6

FILESYSTEM MINIFILTER DRIVERS

While the drivers covered in previous chapters can monitor many important events on the system, they aren't able to detect a particularly critical kind of activity: filesystem operations. Using filesystem minifilter drivers, or *minifilters* for short, endpoint security products can learn about the files being created, modified, written to, and deleted.

These drivers are useful because they can observe an attacker's interactions with the filesystem, such as the dropping of malware to disk. Often, they work in conjunction with other components of the system. By integrating with the agent's scanning engine, for example, they can enable the EDR to scan files.

Minifilters might, of course, monitor the native Windows filesystem, which is called the New Technology File System (NTFS) and is implemented in *ntfs.sys*. However, they might also monitor other important filesystems, including named pipes, a bidirectional inter-process communication mechanism implemented in *npfs.sys*, and mailslots, a unidirectional inter-process communication mechanism implemented in *msfs.sys*. Adversary tools, particularly command-and-control agents, tend to make heavy use of these mechanisms, so tracking their activities provides crucial telemetry. For example, Cobalt Strike's Beacon uses named pipes for tasking and the linking of peer-to-peer agents.

Minifilters are similar in design to the drivers discussed in the previous chapters, but this chapter covers some unique details about their implementations, capabilities, and operations on Windows. We'll also discuss evasion techniques that attackers can leverage to interfere with them.

Legacy Filters and the Filter Manager

Before Microsoft introduced minifilters, EDR developers would write legacy filter drivers to monitor filesystem operations. These drivers would sit on the filesystem stack, directly inline of user-mode calls destined for the filesystem, as shown in Figure 6-1.

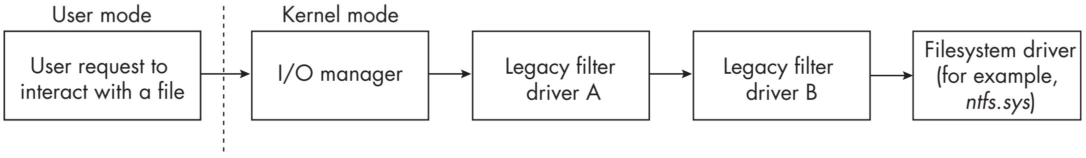
User mode
Kernel mode
User request to interact with a file
I/O manager
Legacy filter driver A
Legacy filter driver B
Filesystem driver (for example, *ntfs.sys*)

Figure 6-1: The legacy filter driver architecture

These drivers were notoriously difficult to develop and support in production environments. A 2019 article published in *The NT Insider*, titled "Understanding Minifilters: Why and How File System Filter Drivers Evolved," highlights seven large problems that developers face when writing legacy filter drivers:

Confusing Filter Layering

In cases when there is more than one legacy filter installed on the system, the architecture defines no order for how these drivers should be placed on the filesystem stack. This prevents the driver developer from knowing when the system will load their driver in relation to the others.

A Lack of Dynamic Loading and Unloading

Legacy filter drivers can't be inserted into a specific location on the device stack and can only be loaded at the top of the stack. Additionally, legacy filters can't be unloaded easily and typically require a full system reboot to unload.

Tricky Filesystem-Stack Attachment and Detachment

The mechanics of how the filesystem stack attaches and detaches devices are extremely complicated, and developers must have a substantial amount of arcane knowledge to ensure that their driver can appropriately handle odd edge cases.

Indiscriminate IRP Processing

Legacy filter drivers are responsible for processing *all* Interrupt Request Packets (IRPs) sent to the device stack, regardless of whether they are interested in the IRPs or not.

Challenges with Fast I/O Data Operations

Windows supports a mechanism for working with cached files, called *Fast I/O*, that provides an alternative to its standard packet-based I/O model. It relies on a dispatch table implemented in the legacy drivers. Each driver processes Fast I/O requests and passes them down the stack to the next driver. If a single driver in the stack lacks a dispatch table, it disables Fast I/O processing for the entire device stack.

An Inability to Monitor Non-data Fast I/O Operations

In Windows, filesystems are deeply integrated into other system components, such as the memory manager. For instance, when a user requests that a file be mapped into memory, the memory manager calls the Fast I/O callback `AcquireFileForNtCreateSection`. These non-data requests always bypass the device stack, making it hard for a legacy filter driver to collect information about them. It wasn't until Windows XP, which introduced `nt!FsRtlRegisterFileSystemFilterCallbacks()`, that developers could request this information.

Issues with Handling Recursion

Filesystems make heavy use of recursion, so filters in the filesystem stack must support it as well. However, due to the way that Windows manages I/O operations, this is easier said than done. Because each request passes through the entire device stack, a driver could easily deadlock or exhaust its resources if it handles recursion poorly.

To address some of these limitations, Microsoft introduced the filter manager model. The filter manager (*fltmgr.sys*) is a driver that ships with Windows and exposes functionality commonly used by filter drivers when intercepting filesystem operations. To leverage this functionality, developers can write minifilters. The filter manager then intercepts requests destined for the filesystem and passes them to the minifilters loaded on the system, which exist in their own sorted stack, as shown in Figure 6-2.

