An Introduction to Standard and Isolation Minifilters

osr.com/nt-insider/2017-issue2/introduction-standard-isolation-minifilters

October 6, 2017

Last reviewed and updated: 10 August 2020

The filter driver concept is one of the most powerful architectural features of the Windows I/O subsystem. A filter can add value to the functionality of an existing device by simply attaching itself over that device. And, of course, filtering a device requires no change to the driver for the underlying device.

Filter drivers are installed at many levels in a typical Windows system. For example, there are standard Windows-supplied filter drivers at the volume level that provide volume snapshot functionality (to support backups), as well as optional full volume encryption (in support of Windows Bitlocker). There may also be manufacturer-specific filter drivers, such as a filter for a mouse or keyboard that adds support for buttons that are unique to a given model.

One of the most common, and also the most powerful, places to insert a filter in a Windows system is over a file system. File system filters intercept I/O operations (from both applications and the system itself) before those I/O operations reach the file system. This allows them to monitor, track, manage, manipulate, and even accept or reject I/O operations before the file system gets to see them. The type of file system filter that most people are familiar with is probably the antivirus filter. This type of filter typically intercepts file open requests and suspends them while the filter (or, more likely, an associated service running in user mode) scans the file being opened for viruses. If any viruses are found, the open request can be canceled. If no viruses are found, the open request can be allowed to complete normally.

File system filters are commonly used for everything from antivirus and malware scanning as just described, to software license tracking and management, to auditing and changed tracking on files, to on access transparent data encryption and decryption. File system filters can also be used for other, less obvious, purposes. For example, because they see which files are created and written, file system filters are often play key roles in backup products and hierarchical storage subsystems. And because file system filters are able to be the first interpreters of the file system “name space” that applications see, they can also perform powerful file redirection operations, such as making a remote file (such as one stored somewhere in the cloud) appear to be local.

Since its introduction in Windows XP SP2, the File System Minifilter model has become the preferred mechanism for implementing file system filters. This is for good reason, because the Minifilter model provides an excellent organization and support framework for file system filter driver development. With reasonably good documentation and a significant set of examples on GitHub, many devs feel that writing a file system Minifilter is well within their ability. And they're right… provided they stay within certain boundaries.

This article describes the basic architectural concepts of Windows file system Minifilters. It then describes two distinct types of Minifilters: **Standard Minifilters**, and **Isolation Minifilters**. Finally, it describes why a project to develop a file system Isolation Minifilter is significantly more complex than a project to develop a Standard Minifilter.

Standard Isolation Minifilters vs. Isolation Minifilters
In this article, we clarify a couple of common terms that are in use in the Windows file system community.

Standard Minifilter

A Standard Minifilter is a Windows file system Minifilter driver that monitors or tracks file system data. Most all antivirus scanners are Standard Minifilters.

Isolation Minifilter

An Isolation Minifilter is a Windows file system Minifilter driver that separates the view(s) of a file's data from the actual underlying data of that same file. A typical example of an Isolation Minifilter is an on access transparent encryption/decryption filter. Isolation Minifilters use the "same stack" concept, and provide their different views by providing unique cache sections for each view.

Filter Manager and Altitudes

The basis for the file system Minifilter model is Filter Manager, which is a standard Windows component. Filter Manager is implemented as a legacy file system filter, and filters all file system instances. Because there can be multiple filters located over any given file system instance (the default installation of Windows 10 includes no less than nine standard file system Minifilters!), Filter Manager provides a system of "altitudes" that allows a developer to decide where in the filter hierarchy their Minifilter should be installed. See *Figure 1*.

In *Figure 1*, you can see Windows Filter Manager (labelled FLTMGR) filtering the instance of the NTFS file system that's mounted as the C drive. In the figure, Filter Manager has three file system Minifilters loaded: MiniFilter A, MiniFilter B and MiniFilter C. The order of these filters, MiniFilter A being above MiniFilter B, and MiniFilter B being above MiniFilter C is not random. Rather, this is determined by the Altitude assigned to each of the Minifilters. Altitudes are unique per Minifilter and are specified during Minifilter installation.

