Project Zero

News and updates from the Project Zero team at Google

Thursday, January 14, 2021

Hunting for Bugs in Windows Mini-Filter Drivers

Posted by James Forshaw, Project Zero

In December Microsoft fixed 4 issues in Windows in the Cloud Filter and Windows Overlay Filter (WOF) drivers (CVE-2020-17103, CVE-2020-17134, CVE-2020-17136, CVE-2020-17139). These 4 issues were 3 local privilege escalations and a security feature bypass, and they were all present in Windows file system filter drivers. I've found a number of issues in filter drivers previously, including 6 in the LUAFV driver which implements UAC file virtualization.

The purpose of a file system filter driver according to Microsoft is:
"A file system filter driver can filter I/O operations for one or more file systems or file system volumes. Depending on the nature of the driver, filter can mean log, observe, modify, or even prevent. Typical applications for file system filter drivers include antivirus utilities, encryption programs, and hierarchical storage management systems."

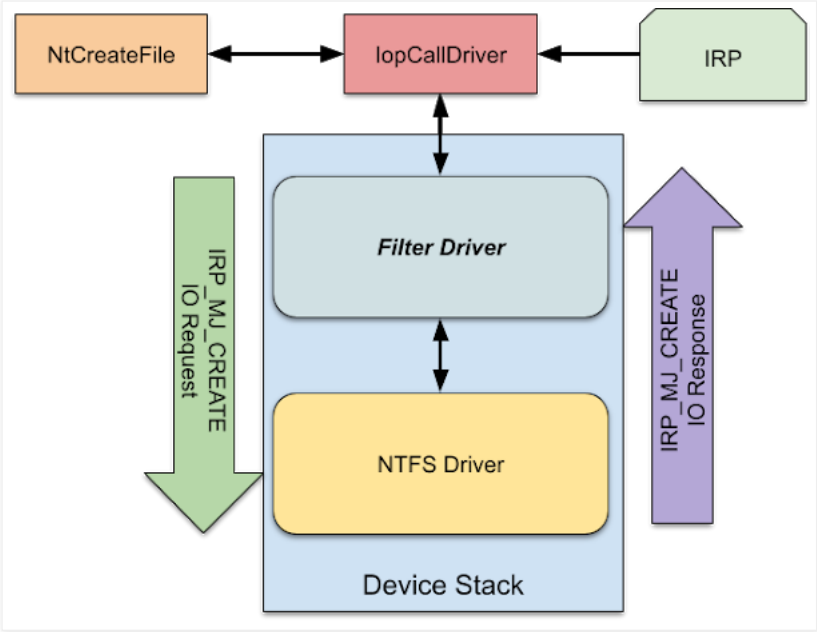
What this boils down to is the filter driver can inspect and modify almost any IO request sent to a file system. This power comes with many responsibilities, and considering the complexity of the IO model on Windows it can be hard to avoid introducing subtle bugs.

With the issues being fixed I thought would be a good opportunity to go into a bit more detail on how you can research file system filter drivers, specifically the kind of things I looked at to find my security vulnerabilities. I'm going to give an overview of how filter drivers work, how you communicate with them, some hints on reverse engineering and some of the common security issues you might discover. I'll also provide some basic example code to give you a basic idea of some common coding patterns. The goal is to allow you to do your own research in this area.

I'm assuming you have some prior knowledge on how the IO Manager works and have experience in finding security issues in non-filter drivers. Also I'm not claiming this to be an exhaustive description of bug hunting in filter drivers as the topic is very deep and complex. With this in mind let's start with an overview of how a filter driver works.

Filter Driver Implementation

A filter driver exploits the way the Windows IO Manager implements file system drivers. When you make a request to access a file, such as calling the *NtCreateFile* system call the IO Manager allocates an IO Request Packet (IRP) structure which contains the operation type and all the parameters for the operation. The IRP is then dispatched to the top of the device stack associated with the request.



A filter driver registers for the IO requests it supports with a callback function which is invoked when a specific IO request type IRP is queued in the device stack. The driver callback can then do a number of different things to the IRP.

- Pass the IRP unmodified directly to the next driver in the stack.
- Modify the IRP then pass to the next driver.
- Modify the IRP response.
- Complete the IRP operation with a success result.
- Complete the IRP operation with an error result.
- Pass the IRP to a different device stack.

This is the basics of how a filter driver works, the driver is attached at a suitable point of a device stack and handles IO requests. When an IRP of interest is received it can perform one of the operations to filter requests. If it wants to inspect or modify the response it can register for the completion routine and handle the operation in the callback.

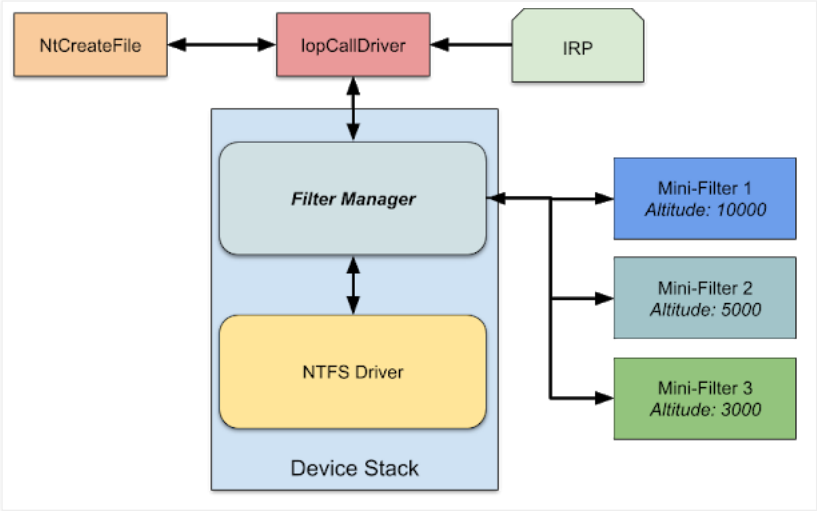
It's important to note that the IRP doesn't automatically propagate down the stack. A driver can choose to complete the IRP which means it'll not be processed by any other driver down the stack. If the driver passes on the IRP the driver must register a completion routine otherwise it'll not be notified when the IRP has been processed by the lower drivers in the stack.

For a file system filter the insertion point would typically be on top of the file system device object which is exposed by a file system driver such as NTFS. However, the driver can insert itself almost anywhere, allowing it to filter not just file system requests but also change data such as disk sectors. For example the Bitlocker Full Disk Encryption driver is a filter which is attached to the top of a volume block device. Any sectors passed in a write IRP are encrypted before passing to the lower driver. Read IRPs are handled in a completion routine and the sectors are decrypted before returning to the caller.

