terms, applying a W^X protection that imposes that an allocated kernel memory might be executable, but not writable (and vice-versa).

- If Memory Integrity is not enabled (and it is not true because is enabled by default on Windows 11 in the current days), the possibilities are better because most processes have Integrity level as Medium (read about it: https://learn.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control) then invoking **NtQuerySystemInformation** and **EnumDeviceDriver** functions would be enough to bypass the KASLR and get the kernel's base address and, as it is well-known, we could need to disable SMEP (Supervision Mode Execution Prevention) via CR4, which would allow a kernel driver to access/execute a code in a user mode buffer and perform an local elevation of privilege. It would not be necessary to mention that arbitrary write vulnerability would be more susceptible for drivers using METHOD_NEITHER method, and glaringly developers should never use or work with such method because they will be losing the entire I/O manager support. That is one of reasons for paying attention to the IOCTL mode, which will end with 11 in case of **METHOD_NEITHER** (based on wdm.h). Readers are encouraged to check **wdm.h file** *("C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\km\wdm.h"*), where you will find interesting information :

```
6832  // Define the method codes for how buffers are
6833  // passed for I/O and FS controls
6834
6835  #define METHOD_BUFFERED                 0
6836  #define METHOD_IN_DIRECT                1
6837  #define METHOD_OUT_DIRECT               2
6838  #define METHOD_NEITHER                  3
```

**[Figure 27] Method codes (wdm.h)**

```
7259  // Define the various device characteristics flags
7260  //
7261
7262  #define FILE_REMOVABLE_MEDIA                        0x00000001
7263  #define FILE_READ_ONLY_DEVICE                       0x00000002
7264  #define FILE_FLOPPY_DISKETTE                        0x00000004
7265  #define FILE_WRITE_ONCE_MEDIA                       0x00000008
7266  #define FILE_REMOTE_DEVICE                          0x00000010
7267  #define FILE_DEVICE_IS_MOUNTED                      0x00000020
7268  #define FILE_VIRTUAL_VOLUME                         0x00000040
7269  #define FILE_AUTOGENERATED_DEVICE_NAME              0x00000080
7270  #define FILE_DEVICE_SECURE_OPEN                     0x00000100
7271  #define FILE_CHARACTERISTIC_PNP_DEVICE              0x00000800
7272  #define FILE_CHARACTERISTIC_TS_DEVICE               0x00001000
7273  #define FILE_CHARACTERISTIC_WEBDAV_DEVICE           0x00002000
7274  #define FILE_CHARACTERISTIC_CSV                     0x00010000
7275  #define FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL    0x00020000
7276  #define FILE_PORTABLE_DEVICE                        0x00040000
7277  #define FILE_REMOTE_DEVICE_VSMB                     0x00080000
7278  #define FILE_DEVICE_REQUIRE_SECURITY_CHECK          0x00100000
```

**[Figure 28] Device characteristics flags (wdm.h)**

- Two notes about the last couple of bullets: previously attackers could overwrite the **HalDispatchTable** function table with a user-mode address to reference our shellcode, however this function's address is randomized these days, and that is the motivation of bypassing KASLR. Additionally, it would be necessary to bypass Memory Integrity to be able to allocate, write and execute the shellcode, and as I mentioned, it is enabled by default.

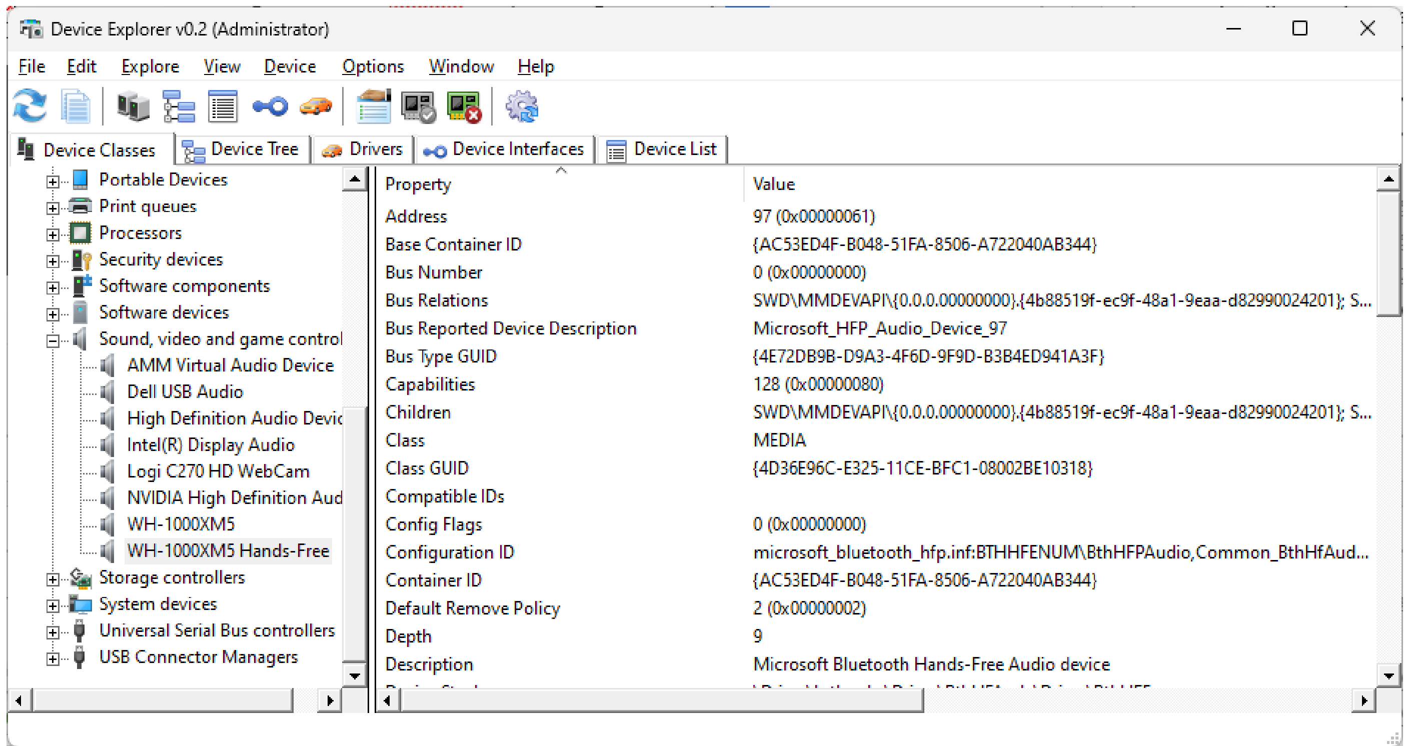
- Another interesting point is that if attackers are trying to bypass **ASLR/KASLR** from a **Low Integrity** process might need a leak and, in this case, a kernel-mode read primitive would be useful to get the kernel base address.

- Security issues can come up from unexpected points like unchecked returned types or even due to passing unprecise type arguments, which causes type confusion vulnerability.

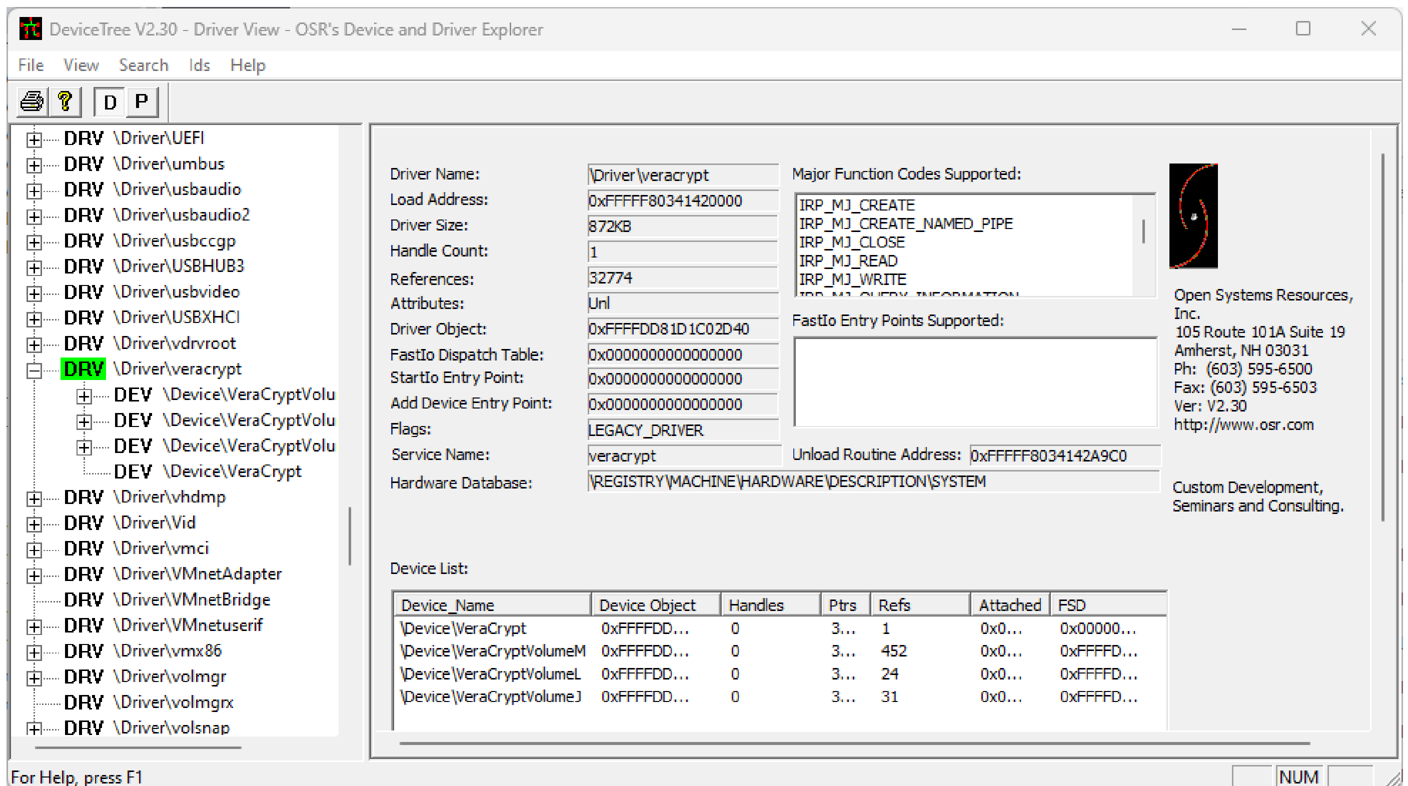
Optionally, readers can obtain relevant information about device drivers from a running system and even interacting with such drivers using excellent tools, which some of them offers a compiled version while other one's demand to compile the project:

- **DeviceTree (excellent):** https://www.osronline.com/article.cfm%5Earticle=97.htm
- **System Informer:** https://systeminformer.sourceforge.io/
- **Object Explorer and Device Explorer:** https://github.com/zodiacon/AllTools
- **IRPMon:** https://github.com/MartinDrab/IRPMon
- **Ioctlplus:** https://github.com/VoidSec/ioctlpus (a fork from https://github.com/jthuraisamy/ioctlpus).
- **Windows Driver IOCTL Tool Suite (DIBF):** https://github.com/nccgroup/DIBF

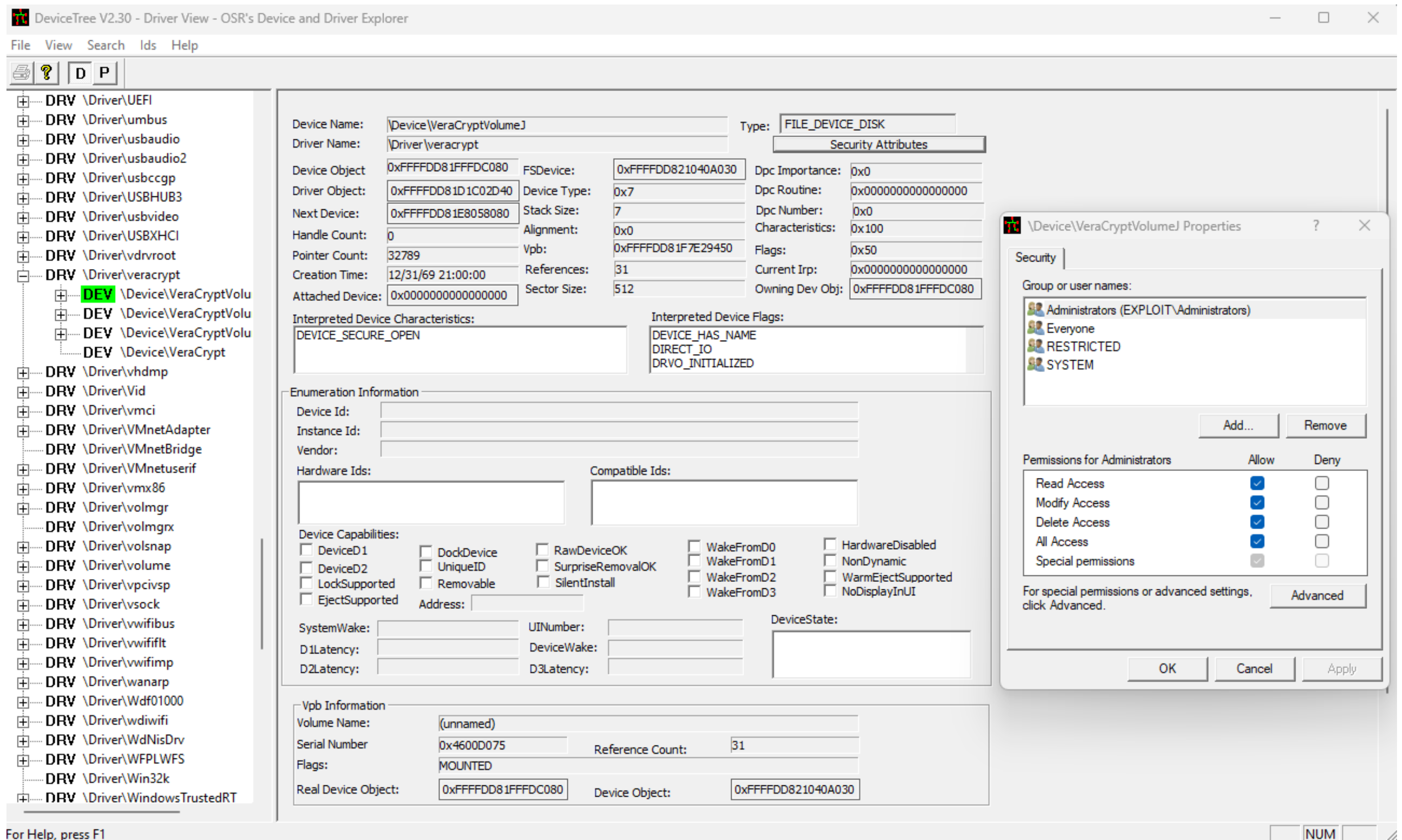
These tools are great and provide readers with excellent and outstanding information about installed device drivers. The three last ones (**IRPMon, Ioctplus** and **DIBF**) are also useful for monitoring IRP requests, sending IRP requests, fuzzing, and translating IOCTL code, respectively. Therefore, even though I will not use all of them right now in this article, it is still recommended to know about them:



**[Figure 29] Device Explorer**

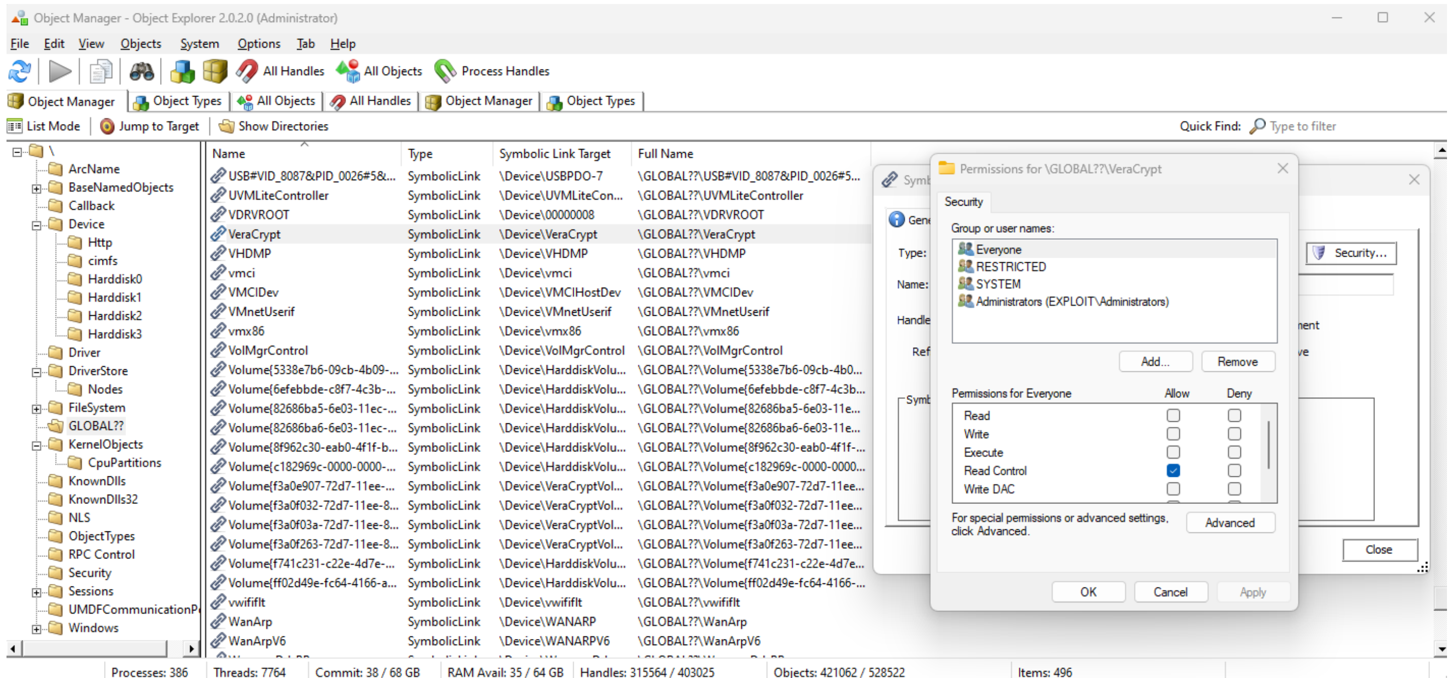


**[Figure 30] DeviceTree (1)**

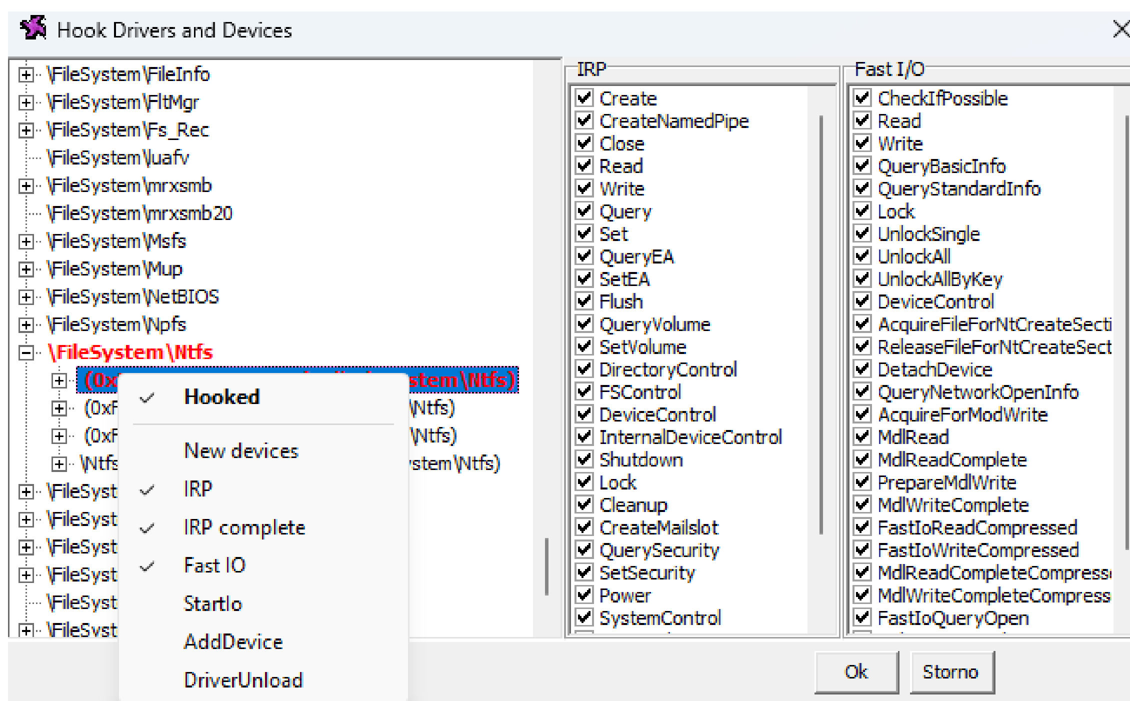


**[Figure 31] DeviceTree (2)**

Readers should notice that **DeviceTree tool** to provide us with all necessary information**: driver's general information, supported major function codes, all device names, device characteristics flags, device flags, security attributes**, and other essential information.