Minifilters are substantially easier to develop than their legacy counterparts, and EDRs can manage them more easily by dynamically loading and unloading them on a running system. The ability to access functionality exposed by the filter manager makes for less complex drivers, allowing for easier maintenance. Microsoft has made tremendous efforts to move developers away from the legacy filter model and over to the minifilter model. It has even included an optional registry value that allows administrators to block legacy filter drivers from being loaded on the system altogether.

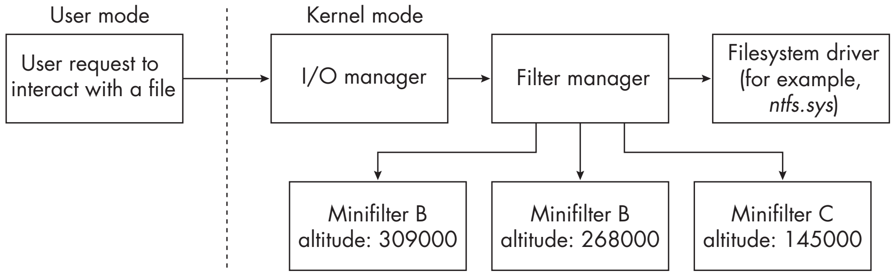
User mode
Kernel mode
User request to interact with a file
I/O manager
Filter manager
Filesystem driver (for example, *ntfs.sys*)
Minifilter B altitude: 309000
Minifilter B altitude: 268000
Minifilter C altitude: 145000

Figure 6-2: The filter manager and minifilter architecture

Minifilter Architecture

Minifilters have a unique architecture in several respects. First is the role of the filter manager itself. In a legacy architecture, filesystem drivers would filter I/O requests directly, while in a minifilter architecture, the filter manager handles this task before passing information about the requests to the minifilters loaded on the system. This means that minifilters are only indirectly attached to the filesystem stack. Also, they register with the filter manager for the specific operations they're interested in, removing the need for them to handle all I/O requests.

Next is how they interact with registered callback routines. As with the drivers discussed in the previous chapters, minifilters may register both pre- and post-operation callbacks. When a supported operation occurs, the filter manager first calls the correlated pre-operation callback function in each of the loaded minifilters. Once a minifilter completes its pre-operation routine, it passes control back to the filter manager, which calls the next callback function in the subsequent driver. When all drivers have completed their pre-operation callbacks, the request travels to the filesystem driver, which processes the operation. After receiving the I/O request for completion, the filter manager invokes the post-operation callback functions in the minifilters in reverse order. Once the post-operation callbacks complete, control is transferred back to the I/O manager, which eventually passes control back to the caller application.

Each minifilter has an *altitude*, which is a number that identifies its location in the minifilter stack and determines when the system will load that minifilter. Altitudes address the issue of ordering that plagued legacy filter drivers. Ideally, Microsoft assigns altitudes to the minifilters of production applications, and these values are specified in the drivers' registry keys, under `Altitude`. Microsoft sorts altitudes into load-order groups, which are shown in Table 6-1.

Table 6-1: Microsoft's Minifilter Load-Order Groups

Altitude range	Load-order group name	Minifilter role
420000–429999	Filter	Legacy filter drivers
400000–409999	FSFilter Top	Filters that must attach above all others
360000–389999	FSFilter Activity Monitor	Drivers that observe and report on file I/O
340000–349999	FSFilter Undelete	Drivers that recover deleted files
320000–329998	FSFilter Anti-Virus	Antimalware drivers
300000–309998	FSFilter Replication	Drivers that copy data to a remote system
280000–289998	FSFilter Continuous Backup	Drivers that copy data to backup media
260000–269998	FSFilter Content Screener	Drivers that prevent the creation of specific files or content
240000–249999	FSFilter Quota Management	Drivers that provide enhanced filesystem quotas that limit the space allowed for a volume or folder
220000–229999	FSFilter System Recovery	Drivers that maintain operating system integrity
200000–209999	FSFilter Cluster File System	Drivers used by applications that provide file server metadata across a network
180000–189999	FSFilter HSM	Hierarchical storage management drivers
170000–174999	FSFilter Imaging	ZIP-like drivers that provide a virtual namespace
160000–169999	FSFilter Compression	File-data compression drivers
140000–149999	FSFilter Encryption	File-data encryption and decryption drivers
130000–139999	FSFilter Virtualization	Filepath virtualization drivers
120000–129999	FSFilter Physical Quota Management	Drivers that manage quotes by using physical block counts
100000–109999	FSFilter Open File	Drivers that provide snapshots of already-opened files
80000–89999	FSFilter Security Enhancer	Drivers that apply file-based lockdowns and enhanced access control
60000–69999	FSFilter Copy Protection	Drivers that check for out-of-band data on storage media
40000–49999	FSFilter Bottom	Filters that must attach below all others
20000–29999	FSFilter System	Reserved
<20000	FSFilter Infrastructure	Reserved for system use but attaches closest to the filesystem

Most EDR vendors register their minifilters in the FSFilter Anti-Virus or FSFilter Activity Monitor group. Microsoft publishes a list of registered altitudes, as well as their associated filenames and publishers. Table 6-2 lists altitudes assigned to minifilters belonging to popular commercial EDR solutions.