The Filter Manager and Mini-Filters

Implementing a filter driver from scratch is quite complicated. You have to handle every single IO request type, even if you don't care about it, so that it can be forwarded to the next driver in the stack. You also have to find the correct point to insert your filter driver into the device stack. It's easy to attach a driver to the top of the stack but trying to insert in the middle of an existing stack can be a recipe for disaster, for example the ordering of the filter drivers in the stack might differ depending on load order.

To make it easier to write a filter driver Windows comes with the Filter Manager Driver which takes care of handling IO requests and device stacks. This allows a developer to write what's called a mini-filter driver instead of a, now named, legacy filter driver. The following diagram shows how the architecture changes when you introduce the filter manager.



As you can see the mini-filters don't add their own device objects to the stack. Instead they are registered with the filter manager and it's the filter manager which inserts its own device. The filter manager handles the IO requests and calls registered mini-filters to process the request. If your mini-filter doesn't support a certain IO request then the filter manager implements a default which handles passing the IRP on to the next driver in the stack.

Another useful feature is the filter manager implements a mechanism for ordering the mini-filters, through an altitude value. The higher the altitude value the higher the priority. For example, a filter at altitude 10000 will be called before a filter at altitude 5000 when making a IO request. When handling responses the altitudes processed in reverse order, so the filter at 5000 will be called first then the one at 10000. Officially the altitude values must be registered with Microsoft. MSDN contains a list of the currently registered altitudes. However, there's nothing to stop a driver from registering itself with a different altitude except it'll likely draw the ire of Microsoft and might fail certification. By formalizing the altitude values you avoid the risk that a filter driver's ordering may change depending on load order.

Mini-Filter Registration

A mini-filter driver registers its presence by calling the FltRegisterFilter filter manager API, normally during the driver's entry point. The main parameter is a FLT_REGISTRATION structure which defines all the various callbacks for handling IO requests and bookkeeping. The important fields are the callbacks which a driver can register to respond to events from the filter manager. You can view what filters are registered with the filter manager using the *fltmc* command line tool (must be run as an administrator).

```
C:\> fltmc

Filter Name                     Num Instances    Altitude    Frame
------------------------------  -------------  ------------  -----
bindflt                             1       409800         0
WdFilter                           17       328010         0
storqosflt                          1       244000         0
wcifs                               0       189900         0
CldFlt                              0       180451         0
FileCrypt                           0       141100         0
luafv                               1       135000         0
npsvctrig                           1        46000         0
Wof                                14        40700         0
FileInfo                           17        40500         0
```

We can see all the mini-filters registered, the number of instances which indicates the number of volumes that's been attached and the altitude. There are 19 volumes available for filtering in the system I tested on (according to running *fltmc volumes*) so no filter is attached to everything. A driver can select and decide what volumes it wants to attach to by assigning an instance setup callback to the *InstanceSetupCallback* field in the filter registration structure. This callback is invoked for every volume on the system, including new ones added after the filter starts. The callback can return the status code *STATUS_FLT_DO_NOT_ATTACH* to block attachment.

You can view what volumes a filter is attached to using *fltmc* again:

```
C:\> fltmc instances -f luafv
```

```
Instances for luafv filter:

Volume Name     Altitude        Instance Name       Frame  VlStatus
------------- ------------  ----------------------  -----  --------
C:               135000     luafv                     0
```

This just shows the volume that LUAFV is attached to. As UAC virtualization only makes sense in the context of the system drive then it's only attached to *C:*. You can manually attach and detach filters on volumes using the *fltmc* tool with the *attach* and *detach* commands, we'll show an example of using these commands later.

NOTE: Just because a filter driver is attached to a volume it doesn't mean it'll filter any IO requests for that volume. For example, the WOF driver is attached to all NTFS volumes, however it'll only enable itself if there's at least one file in the volume which is registered to be handled by WOF. Otherwise it ignores the IO request, letting it complete normally.

Most mini-filters only attach to file system volumes. However, the filter manager also supports attaching to the named pipe and mailslot devices. The filter driver indicates support by setting the *FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS* flag in the *FLT_REGISTRATION* structure.

Mini-Filter IO Request Operation Callbacks

By far the most important field in the *FLT_REGISTRATION* structure is *OperationRegistration* which references a list of FLT_OPERATION_REGISTRATION structures defining the IO request callbacks. Each entry contains the IRP major code for the operation (such as *IRP_MJ_CREATE* or *IRP_MJ_FILE_SYSTEM_CONTROL*) and can have a pre-request and post-request callback. The driver doesn't need to specify both if it doesn't need both. The list is a variable length array, terminated with the major code being set to *IRP_MJ_OPERATION_END* (0x80). Any operation not in the list is handled by the filter manager which typically just ignores it and continues to the next filter in the list. A basic example of what you might see in C code is shown below.

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE,
      0,
      PreCreateOperation,
      PostCreateOperation },
    { IRP_MJ_OPERATION_END }
};
```

A pre-request callback accepts three parameters:

- The parameters for the operation, specified in a FLT_CALLBACK_DATA structure.
- Related kernel objects, in a FLT_RELATED_OBJECTS structure.
- An output pointer which can be assigned a callback context.

The prototype of the callback function pointer is:

