Part 1: Fs Minifilter Hooking

July 11, 2020

Aviad Shamriz

Jul 10, 2020

This post is the first part of series about hooking minifilter/miniport objects. During the course of this series I will explain how the management of these objects works, focusing where various callbacks reside in memory, The manner in which they are called by the manager driver, and how we might be able hook them without triggering PatchGuard.

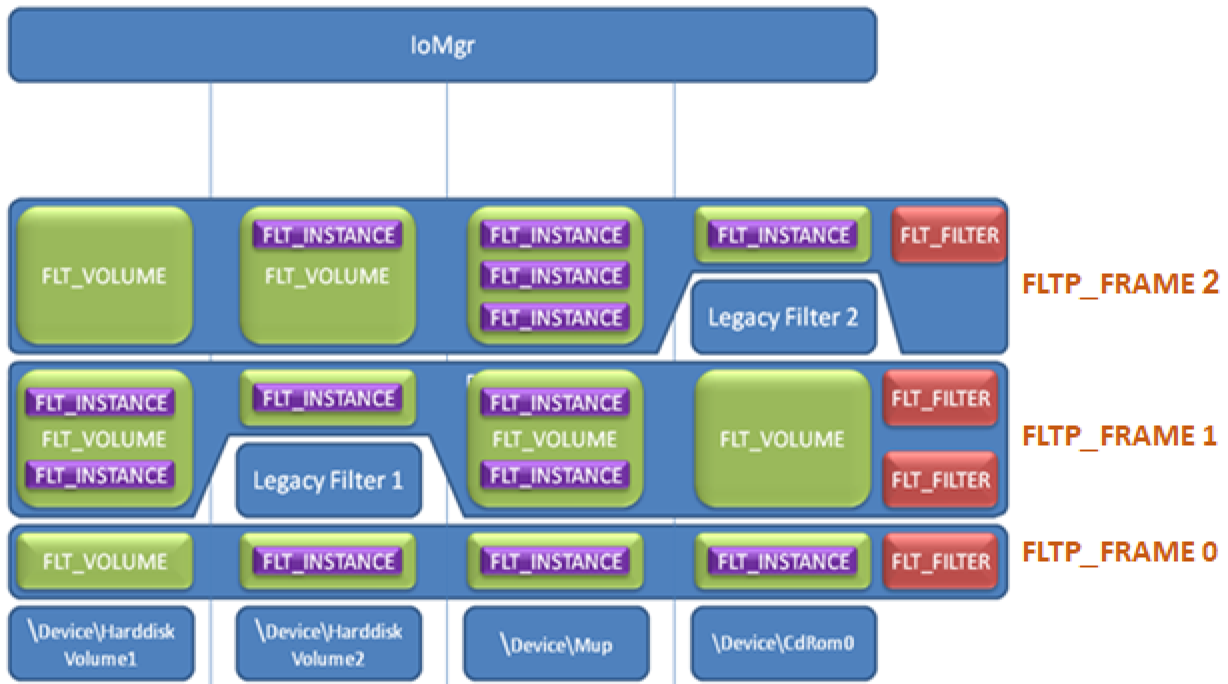
The Filter Manager (FltMgr.sys) is a kernel component that allows other drivers to install callbacks that intercept file system operations. The filter manager is a replacement of the legacy file system filter driver model, and operates at the same level. Instead of consume the callback only by himself it sends callback to all consumers that registered to him.

Minifilter driver is commonly used in security components that have kernel driver (For example: AV, EDR or EPP). There are security components that use it to gather information about operations in file system and use it to make rules to detect unusual behavior, others use it for protects their files from evil that wants to damage their files. I want to show you how rootkit can tamper or filter these callbacks to the minifilter driver.

Before beginning to dig into the internals of the Filter Manager, which will help us understand where we can perform our hook, I recommend that you read a high level description of the Filter Manger here and look at the code for minifilter driver (E.g., You can see this sample of Microsoft's code).

Filter Manager Internals

FltMgr uses a number of structs that simplify the development of a Minifilter. Most of the structs are undocumented and only accessed by an opaque pointer.



From the blog

This diagram shows us the layers of the Filter Manager, each layer represented by the following structs:

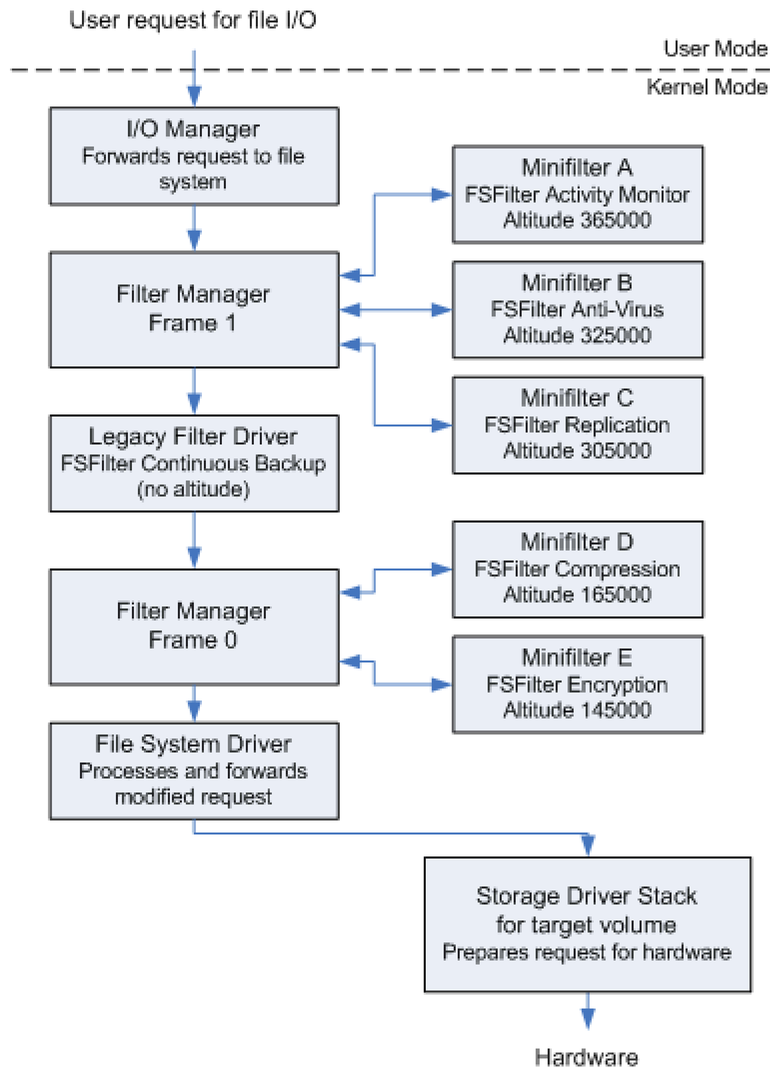
FLTP_FRAME

FltMgr uses a concept called “frames” to enable minifilters to be placed before or after legacy filter drivers. FltMgr can add frame before, after or between legacy filter drivers (If there aren’t legacy filter drivers, FltMgr will use only one frame — “Frame 0”). From the perspective of a legacy filter driver, each frame represents a single legacy filter driver. Each frame contains a range of altitudes that are allowed for minifilters attached to it

```
0: kd> dt fltmgr!_FLTP_FRAME 0xffffba07c66c3010
   +0x000 Type                : _FLT_TYPE
   +0x008 Links               : _LIST_ENTRY [ 0xfffff802`1bd7a640 - 0xfffff802`1bd7a640 ]
   +0x018 FrameID             : 0
   +0x020 AltitudeIntervalLow : _UNICODE_STRING "0"
   +0x030 AltitudeIntervalHigh : _UNICODE_STRING "409800"
```

In this example we can see that the range of altitudes of “Frame 0” is 0–409500.