**[Figure 32] Object Explorer -- Object Manager**




**[Figure 33] IRPMon – Configuration**

IRPMon

Action Monitoring Request Driver Columns Help

Requests

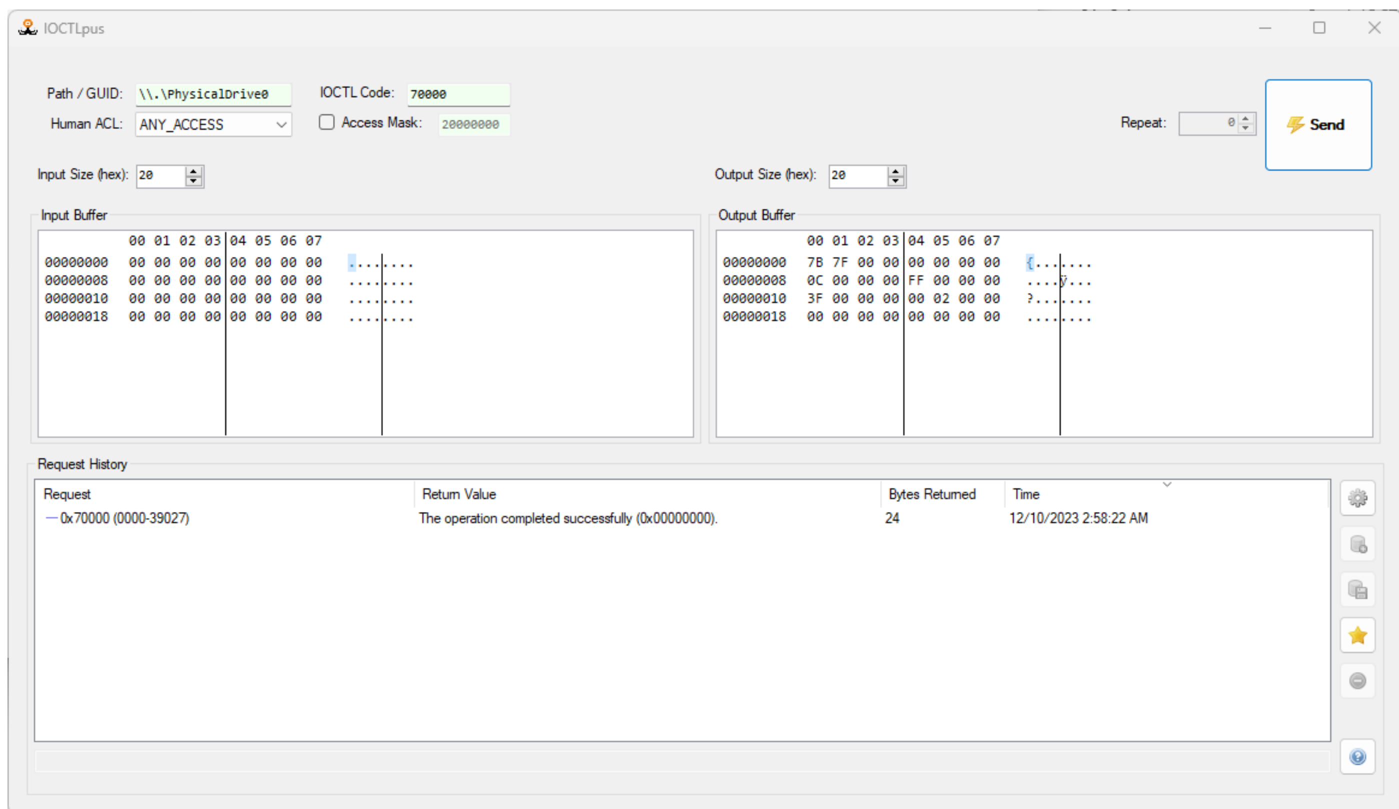
| ID | Process ID | IRQL | Device object | Device name | Driver object | Driver name | IRP address | Subtype | File object | IRP flags | Argument1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | | DeviceControl | 0xFFFF898EABAA9630 | | O: 1024 (0x0000000000000400) |
| 2 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | 0xFFFF898EA8BF48... | DeviceControl: | 0xFFFF898EABAA9630 | 0x62000 | O: 1024 (0x0000000000000400) |
| 3 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | | DeviceControl | 0xFFFF898EABAA9630 | | O: 1024 (0x0000000000000400) |
| 4 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | 0xFFFF898EAE5FC... | DeviceControl: | 0xFFFF898EABAA9630 | 0x62000 | O: 1024 (0x0000000000000400) |
| 5 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | | DeviceControl | 0xFFFF898EABAA9630 | | O: 1024 (0x0000000000000400) |
| 6 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | 0xFFFF898EA8BF5... | DeviceControl: | 0xFFFF898EABAA9630 | 0x62000 | O: 1024 (0x0000000000000400) |
| 7 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | | DeviceControl | 0xFFFF898EABAA9630 | | O: 1024 (0x0000000000000400) |
| 8 | 4604 | Passive | 0xFFFF898EA6C09... | \FileSystem\Filters\FltMgrMsg | 0xFFFF898EA6C11... | \FileSystem\Fl... | 0xFFFF898EA8BF4B00 | DeviceControl: | 0xFFFF898EABAA9630 | 0x62000 | O: 1024 (0x0000000000000400) |
| 9 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Create: | 0xFFFF898EB1503180 | 0x884 | 0xFFFFD48128014820 |
| 10 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 11 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | | QueryBasicInfo | 0xFFFF898EB1503180 | | |
| 12 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Cleanup: | 0xFFFF898EB1503180 | 0x404 | |
| 13 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 14 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Close: | 0xFFFF898EB1503180 | 0x404 | |
| 15 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 16 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Create: | 0xFFFF898EB1503180 | 0x884 | 0xFFFFD48128014820 |
| 17 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 18 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Create: | 0xFFFF898EB1503180 | 0x884 | 0xFFFFD48128014820 |
| 19 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 20 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | | QueryBasicInfo | 0xFFFF898EB1503180 | | |
| 21 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Cleanup: | 0xFFFF898EB1503180 | 0x404 | |
| 22 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 23 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | Close: | 0xFFFF898EB1503180 | 0x404 | |
| 24 | 8876 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD4DF... | | | | |
| 25 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Query: | 0xFFFF898EAFD050C0 | 0x1014 | L: 190 |
| 26 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | | | | |
| 27 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Create: | 0xFFFF898EB1503180 | 0x884 | 0xFFFFD481291E9820 |
| 28 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | | | | |
| 29 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | | QueryBasicInfo | 0xFFFF898EB1503180 | | |
| 30 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Cleanup: | 0xFFFF898EB1503180 | 0x404 | |
| 31 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | | | | |
| 32 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Close: | 0xFFFF898EB1503180 | 0x404 | |
| 33 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | | | | |
| 34 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Create: | 0xFFFF898EB1503180 | 0x884 | 0xFFFFD481291E9910 |
| 35 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | | | | |
| 36 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD8BF... | DirectoryControl... | 0xFFFF898EB1503180 | 0x60000 | 0x000000000000027C |
| 37 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EAD8BF... | | | | |
| 38 | 5516 | Passive | 0xFFFF898EA846E030 | | 0xFFFF898EA636CE20 | \FileSystem\Ntfs | 0xFFFF898EB10BCB20 | Cleanup: | 0xFFFF898EB1503180 | 0x404 | |

**[Figure 34] IRPMon – Monitoring (truncated – there are additional columns)**

The **original IOCTLplus repository** does not offer an already compiled version, so execute the following steps:

- Clone the repository: git clone https://github.com/jthuraisamy/ioctlpus
- Open the ioctlplus.sln solution on Visual Studio 2017.
- There will be a few errors, but do not worry about them.
- Close the solution.
- Open the solution up again and compile it.

The **VoidSec's version** is **updated** and already **compiled**:
https://github.com/VoidSec/ioctlpus/releases/tag/2.4




**[Figure 35] IOCTLplus**

To decode the **IOCTL code** shown above we can **iocode.exe** from DIBF:

```
C:\github\DIBF\Release>iocode.exe 0x70000
DECODING IOCODE 0x00070000:
device type = 0x7 FILE_DEVICE_DISK (MS)
function = 0 (MS)
method = METHOD_BUFFERED
access = FILE_ANY_ACCESS
```

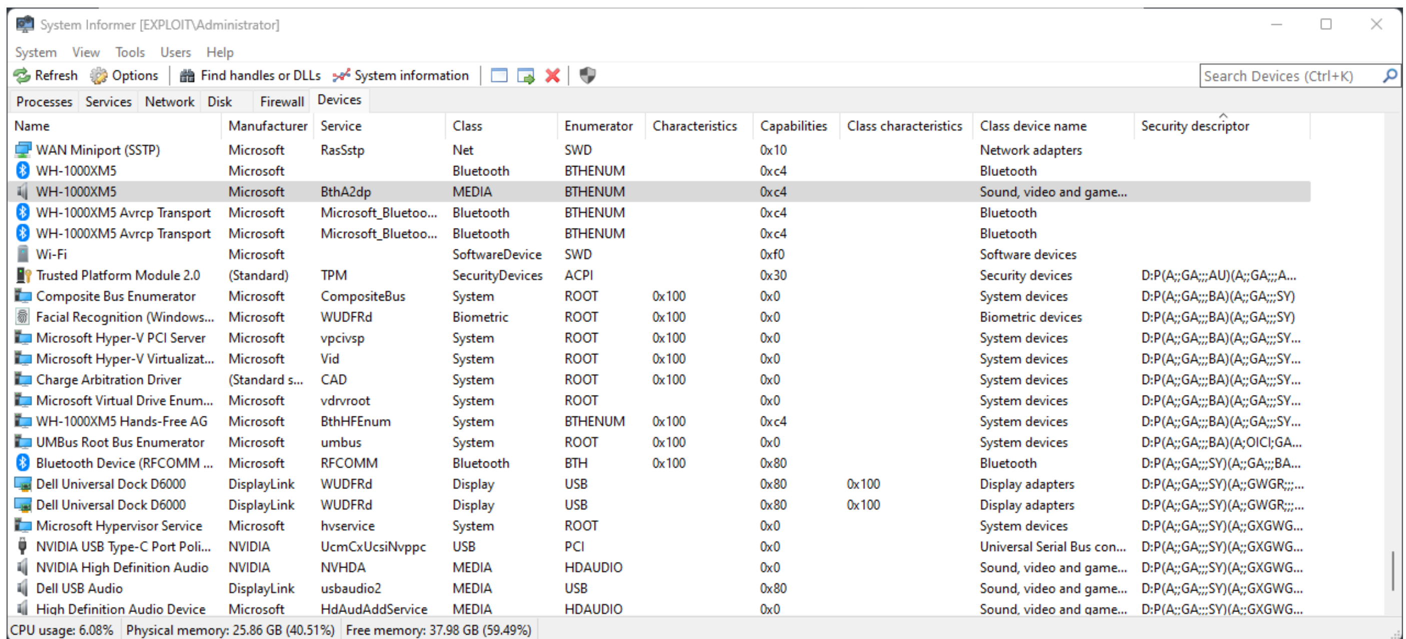
**[Figure 36] iocode.exe**

As expected, we have:

- The IRP is using **METHOD_BUFFERED**, which is safer than other options, but it is not always safe.
- The **Device type** is related to disk, which readers can confirm by checking the **wdm.h file.**
- The access is **FILE_ANY_ACCESS**, so the I/O manager sends the IRP for any caller that has a handle to the file object to the target driver.
- Functions below 0x800 are reserved to Microsoft.

This procedure of decoding IOCTL is integrated into IDA Pro through plugins, which I already commented on previously.

Finally, we have the **System Informer tool**, which is a multi-purpose tool:




**[Figure 37] System Informer – Devices tab**

I have quickly mentioned a small number of classes of vulnerabilities, but there are other ones that are usually found on kernel drivers and user mode codes such as:

- **Buffer overflow:**
  - https://cwe.mitre.org/data/definitions/120.html
  - https://cwe.mitre.org/data/definitions/121.html
  - https://cwe.mitre.org/data/definitions/122.html
  - https://cwe.mitre.org/data/definitions/788.html
- **Use-After-Free (UAF):** https://cwe.mitre.org/data/definitions/416.html
- **Integer Overflow:**
  - https://cwe.mitre.org/data/definitions/190.html
  - https://cwe.mitre.org/data/definitions/680.html
- **Type Confusion:**
  - https://cwe.mitre.org/data/definitions/704.html
  - https://cwe.mitre.org/data/definitions/843.html
- **Uninitialized Memory:**
  - https://cwe.mitre.org/data/definitions/457.html
  - https://cwe.mitre.org/data/definitions/665.html
  - https://cwe.mitre.org/data/definitions/824.html
  - https://cwe.mitre.org/data/definitions/908.html

- **Double Free:** https://cwe.mitre.org/data/definitions/415.html

- **Race Conditions, Double Fetch (subset of Race Conditions) and TOCTOU (Time of Check, Time of Use – it is a subset of Double Fetch and Race Conditions):**

    - https://cwe.mitre.org/data/definitions/362.html
    - https://cwe.mitre.org/data/definitions/366.html
    - https://cwe.mitre.org/data/definitions/367.html

- **Null Pointer Dereference:**

    - https://cwe.mitre.org/data/definitions/476.html
    - https://cwe.mitre.org/data/definitions/690.html

- **Memory Leak:**

    - https://cwe.mitre.org/data/definitions/401.html
    - https://cwe.mitre.org/data/definitions/772.html

- **Out-of-bounds write/read, Write-what-where:**

    - https://cwe.mitre.org/data/definitions/123.html
    - https://cwe.mitre.org/data/definitions/125.html
    - https://cwe.mitre.org/data/definitions/787.html

- **Off-by-one Error:** https://cwe.mitre.org/data/definitions/193.html

- **Elevation of Privilege/Privilege Escalation:**

    - https://cwe.mitre.org/data/definitions/250.html
    - https://cwe.mitre.org/data/definitions/269.html

- **Exceptional Conditionals:** https://cwe.mitre.org/data/definitions/703.html

Most of the time, one vulnerability results in an another one. For example, an integer overflow might cause a buffer overflow, as well as a double fetch could open an opportunity for a buffer overflow. Furthermore, readers should remember about the involved context while referring to memory.

In this article, as we are discussing kernel driver, the word "memory" could mean stack, **NonPagedPool, NonPagedPoolNx** (the recommended non-paged memory), and **PagedPoolSession**, depending on the code being analyzed.

Microsoft has been adding a series of protections of the time which are focused on preventing exploitation from the kernel side, and a limited list of them is:

- **Kernel Space Address Randomization (KASLR):**

    - https://www.offsec.com/vulndev/development-of-a-new-windows-10-kaslr-bypass-in-one-windbg-command/

- **Driver Signature Enforcement (DSE):**

    - https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing

- **Supervisor Mode Execution Protection (SMEP):**

    - https://www.microsoft.com/en-us/security/blog/2017/03/27/detecting-and-mitigating-elevation-of-privilege-exploit-for-cve-2017-0005/
    - https://j00ru.vexillium.org/2011/06/smep-what-is-it-and-how-to-beat-it-on-windows/
    - https://www.coresecurity.com/sites/default/files/2020-06/Windows%20SMEP%20bypass%20U%20equals%20S_0.pdf

- **Supervisor Mode Access Prevention:**

    - https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Evaluating%20the%20feasibility%20of%20enabling%20SMAP%20for%20the%20Windows%20kernel.pdf
    - https://lwn.net/Articles/517475/

- **Virtualization-based Security (VBS):**

    - https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

- **Kernel Data Protection (KDP):**

    - https://www.microsoft.com/en-us/security/blog/2020/07/08/introducing-kernel-data-protection-a-new-platform-security-technology-for-preventing-data-corruption/

- **Memory Integrity or Hypervisor Protected Code Integrity (HVCI):**

    - https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-hvci-enablement

- **Kernel DMA Protection**:

    - https://learn.microsoft.com/en-us/windows/security/hardware-security/kernel-dma-protection-for-thunderbolt

- **Microsoft Vulnerable Driver Blocklist:**

- https://learn.microsoft.com/en-9us/windows/security/application-security/application-control/windows-defender-application-control/design/microsoft-recommended-driver-block-rules

Obviously, I am not including the already existing **Kernel Patch Protection (KPP, that is also known as Patch Guard)** and **HyperGuard**, which readers can read two articles written by my colleague **Yarden Shafir (@yarden_shafir)**:

- https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-1-skpg-initialization/
- https://windows-internals.com/hyperguard-secure-kernel-patch-guard-part-2-skpg-extents/

Returning to the kernel drivers' topic, I would like to leave further considerations:

- Drivers usually present issues related to returned values because most drivers do not evaluate them and do not check whether the result is NULL or not, so it is advisable to check them.

- Never trust on passed arguments because values can have changed along the code and depending on APIs.

- Drivers using **Direct I/O approach (DeviceObject → Flags |= DO_DIRECT_IO)** use MDL, which works in a double mapping to memory, as I mentioned previously, and double mapping might involve vulnerabilities as double fetching (I left about this class of vulnerability on previous pages).

- Using a pointer that supposedly refers to a type of object might cause a **type-confusion vulnerability**, so if the driver code does not evaluate the type or even if the pointer is NULL, there is an opportunity of exploitation.

- A lack of reference counting control might cause a **UAF** or even a **Double Free** vulnerability, and mainly because these controls are usually very away each other and because there are situations where the code might present unexpected decisions.

- It is highly relevant to underscore that since Windows 10 19H1 pool allocations has changed and migrated to Segment Heap (Low Fragmentation Heap + Variable Size + Heap Backend + Large Block Allocation), even though concepts like **NonPaged, NonPagedNx** and **Paged pool** have been kept. Eventually this topic will be detailed when necessary to understand a specific context and concept, and it an interesting theme to be included in next articles.

- Drivers (mainly device drivers) can cancel individual IRPs by invoking **IoCancelIrp** function, which are associated with threads, and have not been processed yet. When cancelling an IRP is mentioned, this action does not "drop" the IRP immediately, but the I/O manager grants a limited time of few minutes to the IRP to be completed before it being considered timed out. In other words, IRP requests should be completed as soon as possible, and the driver should have a cancelation routine to prevent an IRP staying in the queue forever. Additionally, drivers can have and implement their own Cancel routines (**DRIVER_CANCEL callback**) that are invoked by the I/On Manager once the **IoCancelIrp routine** is called by the driver, and it is the **Cancel routine** that completes user cancelled I/O requests.

- In untold opportunities drivers present double fetch vulnerabilities, which happens when the same buffer (for example) is accessed and checked/verified almost in sequence. However, status and conditions might change between the first and the second access/verification (typical race condition vulnerability), and such changes could cause bugs like buffer overflow and integer overflow. Double fetching, which is a subset of race condition class, is also a possibility.

- Kernel drivers can monitor, by registering for notifications, whether processes and threads are created and, consequently, act through callbacks. In this case, readers will be typical functions, callback's prototype, and structures such as:

    - **PsSetCreateProcessNotifyRoutineEx (PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine, BOOLEAN Remove):** this routine registers a list of callbacks for receiving process notification such as creation or deletion.

    - **PCREATE_PROCESS_NOTIFY_ROUTINE[_EX] callback:** It is a callback routine implemented to notify the caller when a process is created or even exits.

```
PCREATE_PROCESS_NOTIFY_ROUTINE PcreateProcessNotifyRoutine;

void PcreateProcessNotifyRoutine(
  HANDLE ParentId,
  HANDLE ProcessId,
  BOOLEAN Create
)
{...}
```

**[Figure 38] PCREATE_PROCESS_NOTIFY_ROUTINE**

    - **PS_CREATE_NOTIFY_INFO structure:** it is a structure used in **ProcessNotifyCallback prototype**.

    - **PsSetCreateThreadNotifyRoutine(PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine):** this routine is responsible for registering a callback that later will be notified and invoked when a new thread to be created and even deleted.

    - **PCREATE_THREAD_NOTIFY_ROUTINE** callback: it is a callback routine that is implemented and invoked to notify the caller when a thread is created or deleted.

```
PCREATE_THREAD_NOTIFY_ROUTINE PcreateThreadNotifyRoutine;

void PcreateThreadNotifyRoutine(
  HANDLE ProcessId,
  HANDLE ThreadId,
  BOOLEAN Create
)
{...}
```

**[Figure 39] PCREATE_THREAD_NOTIFY_ROUTINE**

- At the same way, there are other kind of notifications and callbacks based on image load notification (**PSSetLoadImageNotifyRoutine, PLOAD_IMAGE_NOTIFY_ROUTINE callback function** and **_IMAGE_INFO structure**) and Registry notification (**CmRegisterCallbackEx, EX_CALLBACK_FUNCTION callback function**), for example.

- There is a last possible and quite interesting approach that the drivers perform the registration to receive a notification when a handle for a specific object is opened or duplicated, and we have **ObRegisterCallbacks function** and **OB_CALLBACK_REGISTRATION structure** involved in this process. Such a structure takes us to the **OB_OPERATION_REGISTRATION structure**, which is associated to two different possible callbacks instead of only one: **POB_PRE_OPERATION_CALLBACK** callback function and **POB_POST_OPERATION_CALLBACK** callback function that, as the name suggests, are invoked before and after an operation (**OB_OPERATION_HANDLE_CREATE, OB_OPERATION_HANDLE_DUPLICATED**) occurs, respectively.

During a driver analysis, there is a list of points to pay attention and how to proceed to collect vital information:

- Check the driver using **DeviceTree tool** and write down information **Device Name, Device Flags, Device Characteristics**, and mainly **Security Attributes** to confirm who can access the device and which permissions are bound to users and groups. The first point is to verify **Everyone** and **Authenticated Users group permissions**, and excessive permissions might offer us a better perspective of locating vulnerabilities and even exploiting the vulnerability.

- Always verify whether the driver and its functions are evaluating a possible failed invocation, if they are evaluating a potential returned value as NULL and whether there are wrong types being passed to functions (most common than you can imagine due to returned pointers to buffers on memory). These issues usually represent a quite relevant source of vulnerabilities, mainly when there are associated memory related functions such as **ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePool2** and **ExAllocatePool3**. Additionally, vulnerabilities classes like *buffer overflow* and *type confusion* can come up easily from these problems.

- In specific, type confusion vulnerability may arise from **ObReferenceObjectByHandle function**, which provides validation on a given object handles and should return a corresponding pointer, but eventually a NULL appears as return, which would be used later by other routines and functions. I suggest that readers check such function on the Microsoft website (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obreferenceobjectbyhandle) and pay attention to its third argument (**ObjectType**).

- Search for a possible associated symbolic link, which is an argument of **IoCreateSymbolicLink** and **IoDeleteSymbolicLink** functions. The symbolic link is quite valuable artifact for writing a quick client program to interact with the driver.

- Open the kernel driver on IDA Pro and search for routines such as **DriverEntry, CreateFile, IoCreateDevice/IoCreateDeviceSecure** and **IoControlDevice**. Once they are found, identify, and

interpret their arguments. For example, it would be interesting to verify whether *FILE_DEVICE_SECURE_OPEN* flag has been specified or not as component of **DeviceCharacteristics** parameter of **IoCreateDevice** function. **DeviceTree** tool is great for gathering such information.

- Identify the dispatch function and major functions (dispatch routines) such as **IRP_CREATE, IRP_MJ_CLOSE, IRP_MJ_WRITE, IRP_MJ_READ** and, of course, **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL**.

- Search for **IOCTL codes** and decode them. IOCTL code is an essential artifact and provide us with guidelines about how to navigate in the kernel driver's code through of function, method and access information built in each of those IOCTL codes and understand what the context is and involved operations. Mainly, using the same mentioned information, they offer directions about how we can interact with the driver.

- Search and analyze the **IoGetCurrentIrpStackLocation** function, which accepts an IRP as argument and as expected, returns the location of the current **IO_STACK_LOCATION**. The IRP structure offers crucial details to understand what further operations and data we should expect from this driver.

- IRPs are passed down to another driver by using **IoCallDriver function**, so you should check whether this call exists in the code and try to understand the respective data flow.

- Check for functions involved with memory allocation and manipulation like **MmMapIoSpace** (responsible for mapping a provided physical address range to a non-paged system space), **IoAllocateMdl** (responsible for allocating an MDL) and **ZwMapViewOfSection** (responsible for mapping a view of section into the virtual address space).

- In special, as **MmMapIoSpace** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmapiospace) potentially "copies" user buffer's content (from the process) into the kernel space as a non-paged system, so if its parameter somehow can be controlled then we might have, at the limit, a **write-what-where primitive** on our hands (remember **CWE-123** mentioned previously).

- It would be unnecessary to mention, but equivalent functions to undo actions such as **ZwUnmapViewOfSection** and **MmUnMapIoSpace** also provide us with possibilities of vulnerability and eventually exploitation because we have the same memory's pointers from the allocation involved in both functions, obviously.

- Other two functions that also handle memory in the kernel side and are used by device drivers are **MmGetPhysicalAddress**, which given the non-paged virtual address, it returns the physical address (exactly the reverse of **MmMapIoSpace function**), and **MmAllocateContiguousMemory** (typically called in the **DriverEntry routine**), which allocates a range of contiguous and non-paged physical memory and maps it to the system address space. About the **MmAllocateContiguousMemory** function, it is quite appropriate to emphasize that this physical memory region is being brough into

the virtual address space of the kernel, which may open opportunities for an arbitrary write vulnerability.

- Another critical and memory related function is **MmGetSystemAddressForMdl**, which returns a non-paged memory pool for the buffer described by the MDL, and that driver should examine this returned pointer and check whether is not NULL. As recommended by Microsoft, drivers must use **MmGetSystemAddressForMdlSafe** instead. At the same way, there can be a possibility of finding a vulnerability here.

- **IRP cancellation** (quickly described previously), regarding that different things can happen between the cancellation order and the actual timeout of the IRP, might cause vulnerabilities issues such as double free, UAF and race conditions.

- Write primitives can arise due to unnecessary exposure of MSR registers, which are control registers provided by the processor implementation for range of goals such as debugging, tracing, performance and even enabling processor features. In our case, our focus are registers like **wrmsr** and **rdmsr (Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 4 --** https://cdrdv2.intel.com/v1/dl/getContent/671200), which are involved with system calls through the **MSR _LSTAR register** that holds a function pointer that is called one a system call occurs, and this function pointer might point to an arbitrary code to be executed whether we could change it. Readers can find occurrences of these registers by using its search options.

- Do not forget to check all references to these functions. Most of the time what we are looking for is around such these cross-references.

I will not be reviewing concepts about **KDM**, which is a high-level interface over **MDM** (and has the same security issues) and **mini filter drivers**, which I already explained in good details in my previous article (https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/).