```
typedef FLT_PREOP_CALLBACK_STATUS
(*PFLT_PRE_OPERATION_CALLBACK) (
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext
    );
```

The parameters for the IO request are accessible in the *FLT_CALLBACK_DATA* structure's *Iopb* field which is an FLT_IO_PARAMETER_BLOCK structure. The parameters are similar to the ones exposed through the IRP's current IO_STACK_LOCATION structure. The data parameter also contains the IO_STATUS_BLOCK for the request and the caller's requestor mode (either *KernelMode* or *UserMode*). The return code from the pre-request callback function determines what the filter driver wants to do with the request. The return type *FLT_PREOP_CALLBACK_STATUS* can be one of the following:

Name	Value	Description
FLT_PREOP_SUCCESS_WITH_CALLBACK	0	The callback was successful. Pass on the IO request and get a post-operation callback after completion.
FLT_PREOP_SUCCESS_NO_CALLBACK	1	The callback was successful. Pass on the IO request. No callback required.
FLT_PREOP_PENDING	2	Mark the IO operation as pending.
FLT_PREOP_DISALLOW_FASTIO	3	If handling a Fast IO operation, fail it to force the operation as a normal IO Request.
FLT_PREOP_COMPLETE	4	The operation has been completed. Do not pass on the IO request to any

		other drivers, even other filters in the stack.
FLT_PREOP_SYNCHRONIZE	5	Synchronize the post-operation callback in the same thread.
FLT_PREOP_DISALLOW_FSFILTER_IO	6	Disallow FastIO file creation.

A post-request callback accepts four parameters:

- The parameters for the operation, specified in a FLT_CALLBACK_DATA structure.
- Related kernel objects, in a FLT_RELATED_OBJECTS structure.
- A context pointer which could have been assigned by the pre-operation callback.
- Additional flags.

For post-operation callbacks the prototype is as follows:

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

The parameters are more or less the same as for the pre-operation callback. The *CompletionContext* parameter is the same one assigned in the pre-operation callback. If this value was allocated the post-operation callback needs to free the memory buffer to prevent leaking memory. The *FLT_POSTOP_CALLBACK_STATUS* return type can be one of the following values.

Name	Value	Description
FLT_POSTOP_FINISHED_PROCESSING	0	The callback was successful. No further processing required.
FLT_POSTOP_MORE_PROCESSING_REQUIRED	1	Halts completion of the IO request. The operation will be pending until the filter driver completes it.
FLT_POSTOP_DISALLOW_FSFILTER_IO	2	Disallow FastIO file creation.

Handling IO Requests

Now that we've described registration of the mini-filter and its callbacks let's go through a few examples of how IO requests are handled inside the pre and post operation callbacks. We'll use the six operations I mentioned earlier as a base for this discussion. Any examples are to demonstrate the likely code you'll find in a driver but omits security checks and other unimportant details. This isn't Stack Overflow, so please don't copy and paste them into real drivers.

Pass the IO request unmodified

The simplest way of not modifying an IO request is to not specify a pre-operation callback. Of course we're assuming the driver wants to handle an IO request selectively based on certain criteria so it must implement the callback.

The easiest way to ignore the IO request is to return the *FLT_PREOP_SUCCESS_NO_CALLBACK* status code from the pre-operation callback. That indicates to the filter manager that the mini-filter has completed its processing and is no longer interested in the IO request.

To give an example the following pre-create operation callback will ignore any open requests where the desired access does not request the *FILE_WRITE_DATA* access right. If the request doesn't contain the access then the request is completed with no callback.

```
FLT_PREOP_CALLBACK_STATUS
PreCreateOperation(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID* CompletionContext
) {
    PFLT_IO_PARAMETER_BLOCK ps = &Data->Iopb->Parameters;
    DWORD access = ps->Create.SecurityContext->DesiredAccess;
    if ((access & FILE_WRITE_DATA) == 0) {
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
    }

    // Perform some operation...
```

```
}
```

The example extracts the desired access from the creation parameters. If the FILE_WRITE_DATA access right is not set then the filter driver will ignore the IO request entirely by returning the no callback status code.

Of course depending on the purpose of the filter driver it might still want the post-operation callback to be called. For example if the filter driver is monitoring file access then the post-operation callback will contain valuable information such as the success or failure of opening the file or the data read from the file. In this case it makes sense to return *FLT_PREOP_SUCCESS_WITH_CALLBACK*.

When the driver specified it wants a post-operation callback it can configure the *CompletionContext* with any value it likes. This context can then be used in the post-operation callback. This can be used to pass additional data between the callbacks so that it can perform its operation correctly.

Modify the IO request

During a pre-operation callback the driver can modify the contents of the *FLT_CALLBACK_DATA* structure. For example the driver could change the security context used to open the file or it could even change the name of the file itself. The driver must indicate to the filter manager that the data has been modified by setting the *FLTFL_CALLBACK_DATA_DIRTY* flag in the *Flags* field before returning. The correct way of setting the flag is to call the FltSetCallbackDataDirty API however all that currently does is set the flag.

Modify the IO request response

As with the request you can modify the response in the post-operation callback which will return the changes to higher mini-filters and the IO manager. One trick I've commonly seen is to use this to change the target file by modifying the file name and returning the status code *STATUS_REPARSE* as if the file system hand encountered a symbolic link. The following is the basic approach that the *LUAFV* driver uses to perform the reparse operation to an arbitrary file path in a post-operation callback.

```
FLT_POSTOP_CALLBACK_STATUS LuafvReparse(PFLT_CALLBACK_DATA Data,
                                        PUNICODE_STRING TargetFileName){
  LuafvSetEcp(Data, TargetFileName);
  PFILE_OBJECT FileObject = Data->Iopb->TargetFileObject;
  ExFreePool(FileObject->FileName.Buffer);
  FileObject->FileName.Buffer = ExAllocatePool(PagedPool,
                                        TargetFileName.Length);
  FileObject->FileName.MaximumLength = TargetFileName.Length;
  RtlCopyUnicodeString(&FileObject->FileName, TargetFileName);
  Data->IoStatus.Information = 0;
  Data->IoStatus.Status = STATUS_REPARSE;
  FltSetCallbackDataDirty(Data);
  return FLT_POSTOP_FINISHED_PROCESSING;
}
```

The code deallocates the filename buffer in the target file object and replaces it with its own. It then sets the status code to *STATUS_REPARSE* and indicates that processing has finished. In Windows 7 a IoReplaceFileObjectName API was introduced which makes this operation much less error prone, however LUAFV was written for Vista where the API didn't exist so it had to make do. An official Microsoft example can be found in the SimRep sample driver.

One quirk of this operation is the *FileName* in the file object is volume relative, e.g. if you opened *c:\windows\notepad.exe* then *FileName* is set to *\windows\notepad.exe*. However, you can replace that with an absolute path such as *\??\d:\abc.txt* and that still works. Also the driver doesn't need to create a real mount point or symbolic link reparse point buffer for this to work. The IO manager will just take the path from the file object and restart the create request with the new path.

Complete the IO request with a success result

The driver can immediately complete an IO request by returning *FLT_PREOP_COMPLETE* from a pre-operation callback and updating the *IO_STATUS_BLOCK* in the *FLT_CALLBACK_DATA* parameter. The previous reparse example shows how that update works. If you're only updating the *IO_STATUS_BLOCK* you don't need to mark the data as dirty.

Higher level filter drivers will still get their post-operation callbacks invoked if they're registered for them, however no lower altitude drivers will be called with the IO request.

Complete the IO request with an error result.

This is basically the same as for a success code, just specifying a different NT status. There's nothing stopping a higher level filter driver from ignoring the error code and replacing it with a success.

Pass the IO request to a different file or device stack

The filter driver can redirect the operation to another device stack. For example you could implement a driver which redirects file reads and writes to a completely different file on the disk, making it look like the user is modifying the file when they're not.

The most obvious way of achieving this would be to open the new file during the pre-create operation then use that file object as the target for all subsequent operations. There are two potential issues with this approach.

First, how can a filter driver interact with a file system volume it's attached to without resulting in an infinite loop? For example, if the driver wants to open a file it can call IoCreateFile (and variants). However, the IO manager would dispatch the IO request to the top of the device stack, which would get back to the filter manager which could end up calling the filter driver again, ad infinitum. The same would be the case with any exported APIs from the kernel.

This issue is solved through two mechanisms. The first is the filter manager exposes a set of APIs which mirror the kernel IO APIs but will only dispatch the IO request to filters below the caller. For example you can call FltCreateFileEx or FltWriteFile and be sure you won't end up in a loop.

For file creation requests the driver can also employ a second mechanism called *Extra Create Parameters (ECP)*. An ECP is a GUID along with additional data which can be attached to the create request using the FltInsertExtraCreateParameter API. The filter driver can attach the ECP to the request, then check for its presence using FltFindExtraCreateParameter API, allowing it to ignore the request. For example the earlier code which shows how LUAFV implements a reparse operation shows calling *LuafvSetEcp* which sets an ECP on the request so that the new create request can be ignored by the driver.

The second issue is how do you actually pass on the parameters for the IO request to the new file you've opened? The naive approach would be to extract the parameters then invoke the corresponding filter manager API. For example, for a write IO request, read out the buffer and length then call FltWriteFile. This is error prone and might introduce subtle security issues.

A better approach is the driver can change the *TargetFileObject* field in the pre-operation callback's *FLT_IO_PARAMETER_BLOCK* structure then return a success code for the IO request to continue. This will cause the filter manager to send the original IO request to the new file object. The following is a simple example which could be in a pre-operation callback which will redirect the request to a file object extracted from the file system context:

```
PREDIRECT_CONTEXT context = // Get driver's allocated context.
if (context->FileObject) {
    Data->Iopb->TargetFileObject = context->FileObject;
    FltSetCallbackDataDirty(Data);
    return FLT_PREOP_SUCCESS_NO_CALLBACK;
}
```

Mini-Filter Communication

For there to be a security vulnerability the driver must process some untrustworthy data from a malicious user. What makes mini-filter drivers interesting is there's multiple places where untrusted data can be processed. Let's go through the ways of identifying and analyzing these communication channels.

Device Object

A mini-filter doesn't need to create any device object to perform its function, the filter manager deals with creating any necessary device objects. That doesn't mean the driver can't create one for its own purposes. A typical attack vector is the malicious user opens a handle to the device object and sends device IO control codes to exercise the vulnerable behavior.

I'm not going to go into details about how to analyze Windows kernel drivers for security issues in the IRP dispatch callbacks, as there's plenty of other resources. For example: Reverse Engineering and Bug Hunting on KMDF Drivers (video, slides).

Filter Communication Ports

One unique communication mechanism which is implemented by the filter manager is Filter Communication Ports. A port can be created by a mini-filter driver by calling the exported filter manager API FltCreateCommunicationPort.

```
PSECURITY_DESCRIPTOR SecurityDescriptor;

FltBuildDefaultSecurityDescriptor(
  &SecurityDescriptor,
```

```
  FLT_PORT_ALL_ACCESS
);

UNICODE_STRING Name;
RtlInitUnicodeString(&Name, L"\\FilterPortName");

OBJECT_ATTRIBUTES ObjAttr;
InitializeObjectAttributes(&ObjAttr, &Name, 0, NULL, SecurityDescriptor);

PFLT_PORT Port;
FltCreateCommunicationPort(
  Filter,
  &Port,
  &ObjAttr,
  NULL,
  ConnectNotifyCallback,
  DisconnectNotifyCallback,
  MessageNotifyCallback,
  100
);
```

The name of the port is specified using an OBJECT_ATTRIBUTES structure, in this example the filter port will be called *\FilterPortName* in the Object Manager Namespace (OMNS). The driver should also specify the security descriptor to be associated with the port through the OBJECT_ATTRIBUTES. It's most common to call the FltBuildDefaultSecurityDescriptor API to build a security descriptor which only grants administrators access to the port. However, the driver can configure the security any way it likes.

In *FltCreateCommunicationPort* the filter manager creates a new named kernel object of type *FilterConnectionPort* with the *OBJECT_ATTRIBUTES* and associates it with the callbacks. There's no *NtOpenFilterConnectionPort* system call to open a port. Instead when a user wants to access the port it must first open a handle to the filter manager message device object, *\FileSystem\Filters\FltMgrMsg,* passing an extended attributes structure identifying the full OMNS path to the port.

It is much easier to open a port by calling the FilterConnectCommunicationPort API in user-mode, so you don't need to deal with connecting manually. When opening a port you can also specify an arbitrary context buffer to pass to the connect callback. This can be used to configure the open port instance. On connection the connect notification callback passed to *FltCreateCommunicationPort* will be called. The prototype for the callback is as follows:

```
typedef NTSTATUS
(*PFLT_CONNECT_NOTIFY) (
      PFLT_PORT ClientPort,
      PVOID ServerPortCookie,
      PVOID ConnectionContext,
      ULONG SizeOfContext,
      PVOID *ConnectionPortCookie
      );
```

The *ConnectionContext* and *SizeOfContext* are values passed from user-mode when calling *FilterConnectCommunicationPort*. The *ConnectionContext* has its length verified and copied into kernel memory before use. However, there's no structure for the context so the driver must still carefully verify its contents before using it. The driver can reject a caller by returning an error NT status code. This allows the driver to do things like verify the caller is in a signed binary or similar, which is likely something security products will do.

If the connection is allowed the *ConnectionPortCookie* pointer can be updated with a pointer to an allocated structure unique to the client. This pointer will be passed back to the driver in the message and disconnect notification callbacks.

You can enumerate what ports are currently registered by inspecting the OMNS. For example, to enumerate the ports in the root of the OMNS using my NtObjectManager PowerShell module run the following command:

```
PS> ls NtObject:\ | Where-Object TypeName -eq "FilterConnectionPort"
Name                                    TypeName
----                                    --------
storqosfltport                          FilterConnectionPort
MicrosoftMalwareProtectionRemoteIoPortWD  FilterConnectionPort
MicrosoftMalwareProtectionVeryLowIoPortWD FilterConnectionPort
WcifsPort                               FilterConnectionPort
MicrosoftMalwareProtectionControlPortWD   FilterConnectionPort
```

```
BindFltPort                                 FilterConnectionPort
MicrosoftMalwareProtectionAsyncPortWD       FilterConnectionPort
CLDMSGPORT                                  FilterConnectionPort
MicrosoftMalwareProtectionPortWD            FilterConnectionPort
```

You might notice there is also a FilterCommunicationPort kernel object type. This is the object used for the client-end where FilterConnectionPort is the mini-filter server end. You should never see a FilterCommunicationPort named object in the OMNS.

When the port is opened the kernel will check the security descriptor for access. Unfortunately there's no way to directly query the assigned security descriptor for a port from user-mode. The simplest way to test is to just try and open the port and see if it returns an access denied error.