Each minifilter is registered with a unique altitude that guides the filter manager to load it within the corresponding frame. For example:



From .

In this example we can see that there is a legacy filter driver responsible for some backup functionality. To backing up files before they are encrypted or compressed, the minifilters that perform these tasks belong to “Frame 0” which is below the backup legacy filter driver. This allows the encryption and compression operations to be performed after the backup operations.

FLTP_FRAME is a private struct that is not used by minfilter writers.

FLT_VOLUME

It represents the attachment of FLTP_FRAME to a volume, i.e. each frame has a FLT_VOLUME for each volume (For example: In the above diagram “Frame 0” has four FLT_VOLUME structures). Let’s look at the volume list:

```
0: kd> !fltkd.volumes

Volume List: ffffba07c66c3140 "Frame 0"
   FLT_VOLUME: ffffba07c69da6b0 "\Device\Mup"
      FLT_INSTANCE: ffffba07c68578e0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07c69ea9e0 "FileInfo" "40500"
   FLT_VOLUME: ffffba07c69f35a0 "\Device\HarddiskVolume2"
      FLT_INSTANCE: ffffba07cdc04010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffffba07c68a88a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07cc662010 "luafv" "135000"
      FLT_INSTANCE: ffffba07c68988a0 "Wof Instance" "40700"
      FLT_INSTANCE: ffffba07c689e8a0 "FileInfo" "40500"
   FLT_VOLUME: ffffba07c6bf93a0 "\Device\HarddiskVolume1"
      FLT_INSTANCE: ffffba07c68418a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07c6bfa920 "Wof Instance" "40700"
      FLT_INSTANCE: ffffba07c6c519f0 "FileInfo" "40500"
   FLT_VOLUME: ffffba07cac07010 "\Device\NamedPipe"
      FLT_INSTANCE: ffffba07ca562cb0 "npsvctrig" "46000"
```

Take one of this FLT_VOLUME and see its fields:

```
0: kd> !fltkd.volume 0xffffba07c69f35a0

FLT_VOLUME: ffffba07c69f35a0 "\Device\HarddiskVolume2"
   FLT_OBJECT: ffffba07c69f35a0  [04000000] Volume
      RundownRef               : 0x0000000000000caa (1621)
      PointerCount             : 0x00000001
      PrimaryLink              : [ffffba07c6bf93b0-ffffba07c69da6c0]
   Frame                    : ffffba07c66c3010 "Frame 0"
   Flags                    : [00000564] SetupNotifyCalled EnableNameCaching FilterAttached +500!!
   FileSystemType           : [00000002] FLT_FSTYPE_NTFS
   VolumeLink               : [ffffba07c6bf93b0-ffffba07c69da6c0]
   DeviceObject             : ffffba07c67d8d60
   DiskDeviceObject         : ffffba07c67409f0
   FrameZeroVolume          : ffffba07c69f35a0
   VolumeInNextFrame        : 0000000000000000
   Guid                     : "\??\Volume{a6e9b1fb-0000-0000-0000-501f00000000}"
   CDODeviceName            : "\Ntfs"
   CDODriverName            : "\FileSystem\Ntfs"
   TargetedOpenCount        : 1615
   Callbacks                : (ffffba07c69f36c0)
   ContextLock              : (ffffba07c69f3aa8)
   VolumeContexts           : (ffffba07c69f3ab0)  Count=0
   StreamListCtrls          : (ffffba07c69f3ab8)  rCount=7026
   FileListCtrls            : (ffffba07c69f3b38)  rCount=0
   NameCacheCtrl            : (ffffba07c69f3bb8)
   InstanceList             : (ffffba07c69f3640)
      FLT_INSTANCE: ffffba07cdc04010 "bindflt Instance" "409800"
      FLT_INSTANCE: ffffba07c68a88a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07cc662010 "luafv" "135000"
      FLT_INSTANCE: ffffba07c68988a0 "Wof Instance" "40700"
      FLT_INSTANCE: ffffba07c689e8a0 "FileInfo" "40500"
```

Interesting fields in this struct:

- Frame — The frame that contains the volume.
- DeviceObject — Is the device object that associated with the volume.

- Callbacks — This field is a pointer to a struct that contains an array of callbacks. This is an important field, which we will elaborate upon further along.

FLT_FILTER

As can be inferred from the name of the struct, it represents a minifilter driver. Each filter belongs to a FLTP_FRAME corresponding to its altitude.
The list of filters belonging to Frame 0 on a machine might look something like this:

```
0: kd> !fltkd.filters

Filter List: ffffba07c66c30c0 "Frame 0"
   FLT_FILTER: ffffba07cdac7010 "bindflt" "409800"
      FLT_INSTANCE: ffffba07cdc04010 "bindflt Instance" "409800"
   FLT_FILTER: ffffba07c673e9a0 "WdFilter" "328010"
      FLT_INSTANCE: ffffba07c68578e0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07c68a88a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07c68418a0 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffba07c67d28a0 "WdFilter Instance" "328010"
   FLT_FILTER: ffffba07cc65e430 "storqosflt" "244000"
   FLT_FILTER: ffffba07cc5ee9a0 "wcifs" "189900"
   FLT_FILTER: ffffba07cc4b5a00 "CldFlt" "180451"
   FLT_FILTER: ffffba07c68048a0 "FileCrypt" "141100"
   FLT_FILTER: ffffba07cc660010 "luafv" "135000"
      FLT_INSTANCE: ffffba07cc662010 "luafv" "135000"
   FLT_FILTER: ffffba07c69318a0 "npsvctrig" "46000"
      FLT_INSTANCE: ffffba07ca562cb0 "npsvctrig" "46000"
   FLT_FILTER: ffffba07c66e88a0 "Wof" "40700"
      FLT_INSTANCE: ffffba07c68988a0 "Wof Instance" "40700"
      FLT_INSTANCE: ffffba07c6bfa920 "Wof Instance" "40700"
   FLT_FILTER: ffffba07c66d68a0 "FileInfo" "40500"
      FLT_INSTANCE: ffffba07c69ea9e0 "FileInfo" "40500"
      FLT_INSTANCE: ffffba07c689e8a0 "FileInfo" "40500"
      FLT_INSTANCE: ffffba07c6c519f0 "FileInfo" "40500"
      FLT_INSTANCE: ffffba07c690f8a0 "FileInfo" "40500"
```

In the above case, only one frame ("Frame 0") is available and all the filters are loaded within it. We can observer the pointers, the altitude and the name of each filter belonging to the frame (One of the filters here is luafv, I recommend to read about it in the following blog).