Altitude is an important attribute for file system filters, because filtering at the right altitude can be critical to proper operation. For example, consider two Minifilters that might be installed over an arbitrary file system: An on-access transparent data encryption Minifilter and an antivirus Minifilter. In order for the antivirus Minifilter to do its work, it needs to operate on the decrypted contents of files. Therefore, the antivirus Minifilter would need to be higher in altitude (that is, above) the data encryption Minifilter.

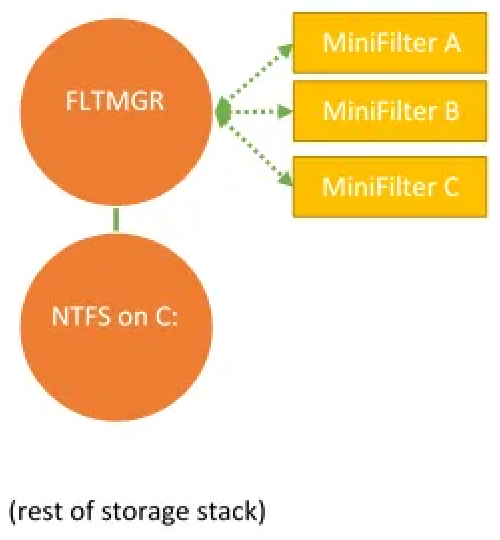

Figure 1 – Filter Manager with Three File System Minifilters

Minifilter Callbacks

When a Minifilter registers with Filter Manager, in addition to other things, it may elect to receive PreOperation and/or PostOperation callbacks for specific I/O operations. PreOperation callbacks are invoked before each I/O operation of the specified type is passed on to the file system being filtered (or, in fact, the next lowest altitude Minifilter). PostOperation Minifilter callbacks are invoked after the file system (and any lower Minifilters) have processed the particular type of I/O operation.

Filter Manager's support for callbacks is very well thought out. For example, when a given Minifilter receives a PreOperation callback, that filter can:

- Complete the operation entirely. This results in lower altitude Minifilters (if any), and even the file system that's being filtered, not seeing this I/O operation.
- Complete the operation, pass it along to lower altitude Minifilters (if any) and the underlying file system, and elect to have a callback when the operation is complete (PostOperation callback).
- Complete the operation, pass it along to lower altitude Minifilters (if any) and the underlying file system, but elect NOT to have a callback when the operation is complete (this is the PostOperation callback).
- Return with the operation in progress. In this case, the Minifilter will inform the Filter Manager later as to the status of the operation, including whether any underlying entities and need to be called or if a PostOperation callback is required.

Within its PreOperation and/or PostOperation callbacks, a Minifilter can perform almost any operation, including examining or modifying the data involved in the operation. Thus, the trick becomes understanding the specific meaning of those operations.

Win32 API vs Native Windows API

The Filter Manager framework for writing Minifilters is so powerful and well thought-out, that it's very easy for new Minifilter developers to be tricked into thinking that file system filtering is easy. The thing is: It *can* be easy, but it can also be get surprisingly confusing very quickly.

The inherent problem is that regardless of how powerful or pleasant the Filter Manager model is, the world of Windows file systems is inherently a complicated one. Some of this complexity comes from the fact that the Win32 API can sometimes be very different from the native Windows API that the I/O subsystem actually uses. A simple example that we often encounter here at OSR is the Win32 API CopyFile. Developers are often surprised that there's no native analog to this function. Rather, internally, **CopyFile** opens the source file, opens the target file, and if both of those operations are successful, issues a series of reads from the source file and writes to the target file until the file is copied. The source file and the target file are then closed.

