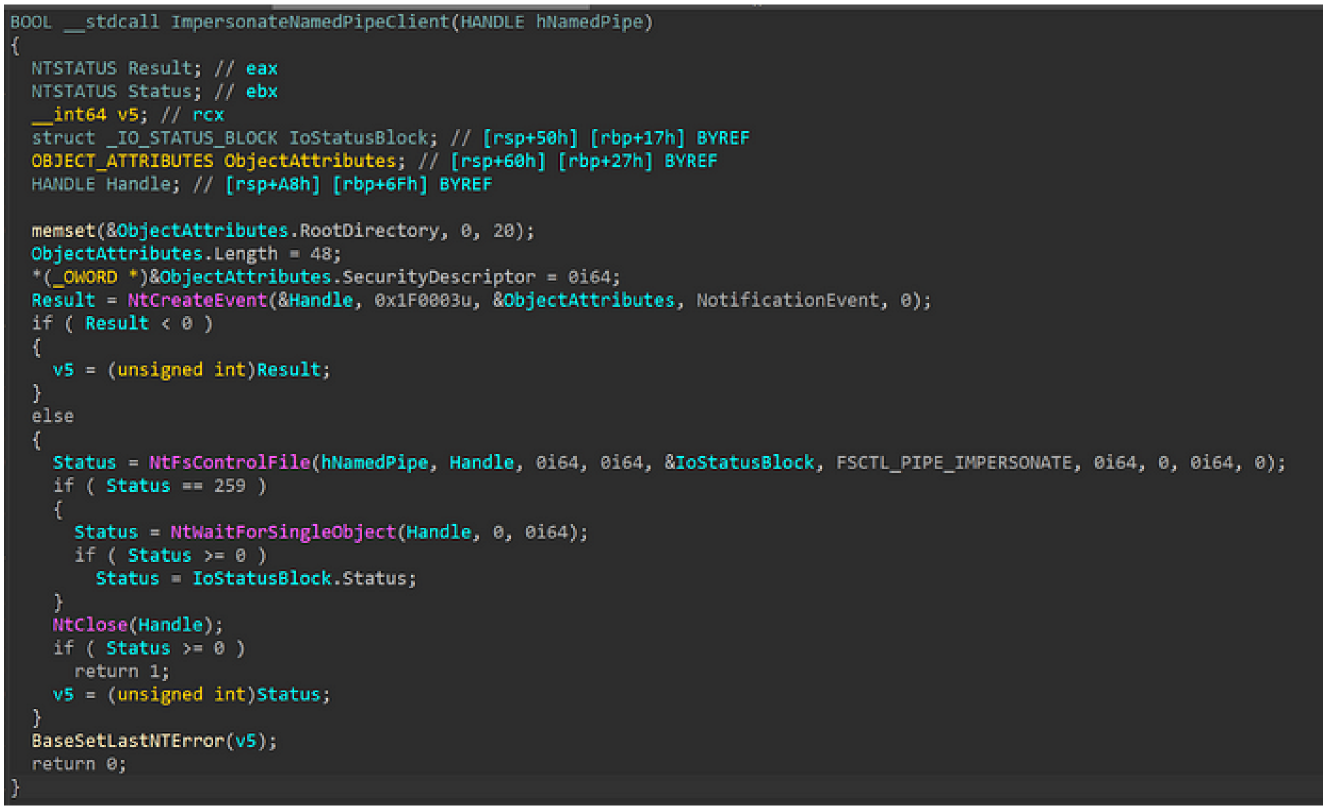
Exploring Impersonation through the Named Pipe Filesystem Driver

posts.specterops.io/exploring-impersonation-through-the-named-pipe-filesystem-driver-15f324dfbaf2