Table 6-2: Altitudes of Popular EDRs

Altitude	Vendor	EDR
389220	Sophos	*sophosed.sys*
389040	SentinelOne	*sentinelmonitor.sys*
328010	Microsoft	*wdfilter.sys*
321410	CrowdStrike	*csagent.sys*
388360	FireEye/Trellix	*fekern.sys*
386720	Bit9/Carbon Black/VMWare	*carbonblackk.sys*

While an administrator can change a minifilter's altitude, the system can load only one minifilter at a single altitude at one time.

Writing a Minifilter

Let's walk through the process of writing a minifilter. Each minifilter begins with a `DriverEntry()` function, defined in the same way as other drivers. This function performs any required global initializations and then registers the minifilter. Finally, it starts filtering I/O operations and returns an appropriate value.

Beginning the Registration

The first, and most important, of these actions is registration, which the `DriverEntry()` function performs by calling `fltmgr!FltRegisterFilter()`. This function adds the minifilter to the list of registered minifilter drivers on the host and provides the filter manager with information about the minifilter, including a list of callback routines. This function is defined in Listing 6-1.

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

Listing 6-1: The `fltmgr!FltRegisterFilter()` function definition

Of the three parameters passed to it, the `Registration` parameter is the most interesting. This is a pointer to an `FLT_REGISTRATION` structure, defined in Listing 6-2, which houses all the relevant information about the minifilter.

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

Listing 6-2: The `FLT_REGISTRATION` *structure definition*

The first two members of this structure set the structure size, which is always `sizeof(FLT_REGISTRATION)`, and the structure revision level, which is always `FLT_REGISTRATION_VERSION`. The next member is *flags*, which is a bitmask that may be zero or a combination of any of the following three values:

`FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP` (1)

The minifilter won't be unloaded in the event of a service stop request.

`FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS` (2)

The minifilter supports named pipe and mailslot requests.

`FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME` (4)

The minifilter supports attaching to a Direct Access (DAX) volume.

Following this member is the context registration. This will be either an array of `FLT_CONTEXT_REGISTRATION` structures or null. These contexts allow a minifilter to associate related objects and preserve state across I/O operations. After this array of context comes the critically important operation registration array. This is a variable length array of `FLT_OPERATION_REGISTRATION` structures, which are defined in Listing 6-3. While this array can technically be null, it's rare to see that configuration in an EDR sensor. The minifilter must provide a structure for each type of I/O for which it registers a pre-operation or post-operation callback routine.

```
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;
```

Listing 6-3: The `FLT_OPERATION_REGISTRATION` *structure definition*

The first parameter indicates which major function the minifilter is interested in processing. These are constants defined in *wdm.h*, and Table 6-3 lists some of those most relevant to security monitoring.

Table 6-3: Major Functions and Their Purposes

Major function	Purpose
`IRP_MJ_CREATE` (0x00)	A new file is being created or a handle to an existing one is being opened.
`IRP_MJ_CREATE_NAMED_PIPE` (0x01)	A named pipe is being created or opened.
`IRP_MJ_CLOSE` (0x02)	A handle to a file object is being closed.
`IRP_MJ_READ` (0x03)	Data is being read from a file.
`IRP_MJ_WRITE` (0x04)	Data is being written to a file.
`IRP_MJ_QUERY_INFORMATION` (0x05)	Information about a file, such as its creation time, has been requested.
`IRP_MJ_SET_INFORMATION` (0x06)	Information about a file, such as its name, is being set or updated.
`IRP_MJ_QUERY_EA` (0x07)	A file's extended information has been requested.
`IRP_MJ_SET_EA` (0x08)	A file's extended information is being set or updated.
`IRP_MJ_LOCK_CONTROL` (0x11)	A lock is being placed on a file, such as via a call to `kernel32!LockFileEx()`.
`IRP_MJ_CREATE_MAILSLOT` (0x13)	A new mailslot is being created or opened.
`IRP_MJ_QUERY_SECURITY` (0x14)	Security information about a file is being requested.
`IRP_MJ_SET_SECURITY` (0x15)	Security information related to a file is being set or updated.
`IRP_MJ_SYSTEM_CONTROL` (0x17)	A new driver has been registered as a supplier of Windows Management Instrumentation.

The next member of the structure specifies the flags. This bitmask describes when the callback functions should be invoked for cached I/O or paging I/O operations. At the time of this writing, there are four supported flags, all of which are prefixed with `FLTFL_OPERATION_REGISTRATION_`. First, `SKIP_PAGING_IO` indicates whether a callback should be invoked for IRP-based read or write paging I/O operations. The `SKIP_CACHED_IO` flag is used to prevent the invocation of callbacks on fast I/O-based read or write cached I/O operations. Next, `SKIP_NON_DASD_IO` is used for requests issued on a Direct Access Storage Device (DASD) volume handle. Finally, `SKIP_NON_CACHED_NON_PAGING_IO` prevents callback invocation on read or write I/O operations that are not cached or paging operations.