## 11. Analysis of binary differences

Returning to analysis of **srv2.sys driver** and its respective versions and binary diffing, I am going to do a quick analysis of binary diffing and also check the reversed codes on IDA Pro. However, these are only a few of the available steps that are necessary while investigating likely security issues. In general, a recommended procedure is:

- Learn fundamentals about the target (kernel driver and mini-filter drivers in your case).
- Search for involved binary's versions and execute follow-on preparation.
- Perform binary diffing on the target drivers.
- Analyze the binary diffing and try to catch any initial security issue.
- Do reverse engineering of one or both files used to binary diffing.
- Prepare one or two virtual machines with exactly the vulnerable driver's version.

- Interact with the driver using different methodologies and tools.
- A usual and good approach is, if the driver is not so complex, try to write a client program (**C/C++, cPython** and **PowerShell**) to submit requests to the driver.
- Using WinDbg, try to set up breakpoints on key routines and functions of the vulnerable driver and find ways to "activate them" (hit them) using daily operations from the own operating system.
- Study the driver and the entire involved subsystem in depth. It takes a long time, but it is worth because you will have better conditions to find other vulnerabilities that have not been found previously.
- Perform fuzzing on the driver.
- Repeat part of this procedure to each new finding.
- **Note 1:** if there is one vulnerability in a piece of code, odds of finding new ones are considerable.
- **Note 2**: it is not because a code has been analyzed by other professionals that do not exist additional vulnerabilities there.

Obviously, this is an incomplete procedure (there are additional steps), but it could offer a small number of directions to readers to know what could be done over the period of the investigation. In this article I will not go too far because I would like to comment about these topics in the next articles of this series, but it is still fair to provide readers with this initial information in this article.

Before we proceeding to the binary diffing analysis, there are two interesting IDA Pro plugins (the second one I have already mentioned previously in this text) that can be useful for readers in different analysis and contexts beyond this article too:

- **DriverBuddyReloaded:** https://github.com/VoidSec/DriverBuddyReloaded
- **MsdocViewer:** https://github.com/alexander-hanel/msdocsviewer.git

**DriverBuddyReloaded** has been written by **Paolo Stagno** (**Twitter/X: @Void_Sec**) and offers interesting resources such as decoding one or all IOCTL codes present in the driver, detecting potential and usual vulnerable functions, finding opcodes in data sections and much more. Once of conditions to install such a plugin is that is necessarily have **IDA Pro 7.5 or newer**. To install it:

1. Make sure that **IDA Python** is using the same Python version that your system:

    - Open **IDA Pro 8.3** and execute the following commands in the **IDA Python prompt**:

        - **import sys**
        - **sys.version**

    - Open a Terminal / Command Prompt and execute **python -V**
    - If versions do not match, it is necessary to adjust the IDA Python version by executing **idapyswitch.exe** (from **C:\Program Files\IDA Pro 8.3** folder) and choosing the correct Python version.

2. Install the plugin:

    a. Clone the **DriverBuddyReloaded** plugin: **git clone** https://github.com/VoidSec/DriverBuddyReloaded

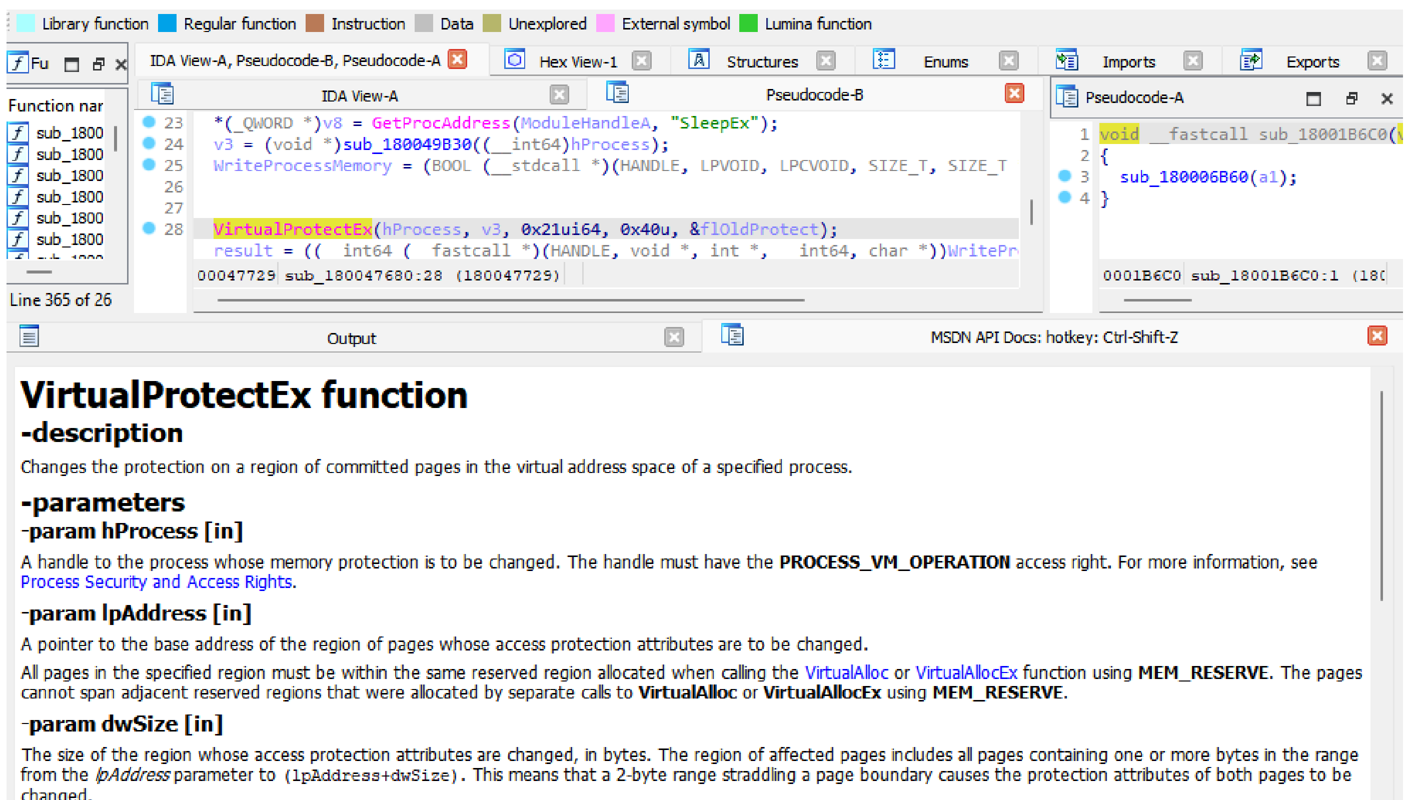
b. Or download it: https://github.com/VoidSec/DriverBuddyReloaded/releases/download/1.6/DriverBuddyReloaded.zip

c. Copy the **DriverBuddyReloaded.py** and **DriverBuddyReloaded folder** to **C:\Program Files\IDA Pro 8.3\plugins\ folder** or **%APPDATA%\Hex-Rays\IDA Pro\plugins** folder.

**Msdocviewer** is a tool created by **Alexander Hanel (Twitter/X: @nullandnull)** consists of two parts, which the first one is the **run_me_first.py script** that searches for all markdown files in Microsoft repositories, checks whether the document is related to a function, copies the document to a directory and then renames the file with their corresponding API name.

The second part is an IDA plugin (**ida_plugin/msdocviewida.py**) that displays the document in IDA. To install it, execute:

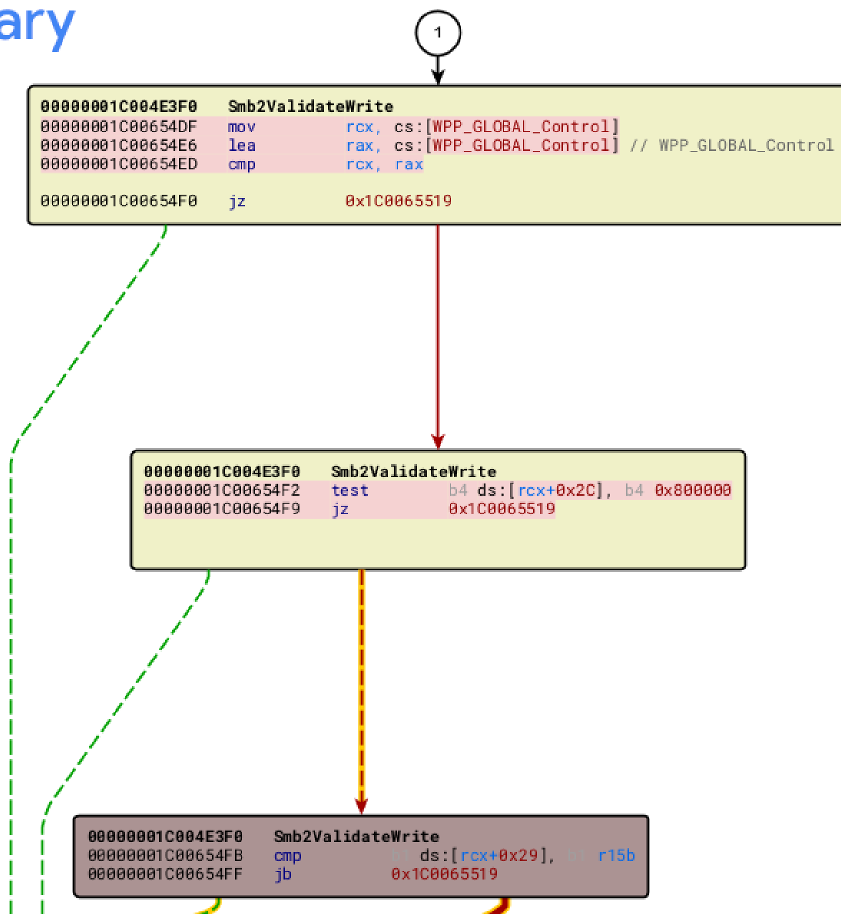
- **git clone** https://github.com/alexander-hanel/msdocsviewer.git
- **cd msdocviewer**
- **git submodule update --init –recursive**
- **python run_me_first.py**
- Edit **ida_plugin/msdocviewida.py** and add the directory path of **apis_md** to the **API_MD** variable (currently on line 18). In my case: **API_MD = r"c:\github\msdocsviewer\apis_md"**
- Copy **msdocviewida.py** to the IDA plugin directory:
  - **cp ida_plugin\msdocviewida.py "C:\Program Files\IDA Pro 8.3\plugins"**
- If all steps have been executed correctly, open a binary on IDA Pro, go to disassembly or pseudo-code, put the mouse's pointer on a Windows API and press **CTRL+SHIFT+Z**:




**[Figure 40] Msdocviewer plugin**

As I have emphasized previously, the goal of this article is to show techniques and procedures to allow readers to make their own analysis, so I am not concerned about interpreting vulnerabilities or even writing exploits right now.

If readers remember about our binary diffing section, we had two functions that are different in **srv2.sys** from AUG/2022 when compared to its version from JUL/2022: **Smb2ValidateWrite** and **Smb2QueryFileNormalizedName**. Refreshing details that we had viewed about **Smb2ValidateWrite**, we have:




**[Figure 41-A] BinDiff: Smb2ValidateWrite**

In terms of Assembly code from IDA Pro view, we have:

```
PAGE:00000001C00654DF loc_1C00654DF:                          ; CODE XREF: Smb2ValidateWrite+1C5↑j
PAGE:00000001C00654DF                 mov     rcx, cs:WPP_GLOBAL_Control
PAGE:00000001C00654E6                 lea     rax, WPP_GLOBAL_Control
PAGE:00000001C00654ED                 cmp     rcx, rax
PAGE:00000001C00654F0                 jz      short loc_1C0065519
PAGE:00000001C00654F2                 test    dword ptr [rcx+2Ch], 800000h
PAGE:00000001C00654F9                 jz      short loc_1C0065519
PAGE:00000001C00654FB                 cmp     [rcx+29h], r15b
PAGE:00000001C00654FF                 jb      short loc_1C0065519
PAGE:00000001C0065501                 mov     rcx, [rcx+18h]
```

**[Figure 41-B] IDA Pro: Smb2ValidateWrite**

In terms of pseudo code (it is recommended to synchronize **IDA-View** and **Pseudocode tabs**) IDA Pro resolves the structure and shows the following code:

```
623 LABEL_19:
624         if ( v17 > *(_DWORD *)(v4 + 36) )
625         {
626           if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
627             && (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x800000) != 0
628             && BYTE1(WPP_GLOBAL_Control->Timer) )
629           {
630             WPP_SF_q(
631               (__int64)WPP_GLOBAL_Control->AttachedDevice,
632               0x2Au,
633               (__int64)&WPP_e4ca1247873838990567653 8f824753d_Traceguids,
634               BugCheckParameter4);
635           }
```

**[Figure 42] IDA Pro: Smb2ValidateWrite (pseudo code)**

No doubt the pseudo code seems better, and considerations and questions come up:

- What is **WPP**?
- What is the **WPP_GLOBAL_Control structure**?

**WPP** means **Windows Software Trace Preprocessor** and it is used to trace operations of a software component as an application, and user-mode or kernel-mode driver. It helps to improve the tracing and it is very useful for debugging code through an approach like Windows event logging services.

**WPP_GLOBAL_Control** is a pointer to structure of type **_DEVICE_OBJECT**, which you have already seen in the previous article, and its first fields are shown below:

```
typedef struct _DEVICE_OBJECT {
  CSHORT                      Type;
  USHORT                      Size;
  LONG                        ReferenceCount;
  struct _DRIVER_OBJECT       *DriverObject;
  struct _DEVICE_OBJECT       *NextDevice;
  struct _DEVICE_OBJECT       *AttachedDevice;
  struct _IRP                 *CurrentIrp;
  PIO_TIMER                   Timer;
  ULONG                       Flags;
  ULONG                       Characteristics;
  __volatile PVPB             Vpb;
  PVOID                       DeviceExtension;
  DEVICE_TYPE                 DeviceType;
  CCHAR                       StackSize;
  union {
    LIST_ENTRY           ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
```

**[Figure 43] _DEVICE_OBJECT structure (first fields only)**

According to **Microsoft Learn**, **the Timer field** holds a pointer to a timer object (**IO_TIMER**), and it allows the I/O manager to call a timer routine every second. Furthermore, **it is read/write member**.

IDA Pro shows us **HIDWORD** and **BYTE1** macros, which are used to access smaller parts of a variable:

- **HIDWORD:** returns the high part of a DWORD (double word).
- **BYTE1:** returns the second byte of a given data in memory.

These macros are defined in **C:\Program Files\IDA Pro 8.3\plugins\hexrays_sdk\defs.h**, where readers can see the following:

```
// first unsigned macros:
#define BYTEn(x, n)   (*((_BYTE*)&(x)+n))
#define WORDn(x, n)   (*((_WORD*)&(x)+n))
#define DWORDn(x, n)  (*((_DWORD*)&(x)+n))

#define LOBYTE(x)  BYTEn(x,LOW_IND(x,_BYTE))
#define LOWORD(x)  WORDn(x,LOW_IND(x,_WORD))
#define LODWORD(x) DWORDn(x,LOW_IND(x,_DWORD))
#define HIBYTE(x)  BYTEn(x,HIGH_IND(x,_BYTE))
#define HIWORD(x)  WORDn(x,HIGH_IND(x,_WORD))
#define HIDWORD(x) DWORDn(x,HIGH_IND(x,_DWORD))
#define BYTE1(x)   BYTEn(x,  1)         // byte 1 (counting from 0)
#define BYTE2(x)   BYTEn(x,  2)
#define BYTE3(x)   BYTEn(x,  3)
#define BYTE4(x)   BYTEn(x,  4)
```

**[Figure 44] Macros definition in defs.h**

A last thing to observe is the prototype of the **Smb2ValidateWrite** function:

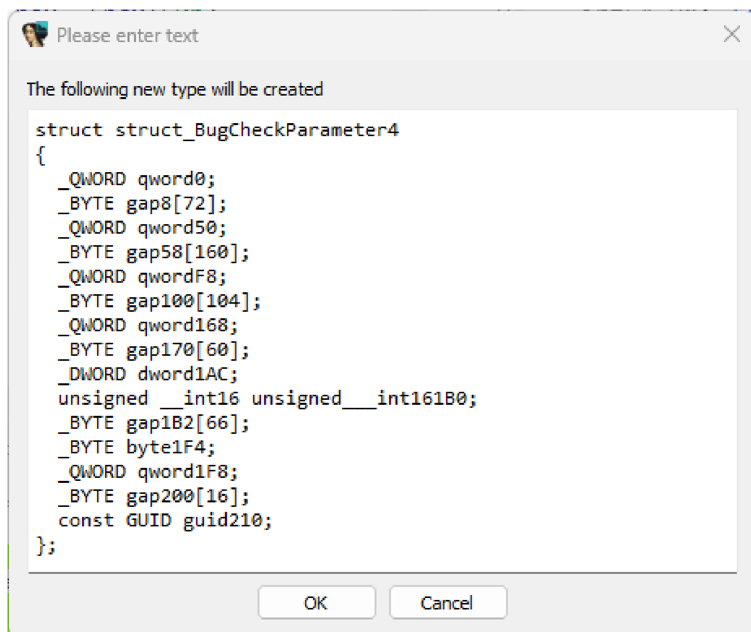
- **__int64 __fastcall Smb2ValidateWrite(__int64 BugCheckParameter4)**

The **__fastcall convention** specifies that four arguments are passed into registers RCX, RDX, R8 and R9, and the remaining one are passed on the stack, and the prototype indicates that this function accepts one argument. If we check the respective Assembly code, we have:

```
PAGE:00000001C004E3F0 ; __int64 __fastcall Smb2ValidateWrite(__int64 BugCheckParameter4)
PAGE:00000001C004E3F0 Smb2ValidateWrite proc near             ; DATA XREF: .rdata:00000€
PAGE:00000001C004E3F0                                         ; .rdata:00000001C0038D081
PAGE:00000001C004E3F0
PAGE:00000001C004E3F0 var_E8          = qword ptr -0E8h
PAGE:00000001C004E3F0 var_E0          = qword ptr -0E0h
PAGE:00000001C004E3F0 var_D8          = qword ptr -0D8h
PAGE:00000001C004E3F0 var_D0          = qword ptr -0D0h
PAGE:00000001C004E3F0 var_C8          = word ptr -0C8h
PAGE:00000001C004E3F0 var_C0          = word ptr -0C0h
PAGE:00000001C004E3F0 var_B8          = dword ptr -0B8h
PAGE:00000001C004E3F0 var_B0          = dword ptr -0B0h
PAGE:00000001C004E3F0 var_A8          = qword ptr -0A8h
PAGE:00000001C004E3F0 var_A0          = qword ptr -0A0h
PAGE:00000001C004E3F0 var_98          = dword ptr -98h
PAGE:00000001C004E3F0 var_90          = dword ptr -90h
PAGE:00000001C004E3F0 var_88          = word ptr -88h
PAGE:00000001C004E3F0 var_80          = word ptr -80h
PAGE:00000001C004E3F0 var_78          = dword ptr -78h
PAGE:00000001C004E3F0 var_70          = qword ptr -70h
PAGE:00000001C004E3F0 var_68          = qword ptr -68h
PAGE:00000001C004E3F0 var_60          = qword ptr -60h
PAGE:00000001C004E3F0 var_58          = qword ptr -58h
PAGE:00000001C004E3F0 var_48          = xmmword ptr -48h
PAGE:00000001C004E3F0 var_38          = qword ptr -38h
PAGE:00000001C004E3F0 arg_0           = dword ptr  8
PAGE:00000001C004E3F0 arg_8           = qword ptr  10h
PAGE:00000001C004E3F0 arg_10          = qword ptr  18h
PAGE:00000001C004E3F0 arg_18          = qword ptr  20h
```

**[Figure 45] Smb2ValidateWrite: assembly code**

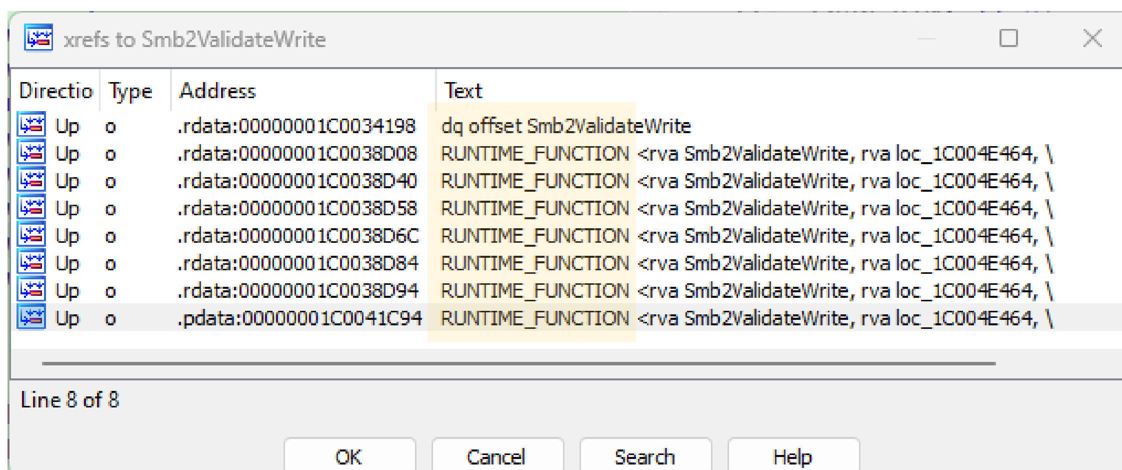
This function would have apparently four arguments, even though the decompiler has assigned only one, and this one is usually used as a **BugCheckParameter**, which is used to inform to callbacks the bug check parameters that were passed to **KeBugcheckEx function**. However, we do not have any information here and, at least for this function, **BugCheckParameter4** represents a structure, which we don't know the associated type and anything else, and initially we can click on this parameter and create a new structure by **right clicking** on the choosing **Create a new structure type**, and IDA Pro will show us the following:




**[Figure 46] Create a new structure type**

Honestly, I do not like this approach because of gap fields, and I prefer to create an array with multiple DWORD/QWORD elements, but right now it helps to improve the code, even though we do not know names and context of each of its fields. You can repeat it with other structures throughout the function.

Checking cross-references, we have:




**[Figure 47] Smb2ValidateWrite: cross references**

**RUNTIME_FUNCTION** is a structure used for exception handling and stack unwinding. This structure represents a table-based exception handling entry with three fields: function start address, function end

address and unwind info address, and you will usually see **UNWIND_INFO** data info structure together, which is used to record associated effects that a function has on the stack pointer and also the location that nonvolatile registers are saved (stored) on the stack.