```
PS> $ports = ls NtObject:\ |
Where-Object TypeName -eq "FilterConnectionPort"
PS> foreach($port in $ports.Name) {
    Write-Host "\$port"
    Use-NtObject($p = Get-FilterConnectionPort "\$port") {}
}
\BindFltPort
Exception: "(0x80070005) - Access is denied."

\CLDMSGPORT
Exception: "(0x8007017C) - The cloud operation is invalid."
```

We can see two ports output in the previous code snippet. The *BindFltPort* port fails with an access denied error, while the *CLDMSGPORT* port (which is part of the Cloud Filter driver) returns "The cloud operation is invalid.". The second error indicates that we've likely opened the port, but you'll need to supply specific parameters in the context buffer when calling the *FilterConnectCommunicationPort* API. You can specify the connection context for the *Get-FilterConnectionPort* command by specifying a byte array to the *Context* parameter.

```
PS> $port = Get-FilterConnectionPort -Path "\PORT" -Context @(0, 1, 2, 3)
```

We can inspect the security descriptor for a port if you've got a Windows system with a kernel debugger enabled and a copy of WinDBG.

```
0: kd> !object \CLDMSGPORT
Object: ffffb487447ff8c0  Type: (ffffb4873d67dc40) FilterConnectionPort
    ObjectHeader: ffffb487447ff890 (new version)
    HandleCount: 1  PointerCount: 4
    Directory Object: ffff8a8889a2d4e0  Name: CLDMSGPORT

0: kd> dx (((nt!_OBJECT_HEADER*)0xffffb487447ff890)->SecurityDescriptor
& ~0x7)
(((nt!_OBJECT_HEADER*)0xffffb487447ff890)->SecurityDescriptor & ~0x7) :
0xffff8a888dccb0a0
0: kd> !sd 0xffff8a888dccb0a0 1
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x9004
            SE_DACL_PRESENT
            SE_DACL_PROTECTED
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544 (Alias: BUILTIN\Administrators)
->Group   : S-1-5-18 (Well Known Group: NT AUTHORITY\SYSTEM)
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x1c
->Dacl    : ->AceCount   : 0x1
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x001f0001
->Dacl    : ->Ace[0]: ->SID: S-1-5-11 (Well Known Group: NT
AUTHORITY\Authenticated Users)

->Sacl    :  is NULL
```

To dump the SD you first query for the object address of the filter communication port using the *!object* command. From the output you take the address of the *OBJECT_HEADER* structure and query the *SecurityDescriptor* field. Note you must clear the lower 3 bits of the address to make a valid security descriptor pointer. Finally we can print the security descriptor using the *!sd* command. The output shows that the security descriptor grants the *Authenticated Users* group access to connect to the port.

With an open handle to the port you can now send and receive messages. The filter manager supports both user to kernel and kernel to user message directions. For the user to kernel messages you call the FilterSendMessage API which sends a raw memory buffer to the filter driver and returns a separate buffer as shown in the following prototype:

```
HRESULT FilterSendMessage(
  HANDLE  hPort,
  LPVOID  lpInBuffer,
  DWORD   dwInBufferSize,
  LPVOID  lpOutBuffer,
  DWORD   dwOutBufferSize,
  LPDWORD lpBytesReturned
);
```

The message is delivered to the filter driver's message notification callback specified when registering the mini-filter. The callback has the following prototype.

```
typedef NTSTATUS
(*PFLT_MESSAGE_NOTIFY) (
      IN PVOID PortCookie,
      IN PVOID InputBuffer OPTIONAL,
      IN ULONG InputBufferLength,
      OUT PVOID OutputBuffer OPTIONAL,
      IN ULONG OutputBufferLength,
      OUT PULONG ReturnOutputBufferLength
      );
```

The handling of the message is similar to a device IO control call. In fact under the hood it's implemented using the device IO control code 0x8801B. As this code uses the METHOD_NEITHER method means the *InputBuffer* and *OutputBuffer* parameters are pointers into user-mode memory. The filter manager does check them before calling the callback with *ProbeForRead* and *ProbeForWrite* calls.

You can send a message to a filter connection port in PowerShell using the *Send-FilterConnectionPort* command specifying the data to send and the maximum size of the output buffer.

```
PS> Send-FilterConnectionPort -Port $port -Input @(0, 1, 2, 3) -
MaximumOutput 0x100
```

For the kernel to user messages the user mode application needs to call FilterGetMessage to wait for the filter driver to send a message to user-mode. The kernel sends a message to the waiting user mode application using the FltSendMessage API which has the following prototype.

```
NTSTATUS FltSendMessage(
  PFLT_FILTER    Filter,
  PFLT_PORT      *ClientPort,
  PVOID          SenderBuffer,
  ULONG          SenderBufferLength,
  PVOID          ReplyBuffer,
  PULONG         ReplyLength,
  PLARGE_INTEGER Timeout
);
```

If there's currently no waiting user mode process the API can wait a specified timeout until the application called *FilterGetMessage*. The returned buffer from *FilterGetMessage* contains a FILTER_MESSAGE_HEADER structure followed by the data. The header contains the size of the reply requested as well as a message ID which is used to correlate any reply to the kernel's message.

To reply the user-mode application calls the FilterReplyMessage API. The user-mode application needs to append any data to a FILTER_REPLY_HEADER structure which contains the NT status code of the operation and the correlated message ID. The *FltSendMessage* API waits for the user-mode application to call *FilterReplyMessage* with the correct ID, and returns a buffer to the kernel-mode code. The message notification callback is not involved when using kernel to user-mode calls.

Filter Callbacks

Typically the purpose of the mini-filter callbacks would be to inspect or modify pre-existing IO requests to a file system. Therefore one way of getting untrusted data to the driver is based on how it handles IO requests. However, it's possible to add additional functionality on top of an existing file system to allow for communication between user mode and kernel mode. The filter driver can add a callback for device or file system IO control code requests and check and handle its own control codes. This allows the filter to implement additional functionality on existing files.

The following is a simple example of adding a *FSCTL_REVERSE_BYTES* FS IO control code to an existing file system. This FSCTL is not really supported by any filesystem.