Defining Pre-operation Callbacks

The next two members of the `FLT_OPERATION_REGISTRATION` structure define the pre-operation or post-operation callbacks to be invoked when each of the target major functions occurs on the system. Pre-operation callbacks are passed via a pointer to an `FLT_PRE_OPERATION_CALLBACK` structure, and post-operation routines are specified as a pointer to an `FLT_POST_OPERATION_CALLBACK` structure. While these functions' definitions aren't too dissimilar, their capabilities and limitations vary substantially.

As with callbacks in other types of drivers, pre-operation callback functions allow the developer to inspect an operation on its way to its destination (the target filesystem, in the case of a minifilter). These callback functions receive a pointer to the callback data for the operation and some opaque pointers for the objects related to the current I/O request, and they return an `FLT_PREOP_CALLBACK_STATUS` return code. In code, this would look like what is shown in Listing 6-4.

```
PFLT_PRE_OPERATION_CALLBACK PfltPreOperationCallback;

FLT_PREOP_CALLBACK_STATUS PfltPreOperationCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in]  PCFLT_RELATED_OBJECTS FltObjects,
 [out]  PVOID *CompletionContext
)
{...}
```

Listing 6-4: Registering a pre-operation callback

The first parameter, `Data`, is the most complex of the three and contains all the major information related to the request that the minifilter is processing. The `FLT_CALLBACK_DATA` structure is used by both the filter manager and the minifilter to process I/O operations and contains a ton of useful data for any EDR agent monitoring filesystem operations. Some of the important members of this structure include:

`Flags` A bitmask that describes the I/O operation. These flags may come preset from the filter manager, though the minifilter may set additional flags in some circumstances. When the filter manager initializes the data structure, it sets a flag to indicate what type of I/O operation it represents: either fast I/O, filter, or IRP operations. The filter manager may also set flags indicating whether a minifilter generated or reissued the operation, whether it came from the non-paged pool, and whether the operation completed.

`Thread` A pointer to the thread that initiated the I/O request. This is useful for identifying the application performing the operation.

`Iopb` The I/O parameter block that contains information about IRP-based operations (for example, `IRP_BUFFERED_IO`, which indicates that it is a buffered I/O operation); the major function code; special flags related to the operation (for example, `SL_CASE_SENSITIVE`, which informs drivers in the stack that filename comparisons should be case sensitive); a pointer to the file object that is the target of the operation; and an `FLT_PARAMETERS` structure containing the parameters unique to the specific I/O operation specified by the major or minor function code member of the structure.

`IoStatus` A structure that contains the completion status of the I/O operation set by the filter manager.

`TagData` A pointer to an `FLT_TAG_DATA_BUFFER` structure containing information about reparse points, such as in the case of NTFS hard links or junctions.

`RequestorMode` A value indicating whether the request came from user mode or kernel mode.

This structure contains much of the information that an EDR agent needs to track file operations on the system. The second parameter passed to the pre-operation callback, a pointer to an `FLT_RELATED_OBJECTS` structure, provides supplemental information. This structure contains opaque pointers to the object associated with the operation, including the volume, minifilter instance, and file object (if present). The last parameter, `CompletionContext`, contains an optional context pointer that will be passed to the correlated post-operation callback if the minifilter returns `FLT_PREOP_SUCCESS_WITH_CALLBACK` or `FLT_PREOP_SYNCHRONIZE`.

On completion of the routine, the minifilter must return an `FLT_PREOP_CALLBACK_STATUS` value. Pre-operation callbacks may return one of seven supported values:

`FLT_PREOP_SUCCESS_WITH_CALLBACK (0)`

Return the I/O operation to the filter manager for processing and instruct it to call the minifilter's post-operation callback during completion.

`FLT_PREOP_SUCCESS_NO_CALLBACK (1)`

Return the I/O operation to the filter manager for processing and instruct it *not* to call the minifilter's post-operation callback during completion.

`FLT_PREOP_PENDING (2)`

Pend the I/O operation and do not process it further until the minifilter calls `fltmgr!FltCompletePendedPreOperation()`.

`FLT_PREOP_DISALLOW_FASTIO (3)`

Block the fast I/O path in the operation. This code instructs the filter manager not to pass the operation to any other minifilters below the current one in the stack and to only call the post-operation callbacks of those drivers at higher altitudes.

`FLT_PREOP_COMPLETE (4)`

Instruct the filter manager not to send the request to minifilters below the current driver in the stack and to only call the post-operation callbacks of those minifilters above it in the driver stack.

FLT_PREOP_SYNCHRONIZE (5)

Pass the request back to the filter manager but don't complete it. This code ensures that the minifilter's post-operation callback is called at IRQL ≤ *APC_LEVEL* in the context of the original thread.

FLT_PREOP_DISALLOW_FSFILTER_IO (6)

Disallow a fast `QueryOpen` operation and force the operation down the slower path, causing the I/O manager to process the request using an open, query, or close operation on the file.

The filter manager invokes the pre-operation callbacks for all minifilters that have registered functions for the I/O operation being processed before passing their requests to the filesystem, beginning with the highest altitude.

