sekoia #CYBERSECURITY

Rootkit analysis
Use case on HideDRV

Version 1.6

2016-10-27

Author: Paul Rascagnères

INDEX

DISCLAIMER

The purpose of this article is to provide a first good overview of the kernel mechanisms and how to handle a rootkit analysis.

This document is issued by Sekoia Cybersecurity and intends to share reverse engineering best practices in order to help organizations in their security jobs. If you are interested in reverse engineering, malware analysis, advanced rootkit analysis or Windows internal working, CERT Sekoia can assist you either for investigations or for trainings.

Contact us at contact@sekoia.fr

INTRODUCTION

ESET's experts published a complete and interesting white paper on the toolkit of a well-known APT actor identified as Sednit (aka APT28, Fancy Bear, Sofacy, STRONTIUM, Tsar Team…). The paper can be downloaded there. This group seems to be the author of several major media hacks such as the attack of the German parliament in December 2014 or the compromise of TV5Monde in April 2015. Thanks to a great collaboration with ESET, CERT Sekoia got access to the rootkit samples in order to perform its own investigation.

The name of the rootkit discovered by ESET is HIDEDRV. This name was chosen by the developer and is present in several comments in the driver file (FsFlt.sys). CERT Sekoia frequently deals with malware and rootkits analysis. Sometimes, several people ask us for tricks for kernel analysis and debugging. After a quick overview of the drivers, we found out that this sample was a perfect case study for beginners. Therefore, we decided to publish a deep analysis on this rootkit. People interested in discovering and practising Windows kernel analysis and rootkit debugging might like it. Indeed, this sample is wonderful for beginners:

- no packer;
- no obfuscation;
- no advanced or undocumented trick;
- really small (< 100 functions);
- all the classical features (such as registries hiding, files hiding, code injection from the kernel).

And there is the icing on the cake: the developers let a lot of comments for helping and guiding the analyst ;).

This article describes how to deal with rootkit analysis step by step: laboratory setup, Windows kernel architecture and API, Windows protection, Windows 10 64 bits… The purpose is to provide a tutorial of the "state of the art" of rootkit analysis on modern x64 Windows systems.

As usual we love feedbacks, so don't be afraid to contact us!

X64 ROOTKIT ANALYSIS

PREPARATION

This article was created based on the following configuration:

- Host: Windows 10 TH2 – x64
- Guest: Windows 10 TH2 – x64
- Hypervisor: Hyper-V
- Sample hash: `4bfe2216ee63657312af1b2507c8f2bf362fdf1d63c88faba397e880c2e39430`
- Sample file name: `fsflt.sys`
- Debugger: WinDBG x64
- Disassembler: IDA Pro

WinDBG is the Microsoft debugger. It allows userland and kernel debugging on Windows environments. If you are not familiar with WinDBG, we recommend reading this article first: http://windbg.info/doc/1-common-cmds.html.

To perform Windows kernel analysis, we need a specific setup of the Virtual Machine and WinDBG in order to remotely debug the VM from the host. We strongly recommend the following Microsoft documentation available there: https://msdn.microsoft.com/en-us/library/windows/hardware/hh439378(v=vs.85).aspx.

Additionnaly, we need:

- To disable the driver signature enforcement (see next chapter):
  `Bcdedit.exe -set TESTSIGNING ON`
- To create some registry keys in order to load the rootkit on demand (we will see why later):

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt]
"Description"="FsFtl minifilter"
"DependOnService"="FltMgr"
"Group"="FSFilter Content Screener"
"ImagePath"="system32\\Drivers\\fsflt.sys"
"DisplayName"="FsFlt"
"Tag"=dword:00000001
"ErrorControl"=dword:00000001
"Type"=dword:00000002
"Start"=dword:00000003

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Enum]
"NextInstance"=dword:00000001
"Count"=dword:00000001
"0"="Root\\\\LEGACY_minifilter\\\\0000"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Instances]
"DefaultInstance"="minifilter Instance"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Instances\minifilter
Instance]
"Flags"=dword:00000000
"Altitude"="262100"
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Parameters]