May 3, 2023

Introduction

Impersonation happens often natively in Windows, however, adversaries also use it to run code in the context of another user. Recently I was researching named pipe impersonation which naturally led me digging into the Win32 API ImpersonateNamedPipeClient. I had never really dug into how ImpersonateNamedPipeClient worked under the hood, so I wanted to do so. During analysis, I saw that a call to NtFsControlFile was made:



```
BOOL __stdcall ImpersonateNamedPipeClient(HANDLE hNamedPipe)
{
  NTSTATUS Result; // eax
  NTSTATUS Status; // ebx
  __int64 v5; // rcx
  struct _IO_STATUS_BLOCK IoStatusBlock; // [rsp+50h] [rbp+17h] BYREF
  OBJECT_ATTRIBUTES ObjectAttributes; // [rsp+60h] [rbp+27h] BYREF
  HANDLE Handle; // [rsp+A8h] [rbp+6Fh] BYREF

  memset(&ObjectAttributes.RootDirectory, 0, 20);
  ObjectAttributes.Length = 48;
  *(_OWORD *)&ObjectAttributes.SecurityDescriptor = 0i64;
  Result = NtCreateEvent(&Handle, 0x1F0003u, &ObjectAttributes, NotificationEvent, 0);
  if ( Result < 0 )
  {
    v5 = (unsigned int)Result;
  }
  else
  {
    Status = NtFsControlFile(hNamedPipe, Handle, 0i64, 0i64, &IoStatusBlock, FSCTL_PIPE_IMPERSONATE, 0i64, 0, 0i64, 0);
    if ( Status == 259 )
    {
      Status = NtWaitForSingleObject(Handle, 0, 0i64);
      if ( Status >= 0 )
        Status = IoStatusBlock.Status;
    }
    NtClose(Handle);
    if ( Status >= 0 )
      return 1;
    v5 = (unsigned int)Status;
  }
  BaseSetLastNTError(v5);
  return 0;
}
```

NtFsControlFile is a function that allows the caller to send a value (FSCTL_PIPE_IMPERSONATE (0x11001C) in the above decompilation), known as a file system control (FSCTL) code, to a file system driver. Upon initial analysis of this function I was reminded of another function — DeviceIoControl. DeviceIoControl serves a similar purpose in the sense that it allows someone to send an input/output control code (known as an IOCTL) to a driver. IOCTLs and FSCTL codes are the same thing, but FSCTLs are a type of IOCTL that are specific to file system drivers. I have encountered this function before so it provided some familiarity with the general architecture of drivers, control codes, and other related concepts, however; I have never interacted with file system drivers themselves.

This post covers file system drivers, specifically the named pipe driver (npfs.sys), as well as shows a proof of concept for calling NtFsControlFile directly to perform named pipe impersonation instead of calling the Win32 API, ImpersonateNamedPipeClient.

This won't be an indepth dive into device drivers, file system drivers, or minifilters. Instead I want to explain some concepts that I learned that I think will be relevant and important to understanding file system operations, specific to named pipes.

Internals

Microsoft exposes a set of APIs that allow applications to interact with drivers — DeviceIoControl, NtDeviceIoControlFile, and NtfsControlFile. Both functions communicate with different types of drivers but the general communication between the application and the driver are the same. Eventually both make a call to IofCallDriver. This function allows the caller to send an input/output request packet (IRP) to the specific driver. IofCallDriver takes in two parameters:

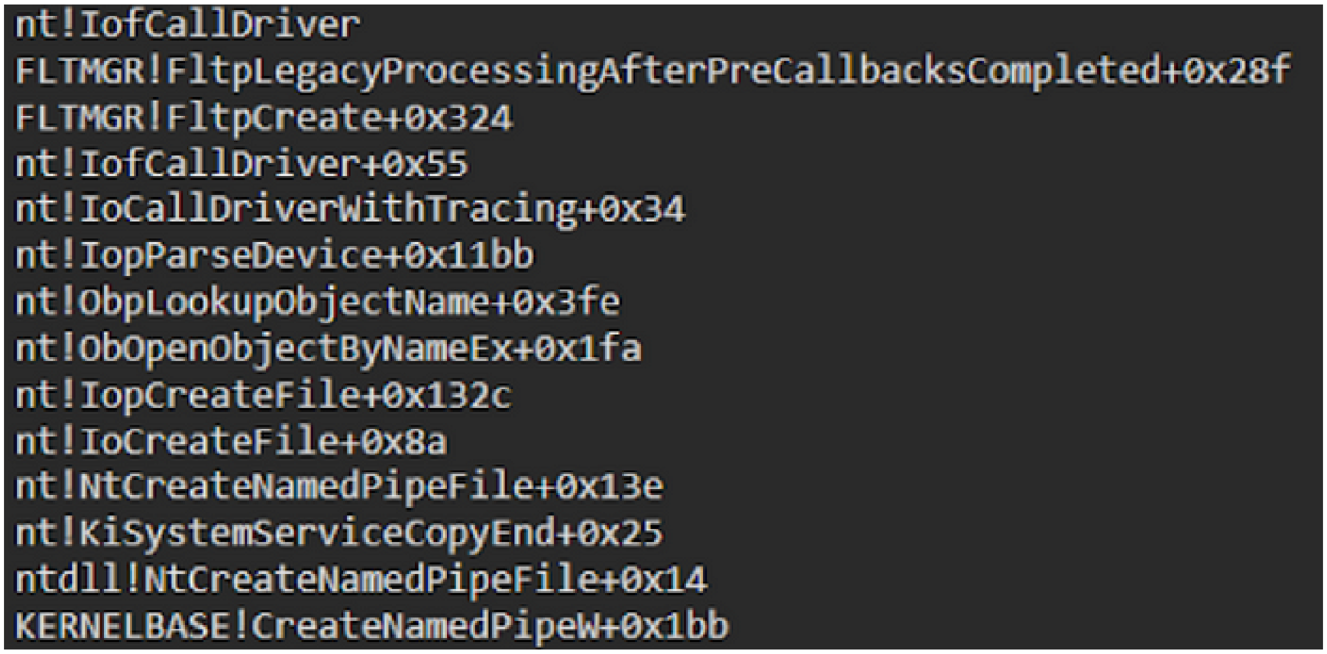
1. A pointer to the structure which is an object that acts as an interface with which the user-mode caller can communicate. Device objects are linked to a driver which will execute the request.
2. A pointer to an structure which packages up the call information about the call to the driver.