Defining Post-operation Callbacks

After the filesystem performs the operations defined in every minifilter's pre-operation callbacks, control is passed up the filter stack to the filter manager. The filter manager then invokes the post-operation callbacks of all minifilters for the request type, beginning with the lowest altitude. These post-operation callbacks have a similar definition to the pre-operation routines, as shown in Listing 6-5.

```
PFLT_POST_OPERATION_CALLBACK PfltPostOperationCallback;

FLT_POSTOP_CALLBACK_STATUS PfltPostOperationCallback(
 [in, out]  PFLT_CALLBACK_DATA Data,
 [in]   PCFLT_RELATED_OBJECTS FltObjects,
 [in, optional] PVOID CompletionContext,
 [in]   FLT_POST_OPERATION_FLAGS Flags
)
{...}
```

Listing 6-5: Post-operation callback routine definitions

Two notable differences here are the addition of the `Flags` parameter and the different return type. The only documented flag that a minifilter can pass is `FLTFL_POST_OPERATION_DRAINING`, which indicates that the minifilter is in the process of unloading. Additionally, post-operation callbacks can return different statuses. If the callback returns `FLT_POSTOP_FINISHED_PROCESSING` (*0*), the minifilter has completed its post-operation callback routine and is passing control back to the filter manager to continue processing the I/O request. If it returns `FLT_POSTOP_MORE_PROCESSING_REQUIRED` (*1*), the minifilter has posted the IRP-based I/O operation to a work queue and halted completion of the request until the work item completes, and it calls `fltmgr!FltCompletePendedPostOperation()`. Lastly, if it returns `FLT_POSTOP_DISALLOW_FSFILTER_IO` (*2*), the minifilter is disallowing a fast `QueryOpen` operation and forcing the operation down the slower path. This is the same as `FLT_PREOP_DISALLOW_FSFILTER_IO`.

Post-operation callbacks have some notable limitations that reduce their viability for security monitoring. The first is that they're invoked in an arbitrary thread unless the pre-operation callback passes the `FLT_PREOP_SYNCHRONIZE` flag, preventing the system from attributing the operation to the requesting application. Next is that post-operation callbacks are invoked at IRQL ≤ *DISPATCH_LEVEL*. This means that certain operations are restricted, including accessing most synchronization primitives (for example, mutexes), calling kernel APIs that require an IRQL ≤ *DISPATCH_LEVEL*, and accessing paged memory. One workaround to these limitations involves delaying the execution of the post-operation callback via the use of `fltmgr!FltDoCompletionProcessingWhenSafe()`, but this solution has its own challenges.

The array of these `FLT_OPERATION_REGISTRATION` structures passed in the `OperationRegistration` member of `FLT_REGISTRATION` may look like Listing 6-6.

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
 {IRP_MJ_CREATE, 0, MyPreCreate, MyPostCreate},
 {IRP_MJ_READ, 0, MyPreRead, NULL},
 {IRP_MJ_WRITE, 0, MyPreWrite, NULL},
 {IRP_MJ_OPERATION_END}
};
```

Listing 6-6: An array of operation registration callback structures

This array registers pre- and post-operation callbacks for `IRP_MJ_CREATE` and only pre-operation callbacks for `IRP_MJ_READ` and `IRP_MJ_WRITE`. No flags are passed in for any of the target operations. Also note that the final element in the array is `IRP_MJ_OPERATION_END`. Microsoft requires this value to be present at the end of the array, and it serves no functional purpose in the context of monitoring.

Defining Optional Callbacks

The last section in the `FLT_REGISTRATION` structure contains the optional callbacks. The first three callbacks, `FilterUnloadCallback`, `InstanceSetupCallback`, and `InstanceQueryTeardownCallback`, may all technically be null, but this will impose some restrictions on the minifilter and system behavior. For example, the system won't be able to unload the minifilter or attach to new filesystem volumes. The rest of the callbacks in this section of the structure relate to various functionality provided by the minifilter. These include things such as the interception of filename requests (`GenerateFileNameCallback`) and filename normalization (`NormalizeNameComponentCallback`). In general, only the first three semi-optional callbacks are registered, and the rest are rarely used.

Activating the Minifilter

After all callback routines have been set, a pointer to the created `FLT_REGISTRATION` structure is passed as the second parameter to `fltmgr!FltRegisterFilter()`. Upon completion of this function, an opaque filter pointer (`PFLT_FILTER`) is returned to the caller in the `RetFilter` parameter. This pointer uniquely identifies the minifilter and remains static as long as the driver is loaded on the system. This pointer is typically preserved as a global variable.

When the minifilter is ready to start processing events, it passes the `PFLT_FILTER` pointer to `fltmgr!FltStartFilter()`. This notifies the filter manager that the driver is ready to attach to filesystem volumes and start filtering I/O requests. After this function returns, the minifilter will be considered active and sit inline of all relevant filesystem operations. The callbacks registered in the `FLT_REGISTRATION` structure will be invoked for their associated major functions. Whenever the minifilter is ready to unload itself, it passes the `PFLT_FILTER` pointer to `fltmgr!FltUnregisterFilter()` to remove any contexts that the minifilter has set on files, volumes, and other components and calls the registered `InstanceTeardownStartCallback` and `InstanceTeardownCompleteCallback` functions.

Managing a Minifilter

Compared to working with other drivers, the process of installing, loading, and unloading a minifilter requires special consideration. This is because minifilters have specific requirements related to the setting of registry values. To make the installation process easier, Microsoft recommends installing minifilters through a *setup information (INF)* file. The format of these INF files is beyond the scope of this book, but there are some interesting details relevant to how minifilters work that are worth mentioning.

The `ClassGuid` entry in the `Version` section of the INF file is a GUID that corresponds to the desired load-order group (for example, `FSFilter Activity Monitor`). In the `AddRegistry` section of the file, which specifies the registry keys to be created, you'll find information about the minifilter's altitude. This section may include multiple similar entries to describe where the system should load various instances of the minifilter. The altitude can be set to the name of a variable (for example, `%MyAltitude%`) defined in the `Strings` section of the INF file. Lastly, the `ServiceType` entry under the `ServiceInstall` section is always set to `SERVICE_FILE_SYSTEM_DRIVER (2)`.

Executing the INF installs the driver, copying files to their specified locations and setting up the required registry keys. Listing 6-7 shows an example of what this looks like in the registry keys for *WdFilter*, Microsoft Defender's minifilter driver.