A slightly more interesting example is the Win32 function DeleteFile. If you're not a file system dev, you're probably thinking "How complicated can delete be? You just… delete the file, right?" Well, not in Windows, no. The native Windows API doesn't actually have a specific delete file operation. Rather, the intention to delete a file is indicated by issuing a Set Information operation (ZwSetInformationFile) to set the file's Disposition (FILE_DISPOSITION_INFORMATION). This allows the caller to specify whether he wants the file to be deleted when its closed. However, it's important to note that the file is not actually deleted until the last handle to the file has been closed. This means that an application can open a file and call the Win32 **DeleteFile** API, but this doesn't actually delete the file. It just set the file's Disposition of the file to be deleted on close. Even when the app subsequently closes the file, this still doesn't necessarily mean that the file will actually be deleted. Consider what could happen if another application has the file open when our example app opens it. That application can also set the file's Disposition at any time. If after our example app has set the file's Disposition to be delete on close, the other app sets the file's Disposition so that it's no longer marked for delete on close, the file will not be deleted. And yes… it happens.

Virtual Memory System Integration

Another source of complexity for file system Minifilters stems from the close working relationship between file systems, the Windows Cache Manager, and the Windows Memory Manager. We all know that when we call ReadFile, data from the specified data buffer is read from the file indicated by the file handle. Most of us probably also know, at least in a general way, that there's typically some caching involved. That is, every call we make to **ReadFile** doesn't always necessarily result in that data being read directly from the media.

Interestingly, except when a file is explicitly opened with no caching, file systems typically do very little to process an application's call to **ReadFile**, beyond calling the Cache Manager to deal with it. At the file system's request, the Cache Manager copies the data from the cache area associated with that file's open instance into the application's data buffer. This is illustrated in *Figure 2*.

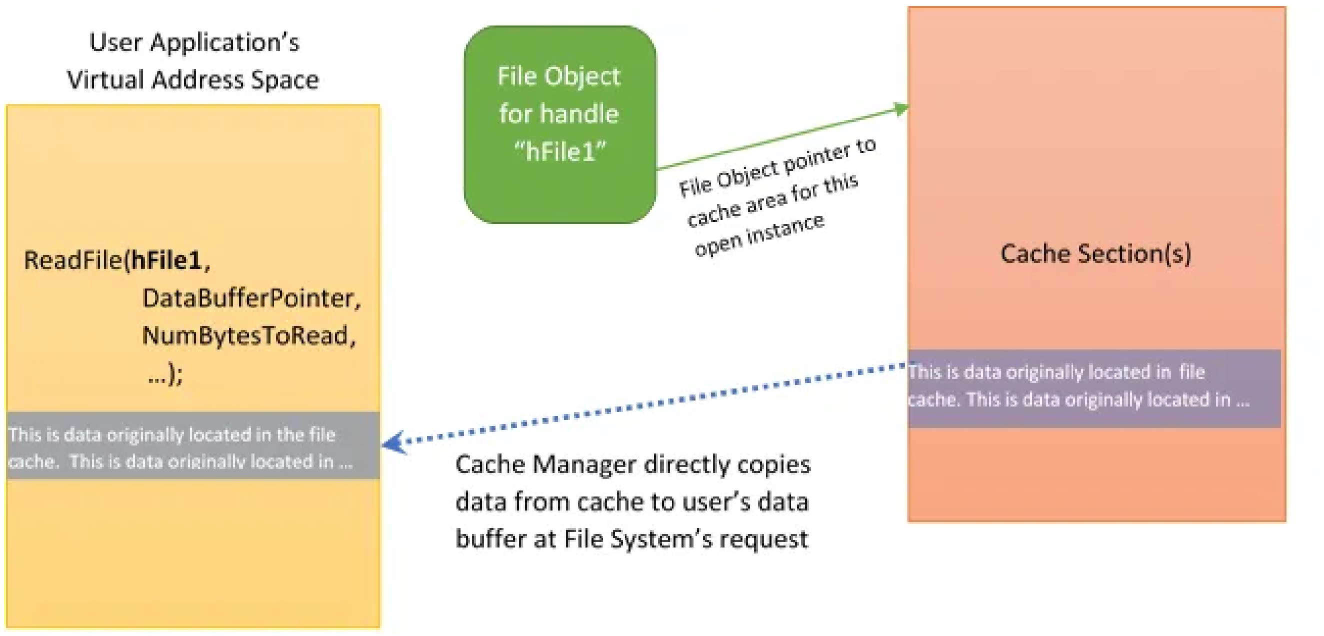
User Application's Virtual Address Space
File Object for handle "hFile1"
File Object pointer to cache area for this open instance
Cache Section(s)
ReadFile(hFile1, DataBufferPointer, NumBytesToRead, ...);
This is data originally located in the file cache. This is data originally located in ...
This is data originally located in file cache. This is data originally located in ...
Cache Manager directly copies data from cache to user's data buffer at File System's request