The DEVICE_OBJECT is going to hold where the call is going to whereas the IRP is going to hold the relevant information about the request for the driver. An IRP is a kernel-based dynamic structure. This structure holds the I/O stack which is backed by the IO_STACK_LOCATION structure that holds information about the functions (known as major and minor functions) being invoked and the appropriate parameters. Major functions are represented as IRP_MJ_ which informs the driver of the operation it should execute. The name of the major function is well-documented in the WDM header with their name and their corresponding number.

Here is a small list of those:

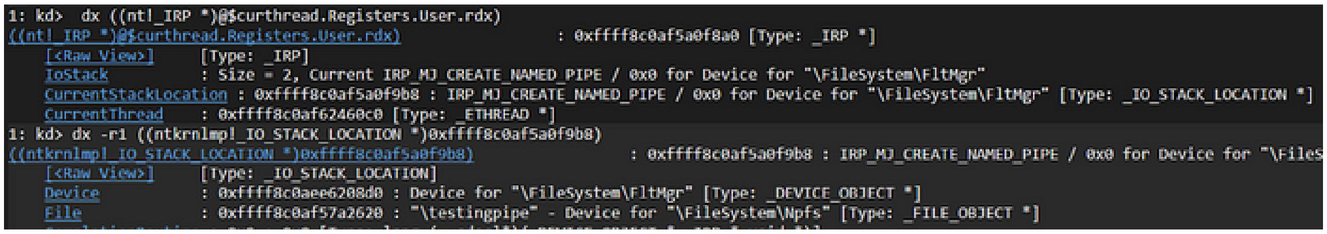
There might be some major functions that look familiar above, but since we are talking about named pipes, let's look at IRP_MJ_CREATE_NAMED_PIPE. This IRP is sent when CreateNamedPipe(A/W) is called. This Win32 API transitions into the kernel via the NtCreateNamedPipeFile syscall. NtCreateNamedPipeFile doesn't perform any file system operations but instead it receives the call from user-mode and forwards it to the appropriate driver via IofCallDriver call. IofCallDriver sends the request to the driver responsible for the named pipe creation, npfs.sys.

Call stack:



```
nt!IofCallDriver
FLTMGR!FltpLegacyProcessingAfterPreCallbacksCompleted+0x28f
FLTMGR!FltpCreate+0x324
nt!IofCallDriver+0x55
nt!IoCallDriverWithTracing+0x34
nt!IopParseDevice+0x11bb
nt!ObpLookupObjectName+0x3fe
nt!ObOpenObjectByNameEx+0x1fa
nt!IopCreateFile+0x132c
nt!IoCreateFile+0x8a
nt!NtCreateNamedPipeFile+0x13e
nt!KiSystemServiceCopyEnd+0x25
ntdll!NtCreateNamedPipeFile+0x14
KERNELBASE!CreateNamedPipeW+0x1bb
```