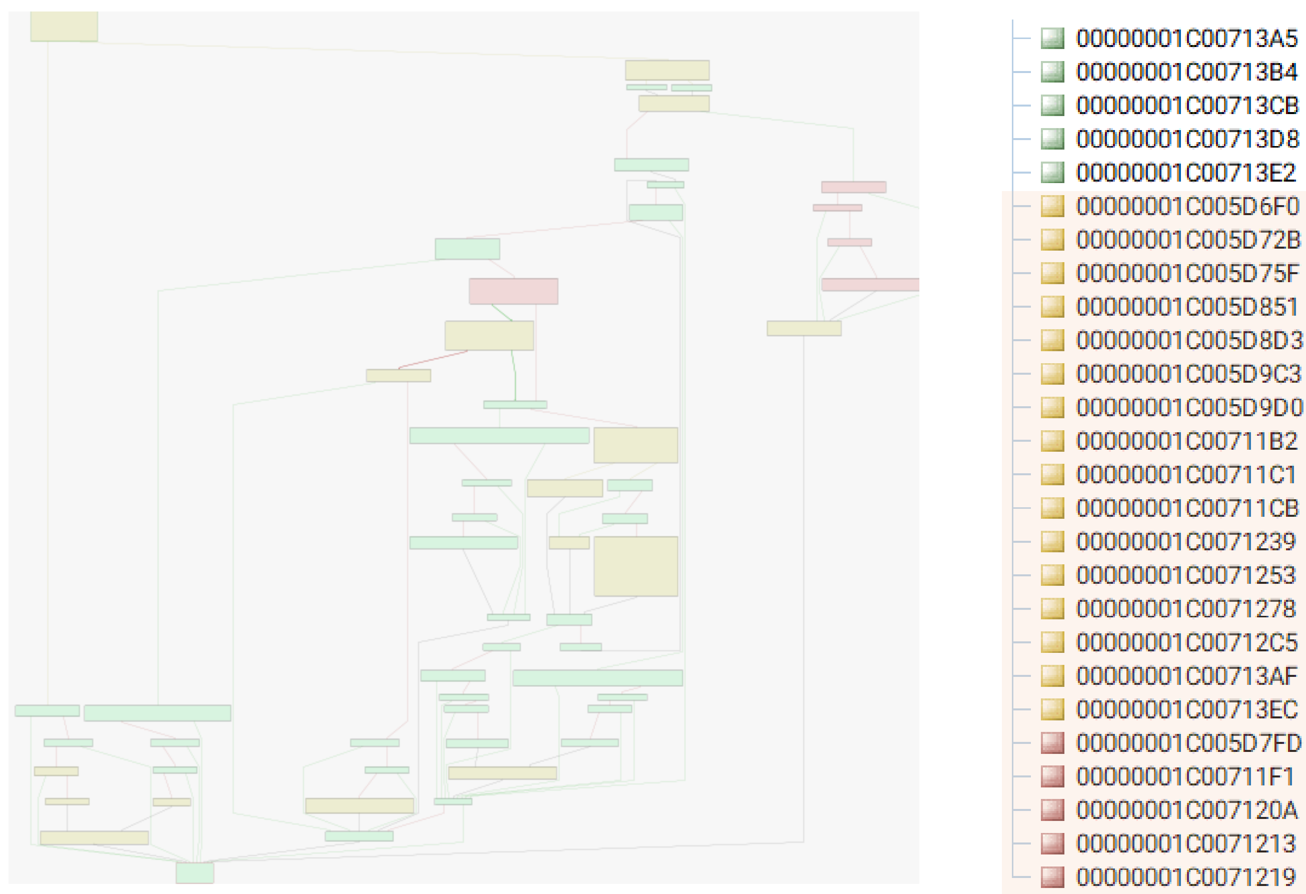
If readers list the functions (**CTRL+F3**) called by **Smb2ValidateWrite**, you are going to see the following ones: **Smb2VerifyFileEx**, **Smb2VerifySessionExEx**, **Smb2VerifyTreeConnect** and **_Smb2SetError**. Although we are not analyzing details right now, this is essential information for deeper analysis in future articles.

Examining the code, we find strings like *"onecore\\base\\fs\\remotefs\\smb\\srv\\srv.v2\\smb2\\write.c"*, which indicates that this function is really related to SMB2 writing operations.

The binary diffing of **Smb2ValidateWrite** function has shown only a few details until now, and only with these facts we do not have means to state anything precise, and it might be that:

- The change is related to performance tracing and debugging (highly likely).
- The change is related to any kind of race condition (very unlikely).

It is time to get a quick view of **Smb2QueryFileNormalizedName function**, which presents many functions' changes between both versions of **srv2.sys**, which a few of blocks did not exist previously and other ones have been patched, as shown below (AUG/2022 – newer version):




**[Figure 48] BinDiff: overview of changed functions and its respective list**

**BinDiff** continues to be one of the most used and powerful tools to find vulnerabilities. However, there is another great option, and it is time to demonstrate **Diaphora**, which offers excellent features.

Right clicking on **Smb2QueryFileNormalizedName function** (**Interesting Matches tab** from **Diaphora**), readers will get the following, as I already shown before for this same function:

```
>AttachedDevice, v24, (__int64)&WPP_e8ea6266aa02331ec1234de41a53c1d4_Traceguids, a1);
51       }
52   }
53   else
54   {
55     ExAcquireResourceSharedLite((PERESOURCE)(*(_QWORD *)v5 + 80i64), 1u);
56     if ( *(_BYTE *)(*(_QWORD *)(v1 + 48) + 137i64) )
57       v6 = *(_QWORD *)(v5 + 280);
58     else
59       v6 = *(_QWORD *)(v5 + 272);
60     v33 = v6;
61     if ( Smb2ValidateVolumeObjectsMatch(a1, v5) )
62     {
63       ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 80i64));
64       v7 = 0;
65       v31 = 0;
66       while ( 1 )
67       {
68         if ( v7 )
69           goto LABEL_21;
70         v30 = 0;
71         v2 = Smb2PopulateShareNormalizedNameCache(a1, v6, &v30);
72         if ( (v2 & 0x80000000) != 0 )
73         {
74           v25 = WPP_GLOBAL_Control;
75           if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
76             && (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) != 0
77             && BYTE1(WPP_GLOBAL_Control->Timer) )
78           {
79             v26 = 38;
80             LODWORD(FileInformationClass) = v2;
81LABEL_61:
82             WPP_SF_qd(
83               (__int64)v25->AttachedDevice,
84               v26,
85               (__int64)&WPP_e8ea6266aa02331ec1234de41a53c1d4_Traceguids,
86               a1,
87               FileInformationClass);
```

```
>AttachedDevice, v7, (__int64)&WPP_6285cecdc58f32c64ce6ba6b855a6046_Traceguids, a1)
53     }
54     return v2;
55   }
56   ExAcquireResourceSharedLite((PERESOURCE)(*(_QWORD *)v5 + 80i64), 1u);
57   if ( *(_BYTE *)(*(_QWORD *)(v1 + 48) + 137i64) )
58     v8 = *(_QWORD *)(v5 + 280);
59   else
60     v8 = *(_QWORD *)(v5 + 272);
61   v35 = v8;
62   if ( (unsigned int)Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled() )
63   {
64     if ( !Smb2ValidateVolumeObjectsMatch(a1, v5) )
65     {
66       v9 = WPP_GLOBAL_Control;
67       if ( WPP_GLOBAL_Control == (PDEVICE_OBJECT)&WPP_GLOBAL_Control
68         || (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) == 0
69         || !BYTE1(WPP_GLOBAL_Control->Timer) )
70       {
71         goto LABEL_22;
```

**[Figure 49] Diaphora – pseudo code diffing (truncated list)**

Although I have truncated the comparison above, it is quite recommended that you pay attention and make note of potential and relevant changes. However, there are good considerations here:

- In srv2.sys, there is a list of five addresses that seemingly (check next sentences) did not exist in the srv2.sys's previous version (JUL/2022) and other 14 functions that have changed.
- Do not assume that blocks have been added or removed because, in innumerable opportunities, they might have been just relocated or reordered over the code.
- Do not focus only on new blocks in the updated version of the driver, but mainly on existing ones from the previous version because it is usually where vulnerabilities have been patched.
- It is always much better to open different IDA Pro instances focusing on the same function (**Smb2QueryFileNormalizedName**) for both versions of functions (older and newer one) and follow the code according to shown by Diaphora, making every single possible note because, according to my experience, there will be useful later.
- It is advisable to examine the code, change and add types, structures, and enumeration you already know to produce a better code to analyze (mainly whether you are using IDA Pro pseudo code).
- Probably you will not have the appropriate context about what is happening, then it is time to examine the entire driver searching and interpreting **critical routines, symbolic links, dispatch functions, permissions, IO_STACK_LOCATION**, and so on.

- Certainly, type confusion and race conditions are harder to spot at the first moment, but you will find them too.

- There is a large list of functions to pay attention to while analyzing Windows C/C++ programs and driver, in special. A concise list of them follows below:

- **CopyMemory**
- **EnterCriticalSection**
- **ExAcquireResourceSharedLite**
- **ExAllocatePool**
- **ExAllocatePool2**
- **ExAllocatePoolWithTag**
- **ExFreePoolWithTag**
- **ExReleaseResourceLite**
- **IoRegisterDeviceInterface**
- **IofCallDriver**
- **MmAdvanceMdl**
- **MmAllocateContiguousMemory**
- **MmAllocateContiguousMemoryEx**
- **MmAllocateMappingAddress**
- **MmAllocateMappingAddressEx**
- **MmAllocateMdlForIoSpace**
- **MmAllocatePagesForMdl**
- **MmAllocatePagesForMdlEx**
- **MmBuildMdlForNonPagedPool**
- **MmGetSystemAddressForMdl**
- **MmGetSystemRoutineAddress**
- **MmGetSystemRoutineAddressEx**
- **MmLockPagableCodeSection**
- **MmLockPagableDataSection**
- **MmMapIoSpace**
- **MmMapIoSpaceEx**
- **MmMapLockedPages**
- **MmMapMdl**
- **MmProtectDriverSection**
- **MmProtectMdlSystemAddress**
- **MmQuerySystemSize**
- **MmSizeOfMdl**
- **MmUnlockPagableImageSection**
- **MmUnlockPages**
- **MmUnmapIoSpace**
- **MmUnmapLockedPages**
- **ObGetObjectSecurity**
- **ObReferenceObjectByPointer**
- **ObRegisterCallbacks**
- **ObfReferenceObject**
- **PsCreateSystemThread**
- **RtlCopyMemory**
- **RtlMoveMemory**
- **RtlSecureZeroMemory**
- **RtlZeroMemory**
- **ZwUnloadDriver**
- **ZwUnmapViewOfSection**
- **ZwWriteFile**
- **_mbslen**
- **_mbstrlen**
- **_memccpy**
- **_snprintf**
- **_sntprintf**
- **fopen**
- **lstrlen**
- **memcpy**
- **memmove**
- **realloc**
- **sprintf**
- **sprintfA**
- **sprintfW**
- **strlen**
- **swprintf**
- **wmemcpy**
- **wmemmove**
- **wnsprintf**
- **wnsprintfA**
- **wnsprintfW**
- **wsprintf**
- **wsprintfA**
- **wsprintfW**
- **wvnsprintf**
- **wvnsprintfA**
- **wvnsprintfW**
- **wvsprintf**
- **wvsprintfA**
- **wvsprintfW**
- **wcslen**

**[Figure 50] A concise list of principal functions that can cause vulnerabilities**

- As I reaffirmed previously, I will not spot bugs or vulnerabilities in this article, but based on the list above, few instructions, functions, and routines from the old version of srv2.sys might be

interesting to take a first look regardless of the Microsoft report about the vulnerability and absolutely without entering in any detail or interpretation.

- Once again, if you need any structure and even check for unknown functions, this is a compact list of options that you have:

  - **Virgilius project:** https://www.vergiliusproject.com/
  - **Phnt:** https://github.com/winsiderss/phnt
  - **NtDoc:** https://ntdoc.m417z.com/
  - **ReactOS:** https://github.com/reactos/reactos
  - **ReactOS docs:** https://doxygen.reactos.org/index.html
  - **Windows SDK:** C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\km (but not only).

- Another point that you are going to learn quickly is that static analysis can have serious limitations and, in different opportunities, you will have to use a debug to understand what is going on. I will talk about it in future articles.

- Certainly, analyzing pseudo code is easier than the respective assembly code, but sometimes the pseudo code will not show a good picture and precise code, so you will have to conduct the analysis through the assembly code.

- Another hint is to use the **graphical mode of IDA** (activated by **TAB key**) to understand the coverage and flow of execution of the code.

- Remember that the most vulnerabilities come from the following classes:

  - **Null-dereference**
  - **Buffer overflow**
  - **UAF**
  - **Double free**
  - **Integer Overflow**
  - **Out-of-bounds write/read**
  - **Off-by-one error**

- One of most common errors you will see is that codes either do not verify that arguments of functions, as pointers, are null or do not check whether their returned are a valid value.

- Regardless of whether there is or not an issue in the code that we are quickly analyzing, there are still relevant points that might deserve attention while analyzing code.

- To these first comments I will use the following pieces of code of srv2.sys from the old (vulnerable) and updated versions of our comparison:

```
10    if ( *(_DWORD *)(v1 + 0xBC) < 4u )
11    {
12      var_STATUS_INFO_LENGTH_MISMATCH = STATUS_INFO_LENGTH_MISMATCH;
13      DEVICE_OBJECT_PTR_2 = WPP_GLOBAL_Control;
14      if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
15        && (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) != 0
16        && BYTE1(WPP_GLOBAL_Control->Timer) )
17      {
18        MessageNumber_1 = 0x24;
19  LABEL_6:
20        WPP_SF_q(
21          (__int64)DEVICE_OBJECT_PTR_2->AttachedDevice,
22          MessageNumber_1,
23          (_GUID *)&WPP_6285cecdc58f32c64ce6ba6b855a6046_Traceguids,
24          a1);
25      }
26      return var_STATUS_INFO_LENGTH_MISMATCH;
27    }
28    ExAcquireResourceSharedLite((PERESOURCE)(*(_QWORD *)v5 + 80i64), 1u);
29    if ( *(_BYTE *)(*(_QWORD *)(v1 + 0x30) + 0x89i64) )
30      v8 = *(_QWORD *)(v5 + 0x118);
31    else
32      v8 = *(_QWORD *)(v5 + 0x110);
33    v35 = v8;
34    if ( (unsigned int)Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled() )
35    {
36      if ( !Smb2ValidateVolumeObjectsMatch(a1, v5) )
37      {
38        PTR_DEVICE_OBJECT_1 = WPP_GLOBAL_Control;
39        if ( WPP_GLOBAL_Control == (PDEVICE_OBJECT)&WPP_GLOBAL_Control
40          || (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) == 0
41          || !BYTE1(WPP_GLOBAL_Control->Timer) )
42        {
43          goto LABEL_22;
44        }
45        MessageNumber_2 = 0x25;
46        goto LABEL_21;
47      }
48    }
49    else if ( !Smb2ValidateVolumeObjectsMatch_Servicing(a1) )
50    {
51      PTR_DEVICE_OBJECT_1 = WPP_GLOBAL_Control;
52      if ( WPP_GLOBAL_Control == (PDEVICE_OBJECT)&WPP_GLOBAL_Control
53        || (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) == 0
54        || !BYTE1(WPP_GLOBAL_Control->Timer) )
55      {
56        goto LABEL_22;
57      }
58      MessageNumber_2 = 0x26;
59  LABEL_21:
60      WPP_SF_qd(
61        (__int64)PTR_DEVICE_OBJECT_1->AttachedDevice,
62        MessageNumber_2,
63        (__int64)&WPP_6285cecdc58f32c64ce6ba6b855a6046_Traceguids,
64        a1,
65        STATUS_NOT_SUPPORTED);
66  LABEL_22:
67      ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 0x50i64));
68      return (unsigned int)STATUS_NOT_SUPPORTED;
69    }
70    ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 0x50i64));
71    v11 = 0;
72    v33 = 0;
73    while ( 1 )
74    {
75      if ( v11 )
76        goto LABEL_65;
77      v32 = 0;
78      var_STATUS_INFO_LENGTH_MISMATCH = Smb2PopulateShareNormalizedNameCache(a1, v8, &v32);
79      if ( (var_STATUS_INFO_LENGTH_MISMATCH & 0x80000000) != 0 )
```

**[Figure 51] Smb2QueryFileNormalizedName routine: first lines of the pseudo code (previous version)**

Observe highlight pieces of the old code and pay attention to key functions in the nextone from the updated version of the driver, which represent approximately the same code from the newer version:

```
23   else
24   {
25     ExAcquireResourceSharedLite((PERESOURCE)(*(_QWORD *)v5 + 80i64), 1u);
26     if ( *(_BYTE *)(*(_QWORD *)(v1 + 48) + 137i64) )
27       v6 = *(_QWORD *)(v5 + 280);
28     else
29       v6 = *(_QWORD *)(v5 + 272);
30     v33 = v6;
31     if ( Smb2ValidateVolumeObjectsMatch(a1, v5) )
32     {
33       ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 80i64));
34       v7 = 0;
35       v31 = 0;
36       while ( 1 )
37       {
38         if ( v7 )
39           goto LABEL_21;
40         v30 = 0;
41         v2 = Smb2PopulateShareNormalizedNameCache(a1, v6, &v30);
42         if ( (v2 & 0x80000000) != 0 )
43         {
44           v25 = WPP_GLOBAL_Control;
45           if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
46             && (HIDWORD(WPP_GLOBAL_Control->Timer) & 0x8000000) != 0
47             && BYTE1(WPP_GLOBAL_Control->Timer) )
48           {
49             v26 = 38;
50             LODWORD(FileInformationClass) = v2;
51 LABEL_61:
52             WPP_SF_qd(
53               (__int64)v25->AttachedDevice,
54               v26,
55               (__int64)&WPP_e8ea6266aa02331ec1234de41a53c1d4_Traceguids,
56               a1,
57               FileInformationClass);
58           }
59 LABEL_21:
```

**[Figure 52] Smb2QueryFileNormalizedName routine: first lines of the pseudo code (updated version)**

There are considerations that are appropriate to this moment, regardless of any specific error or vulnerability, and eventually such comments can bring insights for readers.

Before analyzing any code, I suggest trying to improve the code whether is possible. As readers quickly will notice, it will be not possible in different situations because there are uncountable undocumented functions, variable types, structures, enumeration and so on. We can try to search for them on Google or even on GitHub, but it is quite hard finding something. As this binary is a driver, it is useful to add the following **Type Libraries** (**SHIFT+F11**) into IDA Pro: **ntddk64_win7 and ntddk64_win10**. Additionally, as the code uses types defined in the SDK (*C:\Program Files* (*x86)\Windows Kits\10\Include\10.0.22621.0*\), we must insert the **mssdk_win10** type library too. At the same way, add the following **Signatures** (**SHIFT+F5**) into IDA Pro: **ms64wdk, mssdk64** and **vc64ucrt**. As a series of SMB2 functions are undocumented, we do not have any information about structures, data types, global variables, local variables, and eventual classes that could be exported by **srv2.pdb file**. To prove it, retrieve the **srv2.pdb symbol file** by executing the following command:

- **symchk /v /r C:\Users\Administrator\Desktop\JUL_KB5015882\srv2.sys /s srv*C:\symbols*https://msdl.microsoft.com/download/symbols**

Although there are much better tools to verify a symbol file's content, readers can use **dbh.exe tool** from SDK to do a quick inspection:

```
C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031>dbh srv2.pdb

srv2 [1000000]: enum Smb2Query*

 index            address     name
     1            104eed0 :   Smb2QueryMaximalAccess
     3            100b51c :   Smb2QueryTransportInfo
     4            1050558 :   Smb2QueryNetworkOpenInfo
     5            105d6f0 :   Smb2QueryFileNormalizedName

srv2 [1000000]: enum Smb2QueryFileNormalizedName

 index            address     name
     5            105d6f0 :   Smb2QueryFileNormalizedName

srv2 [1000000]: etypes


srv2 [1000000]: obj

d:\os\obj\amd64fre\minkernel\tools\gs_support\kmodefastfail\mp\objfre\amd64\gs_driverentry.obj
d:\os\obj\amd64fre\minkernel\tools\gs_support\kmodefastfail\mp\objfre\amd64\amdsecgs.obj
d:\os\obj\amd64fre\minkernel\tools\gs_support\kmodefastfail\mp\objfre\amd64\gs_support.obj
d:\os\obj\amd64fre\minkernel\tools\gs_support\kmodefastfail\mp\objfre\amd64\gs_report.obj
ntoskrnl.exe
HAL.dll
TDI.SYS
srvnet.sys
d:\os\obj\amd64fre\minio\netio\published\objfre\amd64\sockdef.obj
ksecdd.sys
d:\os\public\amd64fre\onecore\internal\minwin\priv_sdk\lib\amd64\hotpatchspareglobals.obj
d:\os\obj\amd64fre\onecore\base\fs\remotefs\smb\srv\srv.v2\driver\objfre\amd64\pch.obj
```

**[Figure 53] Examining symbols**

As we can notice from the output above, there is not any exported type, and we only have the list of functions and associated objects (strings).

The first issue to improve the code, as we do not know function prototypes and parameter's types, this prevents us to apply such types on the pseudo code. As an example, the function readers are seeing on **Figure 51** has its prototype as **__int64 __fastcall Smb2QueryFileNormalizedName(__int64 a1)**. As expected, its calling convention is **__fastcall** (usual for Microsoft APIs), but we do not have any idea about the content of **a1** or even if its type is really **__int64**. This function is called from **Smb2QueryFileNormalize routine**, but there we also do not have a concrete fact that provide us with the argument's type.

I have already mentioned previously **WPP (Windows software trace processor)**, and I would like to leave additional information about it. It can be used with kernel drivers and user-mode drivers and works as an instrumentation mechanism that is useful for tracing notifications, I/O activities, and memory allocations, which may be one of reasons of its usage here in the shown code. The **Timer field** is a pointer to a time object, which allows the I/O manager to call a timer routine every second.

Routines with **WPP_SF_** prefix are generated automatically and, as readers will see, they are found in different kernel drivers.

The body of **WPP_SF_q routine** is the following:

```
1  __int64 WPP_SF_q(
2          TRACEHANDLE LoggerHandle,
3          unsigned __int16 MessageNumber,
4          _GUID *MessageGuid,
5          ...)
6  {
7    va_list va; // [rsp+68h] [rbp+20h] BYREF
8
9    va_start(va, MessageGuid);
10   return ((__int64 (__fastcall *)(TRACEHANDLE, _QWORD, _GUID *, _QWORD, va_list *, __int64, _QWORD))pfnWppTraceMessage)(
11            LoggerHandle,
12            TRACE_MESSAGE_SYSTEMINFO_TIMESTAMP_SEQUENCE_GUID,
13            MessageGuid,
14            MessageNumber,
15            (va_list *)va,
16            8i64,
17            0i64);
18 }
```

**[Figure 54] Examining symbols**

According to image above, we can see that it calls **pfnTraceMessage** function pointer that references a real function, whose address is retrieved from **WppLoadTracingSupport** function and is set by **MmGetSystemRoutineAddress** function, which returns a pointer to the **WmiTraceMessage** function. This function is responsible for adding a message to the output log for the WPP software tracing session. Thus, on **line 10**, the function being called is **WmiTraceMessage**. As I know the prototype of **WmiTraceMessage** function from **MSDN** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-wmitracemessage), and use it to rename the possible variable and apply the enumeration's type to one of its parameters. Returning to the main picture (**Figure 51**), pay attention to line 23 and you will see a line as "*(_GUID *)&WPP_6285cecdc58f32c64ce6ba6b855a6046_Traceguids*", which clearly shows us that it is GUID. However, IDA Pro automatically offers us in the IDA View the following information:

```
PAGE:00000001C005D721                 mov     r12, [rax+70h]
PAGE:00000001C005D725                 jnb     short loc_1C005D77A
PAGE:00000001C005D727                 mov     r14d, 0C0000004h
PAGE:00000001C005D72D                 mov     rcx, cs:WPP_GLOBAL_Control ; __annotation("TMF:",
PAGE:00000001C005D72D                         ; "6285cecd-c58f-32c6-4ce6-ba6b855a6046 SRV2.SMB2
PAGE:00000001C005D72D                         ; // SRC=Unknown_cxx00 MJ= MN=",
PAGE:00000001C005D72D                         ; "#typev Unknown_cxx00 36 "%0FAIL: User buffer smaller
PAGE:00000001C005D72D                         ; than ULONG, WI %10!p!"
PAGE:00000001C005D72D                         ; //   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO",
PAGE:00000001C005D72D                         ; "{", "Arg, ItemPtr -- 10", "}",
PAGE:00000001C005D72D                         ; "PUBLIC_TMF:")
PAGE:00000001C005D734                 lea     rax, WPP_GLOBAL_Control
PAGE:00000001C005D73B                 cmp     rcx, rax
PAGE:00000001C005D73E                 jz      loc_1C005DC32
PAGE:00000001C005D744                 test    dword ptr [rcx+2Ch], 8000000h
PAGE:00000001C005D74B                 jz      loc_1C005DC32
PAGE:00000001C005D751                 lea     ebp, [r13+1]
PAGE:00000001C005D755                 cmp     [rcx+29h], bpl
PAGE:00000001C005D759                 jb      loc_1C005DC32
PAGE:00000001C005D75F                 lea     edx, [rbp+23h] ; MessageNumber
PAGE:00000001C005D762
PAGE:00000001C005D762 loc_1C005D762:          ; CODE XREF: Smb2QueryFileNormalizedName+4E5↓j
PAGE:00000001C005D762                 mov     rcx, [rcx+18h] ; LoggerHandle
PAGE:00000001C005D766                 lea     r8, WPP_6285cecdc58f32c64ce6ba6b855a6046_Traceguids ; MessageGuid
PAGE:00000001C005D76D                 mov     r9, r15
PAGE:00000001C005D770                 call    WPP_SF_q
PAGE:00000001C005D775                 jmp     loc_1C005DC32
```

**[Figure 55] IDA View: annotation**

The annotation tells us the referred **GUID (6285cecd-c58f-32c6-4ce6-ba6b855a6046)** is associated with:

- A concern about issues of insufficient memory allocation.
- If the error happens then it is logged as **DEBUG_ERROR.**
- It indicates that it is related to **SMB2_QUERY_INFO**, which is a packet type sent by a client to request information on a file, named pipe or even an underlying volume.