Figure 2 – File System Uses Cache Manager to Process a Read Operation

In *Figure 2*, the file system has called the Windows Cache Manager to process an application's **ReadFile** request. The file system identifies the open instance of the file being accessed using the file handle provided by the application in its **ReadFile** call. The file handle "hFile1" is a handle to the indicated File Object which the Cache Manager uses to locate the sections of cached data that are part of the file system's cache.

Interestingly, the data stored in cache is read into memory from disk by the Windows Memory Manager via page fault handling. That paging read operation will also pass through the file system.

File system Minifilters can of course be involved in the processing of read and write operations, both originating from applications and originating from the Memory Manager.

Keeping It Simple: Standard Minifilters

The most common type of file system Minifilter monitors, and perhaps tracks or records, various operations performed at the file system level. Some Minifilters, such as antivirus scanners, might even approve or disapprove certain operations. These filters do not, however, become involved in changing the view, or the size, of the data in the files they filter. We term such filters **Standard Minifilters**, because they represent the vast majority of the file system Minifilters that exist.

The primary challenges that developers of Standard Minifilters face are:

1. Properly understanding and dealing with native Windows I/O Subsystem semantics.

2. Properly preserving the behavior of all native file system operations, even the ones they don't necessarily understand or care about.

Of course, this is in addition to the basic challenges inherent in writing any Windows kernel-mode driver.

Overcoming the first of these challenges, that is properly understanding and dealing with native Windows I/O Subsystem semantics, simply requires time and experience. There are numerous Microsoft-provided examples that can be used as a starting point. And the documentation on writing basic file system Minifilters is surprisingly good – Assuming you take the time to read it and understand it. There are also good online resources, like our NTFSD list, that can help by answering questions you might have along the way. And, I should also mention, we'll be teaching a seminar in how to write Standard Minifilters starting in January 2018.

The complexity of item B, above, depends greatly on the type of Minifilter you write. The simpler you keep your operations, and the more limited you keep your filtering, the easier things will be. For example, it's usually a mistake to attempt to filter I/O operations that your filter doesn't specifically care about. The HLK tests for file system Minifilters will be very helpful to you in assessing your success in preserving the behavior of the file systems you filter. Be sure you run them!

Thus, if you need to write a Standard Minifilter – that is, one that monitors file system operations – the challenges you face will be reasonable.

Exponentially Harder: Isolation Minifilters

A much less common type of file system Minifilter is the **Isolation Minifilter**. Isolation filters separate (or "isolate") the **view** of a file's data from the **actual underlying data** stored by the file system. Writing this type of Minifilter is usually as complex as writing a standard Windows file system, because it involves direct and close interaction between your Minifilter, the Windows Cache Manager, and the Windows Memory Manager. In fact, some experienced devs consider Isolation Minifilter even more difficult than file system development, because when you're writing an Isolation Minifilter you effectively have to "fit" the implementation of a Windows file system into the API provided by Filter Manager. Thus, in addition to the challenges faced by the developers of Standard Minifilters, developers of Isolation Minifilters deal with numerous significantly more complex issues.

To illustrate what we mean by "separating the view of a file's data form the actual underlying data", let's consider an Isolation Minifilter that implements on-access transparent encryption/decryption. An application opens, and calls **ReadFile** for, a file that is stored on disk in encrypted form. The encryption/decryption Minifilter's job in this case is to transparently decrypt the data that is presented to the application. The general scheme for supporting this type of activity is shown in *Figure 3*.