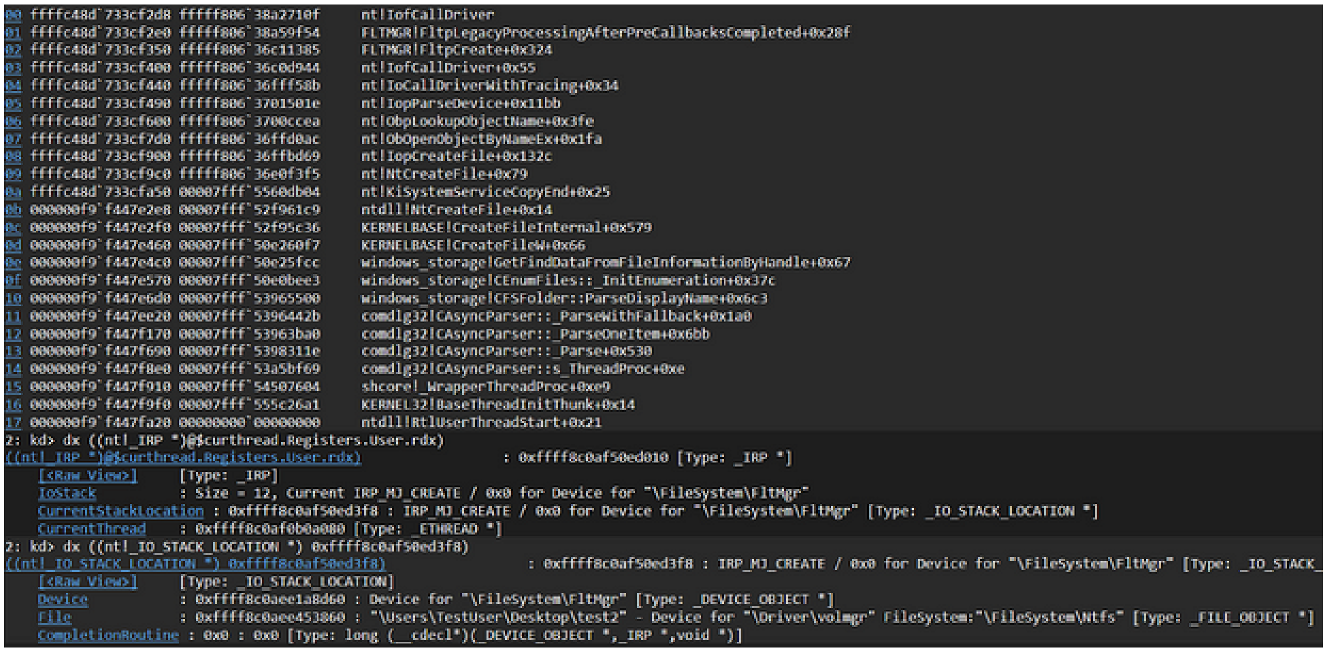
IRP:



```
1: kd>  dx ((nt!_IRP *)@$curthread.Registers.User.rdx)
((nt!_IRP *)@$curthread.Registers.User.rdx)                 : 0xffff8c0af5a0f8a0 [Type: _IRP *]
    [<Raw View>]     [Type: _IRP]
    IoStack          : Size = 2, Current IRP_MJ_CREATE_NAMED_PIPE / 0x0 for Device for "\FileSystem\FltMgr"
    CurrentStackLocation : 0xffff8c0af5a0f9b8 : IRP_MJ_CREATE_NAMED_PIPE / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION *]
    CurrentThread    : 0xffff8c0af62460c0 [Type: _ETHREAD *]
1: kd> dx -r1 ((ntkrnlmp!_IO_STACK_LOCATION *)0xffff8c0af5a0f9b8)
((ntkrnlmp!_IO_STACK_LOCATION *)0xffff8c0af5a0f9b8)                 : 0xffff8c0af5a0f9b8 : IRP_MJ_CREATE_NAMED_PIPE / 0x0 for Device for "\FileS
    [<Raw View>]     [Type: _IO_STACK_LOCATION]
    Device           : 0xffff8c0aee6208d0 : Device for "\FileSystem\FltMgr" [Type: _DEVICE_OBJECT *]
    File             : 0xffff8c0af57a2620 : "\testingpipe" - Device for "\FileSystem\Npfs" [Type: _FILE_OBJECT *]
```

Note: There is even more that could be exposed in WinDbg like the parameters to the IRP_MJ_CREATE_NAMED_PIPE call, the security context, etc. This is all stored in the IO_STACK_LOCATION.