```
#define FSCTL_REVERSE_BYTES CTL_CODE(FILE_DEVICE_FILESYSTEM,
```

```
                                    0x801,
                                    METHOD_BUFFERED,
                                    FILE_ANY_ACCESS)

FLT_PREOP_CALLBACK_STATUS
PreFsControlOperation(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID* CompletionContext
) {
    PFLT_PARAMETERS ps = &Data->Iopb->Parameters;
    if (ps->DeviceIoControl.Common.IoControlCode != FSCTL_REVERSE_BYTES) {
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
    }

    char* buffer = ps->DeviceIoControl.Buffered.SystemBuffer;
    ULONG length = min(ps->DeviceIoControl.Buffered.InputBufferLength,
        ps->DeviceIoControl.Buffered.OutputBufferLength);
    for (ULONG i = 0; i < length; ++i)
    {
        char tmp = buffer[i];
        buffer[i] = buffer[length - i - 1];
        buffer[length - i - 1] = tmp;
    }
    Data->IoStatus.Status = STATUS_SUCCESS;
    Data->IoStatus.Information = length;
    return FLT_PREOP_COMPLETE;
}
```

The parameters for the FSCTL or IOCTL are separated based on the method of buffer access. In this case the FSCTL uses *METHOD_BUFFERED* so the parameters are accessed through the Buffered field. The filter driver needs to ensure it handles correctly all buffer types if it wants to implement its own control codes.

This technique is used by the Windows Overlay Filter (WOF). For example, the FSCTL code FSCTL_SET_EXTERNAL_BACKING is not supported by NTFS. Instead it's intercepted by a pre-operation callback in the WOF filter which completes it before it reaches the NTFS driver. The NTFS driver never sees the control code, unless the WOF driver happens to not be enabled.

Reparse Points

Reparse point buffers are most commonly known for implementing symbolic link support for NTFS. However the reparse point feature of NTFS can store arbitrary tagged data which is used by filter drivers to store additional offline state information for a file. For example, WOF uses its own reparse buffer, with the tag IO_REPARSE_TAG_WOF to store the location of the real file or status of a compressed file.

A user-mode application would set, query and delete using FSCTL control codes, such as FSCTL_SET_REPARSE_POINT. The recommended way a mini-filter driver should set and delete a file's reparse buffer is through the FltTagFile (and FltTagFileEx) and FltUntagFile APIs to set and remove the reparse buffer. Searching for the driver's imported APIs should quickly show whether the driver uses its own reparse buffer format.

To open a file with the supported reparse point buffer the driver could register for the post-create callback and wait for any request which returns the *STATUS_REPARSE* NT status then query for the reparse point data from the *TagData* field in the *FLT_CALLBACK_DATA* parameter. If the reparse tag matches one the filter driver supports it can re-issue the create request but specify the *FILE_OPEN_REPARSE_POINT* flag to open the file and ignore the reparse point. There are many problems with this, not least it requires two IO requests for a single creation and the driver would have to process every reparse event.

To simplify this Windows 10 supports the *ECP_TYPE_OPEN_REPARSE_GUID* ECP. You add the ECP with a buffer containing an OPEN_REPARSE_LIST_ENTRY structure which defines the reparse tag the driver handles. When NTFS encounters a reparse point buffer it checks to see if it's in the open reparse list. If so instead of returning STATUS_REPARSE the *OPEN_REPARSE_POINT_TAG_ENCOUNTERED* flag is set in the *OPEN_REPARSE_LIST_ENTRY* structure, the file is opened and success NT status code is returned. The filter driver can then check for the flag in the post-create callback, if set it can query the reparse tag from the file, for example using FSCTL_GET_REPARSE_POINT and handle accordingly.

The filter manager also exposes the FltAddOpenReparseEntry and FltRemoveOpenReparseEntry to simplify adding and removing these open reparse list entries. Searching for use of these APIs should give you an idea if the filter driver implements its own reparse point format.

The reason I mention this in the context of communication is that a filter driver will process these reparse buffers when accessing the file system. The NTFS driver only checks for the *SeCreateSymbolicLinkPrivilege*

privilege if a user is writing the *IO_REPARSE_TAG_SYMLINK* tag. NTFS delegates the verification of the REPARSE_DATA_BUFFER structure which will be written to the file system by calling the kernel API FsRtlValidateReparsePointBuffer. The kernel API only does basic length checks for non-symlink tag types so the arbitrary bytes set in the *DataBuffer* field can be completely untrusted, which can allow for security issues during parsing.

Security Bug Classes

I've now provided examples of how a mini-filter operates and how you can communicate with it. Let's finish up with an overview of potential bug classes to look for when doing a review. Some of these bug classes are common to any kernel driver, but others are very specifically due to the way mini-filters operate.

Where possible I'll also provide an example of a vulnerability I've discovered to improve understanding. Note, this is not an exhaustive list, I'm sure there are some novel bug classes that I don't know about which are missing from this list. Which is why it's good to describe this process in more detail so others can take advantage of my knowledge and find new and interesting issues.

To aid in analysis I've uploaded my header file I use in IDA Pro to populate the filter manager types. You can get it from github. I've tried to ensure it's correct and up to date, but there's a chance that it is not. YMMV.

Common and garden variety memory safety hazards

Being native C code you can expect the same sorts of issues you'd find in any sizable code base including integer wrapping and incorrect reference counting leading to memory safety hazards. Any of the described communication methods could result in untrusted data being processed and mishandled. I don't think I need to describe this in any detail.

Ignoring the RequestorMode Value

All filtered IO requests have an assigned *RequestorMode* parameter in the *FLT_CALLBACK_DATA* structure which indicates whether it originated from user or kernel mode code. If an IO request is dispatched from kernel mode code the IO manager and file system drivers typically disable security checks, such as file access checking.

There are a couple of related bug classes you'll see with regards to *RequestorMode*. The first class is the filter driver ignoring its value. This can be a problem if the filter driver redirects the IO request to another file either directly or by using a reparse operation during file creation.

For example, CVE-2018-0877 was an issue I found in the WCIFS driver which provides file system virtualization for Desktop Bridge applications. The root cause was the driver would reparse to a user controllable location if the requested file didn't exist in privileged Windows directories.

It's common to find kernel code opening files inside privileged directories with *RequestorMode* set to the kernel. The kernel code can make the assumption this can't be tampered with as only an administrator can normally modify those directories. The end result was a normal user application could get a file opened in the user controllable location but with access checking disabled. In the proof-of-concept in the issue tracker I exploit this to redirect a request for a *National Language Support (NLS)* file to ready arbitrary files on disk such as the SAM hive. The technique was described separately in this blog post.

Incorrect RequestorMode Check.

The second bug class in checking the *RequestorMode* can occur during a file create operation. Specifically the *RequestorMode* field is checked but the driver does not verify if access checking has been re-enabled through the *IO_FORCE_ACCESS_CHECK* flag passed to *IoCreateFile* and variants. For a bit more context on this bug class refer to my blog post from last year where I collaborated with Microsoft on related issues.