Observing the annotation from **Figure 55**, another two good points are: what does mean TMF? How does IDA get this information?

**TMF** comes from **Trace Message Format File**, and it contains instructions for parsing and formatting the message generated by the trace provider, which uses **ETW (Event Tracing for Windows)** to generate trace messages or trace events. These TMFs are inside the symbol file that, in our case, it is **srv2.pdb**. To extract these **trace message format (.tmf) files** we can use **Tracepdb.exe**, which is installed when you install WDK, Visual Studio and Windows SDK. I have created a **tmf folder**, copied the srv2.pdb file to there and executed the **tracepdb.exe:**

```
C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf>tracepdb srv2.pdb

Microsoft (R) TracePDB.Exe (10.0.22621.2428)
⌐ Microsoft Corporation. All rights reserved.

tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\eead5c16-18b1-3491-3be3-ab12a9f517b9.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\78031240-7dc4-35e5-a1b9-2641617685a5.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\60f3abd9-8936-3b27-2a26-4464e18815c7.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\d4be216d-5162-3dd0-61cc-e1745e58168f.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\730b1622-0eff-3927-a775-4624561530d7.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\81284eed-12f1-3d9c-0c4d-d945a4393ede.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\baeab448-35b7-3aea-97a5-5f13c0fab9d1.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\8c6de0d2-9042-3d6b-995a-c4d9651f0eb8.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\419ff270-a959-328c-24ad-6ca01c51a167.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\419ff270-a959-328c-24ad-6ca01c51a167.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\efbf206a-4797-3b92-214d-79f093c52804.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\419ff270-a959-328c-24ad-6ca01c51a167.tmf for srv2.pdb
tracepdb : info : WPPFMT generating C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf\8c1cf058-ca00-3cc3-baa0-a70620c7930d.tmf for srv2.pdb
```

**[Figure 56] Extracting TMF files with tracepdb.exe (truncated output)**

The output is long, but you are going to find the following TMF file named **6285cecd-c58f-32c6-4ce6-ba6b855a6046.tmf**, which represents our GUID found previously. It is trivial to see its content:

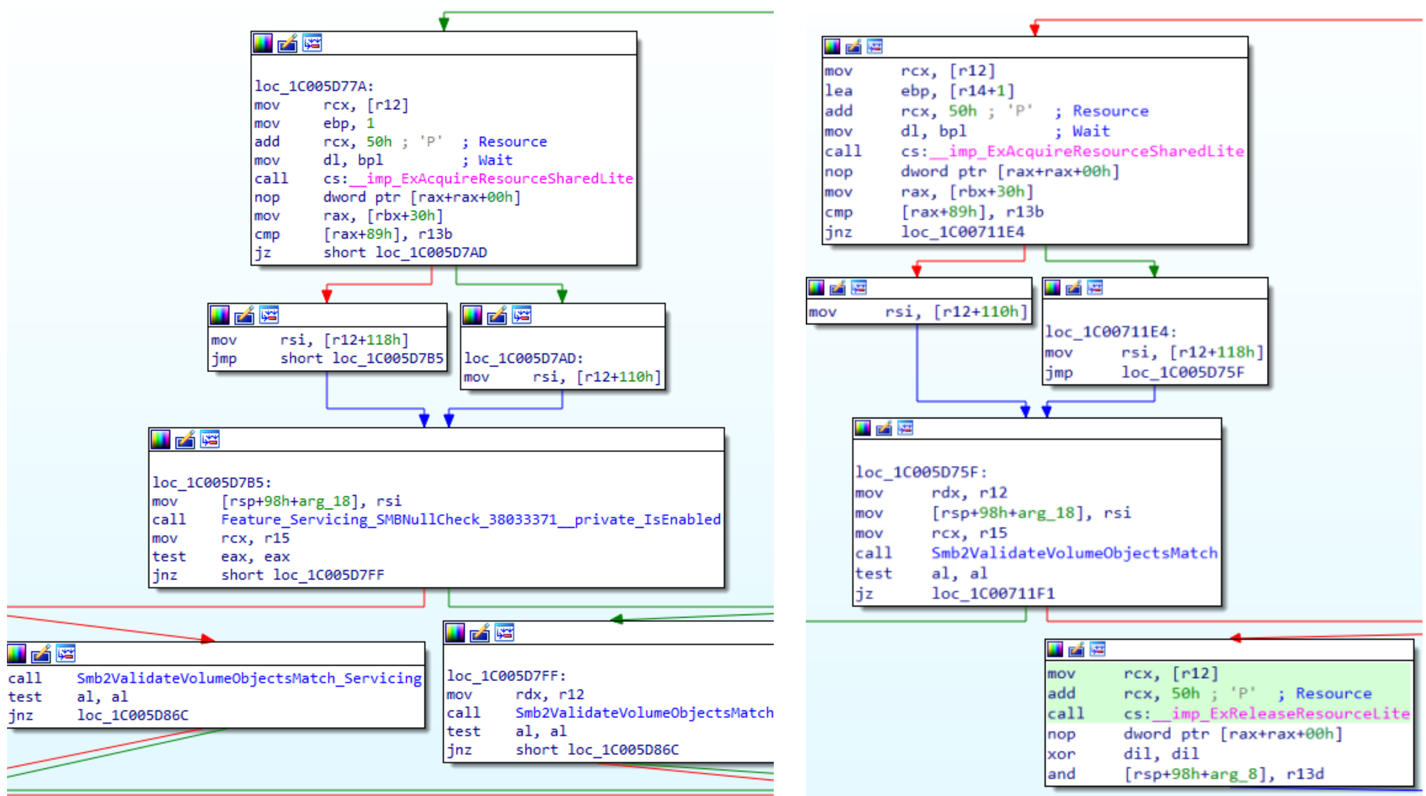
```
C:\Symbols\srv2.pdb\71B56E68A8D67B2EB30072E6ED41C8031\tmf>cat 6285cecd-c58f-32c6-4ce6-ba6b855a6046.tmf | grep typev | sed
 -e s/\/\//\n\/\//g
#typev Unknown_cxx00 34 "%0FAIL: QueryInfo Workitem %10!p! error 0x%11!x! InfoType 0x%12!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2ContinueQueryInfo
#typev Unknown_cxx00 35 "%0FAIL: QueryInfo Workitem %10!p! invalid info level"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2ExecuteQueryInfo
#typev Unknown_cxx00 39 "%0FAIL: Failed to query share normalized name. WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 40 "%0FAIL: Memory allocation failed. WI %10!p!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 42 "%0FAIL: Too few bytes received from ZwQueryInformationFile. WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 43 "%0FAIL: Failed to query normalized share root name after 3 tries.WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 41 "%0FAIL: Querying file normalized name failed. WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 37 "%0FAIL: SMB share volume and file volume mismatch. WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 38 "%0FAIL: SMB share volume and file volume mismatch. WI %10!p!, status 0x%11!x!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 36 "%0FAIL: User buffer smaller than ULONG, WI %10!p!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2QueryFileNormalizedName
#typev Unknown_cxx00 10 "%0FAIL: QueryInfo Workitem %10!p! Invalid FileInfo level %11!d!"
//   LEVEL=DEBUG_ERROR FLAGS=DEBUG_SMB2_QUERY_INFO FUNC=Smb2ValidateGetFileInfoParameters
```

**[Figure 57] Tmf file overview (truncated output)**

The reason I have highlighted **message 36** is because according to **line 18** (**Figure 51**) the message number is 36, and checking such a message in TMF file we can confirm that is exactly the same message presented by the disassembler on the IDA View. At the same way, readers can also observe other messages scattered throughout the code, and it offers good hints (annotations) about **what the code is doing, the kind of SMB2 packet involved and mainly concerns from programmers by have instrumented that portion of the code, which could also indicate possible vulnerabilities' paths to be explored**.

Proceeding with our overview about the code, two first things that is always recommended to do are to check whether the code verifies eventual NULL values are passed to functions/routines and it also checks all returned values. It could be surprising, but such kinds of issues are more usual than could be expected. Honestly, I do not evaluate such issues as mistakes because writing large codes is always a challenging task, and it is natural to forget things.

On line 28 (*previous version of the driver, on figure 51, and remember that I have collapsed the local declarations*), we find the **ExAcquireResourceSharedLite** function, which is used to acquire the given resource for shared access by the calling thread. Therefore, as readers already know, shared resources accessed by threads offer a potential source of race conditions whether the synchronization is incorrectly implemented for writing operations (not in this case). Additionally, the invocation of this routine did not belong to an if-else statement, but it has moved inside the if-else condition, which might indicate a change of logic. In this case, I always recommend readers to check the visual representation offered by IDA Pro:




**[Figure 58] Old driver on the left and new driver on the right.**

No doubt, it is easier to see and understand the logic of programming by using graphs, and as I mentioned previously, it is direct to check whether arguments and returned values (**test [eax] [eax] instruction** or similar) have been verified.

The same **ExAcquireResourceSharedLite function**, from **line 28**, has two parameters that are a pointer to a resource (**PERESOURCE** – from **wdm.h header file**), which is a structure used by drivers to implement shared or exclusive synchronization, and a Boolean flag (Wait), which indicates whether the function must or not wait for the resource until it can be acquired. As it is set to 1 then it will wait for it. Nonetheless, there are few details:

- For now, we do not know what is v1 or v5 (yes, they can be structures), and we only know that **Smb2QueryFileNormalizedName routine** is called from **Smb2ExecQueryInfo routine**. V5 parameter comes from v1 local variable, which is derived from a1 parameter that we also do not do anything about and that serves as parameter for **Smb2QueryFileNormalizedName routine.**

- The returned value is not evaluated as well the first parameter, which the output, is not assessed too, and it is used, as pointer, for setting the v8 value. Of course, it could not be a problem here, but any issues with such an instruction have the potential to cause a DoS, for example.

About the variable's v1 and v5 (mainly v5), it might be difficult to conclude its type. Look at the function once more:

- **ExAcquireResourceSharedLite((PERESOURCE)(*(_QWORD *)v5 + 80i64), 1u);**

As the first parameter of this function is a pointer to **_ERESOURCE type**, so the task is to discover the type we could try on v5 that, once dereferenced, and added to 80 provide us with a **_ERESOURCE type**. There is an additional issue here because we do not know whether it is a well-known type or a type that we do not know previously (it not public or can be a customized type). The context does not help us:

- **(line 01):** __int64 __fastcall Smb2ExecuteQueryInfo(__int64 **a1**)
- **(line 05):** **v1** = *(_QWORD *)(**a1** + 504);
- **(line 09):** **v5** = *(_QWORD *)(*(_QWORD *)(**v1** + 0x40) + 0x70i64);

As readers see, we do not know a1 (parameter), so we do not know about v1 and of course we do not know about v5. If we return to **Smb2ExecuteQueryInfo** (the caller of **Smb2ExecuteQueryInfo routine**), **a1** is also its parameter, and it is involved with a different **Smb2* routines**, but in special as part of one of parameter of **SRV_PERF_ENTER_EX routine**, which seems to help in the instrumentation operation. Finally, **Smb2ExecuteQueryInfo routine** is called at runtime.

Even that v5's type is a public structure, it is still quite difficult to find statically the exact structure to be applied to v5, which should have a member in its offset 0x50 that is a pointer to **_ERESOURCE**, and it is usually worse than it because most structures hold other structures as their members.

Readers could try the following approach that is completely imprecise, but sometimes it produces results. As you are searching for a structure that uses **_ERESOURCE** type as its member, you might check the Virgilius' project page for such structure and at the bottom of the page you will find a section named "**Used in**", which tells you which structure use **_ERESOURCE** type as a type of a member. However, it is time consuming task because most of these structures have other structures as members (all of them available on Virgilius), and you will need to import one by one before importing the target structure. Once you have imported it, you can change the v5's type and check whether get something useful. It is fair to say that adding a new structure or importing a header file into IDA Pro's database demands attention to details and, eventually, it might be a complicated task.

Steps to add structures copied from Virgilius are almost direct:

1. Go to **Local Types tab (SHIFT + F1)**, press the **INSERT key**, and copy the structure from Virgilius's page into the textbox.

2. Change any reference from **PVOID, PVOID* and PVOID**** to **QWORD, QWORD* and QWORD**** respectively because structures here do not accept VOID type and its variants.

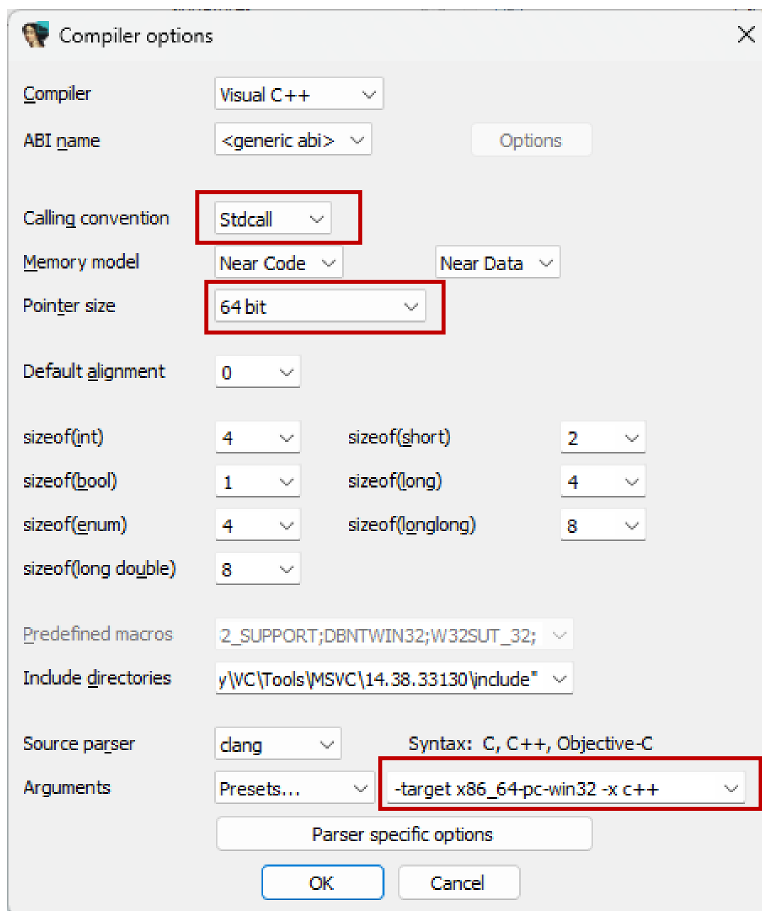
3. Click OK. If everything goes well, IDA Pro will show the structure as imported. If mistakes have occurred, check the **Output window** for errors. Probably one or more structures are missing and should have been added previously.

To import a header file that you already had previously or customized it, it takes a different approach:

1. **Install Visual Studio**, and make sure to install **C++ Clang Compiler for Window** and **MSBuild support for LLVM(clang-cl) toolset components**.

2. **Add the Clang's binary folder to the PATH environment variable**: *C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\Llvm\x64\bin.*

3. Adjust the IDA Pro compiler's setting to Clang and, in this case, for 64-bit binaries:




**If the target's platform was 32-bit, so the argument would be:**

- **-target i386-pc-win32 -x c++**

**For additional reference, check:**

- **https://hex-rays.com/tutorials/idaclang/idaclang_tutorial.pdf**

**[Figure 59] IDA Pro Compiler Settings**

4. Go to **File → Parse C Header File** and pickup its header file.
5. Check the **Output View** and verify if there is any error.
6. Go to **Local Types tab (SHIFT + F1)** and verify whether structures are imported.

It is opportune to mention that this header file (.h extension) should contain only structure definitions and enumerations, and try to avoid complicated header files that, eventually, might cause issues during the parse. Additionally, as numerous header files refer to other header files via include directives, they must be in the same directory or added inline to compose a single header file.

Only to supplement our explanation about creating or event importing a structure into IDA Pro, it could be useful to review about the supported Windows types, which will prevent you to use incompatible types for C/C++: https://learn.microsoft.com/en-us/windows/win32/winprog/windows-data-types.

About inserting a lengthy list of structures into the IDA's database, it is definitely a demanding task and there are measures that might could save our time in a next opportunity. As an example, you can mark all new structures that you have inserted into **Local Types tab**, right clicking them, and choosing **Export to a header file option**. Afterwords, you would have a list of structure's definitions, which it would be possible to choose only a few ones (respecting all dependencies) and insert them into another IDA Pro database. It could save you time to research all the structures again and edit them to adapt to the correct format.

Furthermore, you could consider a more solid and definitive work by creating a custom type-library to import whenever is necessary. It is not hard whether you have a self-contained header file and can save your time for other analysis. A quick review of the procedure follows:

1. Download **Tilib 8.3** from https://hex-rays.com/download-center/.
2. Extract both versions (32-bit and 64-bit) to a folder.
3. Copy the extracted folder into IDA Pro's folder: *C:\Program Files\IDA Pro 8.3*
4. If you have any problem using **tilib64.exe**, copy the executable itself to IDA Pro 8.3 folder, where the IDA Pro executables are kept. It could be easier to use it.
5. Execute: **tilib64.exe -c -Cc1 -h<custom header> <custom header>.til**.
6. If your custom header file also needs other headers (#include directives) then you must specify them:

   **-I<directory holding needed headers>** (use one -I option for each necessary directory).

7. If everything goes well, check the type library's content: **tilib64.exe -l <custom headers>.**
8. Make the newly created type-library available for IDA Pro by copying it to *C:\Program Files\IDA Pro 8.3\til\pc folder*.
9. Go to **View | Open subviews | Type Libraries (SHIFT+F11).**
10. Pressing **insert key** should show a list of available type libraries, and the custom one should be there.

Eventually this brief review about a handful of features from IDA Pro can help you throughout this article or in other opportunities and articles too.

Returning to **Figure 51**, it is time to continue our work and leave further observations. If readers look at the IDA View, you will see an interesting instruction soon after the **cs:__imp_ExAcquireResourceSharedLite** instruction: **nop dword ptr [rax+rax+00h]**

At a first moment, it could represent an error in the disassembling process because this NOP instruction does not make sense. However, it is not an error, and the instruction is used for alignment. An explanation for this instruction is on **page 4-165 from Intel Architectures Software Developer's Manual (SEP/2023)**:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

| Length | Assembly | Byte Sequence |
|---|---|---|
| 2 bytes | 66 NOP | 66 90H |
| 3 bytes | NOP DWORD ptr [EAX] | 0F 1F 00H |
| 4 bytes | NOP DWORD ptr [EAX + 00H] | 0F 1F 40 00H |
| 5 bytes | NOP DWORD ptr [EAX + EAX*1 + 00H] | 0F 1F 44 00 00H |
| 6 bytes | 66 NOP DWORD ptr [EAX + EAX*1 + 00H] | 66 0F 1F 44 00 00H |
| 7 bytes | NOP DWORD ptr [EAX + 00000000H] | 0F 1F 80 00 00 00 00H |
| 8 bytes | NOP DWORD ptr [EAX + EAX*1 + 00000000H] | 0F 1F 84 00 00 00 00 00H |
| 9 bytes | 66 NOP DWORD ptr [EAX + EAX*1 + 00000000H] | 66 0F 1F 84 00 00 00 00 00H |

**[Figure 60] NOP instruction from INTEL Architectures Software Developer's Manual (SEP/2023)**

Additionally, it can be part of the compilation process or even used also as space reservation, like the old **mov edi, edi** instruction, for future hot patches.

Proceeding with our overview, whether we compare **Figures 51** and **52**, and use **Figure 58** for supporting the analysis, we will find a series of details. I will not be commenting on the instrumentation using WPP again because they are there for checking whether any condition is wrong (and return) and logging critical issues related to such conditions, serving as useful debugging resource.

To old driver (vulnerable), we have the following sequence of lines including calls instructions:

1. **ExAcquireResourceSharedLite**
2. **if ( (unsigned int)Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled())**
3. **if ( !Smb2ValidateVolumeObjectsMatch(a1, v5) )** // inside the if-clause (2)
4. **else if ( !Smb2ValidateVolumeObjectsMatch_Servicing(a1) )**
5. **ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 0x50i64))** // inside else-if clause (4)
6. **ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 0x50i64));** // out of any clause.
7. **var_STATUS_INFO_LENGTH_MISMATCH = Smb2PopulateShareNormalizedNameCache**

To the new driver (changed/fixed), we have the following sequence of lines including calls instructions:

1. **ExAcquireResourceSharedLite**
2. **if ( Smb2ValidateVolumeObjectsMatch(a1, v5) )**
3. **ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 80i64));** //inside if clause (2)
4. **if ( Smb2ValidateVolumeObjectsMatch(a1, v5) )** //inside if-clause (2)
5. **ExReleaseResourceLite((PERESOURCE)(*(_QWORD *)v5 + 80i64))** // inside if-clause (4)
6. **v2 = Smb2PopulateShareNormalizedNameCache(a1, v6, &v30);**

We can easily notice that in the old driver there is a **Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled()** function that is not present in the new driver. At the same way, there is another function named **Smb2ValidateVolumeObjectsMatch_Servicing** that has a similar name with the new driver but containing the word *Servicing*, and also with only one argument instead of having two arguments as seen in the new driver.

Another point is that in the old driver uses two invocations of **ExReleaseResourceLite function** (even though only one is really called) while the second one does the same invocation only once. Anyway, resources being allocated are being deallocated in both cases.

For the old driver, we have the following interpretation about the execution sequence:

- If the previously allocated buffer is not less than 4 (line 10), so proceed. Otherwise, log the event and return an error.
- The shared resource is allocated.
- A checking (SMB NullCheck) is performed. If the checking done inside this routine is not successful, it logs the event, releases the shared resource, and returns an error. If it is then it proceeds.
- If the file queried is at the same shared volume, then proceed. If it is not, it logs the event, releases the shared resource, and returns an error.
- Release the resource, which has been used only for volume validation, and call **Smb2PopulateShareNormalizedNameCache routine**.

For the new driver, the following sequence happen:

- If the allocated buffer is not less than 4 (line 10), so proceed. Otherwise, log the event and return an error.
- The shared resource is allocated.
- There is a verification to check if the file queried is at the same shared volume. If such a checking is successful, then proceed. If it is not, the shared resource is released (line 194).
- The resource is released, which has been used only for volume validation and call **Smb2PopulateShareNormalizedNameCache routine**.

To supplement our information, the content of **Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled** function is shown below:

```
1  __int64 Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled()
2  {
3    unsigned int CachedFeatureEnabledState; // eax
4    unsigned int v1; // ebx
5    int v3; // [rsp+30h] [rbp-18h]
6
7    CachedFeatureEnabledState = wil_details_FeatureStateCache_GetCachedFeatureEnabledState(
8                                  (volatile signed __int32 *)&Feature_Servicing_SMBNullCheck_38033371__private_featureState,
9                                  (__int64)&Feature_Servicing_SMBNullCheck_38033371__private_descriptor);
10   v1 = (CachedFeatureEnabledState >> 3) & 1;
11   wil_details_FeatureReporting_ReportUsageToService(
12     (__int64)&Feature_Servicing_SMBNullCheck_38033371__private_reporting,
13     0x24457DBu,
14     (CachedFeatureEnabledState >> 8) & 1,
15     (CachedFeatureEnabledState >> 9) & 1,
16     (__int64)&Feature_Servicing_SetReparsePointImpersonation_37951019_logged_traits,
17     v1,
18     v3);
19   return v1;
20 }
```

**[Figure 61] Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled function**

The code above is using **Windows Implementation Libraries (WIL)**, which is a header-only C++ library, and that can be used in different contexts such as trace logging, resource management and Registry's interaction, and only for mentioning a brief list of them. As another example, WIL is usually used to implement **RAII (Resource Acquisition is Initialization) resource** wrappers, which are used to manage object destruction when a handle to a kernel object is closed.