Another good example of this is when CreateFile(A/W) is called, this will transition into the kernel via NtCreateFile and will eventually make a call to IofCallDriver to communicate with ntfs.sys (or another file system driver) to create the file object:



```
00 ffffc48d`733cf2d8 fffff806`38a2710f     nt!IofCallDriver
01 ffffc48d`733cf2e0 fffff806`38a59f54     FLTMGR!FltpLegacyProcessingAfterPreCallbacksCompleted+0x28f
02 ffffc48d`733cf350 fffff806`36c11385     FLTMGR!FltpCreate+0x324
03 ffffc48d`733cf400 fffff806`36c0d944     nt!IofCallDriver+0x55
04 ffffc48d`733cf440 fffff806`36fff58b     nt!IoCallDriverWithTracing+0x34
05 ffffc48d`733cf490 fffff806`3701501e     nt!IopParseDevice+0x11bb
06 ffffc48d`733cf600 fffff806`3700ccea     nt!ObpLookupObjectName+0x3fe
07 ffffc48d`733cf7d0 fffff806`36ffd0ac     nt!ObOpenObjectByNameEx+0x1fa
08 ffffc48d`733cf900 fffff806`36ffbd69     nt!IopCreateFile+0x132c
09 ffffc48d`733cf9c0 fffff806`36e0f3f5     nt!NtCreateFile+0x79
0a ffffc48d`733cfa50 00007fff`5560db04     nt!KiSystemServiceCopyEnd+0x25
0b 000000f9`f447e2e8 00007fff`52f961c9     ntdll!NtCreateFile+0x14
0c 000000f9`f447e2f0 00007fff`52f95c36     KERNELBASE!CreateFileInternal+0x579
0d 000000f9`f447e460 00007fff`50e260f7     KERNELBASE!CreateFileW+0x66
0e 000000f9`f447e4c0 00007fff`50e25fcc     windows_storage!GetFindDataFromFileInformationByHandle+0x67
0f 000000f9`f447e570 00007fff`50e0bee3     windows_storage!CEnumFiles::_InitEnumeration+0x37c
10 000000f9`f447e6d0 00007fff`53965500     windows_storage!CFSFolder::ParseDisplayName+0x6c3
11 000000f9`f447ee20 00007fff`5396442b     comdlg32!CAsyncParser::_ParseWithFallback+0x1a0
12 000000f9`f447f170 00007fff`53963ba0     comdlg32!CAsyncParser::_ParseOneItem+0x6bb
13 000000f9`f447f690 00007fff`5398311e     comdlg32!CAsyncParser::_Parse+0x530
14 000000f9`f447f8e0 00007fff`53a5bf69     comdlg32!CAsyncParser::s_ThreadProc+0xe
15 000000f9`f447f910 00007fff`54507604     shcore!_WrapperThreadProc+0xe9
16 000000f9`f447f9f0 00007fff`555c26a1     KERNEL32!BaseThreadInitThunk+0x14
17 000000f9`f447fa20 00000000`00000000     ntdll!RtlUserThreadStart+0x21
2: kd> dx ((nt!_IRP *)@$curthread.Registers.User.rdx)
((nt!_IRP *)@$curthread.Registers.User.rdx)                 : 0xffff8c0af50ed010 [Type: _IRP *]
    [<Raw View>]     [Type: _IRP]
    IoStack          : Size = 12, Current IRP_MJ_CREATE / 0x0 for Device for "\FileSystem\FltMgr"
    CurrentStackLocation : 0xffff8c0af50ed3f8 : IRP_MJ_CREATE / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION *]
    CurrentThread    : 0xffff8c0af0b0a080 [Type: _ETHREAD *]
2: kd> dx ((nt!_IO_STACK_LOCATION *) 0xffff8c0af50ed3f8)
((nt!_IO_STACK_LOCATION *) 0xffff8c0af50ed3f8)                 : 0xffff8c0af50ed3f8 : IRP_MJ_CREATE / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_
    [<Raw View>]     [Type: _IO_STACK_LOCATION]
    Device           : 0xffff8c0aee1a8d60 : Device for "\FileSystem\FltMgr" [Type: _DEVICE_OBJECT *]
    File             : 0xffff8c0aee453860 : "\Users\TestUser\Desktop\test2" - Device for "\Driver\volmgr" FileSystem:"\FileSystem\Ntfs" [Type: _FILE_OBJECT *]
    CompletionRoutine : 0x0 : 0x0 [Type: long (__cdecl*)(_DEVICE_OBJECT *,_IRP *,void *)]
```

Chapter 11 in the 2nd Part of the Windows Internals book breaks this down well if you are interested in learning more.