Let's take look at the FLT_FILTER struct of the "wcifs" driver:

```
0: kd> dt fltmgr!_FLT_FILTER 0xffffba07cc5ee9a0
   +0x000 Base             : _FLT_OBJECT
   +0x030 Frame            : 0xffffba07`c66c3010 _FLTP_FRAME
   +0x038 Name             : _UNICODE_STRING "wcifs"
   +0x048 DefaultAltitude  : _UNICODE_STRING "189900"
   +0x058 Flags            : 0x16 (No matching name)
   +0x060 DriverObject     : 0xffffba07`cc589e20 _DRIVER_OBJECT
   +0x068 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x0e8 VerifierExtension : (null)
   +0x0f0 VerifiedFiltersLink : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
   +0x100 FilterUnload     : 0xfffff802`17b0d320     long  wcifs!WcUnload+0
   +0x108 InstanceSetup    : 0xfffff802`17b0c140     long  wcifs!WcInstanceSetup+0
   +0x110 InstanceQueryTeardown : 0xfffff802`17b0cb80     long  wcifs!WcInstanceQueryTeardown+0
   +0x118 InstanceTeardownStart : 0xfffff802`17b0cbe0     void  wcifs!WcInstanceTeardownComplete+0
   +0x120 InstanceTeardownComplete : 0xfffff802`17b0cbe0     void  wcifs!WcInstanceTeardownComplete+0
   +0x128 SupportedContextsListHead : 0xffffba07`cc6269a0 _ALLOCATE_CONTEXT_HEADER
   +0x130 SupportedContexts : [7] (null)
   +0x168 PreVolumeMount   : (null)
   +0x170 PostVolumeMount  : (null)
   +0x178 GenerateFileName : 0xfffff802`17b0cbf0     long  wcifs!WcGenerateFileName+0
   +0x180 NormalizeNameComponent : (null)
   +0x188 NormalizeNameComponentEx : 0xfffff802`17b0ccb0     long  wcifs!WcNormalizeNameComponent+0
   +0x190 NormalizeContextCleanup : (null)
   +0x198 KtmNotification  : (null)
   +0x1a0 SectionNotification : (null)
   +0x1a8 Operations       : 0xffffba07`cc5eec50 _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
```

We will focus at some of its fields

- Frame — The frame that the filter belongs to.
- DriverObject — The driver object that registered the filter.
- FilterUnload — The callback that is invoked when the filter unloads.
- Operations — A pointer to array of type FLT_OPERATION_REGISTRATION, the struct contains the operation callbacks of the minifilter. The structure taken from the struct FLT_REGISTRATION that is argument of the function FltRegisterFilter (This function responsible to register a new filter). Each cell in the array represent another major function (For example: 0x0 is IRP_MJ_CREATE). In FLT_OPERATION_REGISTRATION there is the pre operation callback that invoked before the operation and post operation callback that invoked after the operation complete. The first five operations of the wcifs filter:

```
0: kd> dt -oca60 0xffffba07`cc5eec50 fltmgr!_FLT_OPERATION_REGISTRATION
[0] @ ffffba07`cc5eec50 MajorFunction 0 ''  Flags 0  PreOperation 0xfffff802`17af41d0  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreCreate+0  PostOperation 0xfffff802`17af4910  _FLT_POSTOP_CALLBACK_STATUS  wcifs!WcPostCreate+0  Reserved1 (null)
[1] @ ffffba07`cc5eec70 MajorFunction 0x2 ''  Flags 0  PreOperation 0xfffff802`17b0e140  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreClose+0  PostOperation (null)  Reserved1 (null)
[2] @ ffffba07`cc5eec90 MajorFunction 0x3 ''  Flags 0  PreOperation 0xfffff802`17ae3520  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreRead+0  PostOperation 0xfffff802`17ae3890  _FLT_POSTOP_CALLBACK_STATUS  wcifs!WcPostRead+0  Reserved1 (null)
[3] @ ffffba07`cc5eecb0 MajorFunction 0x4 ''  Flags 2  PreOperation 0xfffff802`17ae38e0  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreWrite+0  PostOperation (null)  Reserved1 (null)
[4] @ ffffba07`cc5eecd0 MajorFunction 0x5 ''  Flags 0  PreOperation 0xfffff802`17afd150  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreQueryInformation+0  PostOperation 0xfffff802`17afd2e0  _FLT_POSTOP_CALLBACK_STATUS  wcifs!WcPostQueryInformation+0  Reserved1 (null)
[5] @ ffffba07`cc5eecf0 MajorFunction 0x6 ''  Flags 0  PreOperation 0xfffff802`17afca40  _FLT_PREOP_CALLBACK_STATUS  wcifs!WcPreSetInformation+0  PostOperation 0xfffff802`17afcb60  _FLT_POSTOP_CALLBACK_STATUS  wcifs!WcPostSetInformation+0  Reserved1 (null)
```

FLT_INSTANCE

This struct represents the attachment of a filter to FLT_VOLUME. The maximum number of instances for a filter is the number of FLT_VOLUME instances (As you can see above, developers can control the number of instances registered. For example: the luafv filter has only one instance).

The instance list can be extracted by the commands "!fltkd.filters" or "!fltkd.volumes", that showed above. Each command shows filter instances from another perspective, "!fltkd.filters" will show all instances registered under each filter and "!fltkd.volumes" show all instances that belong to each volume.