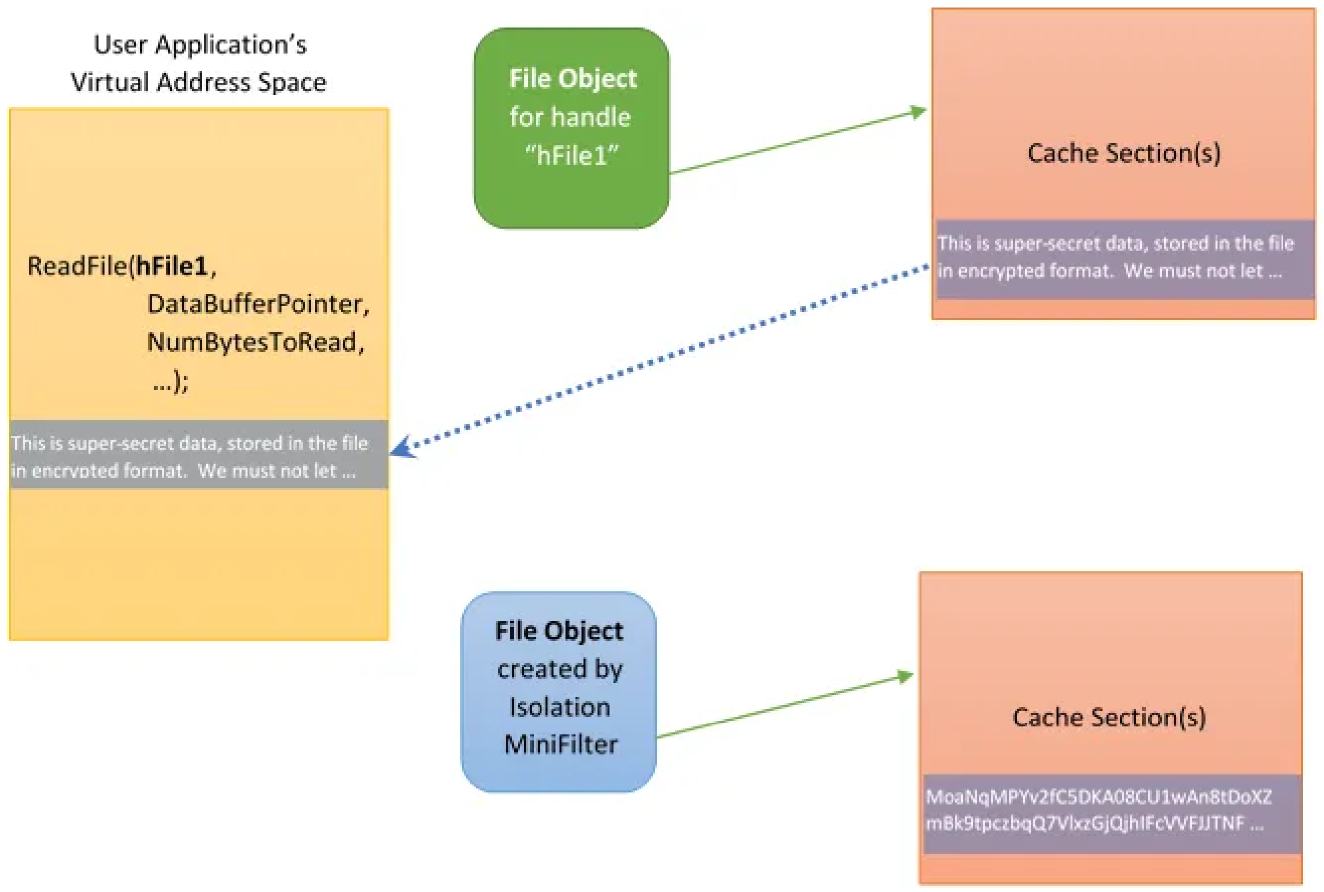
User Application's Virtual Address Space
ReadFile(hFile1, DataBufferPointer, NumBytesToRead, ...);
This is super-secret data, stored in the file in encrypted format. We must not let ...
File Object for handle "hFile1"
Cache Section(s)
This is super-secret data, stored in the file in encrypted format. We must not let ...
File Object created by Isolation MiniFilter
Cache Section(s)
MoaNqMPYv2fC5DKA08CU1wAn8tDoXZ mBk9tpczbqQ7VlxzGjQjhIFcVVFJJTNF ...