Device I/O Control Functions

Earlier I mentioned DeviceIoControl and NtfsControlFile. These are special functions because they relate to unique IRP major functions. Here are the IRP major functions they point to:

- DeviceIoControl — (0xe)
- NtfsControlFile — (0xd)

NtfsControlFile and DeviceIoControl look very similar in functionality but used differently in practice. Simply put, DeviceIoControl is for normal device drivers whereas NtfsControlFile is for file system drivers. DeviceIoControl requires a handle to the device which holds a driver object. NtfsControlFile passes in a handle to a FILE_OBJECT because the action is being executed on that object and later will pass through the filter manager (FltMgr) which will extract the device type and know which file system driver to pass it to.

Since we are primarily talking about file system drivers in this post we will continue using functions/terminology specific for file system drivers. However; it is good to note the majority of the concepts are the same.

When a file system driver wants to expose any functionality, it'll create an internal function implementing that functionality and register it as a major function handler. As an example, below we can see a number of registered major functions within the NpFs driver object:

When a user-mode application wants to execute that functionality, they will call the NtfsControlFile function and pass in the control code which will eventually go to the appropriate file system driver where it will execute its IRP_MJ_FILE_SYSTEM_CONTROL function and in turn execute the internal function associated with the FSCTL code.

We will see an example of this below when ImpersonateNamedPipeClient is called and again in the code I provide where I call NtfsControlFile directly.

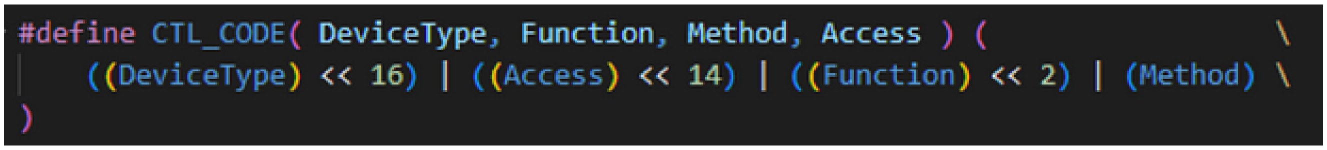
ImpersonateNamedPipeClient

ImpersonateNamedPipeClient is a Win32 API that allows a named pipe server to impersonate the token of client processes connecting to the server's named pipe. This is different than other token impersonation techniques as it requires something or someone else to connect to you before you can steal the token, whereas other capabilities (ImpersonateLoggedOnUser, CreateProcessWithToken, CreateProcessWithLogon, etc) allow for the impersonation of a token by targeting a process, usually ones that are running in a higher integrity level.

Examples in PowerShell can be found in following Atomic Test Harnesses:

NtfsControlFile

As previously mentioned, ImpersonateNamedPipeClient makes a call to NtfsControlFile where it passes in the FSCTL code 0x11001C. Control codes are defined by the CTL_CODE macro which can be found in the ntfis.h:



```
#define CTL_CODE( DeviceType, Function, Method, Access ) (           \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)
```

This can be confusing, but let's manually parse this out. The binary format of 0x11001C is 0001000100000000000011100.