```
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\WdFilter\" | Select *
-Exclude PS* | fl

DependOnService : {FltMgr}
Description  : @%ProgramFiles%\Windows Defender\MpAsDesc.dll,-340
DisplayName  : @%ProgramFiles%\Windows Defender\MpAsDesc.dll,-330
ErrorControl  : 1
Group    : FSFilter Anti-Virus
ImagePath   : system32\drivers\wd\WdFilter.sys
Start    : 0
SupportedFeatures : 7
Type    : 2
```

```
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\WdFilter\Instances\
WdFilter Instance" | Select * -Exclude PS* | fl

Altitude : 328010
Flags : 0
```

Listing 6-7: Viewing WdFilter*'s altitude with PowerShell*

The `Start` key dictates when the minifilter will be loaded. The service can be started and stopped using the Service Control Manager APIs, as well as through a client such as *sc.exe* or the Services snap-in. In addition, we can manage minifilters with the filter manager library, *FltLib*, which is leveraged by the *fltmc.exe* utility included by default on Windows. This setup also includes setting the altitude of the minifilter, which for *WdFilter* is 328010.

Detecting Adversary Tradecraft with Minifilters

Now that you understand the inner workings of minifilters, let's explore how they contribute to the detection of attacks on a system. As discussed in "Writing a Minifilter" on page 108, a minifilter can register pre- or post-operation callbacks for activities that target any filesystem, including NTFS, named pipes, and mailslots. This provides an EDR with an extremely powerful sensor for detecting adversary activity on the host.

File Detections

If an adversary interacts with the filesystem, such as by creating new files or modifying the contents of existing files, the minifilter has an opportunity to detect the behavior. Modern attacks have tended to avoid dropping artifacts directly onto the host filesystem in this way, embracing the "disk is lava" mentality, but many hacking tools continue to interact with files due to limitations of the APIs being leveraged. For example, consider `dbghelp!MiniDumpWriteDump()`, a function used to create process memory dumps. This API requires that the caller pass in a handle to a file for the dump to be written to. The attacker must work with files if they want to use this API, so any minifilter that processes `IRP_MJ_CREATE` or `IRP_MJ_WRITE` I/O operations can indirectly detect those memory-dumping operations.

Additionally, the attacker has no control over the format of the data being written to the file, allowing a minifilter to coordinate with a scanner to detect a memory-dump file without using function hooking. An attacker might try to work around this by opening a handle to an existing file and overwriting its content with the dump of the target process's memory, but a minifilter monitoring `IRP_MJ_CREATE` could still detect this activity, as both the creation of a new file and the opening of a handle to an existing file would trigger it.

Some defenders use these concepts to implement *filesystem canaries*. These are files created in key locations that users should seldom, if ever, interact with. If an application other than a backup agent or the EDR requests a handle to a canary file, the minifilter can take immediate action, including crashing the system. Filesystem canaries provide strong (though at times brutal) anti-ransomware control, as ransomware tends to indiscriminately encrypt files on the host. By placing a canary file in a directory nested deep in the filesystem, hidden from the user but still in one of the paths typically targeted by ransomware, an EDR can limit the damage to the files that the ransomware encountered before reaching the canary.

Named Pipe Detections

Another key piece of adversary tradecraft that minifilters can detect highly effectively is the use of named pipes. Many command-and-control agents, like Cobalt Strike's Beacon, make use of named pipes for tasking, I/O, and linking. Other offensive techniques, such as those that use token impersonation for privilege escalation, revolve around the creation of a named pipe. In both cases, a minifilter monitoring `IRP_MJ_CREATE_NAMED_PIPE` requests would be able to detect the attacker's behavior, in much the same way as those that detect file creation via `IRP_MJ_CREATE`.

Minifilters commonly look for the creation of anomalously named pipes, or those originating from atypical processes. This is useful because many tools used by adversaries rely on the use of named pipes, so an attacker who wants to blend in should pick pipe and host process names that are typical in the environment. Thankfully for attackers and defenders alike, Windows makes enumerating existing named pipes easy, and we can straightforwardly identify many of the common process-to-pipe relationships. One of the most well-known named pipes in the realm of security is *mojo*. When a Chromium process spawns, it creates several named pipes with the format *mojo.PID.TID.VALUE* for use by an IPC abstraction library called Mojo. This named pipe became popular after its inclusion in a well-known repository for documenting Cobalt Strike's Malleable profile options.

There are a few problems with using this specific named pipe that a minifilter can detect. The main one is related to the structured formatting used for the name of the pipe. Because Cobalt Strike's pipe name is a static attribute tied to the instance of the Malleable profile, it is immutable at runtime. This means that an adversary would need to accurately predict the process and thread IDs of their Beacon to ensure the attributes of their process match those of the pipe name format used by Mojo. Remember that minifilters with pre-operation callbacks for monitoring `IRP_MJ_CREATE_NAMED_PIPE` requests are guaranteed to be invoked in the context of the calling thread. This means that when a Beacon process creates the "mojo" named pipe, the minifilter can check that its current context matches the information in the pipe name. Pseudocode to demonstrate this would look like that shown in Listing 6-8.