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Parameters\c1]
@="\\Device\\HarddiskVolume1\\Sekoia\\ApplicationDummy.dll"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Parameters\c2]
@="\\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\services\\FsFlt"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Parameters\c3]
@="\\Device\\HarddiskVolume1\\Sekoia\\ApplicationDummy.dll"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FsFlt\Parameters\c4]
@=\\Device\\HarddiskVolume1\\Sekoia
```

Note: you must adapt the `HarddiskVolume1` to your system and point to the C drive. The directory `C:\Sekoia` must exist and must contain an `ApplicationDummy.dll` file.

DRIVER SIGNATURE ENFORCEMENT & PATCH GUARD

Driver signature enforcement

Since Windows 7 x64, Microsoft implemented the driver signature enforcement. When this feature is enabled, the drivers must be signed by a trusted vendor in order to be loaded by the system. More information about this topic can be found on the Microsoft web site: https://msdn.microsoft.com/en-us/library/windows/hardware/ff544865%28v=vs.85%29.aspx. This validation is performed by `CI.dll` (Code Integrity). The rootkit developers need to sign their rootkits or to bypass the signature control. If you are interested in the driver signature bypasses, we recommend watching our talk during SyScan360 there: https://www.youtube.com/watch?v=lVEaw5SW2DE. In our case, we didn't have the dropper of the driver so we could not identify the signature bypass technique that was used. At this stage, to perform the analysis, we have to disable the driver signature as explained previously.

Patch guard

Since Windows 7 x64, Microsoft implemented the Patch Guard (aka Kernel Patch Protection – KKP). The purpose of the security feature is to protect the integrity of the kernel's code and to avoid modification of sensitive tables such as the System Service Dispatch Table (SSDT), the Interrupt Descriptor Table (IDT) or Global Descriptor Table (GDT). A lot of rootkits performed inline hooks or table modifications in order to modify the behaviour of the kernel. If a driver performs this kind of thing with Patch Guard enabled, the system crashes and the user gets a Blue Screen of Death. Generally, in the 64 bits version of modern Windows rootkit, the developers use legitimate callback procedures provided by Microsoft (the rootkit Turla is an exception however where the developers bypassed Patch Guard and performed inline hooks in the kernel's code directly).

GLOBAL ARCHITECTURE OF THE DRIVER

First steps

WinDBG can be automatically stopped when the driver is loaded (after pressing `CTRL+Break` to break the system):

```
kd> sxe ld FsFlt.sys
```

Using a previously described setup, you can easily load the driver by starting the `FsFlt` service:

```
C:\Windows\System32> sc start FsFlt
```

The debugger will be automatically stopped at the driver loading:

```
kd> !lmi FsFlt
Loaded Module Info: [fsflt]
         Module: fsflt
   Base Address: fffff80198af0000
     Image Name: fsflt.sys
   Machine Type: 34404 (X64)
     Time Stamp: 5305a705 Thu Feb 20 07:56:05 2014
           Size: d000
       CheckSum: b745
Characteristics: 22
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW     58,   82f8,      72f8 RSDS - GUID: {5DBF95F0-1907-435C-996B-1A16AE85070C}
               Age: 1e, Pdb: d:\!work\etc\hideinstaller_kis2013\Bin\Debug\win7\x64\fsflt.pdb
    Symbol Type: DEFERRED - No error - symbol load deferred
    Load Report: no symbols loaded
```

You can notice the name of the .pdb path: `kis2013`. This could be a reference to Kaspersky Internet Security 2013. At this point, you can set a breakpoint on the `DriverEntry()` function of the driver:

Name	Address	Ordinal
DriverEntry	000000000001B064	[main entry]

```
kd> bp fsflt+0xb064
kd> g
Breakpoint 0 hit
fsflt+0xb064:
fffff801`98afb064 4883ec28         sub      rsp,28h
```

Note: the default base address in IDA Pro is 0x10000. The base address of the loading module is 0xfffff800198af0000 as you can see in the `!lmi` command output (so the offset in argument to the breakpoint is IDA Pro Address – 0x10000).

Static overview of the rootkit design

As explained, the entry point of the rootkit is at offset 0x1b064 (`DriverEntry()`). The core of the driver is located at 0x11010 (`Core()`). This chapter will focus on the workflow of this `Core()` function.

Note: If you are lost, do not hesitate to read the debug provided by the developer:

```
lea     rcx, Format      ; "HIDEDRV: "
call    DbgPrint
lea     rcx, aHideDriverStar ; "Hide driver started\n"
call    DbgPrint
```

To display the debug, you can use DebugView from SysInternals: https://technet.microsoft.com/en-us/sysinternals/bb896647.aspx

Step 1

First, (offset 0x11500), the driver gets the registry stored value in `\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\services\FsFlt\Parameters\c4` thanks to the `ZwOpenKey()`, `ZwQueryKey()` and `ZwEnumerateKey()` APIs. Then the driver checks if the value is an existing directory with the `ZwCreateFile()` function with the `CreateOptions` argument at “`FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NOALERT`”:

```
0011571                 mov     [rsp+278h+var_128], 0
001157D                 lea     rdx, [rsp+278h+var_F0]
0011585                 lea     rcx, aRegistryMachin ; "\\REGISTRY\\MACHINE\\SYSTEM\\CurrentCon"...
001158C                 call    OpenKey_QueryKey
0011591                 mov     [rsp+278h+var_70], eax
0011598                 cmp     [rsp+278h+var_70], 0
00115A0                 jge     short loc_115CA
00115A2                 cmp     cs:dword_19110, 2
00115A9                 jl      short loc_115C3
00115AB                 lea     rcx, Format     ; "HIDEDRV: "
00115B2                 call    DbgPrint
00115B7                 lea     rcx, aHidefordriverE ; "HideForDriver: error!\n"
00115BE                 call    DbgPrint
00115C3
00115C3 loc_115C3:                              ; CODE XREF: C4_C5_RegMgmt+A9↑j
00115C3                 xor     eax, eax
00115C5                 jmp     loc_11CAD
00115CA ; ---------------------------------------------------------------------------
00115CA
00115CA loc_115CA:                              ; CODE XREF: C4_C5_RegMgmt+A0↑j
00115CA                 mov     rdx, [rsp+278h+var_F0]
00115D2                 add     rdx, 0Ch        ; SourceString
00115D6                 lea     rcx, [rsp+278h+DestinationString] ; DestinationString
00115DE                 call    cs:RtlInitUnicodeString
00115E4                 mov     [rsp+278h+ObjectAttributes.Length], 30h
00115EF                 mov     [rsp+278h+ObjectAttributes.RootDirectory], 0
00115FB                 mov     [rsp+278h+ObjectAttributes.Attributes], 2C0h
0011606                 lea     rax, [rsp+278h+DestinationString]
001160E                 mov     [rsp+278h+ObjectAttributes.ObjectName], rax
0011616                 mov     [rsp+278h+ObjectAttributes.SecurityDescriptor], 0
0011622                 mov     [rsp+278h+ObjectAttributes.SecurityQualityOfService], 0
001162E                 mov     [rsp+278h+EaLength], 0 ; EaLength
0011636                 mov     [rsp+278h+EaBuffer], 0 ; EaBuffer
001163F                 mov     [rsp+278h+CreateOptions], 21h ; CreateOptions (FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT)
0011647                 mov     [rsp+278h+CreateDisposition], 1 ; CreateDisposition
001164F                 mov     [rsp+278h+ShareAccess], 3 ; ShareAccess (FILE_SHARE_READ | FILE_SHARE_WRITE)
0011657                 mov     [rsp+278h+FileAttributes], 10h ; FileAttributes
001165F                 mov     [rsp+278h+AllocationSize], 0 ; AllocationSize
0011668                 lea     r9, [rsp+278h+IoStatusBlock] ; IoStatusBlock
0011670                 lea     r8, [rsp+278h+ObjectAttributes] ; ObjectAttributes
0011678                 mov     edx, 1          ; DesiredAccess
001167D                 lea     rcx, [rsp+278h+FileHandle] ; FileHandle
0011685                 call    cs:ZwCreateFile
001168B                 mov     [rsp+278h+var_70], eax
0011692                 cmp     [rsp+278h+var_70], 0
```

If the directory does not exist, the driver stops itself. (That's why we created the `C:\Sekoia` directory). In kernel space, the most common way to allocate memory is to use `ExAllocatePoolWithTag()` function. Here is the prototype:

```
PVOID ExAllocatePoolWithTag(
  _In_ POOL_TYPE PoolType,
  _In_ SIZE_T    NumberOfBytes,
  _In_ ULONG     Tag
);
```

The first argument is the pool type (paged, non-paged, etc.), the second argument is the size of the pool and the last argument is the tag name. The tag name is a four characters’ value. We identified two different tag names in the rootkit:

- DCBA
- rbRN

```
mov     r8d, 'NRbr'      ; Tag
mov     edx, 10h         ; NumberOfBytes
mov     ecx, 1           ; PoolType
call    cs:ExAllocatePoolWithTag
```

```
mov     r8d, [rsp+278h+Tag_ABCD]
mov     edx, 200h        ; NumberOfBytes
xor     ecx, ecx         ; PoolType
call    cs:ExAllocatePoolWithTag
```

Step 2

The rootkit registers a file-system minifilter. This feature will be described in detail later.

Step 3

The rootkit parses the configuration stored in the registry in order to get the value of the files, the directories and the registry keys to hide and the path of the library to inject in `explorer.exe`. The parsing is performed at offset 0x12e30 and the values are stored in global variables (`FilePath`, `RegPath` and `DLLPath` in the screenshot):

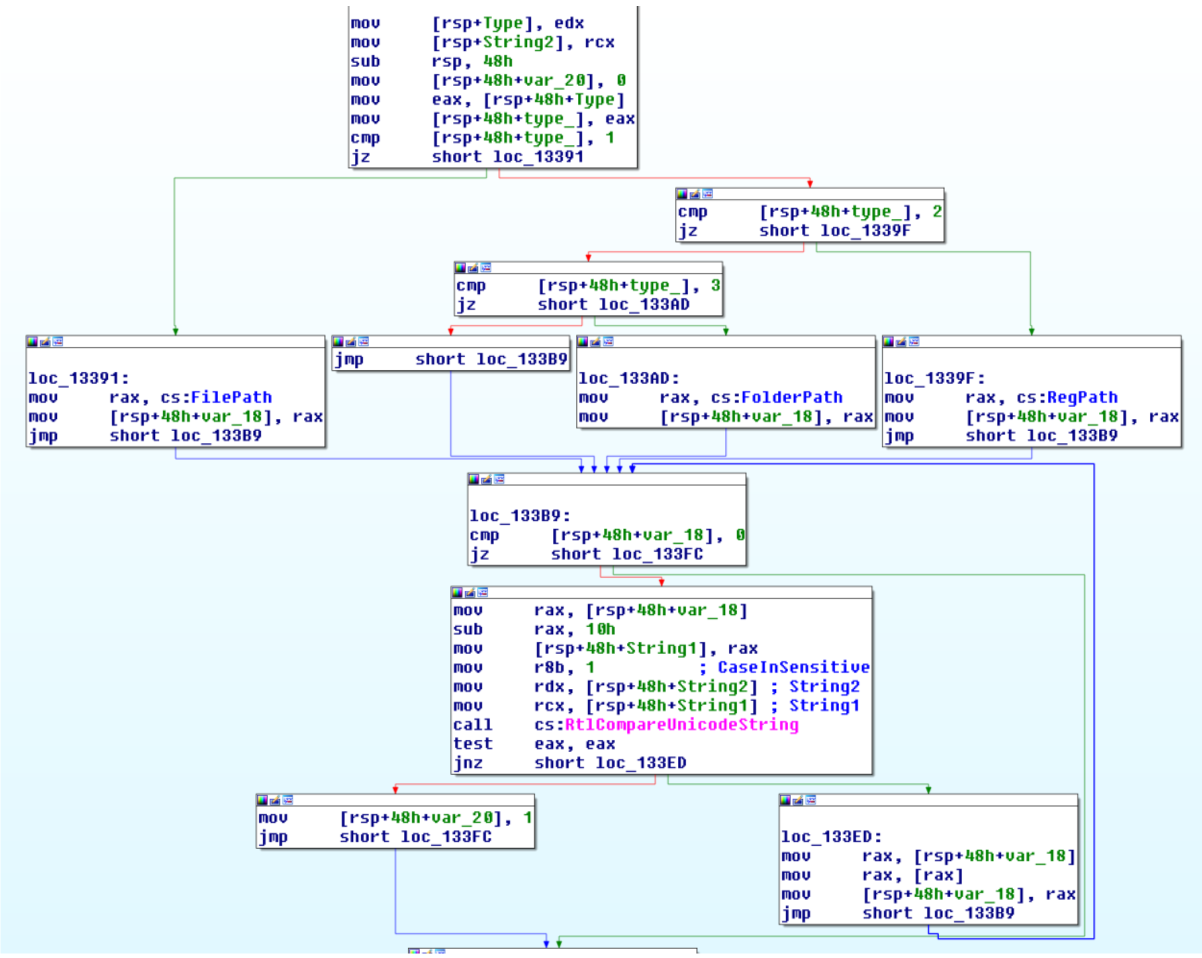
```
012E3C                 lea     rdx, FilePath
012E43                 lea     rcx, aParametersC1 ; "\\Parameters\\c1"
012E4A                 call    OpenQueryKey_
012E4F                 mov     [rsp+38h+var_18], eax
012E53                 cmp     [rsp+38h+var_18], 0
012E58                 jge     short loc_12E7F
012E5A                 cmp     cs:dword_19110, 2
012E61                 jl      short loc_12E7F
012E63                 lea     rcx, Format     ; "HIDEDRV: "
012E6A                 call    DbgPrint
012E6F                 mov     edx, [rsp+38h+var_18]
012E73                 lea     rcx, aInitprotectrul ; "InitProtectRules: ReadHidingConfig erro"...
012E7A                 call    DbgPrint
012E7F