Let's look at the FLT_INSTANCE structure:

```
0: kd> dt fltmgr!_FLT_INSTANCE 0xffffba07cc662010
   +0x000 Base               : _FLT_OBJECT
   +0x030 OperationRundownRef : 0xffffba07`cacc6ea0 _EX_RUNDOWN_REF_CACHE_AWARE
   +0x038 Volume             : 0xffffba07`c69f35a0 _FLT_VOLUME
   +0x040 Filter             : 0xffffba07`cc660010 _FLT_FILTER
   +0x048 Flags              : 20 ( INSFL_IN_STREAM_LIST_CTRL )
   +0x050 Altitude           : _UNICODE_STRING "135000"
   +0x060 Name               : _UNICODE_STRING "luafv"
   +0x070 FilterLink         : _LIST_ENTRY [ 0xffffba07`cc6600e0 - 0xffffba07`cc6600e0 ]
   +0x080 ContextLock        : _EX_PUSH_LOCK
   +0x088 Context            : 0xffffba07`cc661ba0 _CONTEXT_NODE
   +0x090 TransactionContexts : _CONTEXT_LIST_CTRL
   +0x098 TrackCompletionNodes : 0xffffba07`cc489920 _TRACK_COMPLETION_NODES
   +0x0a0 CallbackNodes      : [50] 0xffffba07`cc662b10 _CALLBACK_NODE
```

There are some interesting fields in this struct:

- Volume — The volume that the instance belongs to.
- Filter — The filter that registered the instance.
- Name — The name of the instance (Not to be confused with the name of the filter).
- CallbackNodes — An array of type CALLBACK_NODE. This array contains all the callbacks that belonging to the instance. We can present the array with the following command:

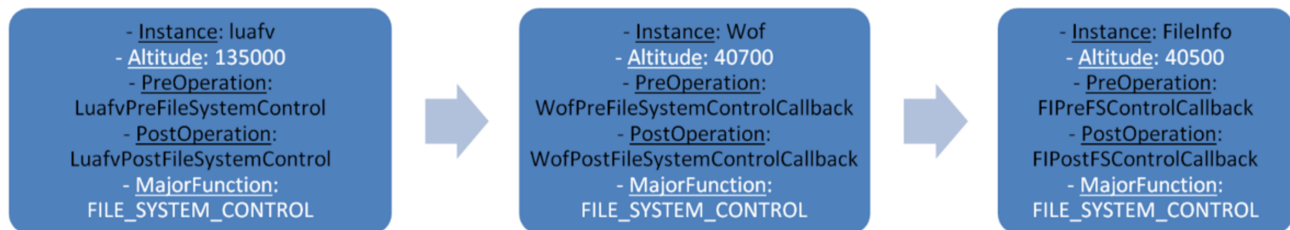
```
0: kd> !fltkd.instance 0xffffba07cc662010 4

FLT_INSTANCE: ffffba07cc662010 "luafv" "135000"
   CallbackNodes             : (ffffba07cc6620b0)
      NORMALIZE_NAME_COMPONENT (-22)
        CALLBACK_NODE: ffffba07cc662b10  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      GENERATE_FILE_NAME (-21)
        CALLBACK_NODE: ffffba07cc662ae0  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      MDL_WRITE_COMPLETE (-18)
        CALLBACK_NODE: ffffba07cc662240  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      PREPARE_MDL_WRITE (-17)
        CALLBACK_NODE: ffffba07cc662270  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      MDL_READ_COMPLETE (-16)
        CALLBACK_NODE: ffffba07cc6622a0  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      MDL_READ (-15)
        CALLBACK_NODE: ffffba07cc6622d0  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
      NETWORK_QUERY_OPEN (-14)
        CALLBACK_NODE: ffffba07cc662300  Inst:(ffffba07cc662010,"luafv","\Device\HarddiskVolume2") "luafv" "135000"
```

As you can see each index in the array represents another MajorFunction (Index = MajorFunction + 0x16h).

Let's look how the CALLBACK_NODE look like:

```
0: kd> dt fltmgr!_CALLBACK_NODE 0xffffba07cc662300
   +0x000 CallbackLinks     : _LIST_ENTRY [ 0xffffba07`c6898b00 - 0xffffba07`cdc04840 ]
   +0x010 Instance          : 0xffffba07`cc662010 _FLT_INSTANCE
   +0x018 PreOperation      : 0xfffff802`17ac36a0     _FLT_PREOP_CALLBACK_STATUS  luafv!LuafvPreNetworkQueryOpen+0
   +0x020 PostOperation     : (null)
   +0x018 GenerateFileName : 0xfffff802`17ac36a0     long  luafv!LuafvPreNetworkQueryOpen+0
   +0x018 NormalizeNameComponent : 0xfffff802`17ac36a0     long  luafv!LuafvPreNetworkQueryOpen+0
   +0x018 NormalizeNameComponentEx : 0xfffff802`17ac36a0     long  luafv!LuafvPreNetworkQueryOpen+0
   +0x020 NormalizeContextCleanup : (null)
   +0x028 Flags             : 0 (No matching name)
```

- PreOperation and PostOperation — Are the callbacks invoked before and after the operation.
- CallbackLinks —Is a dually linked list of CALLBACK_NODE structures that belong to FLT_INSTANCEs belonging to the same volume. All the CALLBACK_NODE structs have callbacks for the same MajorFunction. An example of such a list entry can be seen below:



Only the fields in black belong to CALLBACK_NODE.

With our new found knowledge of callbacks, we can now understand the "Callbacks" field of FLT_VOLUME. The field is of type CALLBACK_CTRL:

```
+0x120 Callbacks          : _CALLBACK_CTRL
   +0x000 OperationLists    : [50] _LIST_ENTRY [ 0xffffb28d`615c2130 - 0xffffb28d`615c2130 ]
   +0x320 OperationFlags    : [50] 0xb (No matching name)
```

The first field in this struct ("OperationList") is array of LIST_ENTRY, each cell in this array represent another major function (similar to the field CallbackNodes at FLT_INSTANCE). The lists in this array are the field CallbackLinks of CALLBACK_NODE that mentioned before.

Invoking Minifilter Callbacks

In order to implement our hook, we need to find where the callbacks are located in memory. To discover this we will explore how the callbacks are invoked. As mentioned before, there are two different callbacks — Pre and Post operation, so let's check them separately:

Pre Operation Callback

To find the mechanism that invokes the callbacks, we'll put a break point at the one of the callbacks in of luafv:

```
2: kd> bp luafv!LuafvPreNetworkQueryOpen
2: kd> g
Breakpoint 3 hit
luafv!LuafvPreNetworkQueryOpen:
fffff802`17ac36a0 b803000000      mov     eax,3
2: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff8402`7266f158 fffff802`1bd54a5d luafv!LuafvPreNetworkQueryOpen
01 ffff8402`7266f160 fffff802`1bd52f5c FLTMGR!FltpPerformPreCallbacks+0x2fd
02 ffff8402`7266f270 fffff802`1bd8c981 FLTMGR!FltpPassThroughFastIo+0x8c
03 ffff8402`7266f2d0 fffff802`1aa8a45f FLTMGR!FltpFastIoQueryOpen+0x131
04 ffff8402`7266f370 fffff802`1aa04daa nt!IopQueryInformation+0x8f
05 ffff8402`7266f3d0 fffff802`1aa0babf nt!IopParseDevice+0x8ea
06 ffff8402`7266f540 fffff802`1aa09f21 nt!ObpLookupObjectName+0x78f
07 ffff8402`7266f700 fffff802`1aa881c6 nt!ObOpenObjectByNameEx+0x201
08 ffff8402`7266f840 fffff802`1a5d3c15 nt!NtQueryAttributesFile+0x1e6
09 ffff8402`7266fb00 00007fff`02d3c814 nt!KiSystemServiceCopyEnd+0x25
0a 0000004f`aa2fe248 00007fff`02cc5e9d ntdll!NtQueryAttributesFile+0x14
0b 0000004f`aa2fe250 00007fff`02cc5981 ntdll!RtlDoesFileExists_UstrEx+0x9d
0c 0000004f`aa2fe340 00007ffe`ffc51966 ntdll!RtlDosSearchPath_Ustr+0x131
```

