Identifying Antivirus Software by enumerating Minifilter String Names

vx-underground collection // by smelly__vx

Introduction:

Lately I've been diving deep into the internals of Antivirus software. I've got quite a bit of information I intend to write about and share. However, for the time being, I have decided to release a small writeup on detecting AVs from their Minifilter port string. I personally have not seen anyone document this technique - which is neat - considering it is so easy to achieve. Unfortunately though, as always, this technique does have some limitations and this should be stated clearly before continuing.

1. This code requires administrative permissions
2. This code returns the AV minifilter name string. This means you'll need to compile a list of known AV minifilter string names. This is easily achievable - but it needs to be hardcoded into your code. Lame. If you'd like a list of these string identifiers check out the MSDN allocated filter altitudes.
3. For the curious: the APIs in this code segment forward to NTDLL. Converting this to a syscall-only proof-of-concept would be a pain in the ass. But it is still feasible.

Anyway, now that is out the way, I hope you enjoy this small writeup. It is a cute little trick.

-smelly

Minifilters, a tl;dr

Although a Red Teamer, or a VXer, may unhook from the user-mode hooks present (i.e. Hells Gate) put into place by the AV system, there still remains the last major obstacle: the minifilter. This minifilter system is typically used to detect ransomware operations, and can be used to perform static analysis on a binary as it is initially executed, so and so forth. It truly is an AVs biggest strength.

Minifilters are a de facto standard AV tech - in Devisha Rochlanis Antivirus Artifacts series - you'll see these from every major AV (as well as EDRs!). They essentially allow a god-mode view of the NTFS filesystem. They allow AVs to perform preoperative and postoperative routines on binaries e.g. perform an action on a binary before its execution and while it is executing.

As a final note, you can read all about minifilters and their uses in this amazing introduction by OSR.These drivers have much more uses besides AVs. If you're curious on how AVs implement minifilters, in a more technical sense, MSDN also provides a proof-of-concept AV you can download and review. It is worth a review. You can download the AV proof-of-concept from their File System Driver Samples.

User-mode Minifilter APIs

Windows provides some user mode APIs to communicate with Minifilters. This is a necessity so user-mode AVs, for example, can communicate with the kernel-mode minifilter. Communications can take 2 forms:

1. FilterConnectCommunicationPort
2. DeviceIoControl

Some AVs may utilize the designed Fltuser.h and relevant APIs to communicate with the kernel mode minifilter. However, some AVs take an alternative approach and communicate directly using DeviceIoControl. The reasoning for this is a bit beyond the scope of this paper as I do not intend to explain pros and cons of minifilter communication methods (very little difference). However, the point remains that Microsoft provides such APIs and some form of flexibility from the user-mode. The only exception here being that both FilterConnectCommunicationPort and DeviceIoControl both require your process to run in high-integrity (administrator). This cannot be achieved otherwise.

Enumerating Minifilter Name Strings

Unlike the traditional meaning of a ‘port’ which usually we think of as an unsigned 16bit integer (1 - 65535), Minifilter ports on the Windows OS use WCHAR strings as unique identifiers. This can be seen in the FilterConnectCommunicationPort parameter lpPortName which is:

Pointer to a NULL-terminated wide-character string containing the fully qualified name of the communication server port (for example, L"\MyFilterPort").

Hence, each minifilter will have a port name we can find using FilterFindFirst, FilterFindNext, and FilterFindClose. This is similar functionality to the APIs used to enumerate files in a directory using FindFileFirst, FindNextFile, and FindClose.

Final remarks

After I demonstrate the code please review the subsequent section on the internals of these APIs. They’re interesting and if you’re a researcher or Red Teamer, you may end up falling down a deeper rabbit hole from this paper. It will expose a relatively unknown part of Windows. Do not hesitate to contact me on whatever you find. This is really cool stuff.

The code

1. Before any APIs are invoked we must allocate a FILTER_FULL_INFORMATION structure. When you invoke FilterFindFirst the parameter lpBytesReturned returns a DWORD that specifies the number of bytes returned to the FILTER_FULL_INFORMATION buffer. Traditionally we'd first invoke FilterFindFirst with a specification of 0 in the dwBufferSize member to determine how much memory we need to allocate. However, in my preliminary testing I discovered a majority of this allocation is due to the string name. Hence I allocate MAX_PATH (260 bytes). This will be more than enough for any minifilter name string.

```
DWORD dwError = ERROR_SUCCESS, dwBufferSize = 0;
HRESULT Result;
HANDLE Filter = INVALID_HANDLE_VALUE, ProcessHeap = GetProcessHeap();
PFILTER_FULL_INFORMATION FilterInformation = NULL;

FilterInformation = (PFILTER_FULL_INFORMATION)HeapAlloc(ProcessHeap, HEAP_ZERO_MEMORY, MAX_PATH);
if (FilterInformation == NULL)
        goto FAILURE;

Result = FilterFindFirst(FilterFullInformation, FilterInformation, MAX_PATH, &dwBufferSize, &Filter);
if (Result != S_OK || Filter == INVALID_HANDLE_VALUE)
{
        SetLastError(Win32FromHResult(Result));
        goto FAILURE;
}

_putws(FilterInformation->FilterNameBuffer);
```

In our first call to FilterFindFirst we print the *FilterNameBuffer* from the *FILTER_FULL_INFORMATION* buffer (*FilterInformation variable*). Note the last variable, *HANDLE Filter*, is returned from FilterFindFirst. This will be used subsequently to continue filter enumeration.