```
FLT_PREOP_CALLBACK_STATUS
PreCreateOperation(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID* CompletionContext
) {
    if (!SeSinglePrivilegeCheck(SeExports->SeTcbPrivilege,
                                Data->RequestorMode)) {
        Data->IoStatus.Status = STATUS_ACCESS_DENIED;
        return FLT_PREOP_COMPLETE;
    }

    // Perform some privileged action.

    return FLT_PREOP_SUCCESS_WITH_CALLBACK;
}
```

The example above shows misuse of the *RequestorMode* field. It passes it directly to SeSinglePrivilegeCheck, if it indicates the call came from the kernel then the privilege check will always return TRUE meaning the privileged action will be taken. If you read the linked blog post, this can happen if the file is opened through calling *IoCreateFileEx* or similar APIs with incorrect flags.

To guard against this issue the driver needs to check if the *SL_FORCE_ACCESS_CHECK* flag has been set in the *OperationFlags* field of the *FLT_IO_PARAMETER_BLOCK* structure. If that flag is set the value of *RequestorMode* should always be assumed to be from user mode.

Driver and Kernel IO Operation Mismatch

The Windows platform is constantly iterating new features, this is even more true since the release of Windows 10 and its six month release cycles. This can introduce new features to the IO stack such as new information classes or IO control codes or additional functionality to existing features.

For the most part the mini-filter driver can just ignore operations it doesn't care about. However, if it does process an IO operation it needs to match with what's implemented in the rest of the OS, which can be difficult if the OS changes around the driver.

An example of this issue is the WOF driver's handling of reparse points. To prevent applications from setting arbitrary reparse points with the *IO_REPARSE_TAG_WOF* tag it handles the *FSCTL_SET_REPARSE_POINT* IO control code and rejects any attempt to set a reparse point buffer with that tag. To complete the trick the driver also hides a file's reparse point from being queried or removed if it's set to *IO_REPARSE_TAG_WOF*.

The issue CVE-2020-17139 resulted from the OS adding a new FSCTL_SET_REPARSE_POINT_EX IO control code which the WOF driver didn't handle. This allowed an application to add or remove the WOF IO tag which resulted in a way of getting an arbitrary file to have a cached code signature to bypass mechanisms such as Windows Defender Application Control.

Altitude sickness.

Sorry, I couldn't resist the pun. This is a bug class which is caused by the ordering of filter operations based on the assigned altitudes of the driver. For example, if you look at the list of filters from the *fltmc* command shown earlier in this blog post you'll notice that *WdFilter* which is the real-time scanner for Windows Defender is at a much higher altitude than *LUAFV* which is the UAC file virtualization driver.

What this means is if *LUAFV* performs some operations, such as calling *FltCreateFileEx* which only dispatches the IO request to filters below *LUAFV* then Windows Defender will miss the file operations and not be able to act on them. Let's show this in action with a simple PowerShell script.

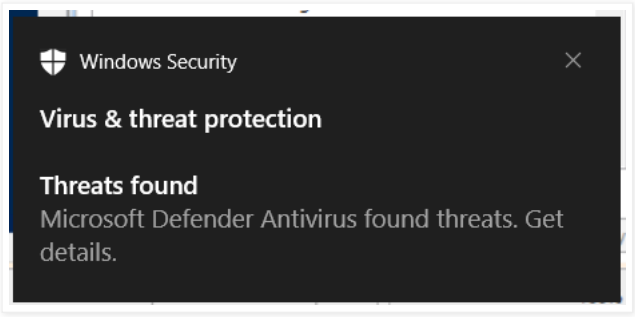
```
function Write-EICAR {
    param([string]$Path)

    # Replace with a real EICAR string.
    $eicar = [System.Text.Encoding]::ASCII.GetBytes("<EICAR>")
    Use-NtObject($f = New-NtFile -Win32Path $Path -Disposition OpenIf -
Access ReadData, WriteData) {
        $f.Length = 0
        Write-NtFile $f $eicar -Offset 0
    }
}

PS> Write-EICAR -Path "$env:TEMP\eicar.txt"
PS> Enable-NtTokenVirtualization
PS> Write-EICAR -Path "$env:windir\system32\license.rtf"
```

The *Write-EICAR* function opens or creates a new file at a specified path, truncates the file to a zero length, writes the EICAR string then closes the file. Note I've replaced the EICAR string with the dummy *<EICAR>*. You'll need to look up the real string online and replace it before running the test. I did this to prevent some overzealous AV detecting the EICAR string and quarantining this web page.

We create an EICAR file in the temporary folder. Once the file has been closed Windows Defender's real-time scanner should scan it and warn the user that it has quarantined the file.



However, once we enable virtualization using *Enable-NtTokenVirtualization* and write to an existing system file the file processing is handled inside the LUAFV driver after *WdFilter* has done its checking. Therefore the second command will succeed, although the file which is actually created is in the user's virtual store, we've not overwritten *license.rtf*.

Worth pointing out that this only allows you to create the file on disk. The instant that virtualized file is used by any application Windows Defender will see it and quarantine it. Therefore it provides no real value to bypass Windows Defender's signature checks. However, I think this is an interesting demonstration of the types of issues you could find due to the differing altitudes.

The mismatch with the filter altitude is also a potential reason you'll miss file events in Process Monitor. Process Monitor runs its mini-filter to capture file events at altitude 385200 which is above *LUAFV*. You will not see most direct virtualization events. However we can do something about this, we can use *fltmc* to detach the Process Monitor filter from a volume and reattach at a much lower altitude. Start Process Monitor then run the following commands to reattach to the C: drive.