The call stack shows us the callback is called by a function named FltpPerformPreCallbacks. FltpPerformPreCallbacks takes a struct of type IRP_CALL_CTRL as an argument:

```
0: kd> dt fltmgr!_IRP_CALL_CTRL
   +0x000 Volume           : Ptr64 _FLT_VOLUME
   +0x008 Irp              : Ptr64 _IRP
   +0x010 IrpCtrl          : Ptr64 _IRP_CTRL
   +0x018 StartingCallbackNode : Ptr64 _CALLBACK_NODE
   +0x020 OperationStatusCallbackListHead : _SINGLE_LIST_ENTRY
   +0x028 Flags            : ICC_FLAGS
```

This struct is used in FltMgr to warp the pointers to callbacks that will be invoked and the information about the operation that will pass as parameters to the callbacks. This struct has some noteworthy fields:

- Volume — A Pointer to volume that contains all filter instances with a callback to be invoked.

- StartingCallbackNode — In order to avoid an infinite loop when a minifilter callback uses a filtered file system routine, FltMgr provides a set of functions that ensure a minifilter's callback will not be invoked for its own file syetem operations (For example — FltCreateFile performs the same operation as ZwCreateFile, but receives an additional FLT_INSTANCE argument. If the argument is not equal to NULL it ensures that only the instances that under the instance in the argument will have a callback invoked for the operation. If the argument is NULL it will invoke the callbacks of all the instances that under the volume which mentioned in the IRP_CALL_CTRL). These functions rely on the field StartingCallbackNode. They insert the next CALLBACK_NODE of the specific operation taken from the FLT_INSTANCE which is passed as an argument (instanceArgument->CallbackNodes[majorFunction + 0x16].CallbackLinks.Flink). The function FltpPerformPreCallbacks checks if the field is not null and if it isn't, it will invoke the callback in the field and all the other callbacks in CallbackLinks (As we can see upon further along).
- IrpCtrl — An internal FltMgr structure that warps the arguments of the callbacks, it is commonly used in FltMgr from the dispatch routines until the data is actually passed as arguments to the callbacks. Let's look at the fields of this struct:

```
1: kd> !fltkd.irpctrl 0xffffba07`cd6cf9a0

IRP_CTRL: ffffba07cd6cf9a0  QUERY_INFORMATION (5) [00000009] Irp SystemBuffer
Flags                    : [10000004] DontCopyParms FixedAlloc
Irp                      : ffffba07cd7da9a0
DeviceObject             : ffffba07c67d8d60 "\Device\HarddiskVolume2"
FileObject               : ffffba07ccf57130
CompletionNodeStack      : ffffba07cd6cfb28   Size=5  Next=0
SyncEvent                : (ffffba07cd6cf9b8)
InitiatingInstance       : 0000000000000000
Icc                      : ffff840271c1aea0
PendingCallbackNode      : 0000000000000000
PendingCallbackContext   : 0000000000000000
PendingStatus            : 0x00000000
CallbackData             : (ffffba07cd6cfa88)
 Flags                    : [00000009] Irp SystemBuffer
 Thread                   : ffffba07cdda0080
 Iopb                     : ffffba07cd6cfae0
 RequestorMode            : [00] KernelMode
 IoStatus.Status          : 0x00000000
 IoStatus.Information     : 0000000000000000
 TagData                  : 0000000000000000
 FilterContext[0]         : 0000000000000000
 FilterContext[1]         : 0000000000000000
 FilterContext[2]         : 0000000000000000
 FilterContext[3]         : 0000000000000000
```

The fields marked in green are passed to the pre operation callback as arguments. We will discuss CompletionNodeStack when we will explore the post operation callback.

Let’s look how the function FltpPerformPreCallbacks work:

```
startingCallbackNode = irpCallCtrl->StartingCallbackNode;
v76 = majorFunctionPlus16;
v83 = 0i64;
if ( startingCallbackNode == -1i64 )
  volume = irpCallCtrl->Volume;
else
  volume = startingCallbackNode->Instance->Volume;
v8 = qword_1C002A7E0;
v77 = volume;
KeEnterGuardedRegion(0i64);
v9 = (ExAcquireCacheAwarePushLockSharedEx)(v8, 0i64);
startingCallbackNodeB = irpCallCtrl->StartingCallbackNode;
v11 = v9;
if ( startingCallbackNodeB == -1i64 )
{
  callbackNode = volume->Callbacks.OperationLists[majorFunctionPlus16].Flink;
}
else
{
  v58 = irpCallCtrl->Flags;
  if ( v58 & 1 )
    callbackNode = startingCallbackNodeB->CallbackLinks.Flink;
  else
    callbackNode = irpCallCtrl->StartingCallbackNode;
  if ( !(v58 & 2) )
    ExReleaseRundownProtectionCacheAwareEx(startingCallbackNodeB->Instance->OperationRundownRef, 1i64);
}
```

The field we should probably rely upon to contain the callbacks of the minifilter is CALLBACK_NODE. As we can see the field CALLBACK_NODE was taken from StartingCallbackNode or from the FLT_VOLUME. After that we can see this loop:

```
  while ( 1 )
  {
    instance = callbackNode->Instance;
    if ( !(instance->Flags & 2) )
      break;
    callbackNode = (_CALLBACK_NODE *)callbackNode->CallbackLinks.Flink;
LABEL_57:
    if ( callbackNode == (_CALLBACK_NODE *)((char *)volume + v90) )
      goto LABEL_58;
    LOBYTE(v15) = v79;
  }
  if ( v14 && callbackNode->Flags & 1
    || !(_BYTE)v15 && callbackNode->Flags & 2 && (unsigned __int8)(irpCtrl->Data.Iopb->MajorFunction - 3) <= 1u )
  {
    callbackNode = (_CALLBACK_NODE *)callbackNode->CallbackLinks.Flink;
    goto LABEL_57;
  }
  v17 = irpCtrl->FileObject;
  if ( v17 && !(v17->Flags & 0x400000) && callbackNode->Flags & 8 )
  {
    callbackNode = (_CALLBACK_NODE *)callbackNode->CallbackLinks.Flink;
    goto LABEL_57;
  }
  if ( irpCallCtrl->Flags & 0x10 && !(instance->Filter->Flags & 4) )
  {
    callbackNode = (_CALLBACK_NODE *)callbackNode->CallbackLinks.Flink;
    goto LABEL_57;
  }
  if ( !(unsigned __int8)ExAcquireRundownProtectionCacheAwareEx(instance->OperationRundownRef, 2i64) )
  {
    callbackNode = (_CALLBACK_NODE *)callbackNode->CallbackLinks.Flink;
    goto LABEL_56;
  }
```

It iterates over the CallbackLinks, extracts each CALLBACK_NODE in turn, and invokes its pre operation callback:

```
(callbackNode->PreOperation)(&irpCtrl->Data, &fltRelatedObjects, &completionContext)
```

Post Operation Callback

Post operation callbacks are invoked by the function FltpPerformPostCallbacks which take an IRP_CTRL structure as argument.