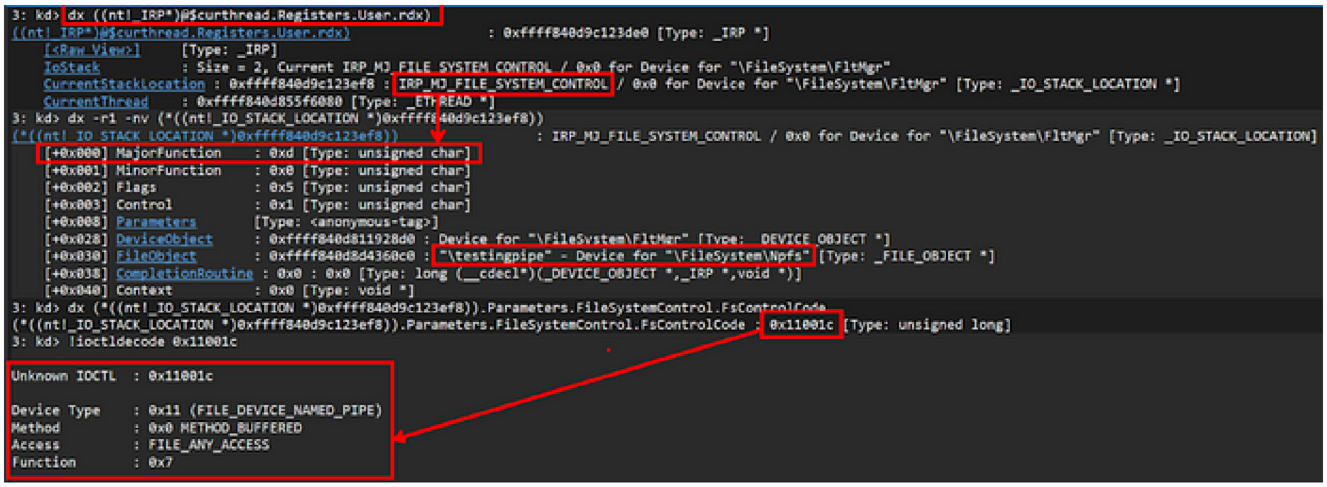
```
DeviceType (shift   bits   bits):    (FILE_DEVICE_NAMED_PIPE)Access (shift   bits
bits)  FILE_ANY_ACCESS (shift   bits   bits):   (  bits):  METHOD_BUFFERED
```

Note: An easier way to do this is via this IOCTL calculator or by using !ioctldecode in WinDbg (thank you Yarden for showing me this WinDbg way!).

Going back to the NTFS header, this value is documented as the FSCTL_PIPE_IMPERSONATE control code.

So why does this FSCTL matter? Remember that NtfsControlFile is eventually going to make a call into IofCallDriver, which passes in the IRP structure as previously mentioned. This IRP holds the relevant information about the call being sent to a device driver, one of which is the Major Function being invoked and the parameters that are packaged with that Major Function. For drivers when they expose a FSCTL code they are exposing this through the IRP_MJ_FILE_SYSTEM_CONTROL Major Function.