```
C:\> fltmc detach PROCMON24 C:
C:\> fltmc attach PROCMON24 C: -i "Process Monitor 24 Instance" -a 100
```

You might need to replace *24* with an appropriate version number for your version of Process Monitor. You should start seeing more events which were previously hidden by *LUAFV* and other filter drivers at lower altitudes. This should help you monitor file access for any interesting behavior. Sadly even though you can try and attach the Process Monitor filter to the named pipe device it won't work as the driver doesn't indicate support for that device.

Note, that stopping and starting the Process Monitor capture will reset the volume instances for the filter driver and remove the low altitude instance. If you create the new instance without the instance name (the string after -i) then it won't get deleted, however Process Monitor will show duplicate entries for any IO request which is the same at both altitudes. The Process Monitor driver does not support attaching at a different altitude through any command line options, this would be one of those cases where it'd be useful for this tooling to be open source so that this feature could be added.

As an example before adding the low altitude instance if you create the EICAR test file you'll see the following events:

ID	Path	Operation	Result	Detail
0	C:\Windows\System32\license.rtf	CreateFile	SUCCESS	Desired Access: Read Data, Write Data
1	C:\Windows\System32\license.rtf	SetEndOfFile	SUCCESS	EndOfFile: 0
2	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	WriteFile	SUCCESS	Offset: 0, Length: 68
3	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	CloseFile	SUCCESS	

I've added an ID column which indicates the event taking place. The events match the code for creating the EICAR file, we open the file for read and write access, set the length to 0, write the EICAR string and then close the file. Note that in event ID 2 the path to the file has changed from the original one in system32 to the virtual store. This is because the file is "delay virtualized" so it'll only be created if a write IO request, such as changing the file length, is dispatched to the file.

Now let's compare the events when the altitude is set to 100:

ID	Path	Operation	Result	Detail
0	C:\Windows\System32\license.rtf	CreateFile	ACCESS DENIED	Desired Access: Read Data, Write Data
	C:\Windows\System32\license.rtf	CreateFile	SUCCESS	Desired Access: Read Data
1	C:\Windows\System32\license.rtf	CreateFile	SUCCESS	Desired Access: Read Data,

				Read Attributes
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	CreateFile	SUCCESS	Desired Access: Write Data, Write Attributes
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	SetEndOfFile	SUCCESS	EndOfFile: 538
	C:\Windows\System32\license.rtf	ReadFile	SUCCESS	Offset: 0, Length: 538
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	WriteFile	SUCCESS	Offset: 0, Length: 538
	C:\Windows\System32\license.rtf	ReadFile	END OF FILE	Offset: 538, Length: 16,384
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	CloseFile	SUCCESS	
	C:\Windows\System32\license.rtf	CloseFile	SUCCESS	
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	CreateFile	SUCCESS	Desired Access: Read Data, Write Data
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	SetEndOfFile	SUCCESS	EndOfFile: 0
2	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	WriteFile	SUCCESS	Offset: 0, Length: 68, Priority: Normal
3	C:\Windows\System32\license.rtf	CloseFile	SUCCESS	
	C:\Users\admin\AppData\Local\VirtualStore\Windows\System32\license.rtf	CloseFile	SUCCESS	

You can see that the list of events is much longer in the second case (I've even removed some for brevity). For event 0 it's no longer a single create IO request for the *license.rtf* file. As the user doesn't have write access when the create call is made to the file system it results in an ACCESS DENIED error. The *LUAFV* driver sees the error in its post-create callback and as virtualization is enabled it makes a second create for only read access. This second create succeeds. Due to the altitude of LUAFV this process is normally hidden from the Process Monitor.

In the first table event ID 2 we saw the caller setting the file length to 0. However in the second table we now see that the virtual file needs to be created and the contents of the original file are copied into the new virtual file. Only after that operation has been completed will the length of the file be set to 0. The last 2 events are more or less the same.

I hope this is a clear demonstration both of how the altitude directly affects the operation of mini-filter drivers as well as how much file information you might be missing in Process Monitor without realizing it.

Concurrency and Reentrancy

The IO manager is designed to operate asynchronously. It's possible that multiple threads could be calling into the same IO driver at the same time and the filter manager is no different. There's no explicit locking in the filter manager which would prevent multiple IO requests being dispatched at the same time to the same file object. This can lead to concurrency and reentrancy issues.

The filter driver can assign shared state based on the file stream or file object. This can be extracted in the filter when operating on the file and used to store and retrieve the current state information. If you dispatch multiple IO requests to the same file it can result in an invalid state or memory corruption issues.

An example of this kind of issue is CVE-2019-0836 which was a race condition in the LUAFV driver related to handling of the SECTION_OBJECT_POINTERS structure in the file object. Basically by racing a read against a write IO request on the same file it was possible to get the wrong *SECTION_OBJECT_POINTERS* structure assigned to the virtual file allowing a normal user to bypass access checks and map a read-only file as writable.

To solve this problem the driver needs to not maintain complex state between pre and post operation callbacks or over any calls out to any API which could be trapped by a user-mode application.

Incorrect Forwarding of IO Operations

We showed earlier how to retarget an IO operation to another file object by switching the *TargetFileObject* pointer. This needs to be done very carefully as when working with file object pointers directly almost any operation can be performed on them. For example, if a file is opened read-only a write operation can still be dispatched to the file object itself and it'll succeed.

The only thing which prevents a user-mode application from doing this is the kernel checks that the handle passed by the application to the *NtWriteFile* system call has the *FILE_WRITE_DATA* access right set. If not the system call can return *STATUS_ACCESS_DENIED*. However, if the handle has write access to a file object, but the filter driver redirects that operation to a read-only file then the check is bypassed and the user can write to a file they don't necessarily control.

Another place this can happen is the dispatch of IO control codes. Each control code has a flag which indicates if the file handle requires read and/or write access to be dispatched. This check is performed in the IO manager before the request ever makes it to the file system. If the filter drivers blindly forward IO control codes to a separate file it could send a code which normally requires write access on the handle bypassing security checks.

The LUAFV driver is a good example of a mini-filter driver where this forwarding takes place. The previously mentioned issue, CVE-2019-0836 while it's a concurrency issue also relies on the fact that the file object can be written to even though it was opened read-only.

Summary

In summary I think that mini-filter drivers are an under-appreciated source of privilege escalation bugs on Windows. In part that's because they're not easy to understand. They have complex interactions with the rest of the IO system which makes understanding difficult but can introduce really subtle and interesting issues. I hope I've given you enough information to better understand how mini-filter drivers function, how you communicate with them and what sorts of unique bug classes you might discover.

If you want some more information a good blog on the inner workings of filters drivers is Of Filesystems and Other Demons. It's not been updated in a long while but it still contains some valuable information. You can also refer to MSDN which has a fairly comprehensive section on mini-filters as well as the Windows Driver Kit sample code. Finally as a reminder I've uploaded a filter manager header file for use in reverse engineering tools such as IDA Pro.

Posted by Ryan at 9:04 AM