The function uses the struct COMPLETION_NODE:

```
0: kd> dt fltmgr!_COMPLETION_NODE
   +0x000 IrpCtrl             : Ptr64 _IRP_CTRL
   +0x008 CallbackNode        : Ptr64 _CALLBACK_NODE
   +0x008 Filter              : Ptr64 _FLT_FILTER
   +0x010 InstanceLink        : _LIST_ENTRY
   +0x020 InstanceTrackingList : Ptr64 _COMPLETION_NODE_TRACKING_LIST
   +0x028 Context             : Ptr64 Void
   +0x030 DataSnapshot        : _FLT_IO_PARAMETER_BLOCK
   +0x078 Flags               : Uint2B
```

COMPLETION_NODE is used to save data from the pre operation callback for post operation callback (For example: The field “Context” in the struct taken from argument of pre operation callback which allows minifilter developers to save data about an operation for the post operation callback).

Before the function FltpPerformPreCallbacks invokes the callback, the function constructs the struct COMPLETION_NODE and in particular takes the CALLBACK_NODE that was extracted at the beginning of the loop and insert it into the field CallbackNode in the struct COMPLETION_NODE.

Subsequently COMPLETION_NODE inserted into CompletionStack (A field of IRP_CTRL) and increments the field NextCompletion (Which is used as an index to the head of the completion stack).

The function FltpPerformPostCallbacks iterates over the CompletionStack and while NextCompletion (The index to the stack) is not zero it extracts the COMPLETION_NODE and get the CALLBACK_NODE from it:

```
while ( 1 )
{
  nextCompletion = irpCtrl->NextCompletion;
  if ( !nextCompletion )
    break;
  completionNode = &irpCtrl->CompletionStack[(unsigned __int64)nextCompletion - 1];
  v22 = completionNode->InstanceTrackingList;
  if ( v22 )
  {
    KeAcquireInStackQueuedSpinLock(v22, &LockHandle);
    v23 = completionNode->Flags;
    if ( v23 & 0x10 )
    {
      KeReleaseInStackQueuedSpinLock(&LockHandle);
      FltpFreeCompletionNodeTracking(completionNode->InstanceTrackingList->TrackCtrl);
      if ( SLOBYTE(completionNode->Flags) < 0 )
      {
        FltpReinstateNameCachingAllFrames(*((PVOID *)irpCtrl->DeviceObject->DeviceExtension + 10), (__int64)irpCtrl);
        completionNode->Flags &= 0xFF7Fu;
      }
      goto LABEL_78;
    }
    if ( v23 & 0x20 )
    {
      IoAcquireCancelSpinLock(&Irql);
      IoReleaseCancelSpinLock((unsigned __int8)Irql);
      v23 = completionNode->Flags;
    }
    if ( v23 & 6 )
    {
      if ( WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_DWORD *)WPP_GLOBAL_Control + 11) & 0x400000 )
      {
        WPP_SF_qqd(
          *((_QWORD *)WPP_GLOBAL_Control + 3),
          18i64,
          &WPP_7840179beb6b390ca475d2dc9baf15fe_Traceguids,
          completionNode->IrpCtrl,
          completionNode,
          completionNode->DataSnapshot.MajorFunction);
        v23 = completionNode->Flags;
      }
      completionNode->Flags = v23 & 0xFFF9;
      v24 = completionNode->InstanceLink.Flink;
      if ( v24->Blink != &completionNode->InstanceLink
        || (v25 = completionNode->InstanceLink.Blink, v25->Flink != &completionNode->InstanceLink) )
      {
        __fastfail(3u);
      }
      v25->Flink = v24;
      v24->Blink = v25;
    }
    KeReleaseInStackQueuedSpinLock(&LockHandle);
  }
  callbackNode = completionNode->CallbackNode;
```

After that it invokes the post operation callback:

```
status = (unsigned int)callbackNode->PostOperation(
                         callbackData,
                         (const _FLT_RELATED_OBJECTS *)&fltRelatedObjects,
                         completionNode->Context,
                         0);
```

How we can extract the list of callbacks?

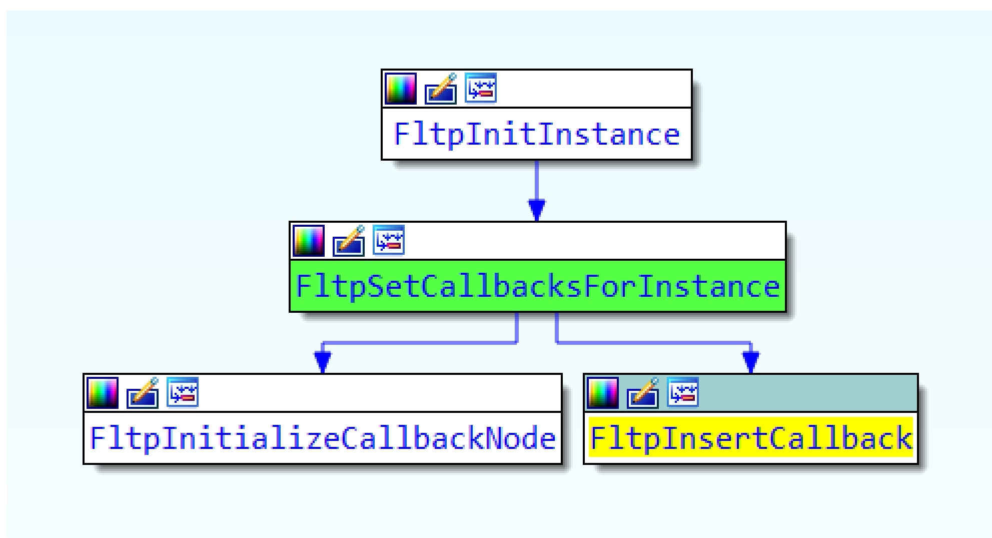
Now that we know that the callbacks stored in the CALLBACK_NODE structure and that FltpPerformPreCallbacks and FltpPerformPostCallbacks invoke the callbacks from it. We have seen that the source of the CALLBACK_NODE can be traced to either FLT_VOLUME or FLT_INSTANCE, so we need to focus on these structures to obtain a pointer to the callbacks.