Figure 3 – Isolation Minifilter

Figure 3 shows two File Objects, each associated with the same file. The upper File Object (in green) is the open instance associated with the application. This File Object represents the application's view of file. Note that the cache section referenced by the File Object contains decrypted data. The data was placed into cache by the Isolation Filter, working in cooperation with the Cache Manager.

The lower File Object (in blue) is created by the Isolation Minifilter, and represents the Minifilter's (and the underlying file system's) view of the file. Note that the data in the cache section for this File Object is encrypted. The Isolation Minifilter uses the lower File Object to interact with the underlying file system.

Thus, when the user application calls **ReadFile**, the Isolation Minifilter will receive that request (as part of its read PreOperation processing, discussed previously) and interact with the Cache Manager to fill the cache with data. In processing the subsequent paging read operation (sent by the Memory Manager in fulfillment of the Cache Manager's attempt to fetch data for the cache) the Isolation Minifilter issues a **ReadFile** operation referencing the lower (blue) File Object. The underlying file system, working with the Cache Manager, fulfills this request and puts data into the indicated cache section.

Things get even more complex when you consider that most transparent data encryption systems utilize some sort of metadata header (to hold keys or access information). This header is often stored within the file itself. Thus, the file offset used for read operations by the application can be different from the file offset that the Isolation Minifilter needs to use when accessing the file as stored on disk.

But there's usually even more complexity than we've discussed so far. Let's say the encryption/decryption Minifilter determines on a per-application basis whether decrypted data or encrypted data should be provided to the application when it does a read. For example, the Isolation Minifilter might automatically decrypt data from an encrypted document when accessed by Microsoft Word. However, when that same encrypted document is accessed from a backup application, the Isolation Minifilter could provide the raw, encrypted, contents of the file. What's even more interesting is a well-designed Isolation Minifilter could allow both views, the decrypted view of the file's contents and the encrypted view of the file's contents, to different applications reading the file simultaneously. And what happens if one of those applications writes data to the file, while the other has a different view of the file open? Let's just say that things get "interesting," fast.

Isolation Minifilters: Not Just Encryption

We should hasten to add that the Isolation Minifilter model is not only used for transparent data encryption and decryption systems. The Isolation Minifilter model is used any time:

- There's a difference between an application's view of the data and the underlying file stored on the file system.
- There's a difference between the applications view of the data and the underlying location of the file data.
- Different application instances need to having unqiue views of the same underlying data stored on the file system.

For example, consider a change tracking and versioning Isolation Minifilter. One copy of Word might be actively editing the latest version of a file, while a different instance of Word (perhaps being run by a different user) might be actively editing an older version of the file.

Or consider a file that's stored in the cloud and only pulled down bit by bit, based on how the file is accessed. The file might appear to applications to be entirely resident on the local file system. However, an Isolation Filter could be accessing and back-filling data from remote storage as required.

Where Does That Leave Us?

The Windows MiniFilter model is flexible, and extremely powerful. With enough time and experience, most developers will be able to write Standard Minifilters. These filters monitor and/or track file operations, and potentially authorize access to files stored in the underlying file systems. The primary challenges inherent in developing Standard Minifilters are understanding native Windows file system semantics, and transparently passing-through to applications all the functionality the underlying file system offers. Both of these challenges are tractable, and can be accomplished without a vast amount of Windows file system expertise.

In contrast to Standard Minifilters, there are Isolation Minifilters. These Minifilters separate the view an application receives of a file, from the actual underlying data of that same file. Developing an Isolation Minifilter is more like developing a full Windows file system than developing a Standard Minifilter, because it requires close interaction with the Windows Cache Manager and Memory Manager. As such, the development of an Isolation Minifilter is not likely to be a task that can be successfully undertaken by most developers who do not also seek to become Windows file system experts.