012E7F loc_12E7F:                              ; CODE XREF: C1_C2_C3_RegMgmt+28↑j
012E7F                                         ; C1_C2_C3_RegMgmt+31↑j
012E7F                 lea     rdx, RegPath
012E86                 lea     rcx, aParametersC2 ; "\\Parameters\\c2"
012E8D                 call    OpenQueryKey_
012E92                 mov     [rsp+38h+var_18], eax
012E96                 cmp     [rsp+38h+var_18], 0
012E9B                 jge     short loc_12EC2
012E9D                 cmp     cs:dword_19110, 2
012EA4                 jl      short loc_12EC2
012EA6                 lea     rcx, Format     ; "HIDEDRV: "
012EAD                 call    DbgPrint
012EB2                 mov     edx, [rsp+38h+var_18]
012EB6                 lea     rcx, aInitprotectr_0 ; "InitProtectRules: ReadHidingConfig erro"...
012EBD                 call    DbgPrint
012EC2
012EC2 loc_12EC2:                              ; CODE XREF: C1_C2_C3_RegMgmt+6B↑j
012EC2                                         ; C1_C2_C3_RegMgmt+74↑j
012EC2                 lea     rdx, DLLPath
012EC9                 lea     rcx, aParametersC3 ; "\\Parameters\\c3"
012ED0                 call    OpenQueryKey_
012ED5                 mov     [rsp+38h+var_18], eax
012ED9                 cmp     [rsp+38h+var_18], 0
012EDE                 jge     short loc_12F05
012EE0                 cmp     cs:dword_19110, 2
012EE7                 jl      short loc_12F05
012EE9                 lea     rcx, Format     ; "HIDEDRV: "
012EF0                 call    DbgPrint
012EF5                 mov     edx, [rsp+38h+var_18]
012EF9                 lea     rcx, aInitprotectr_1 ; "InitProtectRules: ReadHidingConfig erro"...
012F00                 call    DbgPrint
```

In each callback (registry/file/directory), these global variables are used to check if the accessed element returned by the callback matches one of the rules stored in registry. Here is the debug output of the configuration parsing left by the developer:

```
HIDEDRV: >>>>>>>>Hide rules>>>>>>>> rules
HIDEDRV: File rules: \Device\HarddiskVolume1\Sekoia\ApplicationDummy.dll
HIDEDRV: Registry rules: \REGISTRY\MACHINE\SYSTEM\CurrentControlSet\services\FsFlt
HIDEDRV: Inject dll: \Device\HarddiskVolume1\Sekoia\ApplicationDummy.dll
HIDEDRV: Folder rules: \Device\HarddiskVolume1\Sekoia
HIDEDRV: <<<<<<<<XXXXX<<<<<<<<< rules
HIDEDRV: <<<<<<<<Hide rules<<<<<<<< rules
```

The driver callbacks use the full volume path, that's why the content in registry must start by `\Device\HarddiskVolumeX` (the kernel view) and not by `c:\` (the userland view). The comparison function is at the offset 0x13360. This function has 2 arguments:

- a string (the name to be checked returned by the callback)
- an integer:
  - `1`: check if the string matches the file rules global variable;
  - `2`: check if the string matches the registry rules global variable;
  - `3`: check if the string matches the folder rules global variable.



Step 4

The rootkit starts the file-system minifilter registered previously. This feature will be described in detail later.

Step 5

The rootkit registers and starts the registry callbacks at offset 0x144a0. This feature will be described in detail later.

Step 6

The rootkit registers and starts the process creation callbacks at offset 0x15030. This feature will be described in detail later. However, this function starts with an interesting action:

```
sub     rsp, 158h
mov     [rsp+158h+var_18], 0
lea     rdx, a??CWindowsSyst ; "\\??\\C:\\Windows\\System32\\sysprep\\C"...
lea     rcx, [rsp+158h+DestinationString] ; DestinationString
call    cs:RtlInitUnicodeString
mov     [rsp+158h+var_48], 30h
mov     [rsp+158h+var_40], 0
mov     [rsp+158h+var_30], 2C0h
lea     rax, [rsp+158h+DestinationString]
mov     [rsp+158h+var_38], rax
mov     [rsp+158h+var_28], 0
mov     [rsp+158h+var_20], 0
lea     rcx, [rsp+158h+var_48]
call    cs:ZwDeleteFile
mov     [rsp+158h+var_18], eax
cmp     [rsp+158h+var_18], 0
jge     short loc_150EA
```

The rootkit deletes the file `C:\Windows\System32\sysprep\CRYPTBASE.dll`. We don't exactly know why the driver removes this file. However, the library and this path are frequently used to bypass UAC. You can find in Metasploit the code of this kind of privilege escalation: https://github.com/rapid7/metasploit-framework/blob/master/external/source/exploits/bypassuac/Win7Elevate/Win7Elevate_Inject.cpp. We assume that the rootkit removes the trace of a privilege escalation previously realised.

Step 7

Finally, the rootkit defines an event that will be used to perform the library injection in the process `explorer.exe`. This features will be described in detail later.

FILE-SYSTEM MINI-FILTER

Static analysis

A file-system mini-filter is basically registered and started using two functions:

- `FltRegisterFilter()` (MSDN documentation: https://msdn.microsoft.com/en-us/library/windows/hardware/ff544305(v=vs.85).aspx)
- `FltStartFiltering()` (MSDN documentation: https://msdn.microsoft.com/en-us/library/windows/hardware/ff544569(v=vs.85).aspx)

From a reverse engineering point of view, the first one in the most interesting, in particular the second argument (structure) used in the function: `FLT_REGISTRATION`. Here is the prototype of the structure:

```
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;
#if FLT_MGR_LONGHORN
  PFLT_TRANSACTION_NOTIFICATION_CALLBACK      TransactionNotificationCallback;
  PFLT_NORMALIZE_NAME_COMPONENT_EX            NormalizeNameComponentExCallback;
#endif
#ifdef FLT_MFG_WIN8
  PFLT_SECTION_CONFLICT_NOTIFICATION_CALLBACK SectionNotificationCallback;
#endif
} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