[Continued below]

2. This code section is very simple. Once we have successfully gotten a search handle on the filter drivers we simply run a for loop infinitely. If in the event FilterFindNext returns ERROR_NO_MORE_ITEMS (259, 0x103) we have successfully enumerated all minifilter strings. Otherwise, we continue.by printing the *FilterNameBuffer* again. Once we break from the loop if our search handle is still valid we invoke FilterFindClose to close the search handle. Additionally, as always, we free the heap.

```
for (;;)
{
	ZeroMemory(FilterInformation, dwBufferSize);
	Result = FilterFindNext(Filter, FilterFullInformation, FilterInformation,
				MAX_PATH, &dwBufferSize);

	if (Result != S_OK || Filter == INVALID_HANDLE_VALUE)
	{
		if (Win32FromHResult(Result) == ERROR_NO_MORE_ITEMS)
			break;

		SetLastError(Win32FromHResult(Result));
		goto FAILURE;
	}

	_putws(FilterInformation->FilterNameBuffer);
}


if (Filter)
	FilterFindClose(Filter);

if (FilterInformation)
	HeapFree(ProcessHeap, HEAP_ZERO_MEMORY, FilterInformation);

return ERROR_SUCCESS;
```

A small nuance about this API is it does not return DWORD error codes rather it returns HRESULT error codes. To compensate for this I wrote a Win32FromHResult function. This is extremely common and can be found all over the web. It is nothing major.

```
DWORD Win32FromHResult(HRESULT Result)
{
	if ((Result & 0xFFFF0000) == MAKE_HRESULT(SEVERITY_ERROR, FACILITY_WIN32, 0))
		return HRESULT_CODE(Result);

	if (Result == S_OK)
		return ERROR_SUCCESS;

	return ERROR_CAN_NOT_COMPLETE;
}
```

The output

```
bindflt
WdFilter
storqosflt
wcifs
CldFlt
FileCrypt
luafv
npsvctrig
Wof
FileInfo
```

These are all minifilter port strings found on my machine. You can review them on MSDNs Minifilter Altitude listing. My friend who ran this proof-of-concept with AVAST installed got the following results:

aswSP, aswMonFlt, and aswSnx being the Minifilter strings.

The rabbit hole

The function FilterConnectCommunicationPort invokes NtCreateFile on the symbolic link *\\Global??\\FltMgrMsg* which points to *\FileSystem\Filters\FltMgrMsg.* The string name specified in FilterConnectCommunicationPort is passed in the Extended Attributes parameter in NtCreateFile.

Here is a snippet from IDA:

```
RtlInitUnicodeString(&DestinationString, lpcwpPortName);
v26 = DestinationString.Buffer;
v25[0] = DestinationString.Length;
v25[1] = DestinationString.MaximumLength;
*(_QWORD *)&EaBuffer[v13 + 9] = &DestinationString;
*(_QWORD *)&EaBuffer[v13 + 17] = v25;
*(_WORD *)&EaBuffer[v13 + 25] = wSizeOfContext;
if ( wSizeOfContext )
  memcpy_0(&EaBuffer[v13 + 33], Src, wSizeOfContext);
RtlInitUnicodeString(&v23, L"\\Global??\\FltMgrMsg");
ObjectAttributes.Length = 48;
v14 = 64;
ObjectAttributes.Attributes = 64;
ObjectAttributes.ObjectName = &v23;
ObjectAttributes.RootDirectory = 0i64;
*(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0i64;
if ( lpSecurityAttributes )
{
  v15 = !lpSecurityAttributes->bInheritHandle;
  ObjectAttributes.SecurityDescriptor = lpSecurityAttributes->lpSecurityDescriptor;
  if ( !v15 )
    v14 = 66;
  ObjectAttributes.Attributes = v14;
}
v16 = NtCreateFile(
        &FileHandle,
        0x100003u,
        &ObjectAttributes,
        &IoStatusBlock,
        0i64,
        0,
        0,
        3u,
        32 * (v8 & 1),
        EaBuffer,
        (unsigned __int16)(wSizeOfContext + 24) + 19);
```

Meanwhile, FilterFindFirst and FilterFindNext rely heavily on NTDLL functionality and invoke internal functions *FilterpDeviceIoContro*l which makes heavy usage of other low-level APIs such as NtDeviceIoControlFile and NtFsControlFile.

Here is a snippet from IDA:

```
if ( IoStatusBlock )
 {
   IoStatusBlock->Pointer = (PVOID)259;
   v13 = IoStatusBlock[1].Information;
   v14 = IoStatusBlock;
   if ( v12 == 9 )
   {
     if ( (v13 & 1) != 0 )
       v14 = 0i64;
     v15 = NtFsControlFile(
             Handle,
             (HANDLE)v13,
             0i64,
             v14,
             IoStatusBlock,
             FsControlCode,
             InputBuffer,
             InputBufferLength,
             OutputBuffer,
             OutputBufferLength);
   }
   else
   {
     if ( (v13 & 1) != 0 )
       v14 = 0i64;
     v15 = NtDeviceIoControlFile(
             Handle,
             (HANDLE)v13,
             0i64,
             v14,
             IoStatusBlock,
             FsControlCode,
             InputBuffer,
             InputBufferLength,
             OutputBuffer,
             OutputBufferLength);
   }
```

This exposes a lot of really interesting internals which I encourage you to explore. Adrien Chevalier of amossys.fr has done some research on this matter and unveiled more to this. You can check out his blog entry on his research here.

Thanks for reading. More to come soon.