Based on information exposed previously, comments follow:

- The execution flow between them is similar (not equal), and there are minor changes in terms of functions being invoked and instrumented.

- The "servicing" suffix for two routines in the old driver (JUL/2022) suggests that these routines can be a kind of "intermediate" or transitory fix due to previous issues in this part of the code, which will be consolidated in the next version (our updated version from AUG/2022).

- This impression is reinforced since there is not any message based on **TMF** (**Trace Message Format File)** on the assembly code. Even the presence of multiples points of instrumentation also suggests previous issues in this routine (**Smb2QueryFileNormalizedName**) or around it.

- Another curious fact is that If **Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled** function reports a positive condition, the **Smb2ValidateVolumeObjectsMatch** is called. However, if there is something wrong, it executes **Smb2ValidateVolumeObjectsMatch_Servicing**. The fact of having two routines that should do the same tasks but implemented slightly different also provides us with additional foundations to believe in this transition phase. A comparison can help, and remember that both are coming from the old (vulnerable) driver:

Pseudocode-C

```
1  bool __fastcall Smb2ValidateVolumeObjectsMatch(
2          __int64 a1,
3          __int64 a2)
4  {
5    __int64 v2; // rdi
6    __int64 v3; // rcx
7    PDEVICE_OBJECT BaseFileSystemDeviceObject; // rbx
8
9    v2 = *(_QWORD *)(a1 + 0x1F8);
10   if ( *(_BYTE *)(*(_QWORD *)(v2 + 0x30) + 0x89i64) )
11     v3 = *(_QWORD *)(a2 + 0x118);
12   else
13     v3 = *(_QWORD *)(a2 + 0x110);
14   if ( !v3 )
15     return 0;
16   BaseFileSystemDeviceObject = IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(v3 + 0x50));
17   return BaseFileSystemDeviceObject == IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(*(_QWORD *)(v2 + 0x48) + 0x50i64));
18 }

0005DDEE Smb2ValidateVolumeObjectsMatch:8 (1C005DDEE) (Synchronized with IDA View-A, Hex View-1, Pseudocode-A)
```

**V3 is checked to prevent to null pointer dereference attack (CWE-476).**

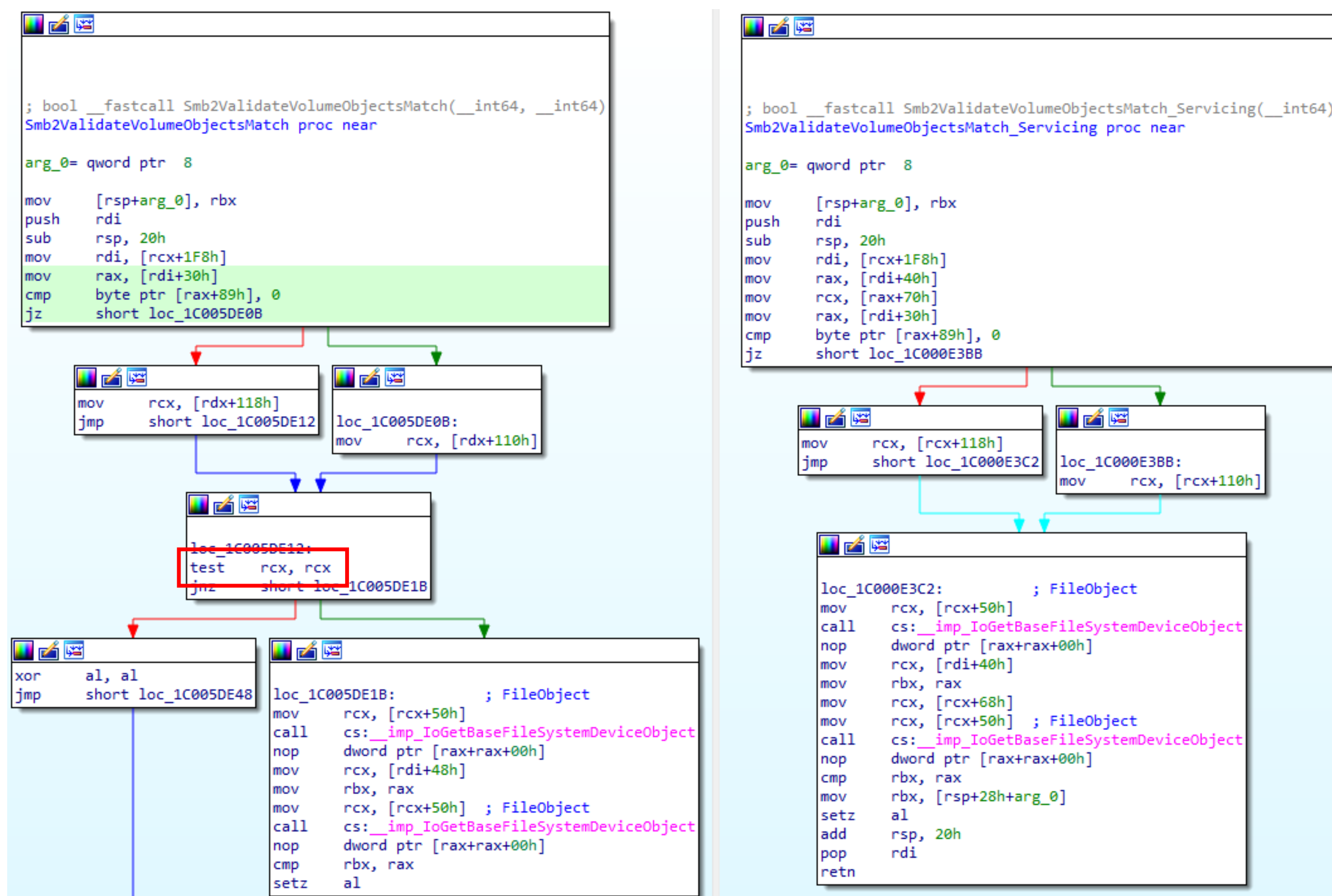
Pseudocode-B

```
1  bool __fastcall Smb2ValidateVolumeObjectsMatch_Servicing(
2          __int64 a1)
3  {
4    __int64 v1; // rdi
5    __int64 v2; // rcx
6    __int64 v3; // rcx
7    PDEVICE_OBJECT BaseFileSystemDeviceObject; // rbx
8
9    v1 = *(_QWORD *)(a1 + 0x1F8);
10   v2 = *(_QWORD *)(*(_QWORD *)(v1 + 0x40) + 0x70i64);
11   if ( *(_BYTE *)(*(_QWORD *)(v1 + 0x30) + 0x89i64) )
12     v3 = *(_QWORD *)(v2 + 0x118);
13   else
14     v3 = *(_QWORD *)(v2 + 0x110);
15   BaseFileSystemDeviceObject = IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(v3 + 0x50));
16   return BaseFileSystemDeviceObject == IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(*(_QWORD *)(*(_QWORD *)(v1 + 0x40) + 0x68i64) + 0x50i64));
17 }
```

**V3 is not checked likely due to the assumption that it represents a member of a structure/class (v2) that should be initialized and never should be equal to zero.**

**[Figure 62] Smb2ValidateVolumeObjectsMatch and Smb2ValidateVolumeObjectMatch_Servicing**

One important aspect is that sometimes might be easier to notice this kind of issue by using the IDA View than the pseudo-code, which shows the assembly code, and that is also a strong reason for always examining both to make sure that they are corresponding to each other:




**[Figure 63] Smb2ValidateVolumeObjectsMatch and Smb2ValidateVolumeObjectMatch_Servicing (2)**

From a different point of view, readers can notice that the instruction **test rcx, rcx** (**!v3** in pseudo code) exists on the left side (**Smb2ValidateVolumeObjectsMatch**) and does not exist on the right side (**Smb2ValidateVolumeObjectMatch_Servicing).**

If readers compare **Smb2ValidateVolumeObjectsMatch routine** from **Figure 62** (old driver – JUL/2022) to the new driver, you will notice that they are the same (including the null pointer dereference verification):

```
 1 bool __fastcall Smb2ValidateVolumeObjectsMatch(__int64 a1, __int64 a2)
 2 {
 3   __int64 v2; // rdi
 4   __int64 v3; // rcx
 5   PDEVICE_OBJECT BaseFileSystemDeviceObject; // rbx
 6
 7   v2 = *(_QWORD *)(a1 + 0x1F8);
 8   if ( *(_BYTE *)(*(_QWORD *)(v2 + 0x30) + 0x89i64) )
 9     v3 = *(_QWORD *)(a2 + 0x118);
10   else
11     v3 = *(_QWORD *)(a2 + 0x110);
12   if ( !v3 )
13     return 0;
14   BaseFileSystemDeviceObject = IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(v3 + 0x50));
15   return BaseFileSystemDeviceObject == IoGetBaseFileSystemDeviceObject(*(PFILE_OBJECT *)(*(_QWORD *)(v2 + 0x48) + 0x50i64));
16 }
```

**[Figure 64] Smb2ValidateVolumeObjectsMatch (newer driver – AUG/2022)**

Curiously, according to the Microsoft registers, this issue has been reported in May/2022 (**CVE-2022-32230**), but it was not considered a vulnerability in June/2022. Furthermore, if readers list SMB vulnerabilities from January to August (our vulnerability), you will notice that there were plenty of issues:

| Release date | CVE Number ↓ | CVE Title | Impact | Tag |
|---|---|---|---|---|
| Aug 9, 2022 | CVE-2022-35804 | SMB Client and Server Remote Code Execution Vulnerability | Remote Code Execution | Windows Kernel |
| Jun 14, 2022 | CVE-2022-32230 | Windows SMB Denial of Service Vulnerability | Not a Vulnerability | Windows SMB |
| Jun 14, 2022 | CVE-2022-30154 | Microsoft File Server Shadow Copy Agent Service (RVSS) Elevation of F | Elevation of Privilege | Remote Volume Shadow Copy Service (RVSS) |
| Apr 12, 2022 | CVE-2022-26830 | DiskUsage.exe Remote Code Execution Vulnerability | Remote Code Execution | diskusage |
| Apr 12, 2022 | CVE-2022-26809 | Remote Procedure Call Runtime Remote Code Execution Vulnerability | Remote Code Execution | Windows Remote Procedure Call Runtime |
| Apr 12, 2022 | CVE-2022-24541 | Windows Server Service Remote Code Execution Vulnerability | Remote Code Execution | Windows SMB |
| Apr 12, 2022 | CVE-2022-24534 | Win32 Stream Enumeration Remote Code Execution Vulnerability | Remote Code Execution | Windows kernel32.dll |
| Mar 8, 2022 | CVE-2022-24508 | Win32 File Enumeration Remote Code Execution Vulnerability | Remote Code Execution | Windows SMB Server |
| Apr 12, 2022 | CVE-2022-24500 | Windows SMB Remote Code Execution Vulnerability | Remote Code Execution | Windows SMB |
| Apr 12, 2022 | CVE-2022-24485 | Win32 File Enumeration Remote Code Execution Vulnerability | Remote Code Execution | Windows kernel32.dll |

**[Figure 65] SMB vulnerabilities from January/2022 to August/2022**

All routines that we have been reviewing so far belong to **Smb2QueryFileNormalizedName routine**, which is called from **Smb2ExecuteQueryInfo routine**. This routine is associated to a **SMB2 QUERY_INFO Request packet** that, once sent by a client, it is used to request information about a file, named pipe or volume, according to the Microsoft Documentation (**MS-SMB2 protocol guide**: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/5606ad47-5ee0-437a-817e-70c366052962). The request structure is formed by a series of fields, which one of them is **FileInfoClass** that, as expected, it is used for file information queries and can have one of the **FILE_INFORMATION_CLASS values**, and one of them is **FileNormalizedNamInformation**. Following the same documentation reference, once the request is for a **FileNormalizedNameInfomation**, the server must convert the information returned by the object store to a normalized path, which is a full pathname of a directory or even a file related to the root or share on which it stored. If readers check **MS-FCC protocol guide** (https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/20bcadba-808c-4880-b757-4af93e41edf6) in 2.4.30 section (*FileNormalizedNameInformation*), you will confirm that there two possible answers to the request: **STATUS_NOT_SUPPORTED (0xC0000BB – it appear along the code several times)** and **STATUS_BUFFER_OVERFLOW**. The **MS-SMB2 protocol guide** tells us that **FileNormalizedNameInformation** information class was not supported by Windows 10 v1709 and prior versions.

We should remember that **CVE-2022-35804** is our chosen vulnerability for analysis since the beginning of this article. Additionally, Microsoft recommended at that time disabling SMBv3 compression to block unauthenticated attackers from exploiting the vulnerability against an SMBv3 Client and Server. If we remember of the **FileInfoClass class**, there is also **FILE_INFORMATION_CLASS** value for compression named **FileCompressionInformation**. For now, we do not know whether is important, or even how important it can be, but we should keep it registered.

Our analysis has been concentrated only on a piece of **Smb2QueryFileNormalizedName routine,** there is much more code in the routine and interesting details. It is not my intention to analyze line by line such a routine right now, but only show the process and how reverse engineering is also useful for understanding what you are seeing in front of you. Of course, it would be much simpler to search for “patterns” of vulnerabilities, but in this case, it would be not so useful for you.

There are other aspects that should be commented on or even refreshed. As we learned from previous sections, there are other function that presented a partial match in terms of similarity:

Partial matches

| Line | Address | Name | Address 2 | Name 2 | Ratio | BBlocks 1 | BBlocks 2 |
|---|---|---|---|---|---|---|---|
| 00000 | 1c004d420 | Smb2ValidateGetFileInfoParameters | 1c004d420 | Smb2ValidateGetFileInfoParameters | 0.9995120 | 11 | 11 |
| 00001 | 1c0050150 | Smb2ExecuteQueryInfo | 1c0050150 | Smb2ExecuteQueryInfo | 0.9991063 | 41 | 41 |
| 00002 | 1c00881b0 | DriverEntry | 1c00881b0 | DriverEntry | 0.9987805 | 33 | 33 |
| 00003 | 1c00535c0 | Smb2ContinueQueryInfo | 1c00535c0 | Smb2ContinueQueryInfo | 0.9985975 | 26 | 26 |
| 00004 | 1c0073400 | DriverUnload | 1c00733f0 | DriverUnload | 0.9957806 | 9 | 9 |
| 00005 | 1c004d850 | Smb2ValidateQueryInfo | 1c004d850 | Smb2ValidateQueryInfo | 0.9930744 | 187 | 187 |
| 00006 | 1c005d6f0 | Smb2QueryFileNormalizedName | 1c005d6f0 | Smb2QueryFileNormalizedName | 0.9778123 | 58 | 63 |

Line 6 of 7

Unmatched in secondary

| Line | Address | Name |
|---|---|---|
| 00000 | 1c000e38c | Smb2ValidateVolumeObjectsMatch_Servicing |
| 00001 | 1c000e470 | Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled |
| 00002 | 1c0018468 | McGenEventUnregister_EtwUnregister |

Line 3 of 3

Unmatched in primary

| Line | Address | Name |
|---|---|---|
| 00000 | 1c0018368 | Real_Driver_Entry |

**[Figure 66 - A] Partial and unmatched routines in srv2.sys (according to Diaphora)**

From **BinDiff tool** view, we have a similar view:

Matched Functions

| Similarity | Confidence | Change | EA Primary | Name Primary | EA Secondary | Name Secondary | Comr | Algorithm |
|---|---|---|---|---|---|---|---|---|
| 0.98 | 0.99 | GI----- | 00000001C004E3F0 | Smb2ValidateWrite | 00000001C004E3F0 | Smb2ValidateWrite | | Name Hash |
| 0.87 | 0.98 | GI-JE-C | 00000001C005D6F0 | Smb2QueryFileNormalizedName | 00000001C005D6F0 | Smb2QueryFileNormalizedName | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001008 | RfsHashTableEnumerate | 00000001C0001008 | RfsHashTableEnumerate | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C00010AC | Smb2LogNetNameChange_i | 00000001C00010AC | Smb2LogNetNameChange_i | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001140 | Srv2UpdateNetnameTable | 00000001C0001140 | Srv2UpdateNetnameTable | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C000132C | Smb2DereferenceNetname | 00000001C000132C | Smb2DereferenceNetname | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C000135C | RfsHashTableLookup | 00000001C000135C | RfsHashTableLookup | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001440 | RfsHashBucketReleaseLockSha... | 00000001C0001440 | RfsHashBucketReleaseLockShared | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001468 | RfsHashBucketAcquireLockShar... | 00000001C0001468 | RfsHashBucketAcquireLockShared | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001498 | RfsHashGenerateKey | 00000001C0001498 | RfsHashGenerateKey | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C00014D0 | Smb2NetnameEqualKey | 00000001C00014D0 | Smb2NetnameEqualKey | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001540 | RfsHashTableInsertEx | 00000001C0001540 | RfsHashTableInsertEx | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001770 | RfsHashTableRemove | 00000001C0001770 | RfsHashTableRemove | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001910 | Srv2DereferenceEndpoint | 00000001C0001910 | Srv2DereferenceEndpoint | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001934 | Srv2RemoveConnectionFromEn... | 00000001C0001934 | Srv2RemoveConnectionFromEndpoin... | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001A58 | RfsTable64Cleanup | 00000001C0001A58 | RfsTable64Cleanup | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001A90 | RfspTable64Cleanup | 00000001C0001A90 | RfspTable64Cleanup | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001B90 | RfsHashBucketReleaseLockExcl... | 00000001C0001B90 | RfsHashBucketReleaseLockExclusive | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001BBC | RfsHashBucketAcquireLockExcl... | 00000001C0001BBC | RfsHashBucketAcquireLockExclusive | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001BEC | Srv2GlobalConnectionListRemove | 00000001C0001BEC | Srv2GlobalConnectionListRemove | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001C54 | Smb2CloseConnection | 00000001C0001C54 | Smb2CloseConnection | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001CD8 | RfsHashTableLookupFirstMatch... | 00000001C0001CD8 | RfsHashTableLookupFirstMatchEntry | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001D3C | Srv2CancelOutstandingWorkIt... | 00000001C0001D3C | Srv2CancelOutstandingWorkItems | | Name Hash |
| 1.00 | 0.99 | ------- | 00000001C0001DB4 | Smb2InsertConnectionIntoClient | 00000001C0001DB4 | Smb2InsertConnectionIntoClient | | Name Hash |

Secondary Unmatched

| EA | Name | Basic Blocks | Instruction: | Edges |
|---|---|---|---|---|
| 00000001C000E38C | Smb2ValidateVolumeObjectsMatch_Servicing | 4 | 27 | 4 |
| 00000001C000E470 | Feature_Servicing_SMBNullCheck_38033371__private_IsEnabled | 1 | 24 | 0 |

**[Figure 66 - B] BinDiff (srv2.sys)**

There are seven routines that presented partial matches, and we quickly analyzed one of them, and the last one is the routine that presents major differences, which we quickly analyzed. Unfortunately, the other six routines with partial matches do not present anything quite interesting to highlight at this point, and most of the changes between versions come from different GUIDs. Additionally, we already commented about **Smb2ValidateVolumeObjectsMatch_Servicing** and **Feature_Servicing_SMBNullCheck** routines.

## 12. Reversing and collecting additional information

It is time to go to the driver's entry point (in the previous version) to examine its content. If readers press **CTRL+E**, IDA Pro shows the **GsDriverEntry routine** and, from there, you will land at **DriverEntry()**:

```
 1 NTSTATUS __stdcall DriverEntry(
 2         _DRIVER_OBJECT *DriverObject,
 3         PUNICODE_STRING RegistryPath)
 4 {
 5   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
 6
 7   DeviceObject = 0i64;
 8   DestinationString = 0i64;
 9   EventHandle = 0i64;
10   EventName = 0i64;
11   McGenEventRegister_EtwRegister();
12   *(_QWORD *)&WPP_MAIN_CB.Type = 0i64;
13   WPP_MAIN_CB.DriverObject = (struct _DRIVER_OBJECT *)&WPP_ThisDir_CTLGUID_Srv2Log;
14   WPP_MAIN_CB.NextDevice = 0i64;
15   WPP_MAIN_CB.CurrentIrp = 0i64;
16   WPP_MAIN_CB.Timer = (PIO_TIMER)1;
17   WppLoadTracingSupport();
18   WPP_MAIN_CB.CurrentIrp = 0i64;
19   WppInitKm();
20   TlgRegisterAggregateProvider();
21   if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
22     && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
23     && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
24   {
25     WPP_SF_(
26       (__int64)WPP_GLOBAL_Control->AttachedDevice,
27       0x16u,
28       (__int64)&WPP_419ff270a959328c24ad6ca01c51a167_Traceguids);
29   }
30   if ( !(unsigned __int8)SrvNetIsDriverLoaded() )
31   {
32     PDEVICE_OBJECT_VAR = WPP_GLOBAL_Control;
33     if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
34       && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
35       && BYTE1(WPP_GLOBAL_Control->Timer) )
36     {
37       WPP_SF_(
38         (__int64)WPP_GLOBAL_Control->AttachedDevice,
39         0x17u,
40         (__int64)&WPP_419ff270a959328c24ad6ca01c51a167_Traceguids);
41     }
42     STATUS = STATUS_DRIVER_UNABLE_TO_LOAD;
43     goto LABEL_32;
44   }
```

**[Figure 67] DriverEntry (first part)**

The beginning of this **DriverEntry routine** is not attractive but deserves brief comments. A **_DEVICE_OBJECT structure (WPP_MAIN_CB)** is initialized, and a kernel-mode event provider, which is associated with a callback (**tlgEnableCallback**) that is called whenever a tracing session interacts with

the provider will be registered, or even the caller will be registered as WMI data provider depending on the value of **WPPTraceSuite variable**. Afterwards there is a tracing indicating that the **DriverEntry routine** has been invoked and there is also a routine (**SrvNetIsDriverLoaded**) that checks whether the **srvnet.sys driver** is loaded before continuing. If it is not, the message *“SrvNet driver is not loaded”* will be logged and **STATUS_DRIVER_UNABLE_TO_LOAD error**.

The **srvnet.sys driver** (“**srv**” means Server Network Driver) is one the most important parts of the SMB (Server Message Block) protocol, and fundamentally acts as a kind of interface between the file network protocol and file sharing protocol. By working as an interface, its key role is to monitor incoming messages and forward any incoming message (SMB2 or SMB3) to **srv2.sys driver**. Additionally, the import table of **srv2.sys** holds diverse SMB routines belonging to **srvnet.sys**. About the code from **Figure 67**, I have renamed variables (**N key**) and applying enumerations (**M key**), as I am going to do from this point onward.