FLT_VOLUME

We can enumerate all instances of FLT_VOLUME (with FltEnumerateVolumes) and for each of them extract the list of CALLBACK_NODE (volume->Callbacks->OperationLists[index]), and iterate over list to find the callbacks that belong to the minifilter which we wish to hook.

FLT_INSTANCE

If the pointer to CALLBACK_NODE in the field CallbackNodes is equal to the pointer to the CALLBACK_NODE in FLT_VOLUME, we can rely solely on FLT_INSTANCE and we will be able to enumerate all the instances that belong to the target filter and extract the callbacks in the field CallbacksNodes. Let's check this hypothesis. The function that responsible to initialize a new FLT_INSTANCE is FltpInitInstance.



The function uses FltpSetCallbacksForInstance to initialize a CALLBACK_NODE array. For each CALLBACK_NODE it calls the function FltpInitializeCallbackNode which takes the PreOperation and PostOperation from FLT_FILTER and inserts it into the CALLBACK_NODE passed as an argument. At the end of the function it uses the FltpInsertCallback, which shows us the connection between the CALLBACK_NODE at volume and the CALLBACK_NODE at instance:

```
_CALLBACK_NODE *__fastcall FltpInsertCallback(_FLT_INSTANCE *instance, _FLT_VOLUME *volume, unsigned int index)
{
  __int64 callbackIndex; // r9
  _LIST_ENTRY *instancesListEntryFromVolume; // r10
  _LIST_ENTRY *instancesListEntryFromInstance; // r8
  _CALLBACK_NODE *result; // rax
  _CALLBACK_NODE *callbackNode; // rcx
  _LIST_ENTRY *entryCallbackNode; // rdx

  callbackIndex = index;
  instancesListEntryFromVolume = &volume->InstanceList.rList;
  instancesListEntryFromInstance = instance->Base.PrimaryLink.Blink;
  result = instance->CallbackNodes[callbackIndex];
  while ( instancesListEntryFromInstance != instancesListEntryFromVolume )
  {
    callbackNode = (_CALLBACK_NODE *)*((_QWORD *)&instancesListEntryFromInstance[9].Flink + callbackIndex);
    if ( callbackNode )
    {
      entryCallbackNode = callbackNode->CallbackLinks.Flink;
      if ( callbackNode->CallbackLinks.Flink )
        goto LABEL_4;
    }
    instancesListEntryFromInstance = instancesListEntryFromInstance->Blink;
  }
  callbackNode = (_CALLBACK_NODE *)&volume->Callbacks.OperationLists[callbackIndex];
  entryCallbackNode = callbackNode->CallbackLinks.Flink;
LABEL_4:
  if ( (_CALLBACK_NODE *)entryCallbackNode->Blink != callbackNode )
    __fastfail(3u);
  result->CallbackLinks.Flink = entryCallbackNode;
  result->CallbackLinks.Blink = &callbackNode->CallbackLinks;
  entryCallbackNode->Blink = &result->CallbackLinks;
  callbackNode->CallbackLinks.Flink = &result->CallbackLinks;
  return result;
}
```

We can see that the our hypothesis is true. If CallbackLinks doesn't exist in the CALLBACK_NODE of a particular instance, The function will take the LIST_ENTRY from the volume and will insert it into the field CallbackLinks. Afterwards it updates the LIST_ENTRY in the volume structure.

How To Hook a FS Minifilter?

The POC code can be found in my GitHub repo (The POC was tested at windows 7 32 bit version 6.1 and windows 10 64 bit version 1903).

First step: Get a pointer to the target FLT_FILTER

The function FltGetFilterFromName is a documented function that takes a name of a filter as an argument and returns the opaque pointer to FLT_FILTER:

```
if (NT_SUCCESS(FltGetFilterFromName(name, &_object)))
{
    _isValid = true;
}
```

Second step: Enumerate an Array of FLT_INSTANCE under the target FLT_FILTER

I use the function FltEnumerateInstances to find the FLT_INSTANCE structures:

```
do
{
    if (fltInstancesNumber != 0)
    {
        fltInstancesArray.allocate(NonPagedPool, fltInstancesNumber);
    }

    status = FltEnumerateInstances(
        NULL,
        get(),
        fltInstancesArray.get(),
        sizeof(PFLT_INSTANCE) * fltInstancesNumber,
        &fltInstancesNumber);

} while ((status == STATUS_BUFFER_TOO_SMALL) && (fltInstancesNumber != 0));
```

Third step: Extract the CALLBACK_NODE array from FLT_INSTANCE

FLT_INSTANCE is an undocumented structure that changes from build to build. Because of that we need to find the offset to the field CallbackNodes in it, which is the array that contains the pointers to CALLBACK_NODE. There are a number of options to do that:

(1) Hardcoded offset — we can extract the offset from the appropriate symbols and set with it constant variables to use for each build. When initializing our driver, we can check the specific build that we run and choose the corresponding offset. This option is not good enough, because the hook will not be supported in future builds.

(2) FltpGetCallbackNodeForInstance — This is private function of FltMgr. The function takes a pointer to instance and an index to a specific CALLBACK_NODE in CallbackNodes as arguments. The function returns the pointer to CALLBACK_NODE by its index:

```
_CALLBACK_NODE *__fastcall FltpGetCallbackNodeForInstance(_FLT_INSTANCE *instance, unsigned int index)
{
  _CALLBACK_NODE *callbackNode; // rbx
  _CALLBACK_NODE *result; // rax

  callbackNode = instance->CallbackNodes[index];
  if ( callbackNode
    && !(instance->Flags & 6)
    && ExAcquireRundownProtectionCacheAwareEx(instance->OperationRundownRef, 1i64) )
  {
    result = callbackNode;
  }
  else
  {
    result = 0i64;
  }
  return result;
}
```

It seems like a wonderful function, we can enumerate all the CALLBACK_NODE at the instance and hook them. Unfortunately there are two problems with this option:

1. The function is not exported. To find the pointer to this function, we can search for hardcoded opcodes before the call to this function and get the pointer from it, we can search the hash of the function or we can do an heuristic search of the function (For example: By checking the number of calls and the number of add operations). All these checks are not good enough because we get the same problem of option one (new builds are bound to break our assumptions).
2. Besides returning the CALLBACK_NODE from the instance, the function FltpGetCallbackNodeForInstance also acquires rundown protection for the instance. The pointer for the rundown lock is located in the instance structure and is not exported (Like the field CallbackNodes). So failing release the rundown after we complete using the CALLBACK_NODE, the instance will be locked for rundown (something we would strongly wish to avoid).