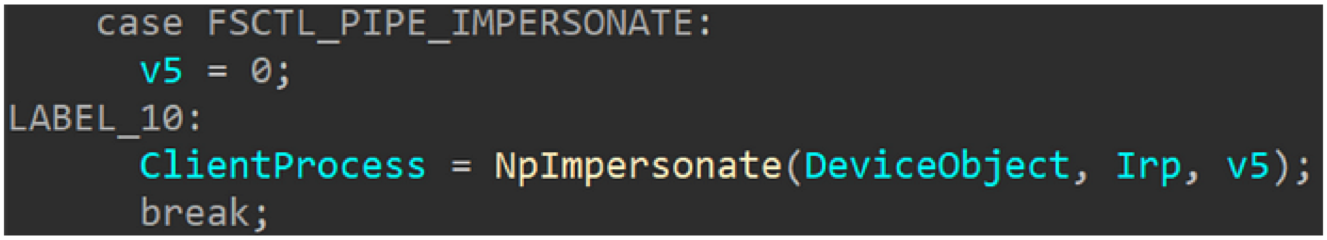
The driver then is going to call the IoGetCurrentIrpStackLocation to get the stack location in the IRP. This is important because under Parameters. FileSystemControl, you can see the parameters passed into this major function, one of which is the FSCTL code:


```
3: kd> dx ((nt!_IRP*)@$curthread.Registers.User.rdx)
((nt!_IRP*)@$curthread.Registers.User.rdx)                 : 0xffff840d9c123de0 [Type: _IRP *]
    [<Raw View>]     [Type: _IRP]
    IoStack          : Size = 2, Current IRP_MJ_FILE_SYSTEM_CONTROL / 0x0 for Device for "\FileSystem\FltMgr"
    CurrentStackLocation : 0xffff840d9c123ef8 : IRP_MJ_FILE_SYSTEM_CONTROL / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION *]
    CurrentThread    : 0xffff840d855f6080 [Type: _ETHREAD *]
3: kd> dx -r1 -nv (*((nt!_IO_STACK_LOCATION *)0xffff840d9c123ef8))
(*((nt!_IO_STACK_LOCATION *)0xffff840d9c123ef8))                 : IRP_MJ_FILE_SYSTEM_CONTROL / 0x0 for Device for "\FileSystem\FltMgr" [Type: _IO_STACK_LOCATION]
    [+0x000] MajorFunction    : 0xd [Type: unsigned char]
    [+0x001] MinorFunction    : 0x0 [Type: unsigned char]
    [+0x002] Flags            : 0x5 [Type: unsigned char]
    [+0x003] Control          : 0x1 [Type: unsigned char]
    [+0x008] Parameters       [Type: <anonymous-tag>]
    [+0x028] DeviceObject     : 0xffff840d811928d0 : Device for "\FileSystem\FltMgr" [Type: _DEVICE_OBJECT *]
    [+0x030] FileObject       : 0xffff840d8d4360c0 : "\testingpipe" - Device for "\FileSystem\Npfs" [Type: _FILE_OBJECT *]
    [+0x038] CompletionRoutine : 0x0 : 0x0 [Type: long (__cdecl*)(_DEVICE_OBJECT *,_IRP *,void *)]
    [+0x040] Context          : 0x0 [Type: void *]
3: kd> dx (*((nt!_IO_STACK_LOCATION *)0xffff840d9c123ef8)).Parameters.FileSystemControl.FsControlCode
(*((nt!_IO_STACK_LOCATION *)0xffff840d9c123ef8)).Parameters.FileSystemControl.FsControlCode : 0x11001c [Type: unsigned long]
3: kd> !ioctldecode 0x11001c

Unknown IOCTL  : 0x11001c

Device Type    : 0x11 (FILE_DEVICE_NAMED_PIPE)
Method         : 0x0 METHOD_BUFFERED
Access         : FILE_ANY_ACCESS
Function       : 0x7
```

This call leads to the NPFS FileSystemControl function NpCommonFileSystemControl. Internally, this function checks the FSCTL code and executes an internal function, in this case NpImpersonate:


```
    case FSCTL_PIPE_IMPERSONATE:
      v5 = 0;
LABEL_10:
      ClientProcess = NpImpersonate(DeviceObject, Irp, v5);
      break;
```

NpImpersonate then calls SeImpersonateClientEx which in turn calls PsImpersonateClient to impersonate the token of the thread that is connecting to the named pipe.

One thing to note is that there are quite a few FSCTLs exposed in npfs.sys that could be used for things like named pipe peeking, connecting to a named pipe, etc. Luckily it seems all of them are exposed in the NTFS header within the SDK.

Proof of Concept

When I see an opportunity to call either a lower level call instead of a Win32 API or the ability to send a control code to a driver, I try to take it. The proof of concept is simple as it changes ImpersonateLoggedOnUser out for NtfsControlFile in a named pipe server implementation. Once NtfsControlFile completes successfully, the user will have impersonated the client that connected to the “npfs” named pipe (i.e., \\pipe\npfs).

The code for this POC can be found on my GitHub at:
https://github.com/jsecurity101/RandomPOCs/tree/main/NtfsControlFile

Function Flow

I have provided the function call stack for those that are interested:

Conclusion

I think oftentimes the thought of interacting directly with a driver to perform an action is overlooked. As an industry this has been touched on but most recently talk has been around interfacing with technologies like RPC, COM, etc. It is important to note that through device objects user mode applications can interact with drivers as well and depending on the functionality that the driver supports those actions could be useful to the callee. Although this isn't a vulnerability or a vulnerable driver, this has been seen with other vulnerable drivers quite a bit.

This was also a fun project that gave me an opportunity to learn about file system drivers and how one may interact with them. I wanted to share this process as well as expose that when dealing with files there are ways one can interact directly with a file system driver to execute some functionality.

In the future I might do a write-up about the differences between file system drivers and file system filter drivers as well as an in-depth look into how someone can capture information about file system activity both from a driver perspective as well as ETW.

Remarks

A very big thank you to Yarden Shafir for answering some very critical questions, providing me with resources to learn this topic, as well as giving me inspiration to write this post.