```
DetectMojoMismatch(string mojoPipeName)
{
 pid = GetCurrentProcessId();
 tid = GetCurrentThreadId();
❶ if (!mojoPipeName.beginsWith("mojo. " + pid + "." + tid + "."))

  {
  // Bad Mojo pipe found
  }
}
```

Listing 6-8: Detecting anomalous Mojo named pipes

Since the format used in Mojo named pipes is known, we can simply concatenate the PID and TID ❶ of the thread creating the named pipe and ensure that it matches what is expected. If not, we can take some defensive action.

Not every command inside Beacon will create a named pipe. There are certain functions that will create an anonymous pipe (as in, a pipe without a name), such as `execute-assembly`. These types of pipes have limited operational viability, as their name can't be referenced and code can interact with them through an open handle only. What they lose in functionality, however, they gain in evasiveness.

Riccardo Ancarani's blog post "Detecting Cobalt Strike Default Modules via Named Pipe Analysis" details the OPSEC considerations related to Beacon's usage of anonymous pipes. In his research, he found that while Windows components rarely used anonymous pipes, their creation could be profiled, and their creators could be used as viable *spawnto* binaries. These included *ngen.exe*, *wsmprovhost.exe*, and *firefox.exe*, among others. By setting their sacrificial processes to one of these executables, attackers could ensure that any actions resulting in the creation of anonymous pipes would likely remain undetected.

Bear in mind, however, that activities making use of named pipes would still be vulnerable to detection, so operators would need to restrict their tradecraft to activities that create anonymous pipes only.

Evading Minifilters

Most strategies for evading an EDR's minifilters rely on one of three techniques: unloading, prevention, or interference. Let's walk through examples of each to demonstrate how we can use them to our advantage.

Unloading

The first technique is to completely unload the minifilter. While you'll need administrator access to do this (specifically, the `SeLoadDriverPrivilege` token privilege), it's the most surefire way to evade the minifilter. After all, if the driver is no longer on the stack, it can't capture events.

Unloading the minifilter can be as simple as calling `fltmc.exe unload`, but if the vendor has put a lot of effort into hiding the presence of their minifilter, it might require complex custom tooling. To explore this idea further, let's target Sysmon, whose minifilter, *SysmonDrv*, is configured in the registry, as shown in Listing 6-9.

```
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv" | Select *
-Exclude PS* | fl

Type   : 1
Start  : 0
ErrorControl : 1
ImagePath : SysmonDrv.sys
DisplayName : SysmonDrv
Description : System Monitor driver

PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv\Instances\
Sysmon Instance\" | Select * -Exclude PS* | fl