(3) Register a callback — If we will have a pointer to a minifilter instance that exists in all builds and we can get a pointer to one of its callback, which will reliably exist on all systems. We could search for this pointer and get the offset of CallbackNodes from it. I didn't find a filter and a callback like this, so I decided to register a new filter with my own set of callbacks. I chose this option, dynamically found the offset of CallbackNodes in the instance structure by using the callback functions of my driver as sentinel values. After finding the offset we can unregister the sentinel minifilter.

Fourth step: Hook the callback at CALLBACK_NODE

After finding the array of CALLBACK_NODE we can simply hook the PreOperation and the PostOperation of the major functions that we want to hook. I saved the original pointer at global variable to return the CALLBACK_NODE back to its original state when I unload the POC driver:

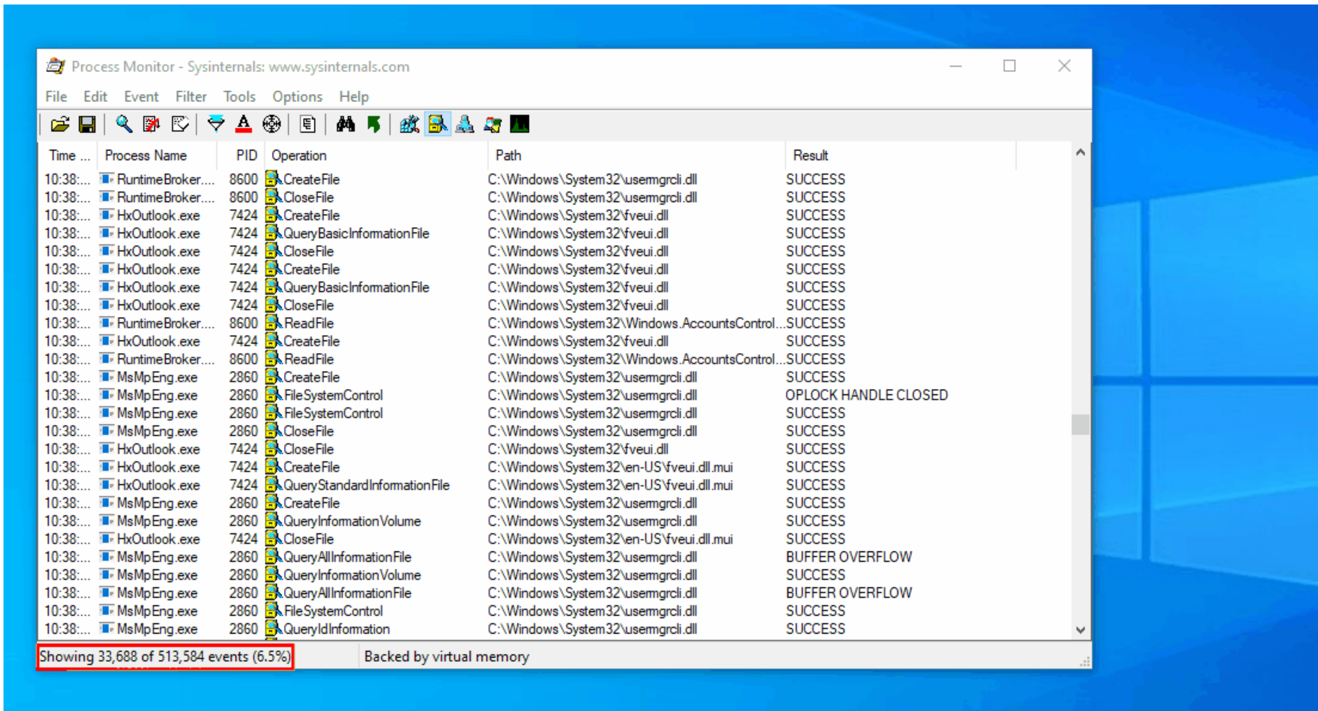
```
originalInstances = new ArrayGuard<FltInstanceGuard*, true>();
originalInstances->allocate(NonPagedPool, instancesCount);

for (size_t instanceIndex = 0; instanceIndex < instancesCount; instanceIndex++)
{
    (*originalInstances)[instanceIndex] =
        new FltInstanceGuard(instancesArray[instanceIndex]->get(), false, false, true);
    (*originalInstances)[instanceIndex]->cachingCallbackNodes();

    PCALLBACK_NODE* callbackNodesList =
        (*originalInstances)[instanceIndex]->getPointerToCallbackNodesField();

    for (size_t callbackIndex = 0X16;
         callbackIndex <= IRP_MJ_MAXIMUM_FUNCTION + 0X16;
         callbackIndex++)
    {
        PCALLBACK_NODE callbackNode = callbackNodesList[callbackIndex];

        if (MmIsAddressValid(callbackNode))
        {
            hookFunction(&(callbackNode->PreOperation), &hookPreOperationFunction);
            hookFunction(&(callbackNode->PostOperation), &hookPostOperationFunction);
        }
    }
}
```

The result

After performing the hook, I set a breakpoint at the original function of the filter, and we can see that our function now resides between FltPerformPreCallback and the original function:

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff8b8a`a5621608 fffff803`c2e6212e luafv!LuafvPreCleanup
01 ffff8b8a`a5621610 fffff806`53014a5d FsMinifilterHooking!hookPreOperationFunction+0x7e
02 ffff8b8a`a5621660 fffff806`530145a0 FLTMGR!FltpPerformPreCallbacks+0x2fd
03 ffff8b8a`a5621770 fffff806`53014112 FLTMGR!FltpPassThroughInternal+0x90
04 ffff8b8a`a56217a0 fffff806`53013efe FLTMGR!FltpPassThrough+0x162
05 ffff8b8a`a5621820 fffff806`4e90a939 FLTMGR!FltpDispatch+0x9e
06 ffff8b8a`a5621880 fffff806`4eead8c8 nt!IofCallDriver+0x59
07 ffff8b8a`a56218c0 fffff806`4eeb59f8 nt!IopCloseFile+0x188
08 ffff8b8a`a5621950 fffff806`4eebad3e nt!ObCloseHandleTableEntry+0x278
09 ffff8b8a`a5621a90 fffff806`4e9d3c15 nt!NtClose+0xde
0a ffff8b8a`a5621b00 00007ffa`6f13c2a4 nt!KiSystemServiceCopyEnd+0x25
0b 000000ff`f027daa8 00007ffa`6cd34a35 ntdll!NtClose+0x14
0c 000000ff`f027dab0 00007ffa`57d4a6ff KERNELBASE!CloseHandle+0x45
0d 000000ff`f027dae0 00007ffa`57d4f9b5 defragsvc!CVolume::MovePieceOfFile+0x12f
0e 000000ff`f027db70 00007ffa`57d22494 defragsvc!CFileOperation::MoveFileExtents+0x4a5
```

Another POC that I done is to block the callbacks to PROCMON minifilter. So its minifilter will not get file system events after my driver loaded.



Thanks to Philip Tsukerman and to Liron Zuartz for the helping with problems in the research.

References