To check functions imported from **svrnet.sys** in **srv2.sys driver** from the command line (PowerShell), readers can install the **NtObjectManager** authored by **James Forshaw** (**@tiraniddo**) as shown below:

- **Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned -Force**
- **Install-Module NtObjectManager -Scope CurrentUser -Force**
- **Update-Module NtObjectManager**

To import the module, execute:

- **Import-Module NtObjectManager**

PS C:\Users\Administrator> **Get-Win32ModuleImport -Path "C:\Windows\System32\drivers\srv2.sys"**

| DllName | FunctionCount | DelayLoaded |
|---|---|---|
| ntoskrnl.exe | 267 | False |
| HAL.dll | 1 | False |
| TDI.SYS | 1 | False |
| **srvnet.sys** | **157** | **False** |
| ext-ms-win-ntos-werkernel-l1-1-1.dll | 5 | False |
| ksecdd.sys | 17 | False |

**PS C:\Users\Administrator> Get-Win32ModuleImport -Path "C:\Windows\System32\drivers\srv2.sys" -Dllname "srvnet.sys" | Select-Object Name -First 8**

```
Name
--------
SrvNetQueryConnectionInformation
SrvNetCloseConnection
SrvNetDisconnectConnection
SrvNetSetConnectionInstanceId
SmbCryptoKeyTableDestroy
SmbCryptoKeyTableCreate
SrvLibApplySrvDeviceAcl
SrvNetIsDriverLoaded
```

PS C:\Users\Administrator> **Get-NtKernelModule | Sort-Object -Property Name | Where-Object Name -Match "^srv"**

```
Name            ImageBase               ImageSize
-------------   --------------------    --------------
srvnet.sys      FFFFF8041B230000        385024
srv2.sys        FFFFF8041B420000        937984
```

Proceeding to the next lines of code, we have:

```
45  RtlInitUnicodeString(
46    &DestinationString,
47    L"\\Device\\Srv2");
48  KeInitializeSpinLock((PKSPIN_LOCK)&WPP_MAIN_CB.Dpc.SystemArgument2);
49  *(_QWORD *)&WPP_MAIN_CB.ActiveThreadCount = &WPP_MAIN_CB.Dpc.DpcData;
50  WPP_MAIN_CB.Dpc.DpcData = &WPP_MAIN_CB.Dpc.DpcData;
51  STATUS_IoCreateDevice = IoCreateDevice(
52                            DriverObject,
53                            0,
54                            &DestinationString,
55                            FILE_DEVICE_NETWORK_FILE_SYSTEM,
56                            FILE_DEVICE_SECURE_OPEN,
57                            0,
58                            &DeviceObject);
59  STATUS = STATUS_IoCreateDevice;
60  if ( STATUS_IoCreateDevice < 0 )
61  {
62    PDEVICE_OBJECT_VAR = WPP_GLOBAL_Control;
63    if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control )
64    {
65      Timer_high = HIDWORD(WPP_GLOBAL_Control->Timer);
66      if ( (Timer_high & 8) != 0 )
67      {
68        if ( BYTE1(WPP_GLOBAL_Control->Timer) )
69          WPP_SF_D(
70            (__int64)WPP_GLOBAL_Control->AttachedDevice,
71            0x18u,
72            (__int64)&WPP_419ff270a959328c24ad6ca01c51a167_Traceguids,
73            STATUS_IoCreateDevice);
74      }
75    }
76    goto LABEL_32;
77  }
78  STATUS = SrvLibApplySrvDeviceAcl(
79             DeviceObject,
80             0x1F01FFi64,
81             0x1200A0i64,
82             0x12019Fi64,
83             0x1F01FF,
84             0x1200A0);
```

**[Figure 68] DriverEntry (second part)**

There is a brief list of observations here:

- **IoCreateDevice function** creates a **device object** for applications that interact with the driver.
- The **device name** is \\Device\\Srv2, as expected.
- The **device type** is **FILE_DEVICE_NETWOR_FILE_SYSTEM**, as expected.
- **FILE_DEVICE_SECURITY_OPEN** was specified for **DeviceCharacteristics parameter**, as recommended.
- **SrvLibApplySrvDeviceAcl function**, from **srvnet.sys**, is used by the driver to apply an ACL (Access Control List) to the device object specified at the first parameter. As I am not sure about the correct order of parameters, I left them as hexadecimal for now.

It follows third part of the code:

```
85   if ( STATUS < 0 )
86   {
87     if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
88       && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
89       && BYTE1(WPP_GLOBAL_Control->Timer) )
90     {
91       WPP_SF_D(
92         (__int64)WPP_GLOBAL_Control->AttachedDevice,
93         0x19u,
94         (__int64)&WPP_419ff270a959328c24ad6ca01c51a167_Traceguids,
95         STATUS);
96     }
97 LABEL_31:
98     IoDeleteDevice((PDEVICE_OBJECT)Srv2DeviceObject);
99     Srv2DeviceObject = 0i64;
100    Srv2ServerProcess = 0i64;
101 LABEL_32:
102    TlgUnregisterAggregateProvider(
103      (__int64)PDEVICE_OBJECT_VAR,
104      Timer_high,
105      v4);
106    WppCleanupKm();
107    Real_Driver_Entry();
108    return STATUS;
109  }
110  Srv2DeviceObject = DeviceObject;
111  CurrentProcess = IoGetCurrentProcess();
112  Srv2DriverState = 0;
113  Srv2ServerProcess = CurrentProcess;
114  memset64(
115    DriverObject->MajorFunction,
116    (unsigned __int64)Srv2DefaultDispatch,
117    0x1Cui64);
118  DriverObject->MajorFunction[IRP_MJ_CLEANUP] = (PDRIVER_DISPATCH)Srv2Cleanup;
119  DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)Srv2Close;
120  DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)Srv2Create;
121  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)Srv2DeviceControl;
122  wil_InitializeFeatureStaging();
123  DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
124  ExInitializeResourceLite((PERESOURCE)&WPP_MAIN_CB.DeviceLock.Header.WaitListHead);
125  RtlInitUnicodeString(
126    &EventName,
127    L"\\KernelObjects\\HighNonPagedPoolCondition");
128  ptr_event_obj = IoCreateNotificationEvent(
129                    &EventName,
130                    &EventHandle);
131  Srv2HighNonPagedPoolConditionEvent = ptr_event_obj;
132  if ( !ptr_event_obj )
133  {
134    STATUS = STATUS_INSUFFICIENT_RESOURCES;
135 LABEL_28:
136    Srv2UnregisterPerfCounterProvider();
137    if ( Srv2HighNonPagedPoolConditionEvent )
138    {
139      ObfDereferenceObject(Srv2HighNonPagedPoolConditionEvent);
140      Srv2HighNonPagedPoolConditionEvent = 0i64;
141    }
142    ExDeleteResourceLite((PERESOURCE)&WPP_MAIN_CB.DeviceLock.Header.WaitListHead);
143    wil_UninitializeFeatureStaging();
144    goto LABEL_31;
145  }
146  ObfReferenceObject(ptr_event_obj);
147  ZwClose(EventHandle);
148  STATUS = Srv2RegisterPerfCounterProvider();
149  if ( STATUS < 0 )
```

**[Figure 69] DriverEntry (third part)**

The third part of the code starts on line 85, and if it didn't get to set Security Descriptor on the device object, so such an object is deleted, both counters (**Srv2DeviceObject** and **Srv2ServerProcess**) are zeroed and the kernel-mode event provider (**EtwUnregister** function within **Real_Driver_Entry** routine) is unregistered before exiting. Additionally, the device object is removed by calling **IoDeleteDevice** function.

If any error has happened, the current processed ID is retrieved by calling **IoGetCurrentProcess function** and the driver dispatch table is filled (remember that there are 28 slots) with pointer to **Srv2DefaultDispatch routine** (lines 114 to 117). Afterwards, a pointer to a specific and meaningful routine is attributed to each relevant slot (dispatch routines): **Srv2Cleanup, Srv2Close, Srv2Create** and **Srv2DeviceControl.** Furthermore, the **DriverUnload routine** is also set.

The call for **ExInitializeResourceLite function** (line 124) can be interesting for readers because, as you remember from previous pages, we did not know about the type of an argument for the same function (check Figure 51, on page 55), and here this line can provide you with a great a direction about how to handle that problem (search for _DRIVER_OBJECT, _KEVENT, and associated structures).

The **IoCreateNotificationEvent routine** is called to create a named notification event (*\\KernelObjects\\HighNonPagedPoolCondition*), which is created, opened and set it up as signaled, and that is employed to notify threads that an event has occurred. If the notification event creation fails, everything is undone and expected tasks follow such as dereferencing objects, releasing resources and locks and closing handles. If the code failed to register srv2.sys as a SMB2 performance counter provider (**Srv2RegisterPerfCounterProvider function** on line 148), so it will send a log message to notify it too.

I will not analyze all dispatch routines but will do only a quick overview about **Srv2Create** and **SrvDeviceContro**l. First, we will examine the **Srv2Create routine**:

```
1  __int64 __fastcall Srv2Create(__int64 a1, IRP *ptr_IRP)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    STATUS = 0;
6    if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
7      && (HIDWORD(WPP_GLOBAL_Control->Timer) & 4) != 0
8      && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
9    {
10     WPP_SF_(
11       (__int64)WPP_GLOBAL_Control->AttachedDevice,
12       0xBu,
13       (__int64)&WPP_8c6de0d290423d6b995ac4d9651f0eb8_Traceguids);
14   }
15   RequestorProcess = IoGetRequestorProcess(ptr_IRP);
16   ProcessServerSilo = (struct _EJOB *)PsGetProcessServerSilo(RequestorProcess);
17   if ( PsIsHostSilo(ProcessServerSilo) )
18   {
19     CurrentStackLocation = ptr_IRP->Tail.Overlay.CurrentStackLocation;
20     CurrentStackLocation->FileObject->FsContext = 0i64;
21     CurrentStackLocation->FileObject->FsContext2 = 0i64;
22   }
23   else
24   {
25     if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
26       && (HIDWORD(WPP_GLOBAL_Control->Timer) & 4) != 0
27       && BYTE1(WPP_GLOBAL_Control->Timer) )
28     {
29       WPP_SF_(
30         (__int64)WPP_GLOBAL_Control->AttachedDevice,
31         0xCu,
32         (__int64)&WPP_8c6de0d290423d6b995ac4d9651f0eb8_Traceguids);
33     }
34     STATUS = STATUS_ACCESS_DENIED;
35   }
36   ptr_IRP->IoStatus.Status = STATUS;
37   IofCompleteRequest(ptr_IRP, IO_NETWORK_INCREMENT);
38   return STATUS;
39 }
```

**[Figure 70] Srv2Create routine**

There are a small number of things happening in this routine, and readers should know that I renamed a few local variables and apply enumerations, which the last one (**IO_NETWORK_INCREMENT**) can be a bit hard to know because of the documentation is a bit vague, but readers can learn about available priority boosts from the following website: https://github.com/MicrosoftDocs/windows-driver-docs/blob/staging/windows-driver-docs-pr/wdf/specifying-priority-boosts-when-completing-i-o-requests.md. The first act is calling the **IoGetRequestorProcess function**, which returns a process' pointer (into the **RequestorProcess variable**) for the thread that requested the I/O operation. if there is not an attached thread, the function returns a point to the process itself. Soon afterwards the **PsGetProcessServerSilo function** returns a reference to sever silo object (basically, a container) for the current process. If **PsGetProcessServerSilo function** returns successfully, the current **IO_STACK_LOCATION** is retrieved and both **FsContext** and **FsContext2** members, which have a meaningful importance for file system drivers, are initialized to zero. If **PsGetProcessServerSilo** call fails, then **STATUS_ACCESS_DENIED** is returned. Finally, **IofCompleteRequest** is called to indicate that the caller (the thread) has completed the necessary processing and is returning the IRP to the I/O manager.

The **Srv2DeviceControl routine** is a quite relevant and, in this case, tricky routine, which is shown below:

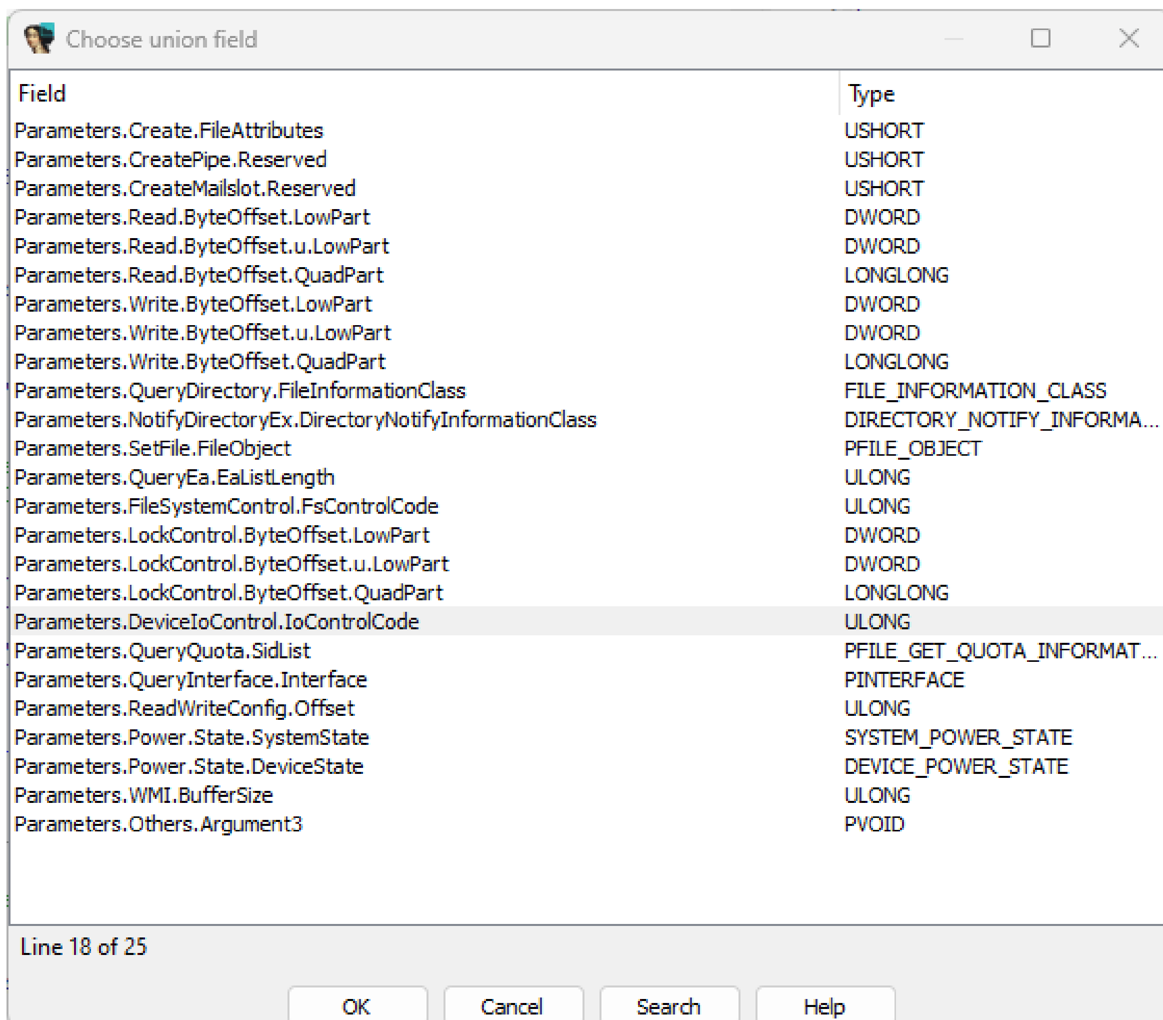
```
 1  __int64 __fastcall Srv2DeviceControl(
 2          __int64 a1,
 3          IRP *ptr_IRP)
 4  {
 5    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
 6
 7    UserBuffer = 0i64;
 8    if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
 9      && (HIDWORD(WPP_GLOBAL_Control->Timer) & 4) != 0
10      && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
11    {
12      WPP_SF_(
13        (__int64)WPP_GLOBAL_Control->AttachedDevice,
14        0xFu,
15        (__int64)&WPP_8c6de0d290423d6b995ac4d9651f0eb8_Traceguids);
16    }
17    if ( IoGetCurrentProcess() == Srv2ServerProcess )
18    {
19      CurrentStackLocation = ptr_IRP->Tail.Overlay.CurrentStackLocation;
20      LowPart = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart;
21      Options = CurrentStackLocation->Parameters.Create.Options;
22      Length = CurrentStackLocation->Parameters.Read.Length;
23      if ( (LowPart & 3) != 0 )
24      {
25        if ( (CurrentStackLocation->Parameters.Read.ByteOffset.LowPart & 3) == 3 )
26        {
27          Parameters = (unsigned __int8 *)CurrentStackLocation->Parameters.CreatePipe.Parameters;
28          UserBuffer = ptr_IRP->UserBuffer;
29          if ( ptr_IRP->RequestorMode == 1 )
30          {
31            ProbeForRead(
32              CurrentStackLocation->Parameters.CreatePipe.Parameters,
33              CurrentStackLocation->Parameters.Create.Options,
34              1u);
35            ProbeForRead(UserBuffer, Length, 1u);
36          }
37        }
```

**[Figure 71] Srv2DeviceControl routine – WRONG decompiling representation**

Before starting any discussion about the **Srv2DeviceControl routine** we must fix its pseudo-code because it is incorrect. Unfortunately, **IO_STACK_LOCATION** is a structure composed of individual fields, but one of them (**Parameters**) is a large union. Therefore, we need to adjust the **Parameter field** for that its member reflects the exact representation according to the dispatch routine (**Create, Read, Write, QueryFile, DeviceIoControl** and so on). If readers analyze the code shown on the **Figure 71**, you will see that parameters are not related to the **DeviceIoControl dispatch routine** that should have the following names:

- **Parameters.DeviceIoControl.OutputBufferLength**
- **Parameters.DeviceIoControl.InputBufferLength**
- **Parameters.DeviceIoControl.IoControlCode**
- **Parameters.DeviceIoControl.Type3InputBuffer**

How to fix this imprecision? The Parameter field's type is correct, and its type is, in fact, **_IO_STACK_LOCATION**. However, the member's type is wrong, and readers need to right click on each one (Read, Create) and change their types by picking up **Select Union Field (ALT+Y).** Once again: **do not** right-click on **Parameters**, but **on the field member of Parameters**. For example, to alter the union field for **LowPart = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart** instruction (**line 20 – wrong representation**), **right-click on Read sub-field** and **press CTRL+Y**, which will show the image below:




**[Figure 72] Altering a Union Field**

As readers can notice, I searched for **DeviceIoControl**, picked up it and clicked OK. Repeat the same steps for similar lines. My suggestion is that for other functions and drivers, if you are not sure about the correct fields, so check this page from Microsoft documentation here: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_io_stack_location. Additionally, you need to rename the fields to mirror its new member's names and apply enumerations. The improved code is shown below:

```
 1 __int64 __fastcall Srv2DeviceControl(
 2         __int64 a1,
 3         IRP *ptr_IRP)
 4 {
 5   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
 6
 7   UserBuffer = (_DWORD *)FALSE;
 8   if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
 9     && (HIDWORD(WPP_GLOBAL_Control->Timer) & 4) != 0
10     && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
11   {
12     WPP_SF_(
13       (__int64)WPP_GLOBAL_Control->AttachedDevice,
14       0xFu,
15       (__int64)&WPP_8c6de0d290423d6b995ac4d9651f0eb8_Traceguids);
16   }
17   if ( IoGetCurrentProcess() == Srv2ServerProcess )
18   {
19     CurrentStackLocation = ptr_IRP->Tail.Overlay.CurrentStackLocation;
20     IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
21     InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
22     OutputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength;
23     if ( (IoControlCode & 3) != 0 )
24     {
25       if ( (CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode & 3) == IRP_MJ_READ )
26       {
27         Type3InputBuffer = (unsigned __int8 *)CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer;
28         UserBuffer = ptr_IRP->UserBuffer;
29         if ( ptr_IRP->RequestorMode == UserMode )
30         {
31           ProbeForRead(
32             CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer,
33             CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength,
34             1u);
35           ProbeForRead(UserBuffer, OutputBufferLength, 1u);
36         }
37       }
38       else
39       {
40         Type3InputBuffer = (unsigned __int8 *)ptr_IRP->AssociatedIrp.SystemBuffer;
41         if ( (_DWORD)OutputBufferLength )
42         {
43           MdlAddress = ptr_IRP->MdlAddress;
44           UserBuffer = (MdlAddress->MdlFlags & (MDL_SOURCE_IS_NONPAGED_POOL|MDL_MAPPED_TO_SYSTEM_VA)) != 0
45                      ? MdlAddress->MappedSystemVa
46                      : MmMapLockedPagesSpecifyCache(
47                          MdlAddress,
48                          KernelMode,
49                          MmCached,
50                          (PVOID)KernelMode,
51                          FALSE,
52                          MdlMappingNoExecute|NormalPagePriority);
53           if ( !UserBuffer )
54           {
55             STATUS = STATUS_INSUFFICIENT_RESOURCES;
56             goto LABEL_6;
57           }
58         }
59       }
60     }
```

**[Figure 73] Improved representation of Srv2DeviceControl routine (part 1)**

```
61      else
62      {
63        UserBuffer = ptr_IRP->AssociatedIrp.SystemBuffer;
64        Type3InputBuffer = (unsigned __int8 *)UserBuffer;
65      }
66      ret_status = Srv2ProcessFsctl(
67                     (__int64)CurrentStackLocation->FileObject,
68                     v4,
69                     Type3InputBuffer,
70                     InputBufferLength,
71                     UserBuffer,
72                     OutputBufferLength,
73                     IoControlCode,
74                     (__int64)&ptr_IRP->IoStatus,
75                     v14,
76                     ptr_IRP);
77    }
78    else
79    {
80      ret_status = Srv2QueueIrpToSystem(ptr_IRP);
81    }
82    STATUS = ret_status;
83    if ( ret_status != STATUS_PENDING )
84    {
85  LABEL_6:
86      ptr_IRP->IoStatus.Status = STATUS;
87      IofCompleteRequest(ptr_IRP, 2);
88    }
89    return STATUS;
90  }
```

**[Figure 74] Improved representation of Srv2DeviceControl routine (part 2)**

There is a series of comments to do about the improved version of the code from Figures 73 and 74:

- On **lines 20, 21, 22, 25, 27, 32** and **33,** the previously described approach of adapting the correct structure to the union field (**ALT+Y**) has been applied.

- On the same lines mentioned above, pay attention to the fact that I renamed all local variables to the same name as their field members.

- Using enumerations (**M**), I applied a symbolic representation on **lines 25** and **29**, and I needed to insert **_MODE enumeration** (check **Figure 75**). However, readers need always to take care not to apply the wrong symbol, which demands attention on the interpretation and the involved context.

- The **ProbeForRead function** (**lines 31 and 35**) checks whether the user-mode buffer resides in the user-mode space, and the value (**UserMode** symbol) on **line 29** matches exactly with this purpose.

- **On line 40**, I applied the same technique of changing to the correct structure in a union field (**ALT+Y**), but this time the trouble structure was **AssociatedIrp**. If readers check the IRP documentation, you will see that as the code manages data's content then I needed to change to **SystemBuffer**. Other options would be **MasterIrp** and **IrpCount**. Check the documentation: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_irp.

- **On line 44**, to apply flags I had to add **MACRO_MDL enumeration** first: **SHIFT+F10 → INSERT → Add standard enum by enum name → MACRO_MDL**. Check **Figure 75**:

```
FFFFFFFF ; enum _MODE, copyof_578, width 4 bytes
FFFFFFFF KernelMode        = 0
FFFFFFFF UserMode          = 1
FFFFFFFF MaximumMode       = 2
FFFFFFFF
00000001 ; ---------------------------------------------------
00000001
00000001 ; enum MACRO_MDL, copyof_579, bitfield, width 4 bytes
00000001 MDL_MAPPED_TO_SYSTEM_VA  = 1
00000002 MDL_PAGES_LOCKED  = 2
00000004 MDL_SOURCE_IS_NONPAGED_POOL  = 4
00000008 MDL_ALLOCATED_FIXED_SIZE  = 8
00000010 MDL_PARTIAL       = 10h
00000020 MDL_PARTIAL_HAS_BEEN_MAPPED  = 20h
00000040 MDL_IO_PAGE_READ  = 40h
00000080 MDL_WRITE_OPERATION  = 80h
00000100 MDL_LOCKED_PAGE_TABLES  = 100h
00000100 MDL_PARENT_MAPPED_SYSTEM_VA  = 100h
00000200 MDL_FREE_EXTRA_PTES  = 200h
00000400 MDL_DESCRIBES_AWE  = 400h
00000800 MDL_IO_SPACE      = 800h
00001000 MDL_NETWORK_HEADER  = 1000h
00002000 MDL_MAPPING_CAN_FAIL  = 2000h
00004000 MDL_PAGE_CONTENTS_INVARIANT  = 4000h
00004000 MDL_ALLOCATED_MUST_SUCCEED  = 4000h
00008000 MDL_INTERNAL      = 8000h
```

**[Figure 75] _MODE and _MDL enumerations**

- Initially, the code is looking for a corresponding system virtual memory mapping that already represents and describes buffer described by the MDL. As the instruction makes part of a ternary operator then once the system virtual address is already available, it returns the pointer to such a memory. If it is not available then the **MnMapLockedPagesSpecifyCache function** will map physical pages made accessible and available by a MDL which describes the layout of the virtual memory buffer in physical memory, to a virtual address. Additionally, it also allows the caller to specify the cache attribute (from **_MEMORY_CACHING_TYPE** enumeration) during the mapping process. As expected, **MmCached** means that the memory request should be cached.