Altitude : 385201
Flags : 0
```

Listing 6-9: Using PowerShell to view SysmonDrv*'s configuration*

By default, *SysmonDrv* has the altitude 385201, and we can easily unload it via a call to `fltmc.exe unload SysmonDrv`, assuming the caller has the required privilege. Doing so would create a *FilterManager* event ID of 1, which indicates that a filesystem filter was unloaded, and a Sysmon event ID of 255, which indicates a driver communication failure. However, Sysmon will no longer receive events.

To complicate this process for attackers, the minifilter sometimes uses a random service name to conceal its presence on the system. In the case of Sysmon, an administrator can implement this approach during installation by passing the `-d` flag to the installer and specifying a new name. This prevents an attacker from using the built-in *fltmc.exe* utility unless they can also identify the service name.

However, an attacker can abuse another feature of production minifilters to locate the driver and unload it: their altitudes. Because Microsoft reserves specific altitudes for certain vendors, an attacker can learn these values and then simply walk the registry or use `fltlib!FilterFindNext()` to locate any driver with the altitude in question. We can't use *fltmc.exe* to unload minifilters based on an altitude, but we can either resolve the driver's name in the registry or pass the minifilter's name to `fltlib!FilterUnload()` for tooling that makes use of `fltlib!FilterFindNext()`. This is how the Shhmon tool, which hunts and unloads *SysmonDrv*, works under the hood.

Defenders could further thwart attackers by modifying the minifilter's altitude. This isn't recommended in production applications, however, because another application might already be using the chosen value. EDR agents sometimes operate across millions of devices, raising the odds of an altitude collision. To mitigate this risk, a vendor might compile a list of active minifilter allocations from Microsoft and choose one not already in use, although this strategy isn't bulletproof.

In the case of Sysmon, defenders could either patch the installer to set the altitude value in the registry to a different value upon installation or manually change the altitude after installation by directly modifying the registry value. Since Windows doesn't place any technical controls on altitudes, the engineer could move *SysmonDrv* to any altitude they wish. Bear in mind, however, that the altitude affects the minifilter's position in the stack, so choosing too low a value could have unintended implications for the efficacy of the tool.

Even with all these obfuscation methods applied, an attacker could still unload a minifilter. Starting in Windows 10, both the vendor and Microsoft must sign a production driver before it can be loaded onto the system, and because these signatures are meant to identify the drivers, they include information about the vendor that signed them. This information is often enough to tip an adversary off to the presence of the target minifilter. In practice, the attacker could walk the registry or use the `fltlib!FilterFindNext()` approach to enumerate minifilters, extract the path to the driver on disk, and parse the digital signatures of all enumerated files until they've identified a file signed by an EDR. At that point, they can unload the minifilter using one of the previously covered methods.

As you've just learned, there are no particularly great ways to hide a minifilter on the system. This doesn't mean, however, that these obfuscations aren't worthwhile. An attacker might lack the tooling or knowledge to counter the obfuscations, providing time for the EDR's sensors to detect their activity without interference.

Prevention

To prevent filesystem operations from ever passing through an EDR's minifilter, attackers can register their own minifilter and use it to force the completion of I/O operations. As an example, let's register a malicious pre-operation callback for `IRP_MJ_WRITE` requests, as shown in Listing 6-10.

```
PFLT_PRE_OPERATION_CALLBACK EvilPreWriteCallback;

FLT_PREOP_CALLBACK_STATUS EvilPreWriteCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in] PCFLT_RELATED_OBJECTS FltObjects,
 [out] PVOID *CompletionContext
)
{
 --snip--
}
```

Listing 6-10: Registering a malicious pre-operation callback routine

When the filter manager invokes this callback routine, it must return an `FLT_PREOP_CALLBACK_STATUS` value. One of the possible values, `FLT_PREOP_COMPLETE`, tells the filter manager that the current minifilter is in the process of completing the request, so the request shouldn't be passed to any minifilters below the current altitude. If a minifilter returns this value, it must set the `NTSTATUS` value in the `Status` member of the I/O status block to the operation's final status. Antivirus engines whose minifilters communicate with user-mode scanning engines commonly use this functionality to determine whether malicious content is being written to a file. If the scanner indicates to the minifilter that the content is malicious, the minifilter completes the request and returns a failure status, such as `STATUS_VIRUS_INFECTED`, to the caller.

But attackers can abuse this feature of minifilters to prevent the security agent from ever intercepting their filesystem operations. Using the earlier callback we registered, this would look something like what's shown in Listing 6-11.

```
FLT_PREOP_CALLBACK_STATUS EvilPreWriteCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in]  PCFLT_RELATED_OBJECTS FltObjects,
 [out]  PVOID *CompletionContext
)
{
    --snip--
    if (IsThisMyEvilProcess(PsGetCurrentProcessId())
    {
        --snip--
     ❶ Data->IoStatus.Status = STATUS_SUCCESS;
        return FLT_PREOP_COMPLETE
    }
    --snip--
}
```

Listing 6-11: Intercepting write operations and forcing their completion

The attacker first inserts their malicious minifilter at an altitude higher than the minifilter belonging to the EDR. Inside the malicious minifilter's pre-operation callback would exist logic to complete the I/O requests coming from the adversary's processes in user mode ❶, preventing them from being passed down the stack to the EDR.

Interference

A final evasion technique, interference, is built around the fact that a minifilter can alter members of the `FLT_CALLBACK_DATA` structure passed to its callbacks on a request. An attacker can modify any members of this structure except the `RequestorMode` and `Thread` members. This includes the file pointer in the `FLT_IO_PARAMETER_BLOCK` structure's `TargetFileObject` member. The only requirement of the malicious minifilter is that it calls `fltmgr!FltSetCallbackDataDirty()`, which indicates that the callback data structure has been modified when it is passing the request to minifilters lower in the stack.

An adversary can abuse this behavior to pass bogus data to the minifilter associated with an EDR by inserting itself anywhere above it in the stack, modifying the data tied to the request and passing control back to the filter manager. A minifilter that receives the modified request may evaluate whether `FLTFL_CALLBACK_DATA_DIRTY`, which is set by `fltmgr!FltSetCallbackDataDirty()`, is present and act accordingly, but the data will still be modified.

Conclusion

Minifilters are the de facto standard for monitoring filesystem activity on Windows, whether it be for NTFS, named pipes, or even mailslots. Their implementation is somewhat more complex than the drivers discussed earlier in this book, but the way they work is very similar; they sit inline of some system operation and receive data about the activity. Attackers can evade minifilters by abusing some logical issue in the sensor or even unloading the driver entirely, but most adversaries have adapted their tradecraft to drastically limit creating new artifacts on disk to reduce the chances of a minifilter picking up their activity.