- Another note is that this part of the code (from **line 39 to line 59**) handles with **KernelMode** access mode, while code from **line 23 to line 37** handles with **Usermode access**.

- **On Line 52** there is a good point to comment and that certainly readers could wonder how I got both values appeared as an OR operation. The natural operation would be to add the **_MM_PAGE_PRIORITY enumeration**, however it holds only the standard values that are **LowPagePriority, NormalPagePriority** and **HighPagePriority**. The issue is that since Windows 8 there are another two available values, **MdlMappingNoWrite** and **MdlMappingNoExecute**, which are bitwise-Ored and are added separately, each one in its own enumeration, into the IDB database. Thus, one of potential approaches to manage this problem is:

  - Create a new enumeration and provide any name you want for it.

  - Add **NormalPagePriority (0x10)** and **HighPagePriority (0x20)** as members.

- Add **MdlMappingNoExecute (0x40000000)** as a **bitwised member** (there is a check box for this purpose) and attribute the same value for the mask. It may want to provide **MM_PAGE_PRIORITY** as mask name.

- Add **MdlMappingNoWrite (0x80000000)** as a **bitwised member**, and pickup as mask the same value and mask name.

- Once the enumeration is created, it is just to apply it on the value. The result should be identical to shown on **line 52**.

- The **enumeration** will be look like as shown below:

```
00000010 ; enum MACRO_MM_PAGE_PRIORITY_COMPLETE, mappedto_586, bitfield, width 4 bytes
00000010 NormalPagePriority  = 10h
00000020 HighPagePriority  = 20h
40000000 MdlMappingNoExecute  = 40000000h
80000000 MdlMappingNoWrite  = 80000000h
80000000
```

**[Figure 76] Customized enumerations**

- On **line 63**, it is the same case as line 40, where it is requested to change the field to be used because it is involved with manipulation of the buffer.

- On **line 66**, the **Srv2ProcessFsctl routine** processes **FSCTLs (File System Control Requests)** that comes from the client, and in this case the routine seems to receive (read) and process such data. Readers might remember that a **file system control code (FSCTL)** is a command sent to the file system for querying or changing a current behavior.

- On **line 80**, the **Srv2QueueIrptoSystem routine** goal is to queue an IRP to a system thread pool with the objective of improving the performance:

```
 1 MACRO_STATUS __fastcall Srv2QueueIrpToSystem(_IRP *Context)
 2 {
 3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
 4
 5   WorkItem = IoAllocateWorkItem((PDEVICE_OBJECT)Srv2DeviceObject);
 6   if ( !WorkItem )
 7     return 0xC000009Ai64;      // STATUS_INSUFFICIENT_RESOURCES
 8   Context->Tail.Overlay.CurrentStackLocation->Control |= SL_PENDING_RETURNED;
 9   IoQueueWorkItemEx(
10     WorkItem,
11     (PIO_WORKITEM_ROUTINE_EX)Srv2ProcessQueuedIrp,
12     DelayedWorkQueue,
13     Context);
14   return STATUS_PENDING;
15 }
```

**[Figure 77] Srv2QueueIrpToSystem**

- There are good details about this routine:

- The driver uses **IoAllocateWorkItem function** to allocate a work item, which is a kind of task represented by a **_IO_WORKITEM structure** that is used to perform delayed processing and represents an associated callback that it is the responsible for executing any related and necessary processing. Alternatively, the driver could use the **IoInitializeWorkItem function** to initialize a given and existing buffer as a work item.

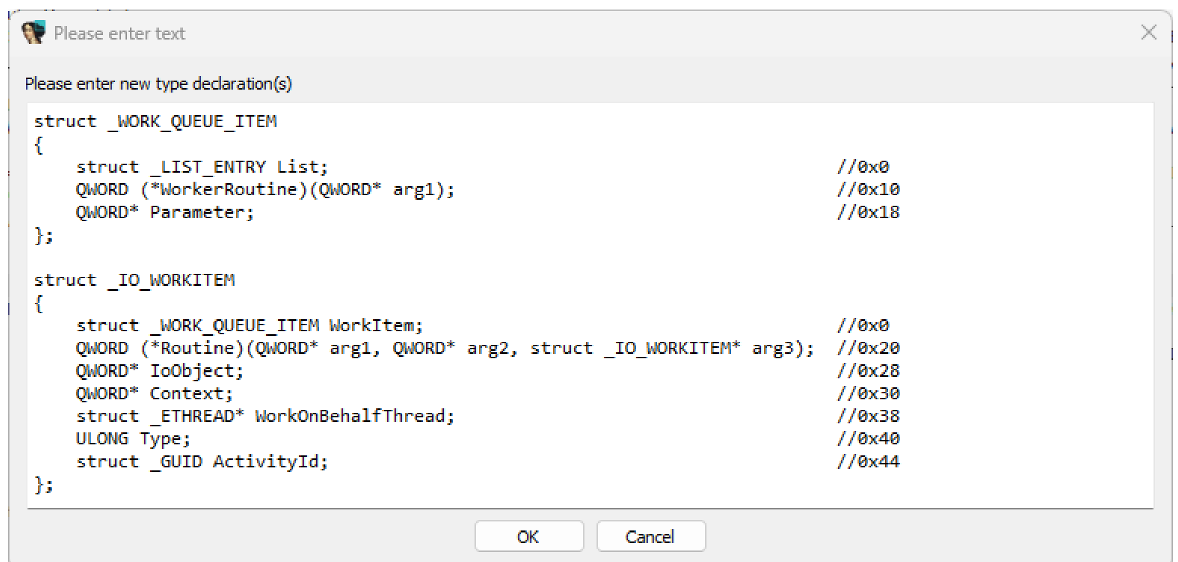
- The **_IO_WORKITEM structure** is an opaque (not documented by Microsoft) structure that describes a work item while using it for a system worker thread.

- As readers probably have already noticed, work items are very similar to **DPC (Deferred Procedure Call)** runs, but work item always executes at **IRQL == PASSIVE_LEVEL** while **DPC at IRQ 2**, even though DPC can delegate additional operations to work items to execute them at **IRQL == PASSIVE_LEVEL**, which could be more appropriate.

- The **work item** is associated with a **Workitem routine (IO_WORKITEM_ROUTINE)** by calling **IoQueueWorkItem** (or **IoQueueWorkItemEx**) function (check documentation: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioqueueworkitemex). This function also adds the work item in a queue for later processing, so working as an asynchronous operation.

- **Virgilius Project** provides us with necessary structure definitions for creating the **_IO_WORKITEM structure** in the IDB, whether this is necessary, by going to **Views → Open subviews → Local Types** or **SHIFT+F1 shortcut**. Afterwards, press **INSERT key** and type definitions as shown below:

    - https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_IO_WORKITEM.
    - https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_WORK_QUEUE_ITEM.




**[Figure 78] Creating an _IO_WORKITEM in IDB**

- I have avoided commenting about other routines because this discussion is already too long, but **Srv2ProcessFsctl routine** is involved with the starting of the driver, instance and server process related to SMB. Curiously, the second and nineth parameters are not initially used, but this fact deserves to be analyzed carefully and we will not do it here.

It is time to return to **DriverEntry routine** (Figure 69), where **IoCreateNotificationEvent function** (**line 129**) creates (or open) a named notification event (*HighNonPagedPoolConditions*) to notify threads when this event occurs. In general, synchronization and notification events are used to control the execution of a piece of code.

In terms of code, it is enough, and I hope readers have realized how complex can be a real kernel driver, and it takes time to make a simple analysis and get a reasonable comprehension about its working.

Change directions of our analysis, there other methods to collect additional information about the driver, and one of them is to use **Driver Buddy Reload plugin** by going to menu **Edit → Plugins → Driver Buddy Reloaded** or pressing **CTRL+ALT+A**. The output is long and saved into a log file, but in our case the plugins reported a summary of the following items:

- **Two DeviceNames**: **\Device\Srv2** and **\Device\NamedPipes**

- Countless routines using **memmove function**, which is always a candidate a security issues such as **out-of-bounds read/write vulnerabilities** and **overflows**.

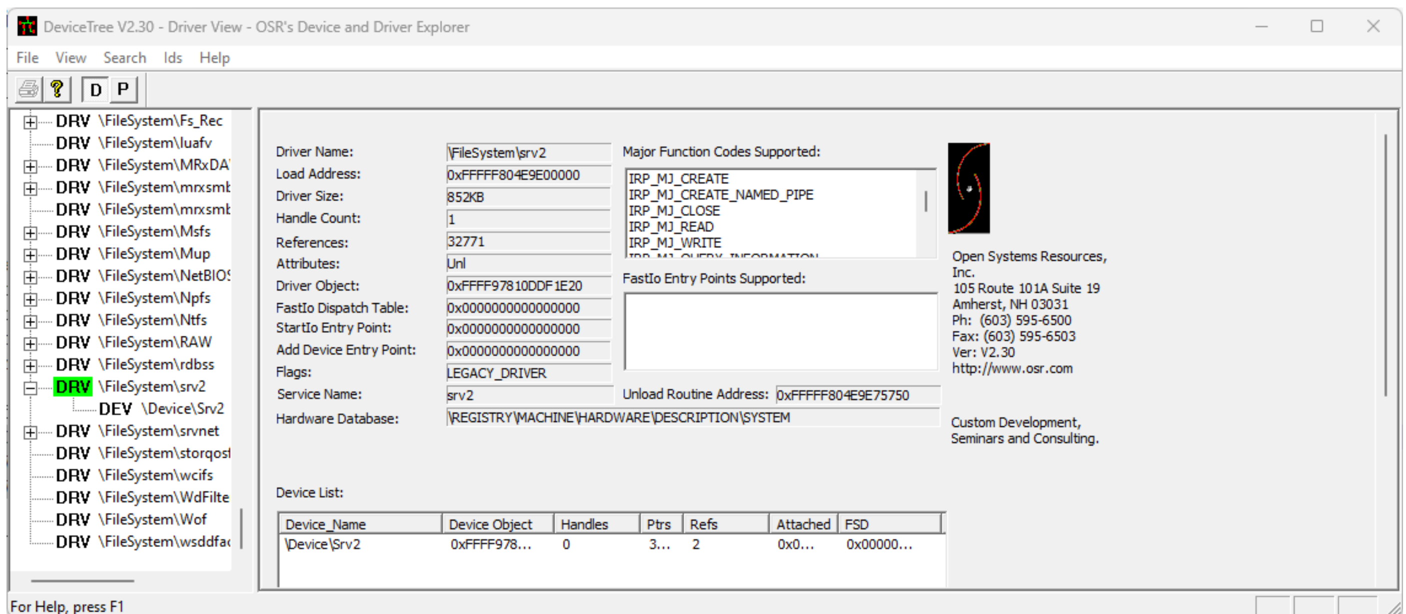
- A brief list of interesting APIs such as:

    - **MmBuildMdlForNonPagedPool**
    - **MmMapLockedPagesSpecifyCache**
    - **MmGetSystemRoutineAddress**
    - **ObfReferenceObject, ObfDereferenceObject**
    - **IofCallDriver**
    - **ZwFsControlFile, ZwCreateEvent, ZwClose, ZwReadFile**
    - **MmSizeOfMdl, MmUnlockPages**
    - **RtlCompareMemory**

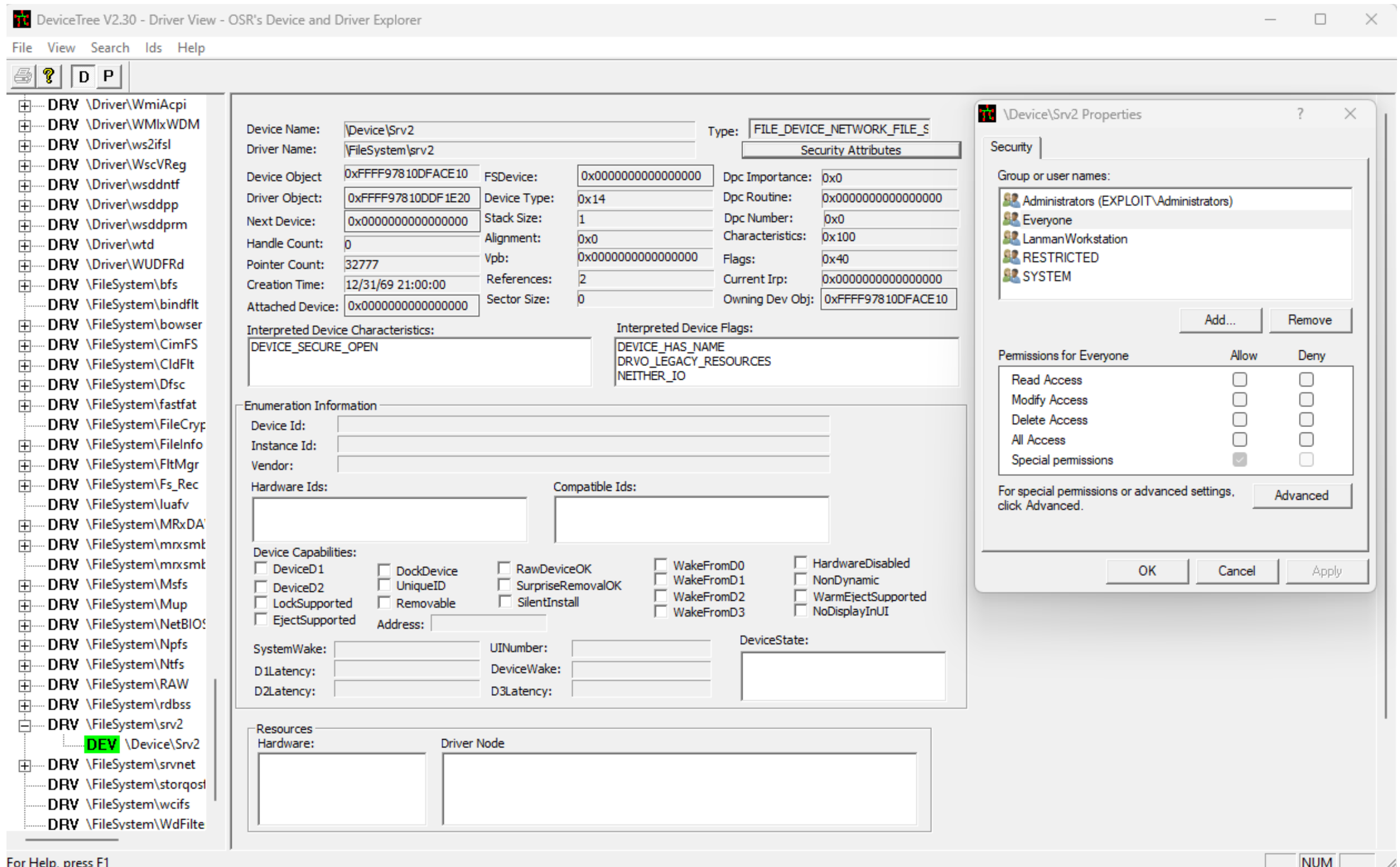
- Three IOCTLs:

    - **0x70214 | FILE_DEVICE_DISK | METHOD_BUFFERED 0 | FILE_ANY_ACCESS (0)**
    - **0x4D0000 | FILE_DEVICE_VIRTUAL_BLOCK | METHOD_BUFFERED 0 | FILE_ANY_ACCESS (0)**
    - **0x4D0008 | FILE_DEVICE_VIRTUAL_BLOCK | METHOD_BUFFERED 0 | FILE_ANY_ACCESS (0)**

The followed approaches so far and artifacts that we just found provide readers with guidelines and directions (the report provides you with each artifact and its respective address) to research and understand whether there is any security issue that can be found in this driver (and any other by following the same approach) beyond vulnerabilities officially reported. Although a dynamic approach will be left for future articles, readers can gather relevant information about **srv2.sys** through **DeviceTree tools**, as shown previously:



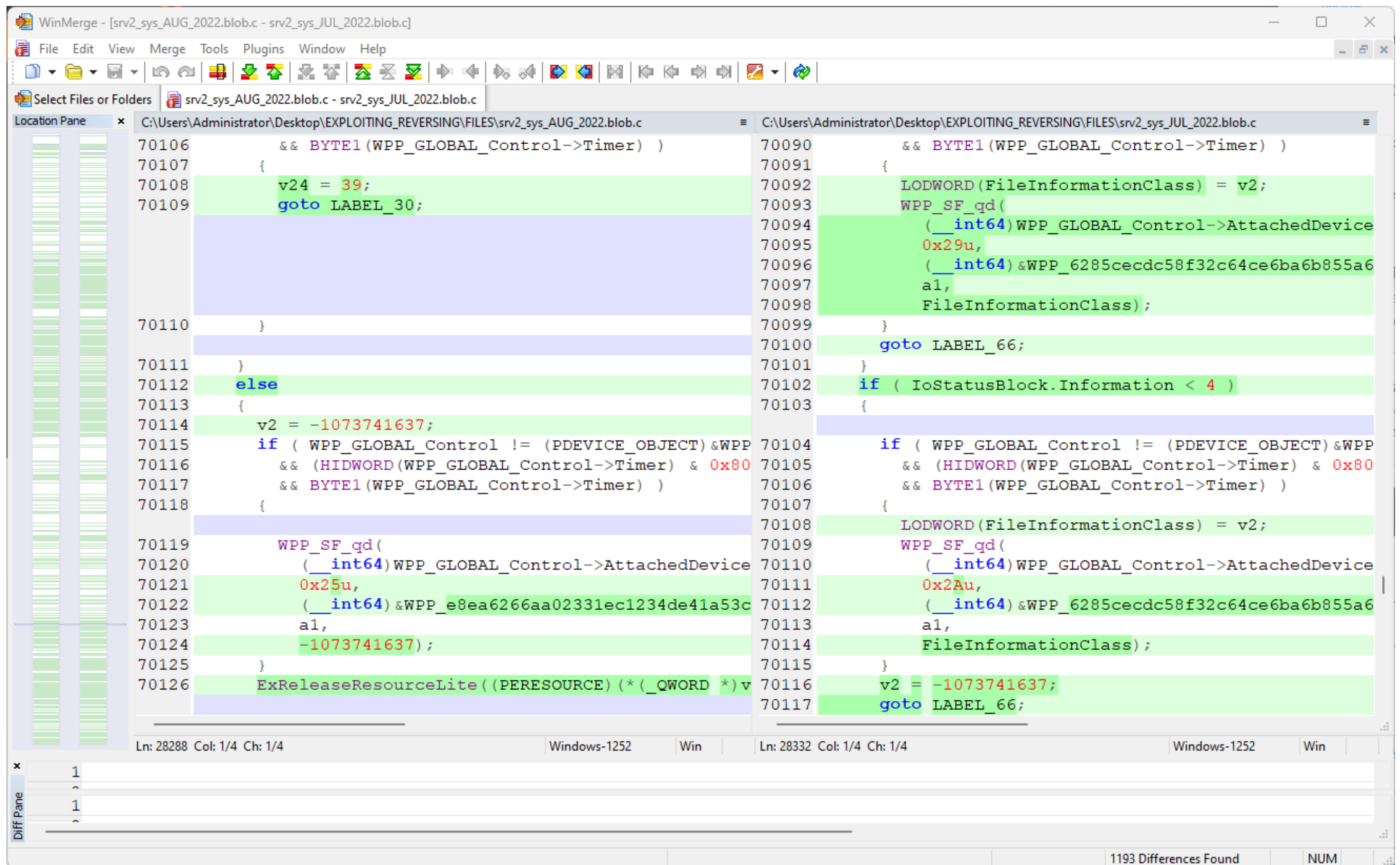
**[Figure 79] DeviceTree: checking srv2.sys (1)**



**[Figure 80] DeviceTree: checking srv2.sys (2)**

We got valuable information from **DeviceTree tool** and **Driver Buddy Reloaded plugin**, and this set of artifacts will provide us with valuable information that will help to device how proceed to interact with the driver dynamically, when it will be necessary to join information collected from static analysis with the understanding of the real time interaction with the driver, which can confirm hypothesis or not, but that fundamentally brings context and a better picture for the analysis.

As a last suggestion, one simple and sometimes good approach for quickly locating potential differences between two binaries is to generate (export) a pseudo code from IDA Pro (**File → Produce File → Create C File or CTRL+F5 shortcut**) for both versions of the srv2.sys (or any other binary) and compare them using **WinMerge** (https://winmerge.org/?lang=en), which does a good job:




**[Figure 81] WinMerge: comparing pseudo codes**

WinMerge's purpose is different from **BinDiff** and **Diaphora**, but it can be useful for initial verification.

## 13. References

For readers that might be interested in learning details about topics mentioned here, a brief list of valuable resources follows below:

- **Microsoft Learn:** https://learn.microsoft.com/en-us/windows-hardware/drivers/
- **Windows drivers samples:** https://github.com/Microsoft/Windows-driver-samples
- **Windows Internals 7th edition book (Parts 1 and 2)** by Pavel Yosifovich , Alex Ionescu, Mark Russinovich, David Solomon, and Andrea Allievi.
- **Practical Reverse Engineering** by Bruce Dang, Alexandre Gazet and Elias Bachaalany.
- **Developing Drivers with the Windows Driver Foundation** by Penny Orwick**.**
- **Authors of prominent articles and presentations (alphabetic order)**: Ilja Van Sprundel, Mateusz Jurczyk, Morten Schenk, Nícolas Economou, Sébastien Duquette, Tarjei Mandt, and others.

## 14. Blogs and channels:

A list of excellent websites, channels, and their respective Twitter handles, sorted by author **in alphabetical order**, follows below:

- https://hasherezade.github.io/articles.html **(by Aleksandra Doniec: @hasherezade)**
- https://malwareunicorn.org/#/workshops **(by Amanda Rousseau: @malwareunicorn)**
- https://russianpanda.com/ **(by Ann: @AnFam17)**
- https://captmeelo.com/ **(by Capt. Meelo: @CaptMeelo)**
- https://csandker.io/ **(by Carsten Sandker: @0xcsandker)**
- https://chuongdong.com/ **(by Chuong Dong: @cPeterr)**
- https://doar-e.github.io/ **(Diary of a reverse-engineer)**
- https://elis531989.medium.com/ **(by Eli Salem: @elisalem9)**
- https://www.youtube.com/@allthingsida **(by Elias Bachaalany: @0xeb and @allthingsida)**
- https://googleprojectzero.blogspot.com/ **(Google Project Zero)**
- https://www.hexacorn.com/index.html **(@Hexacorn)**
- https://hex-rays.com/blog/ **(by Hex-Rays: @HexRaysSA)**
- https://github.com/Dump-GUY/Malware-analysis-and-Reverse-engineering **(by Jiří Vinopal: @vinopaljiri)**
- https://kienmanowar.wordpress.com/ **(by Kien Tran Trung: @kienbigmummy)**
- https://www.inversecos.com/ **(by Lina Lau: @inversecos)**
- https://maldroid.github.io/ **(Łukasz Siewierski: @maldr0id)**
- https://github.com/mnrkbys **(by Minoru Kobayashi: @unkn0wnbit)**
- https://voidsec.com/member/voidsec/ **(by Paolo Stagno: @Void_Sec)**
- https://www.youtube.com/@OffByOneSecurity **(by Stephen Sims: @Steph3nSims)**
- https://windows-internals.com/author/yarden/ (**by Yarden Shafir @yarden_shafir**)

## 15. Conclusion

This article has only scratched the surface of this topic, but it can be used as an introduction to binary diffing and also as the second part of the static analysis for drivers. A natural continuation of this topic is the dynamic analysis to learn how to interact as a client, how to debug and how to perform fuzzing the driver.

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

**Alexandre Borges**