Here is the view of this structure in IDA Pro:

```
018290 FilterRegistration db  68h ; h          ; DATA XREF: Core+90↑o
018291                    db    0
018292                    db    2
018293                    db    2
018294                    db    0
018295                    db    0
018296                    db    0
018297                    db    0
018298                    db    0
018299                    db    0
01829A                    db    0
01829B                    db    0
01829C                    db    0
01829D                    db    0
01829E                    db    0
01829F                    db    0
0182A0                    dq offset unk_18210      ; FLT_OPERATION_REGISTRATION
0182A8                    dq offset FilterUnloadCallback
0182B0                    dq offset InstanceSetupCallback
0182B8                    dq offset InstanceQueryTeardownCallback
0182C0                    dq offset InstanceTeardownStartCallback
0182C8                    dq offset InstanceTeardownStartCallback
0182D0                    db    0
0182D1                    db    0
0182D2                    db    0
```

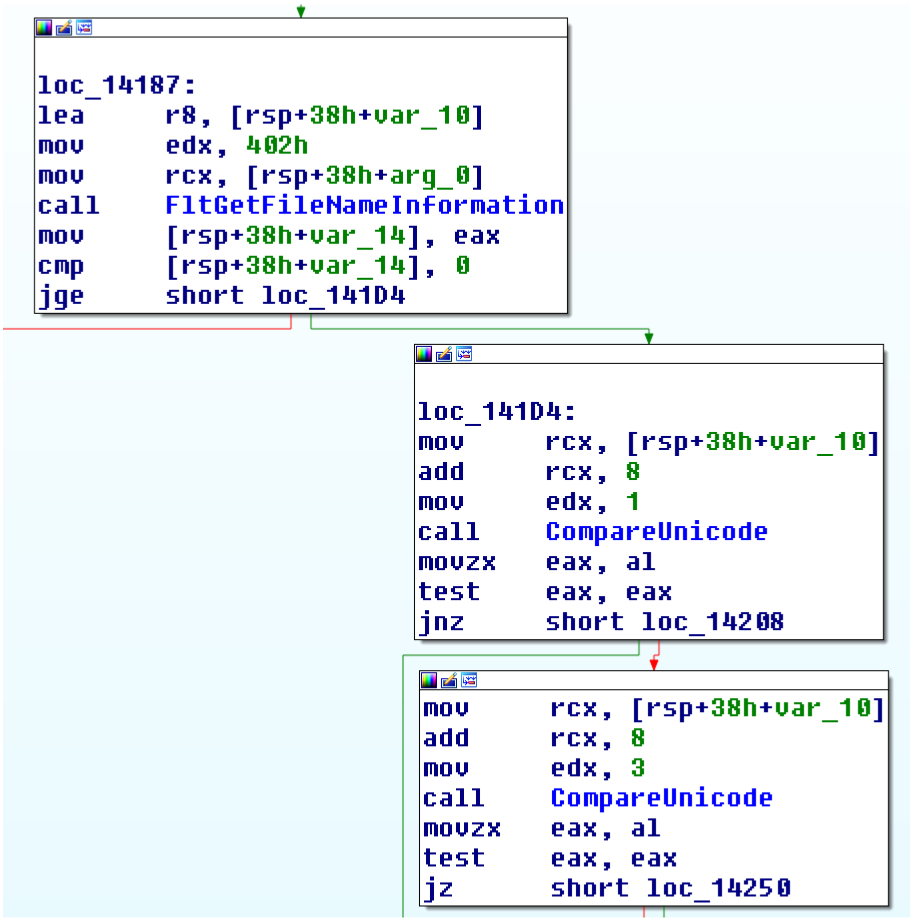
The 5 callbacks functions do not contain relevant codes. The interesting code in located in the `FLT_OPERATION_REGISTRATION` structure. Here is the prototype of the structure:

```
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;
```

The content can also be viewedin IDA Pro:

```
018210 unk_18210       db    0
018211                 db    0
018212                 db    0
018213                 db    0
018214                 db    0
018215                 db    0
018216                 db    0
018217                 db    0
018218                 dq offset PreOperation
018220                 db    0
018221                 db    0
```

The `PreOperation()` function (0x14100) contains the code used to get the file/directory name returned by the callback in order to compare it with the path stored in the registry. Here is the assembly code:

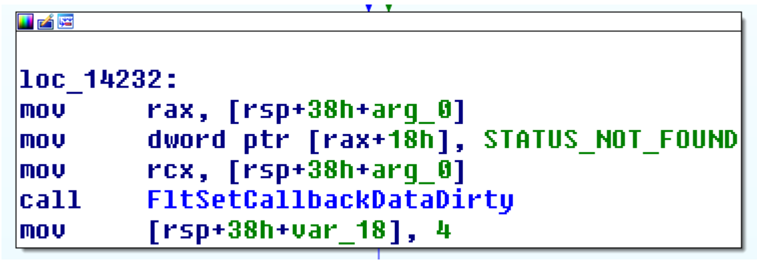


```
loc_14187:
lea     r8, [rsp+38h+var_10]
mov     edx, 402h
mov     rcx, [rsp+38h+arg_0]
call    FltGetFileNameInformation
mov     [rsp+38h+var_14], eax
cmp     [rsp+38h+var_14], 0
jge     short loc_141D4

loc_141D4:
mov     rcx, [rsp+38h+var_10]
add     rcx, 8
mov     edx, 1
call    CompareUnicode
movzx   eax, al
test    eax, eax
jnz     short loc_14208

mov     rcx, [rsp+38h+var_10]
add     rcx, 8
mov     edx, 3
call    CompareUnicode
movzx   eax, al
test    eax, eax
jz      short loc_14250
```

If the callback name returned matches a value in registry, the code modifies the contents of the callback data structure by setting `STATUS_NOT_FOUND` via the `FltSetCallbackDataDirty()` API. The file or the directory will finally be hidden to the user:



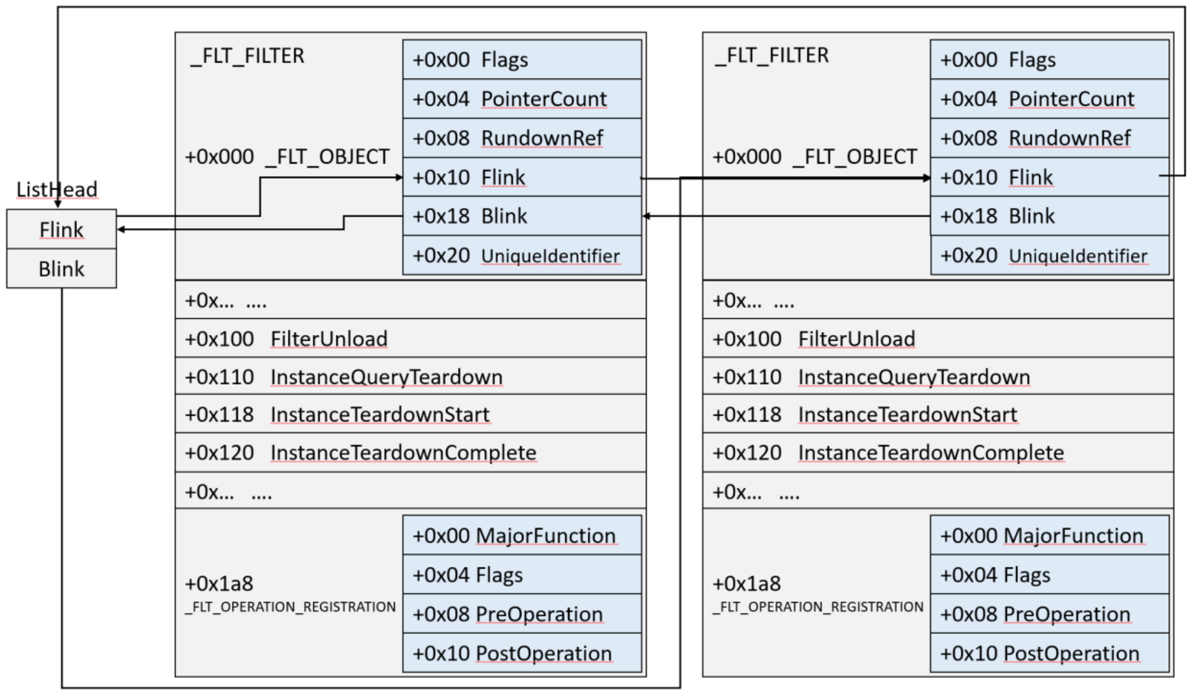
```
loc_14232:
mov     rax, [rsp+38h+arg_0]
mov     dword ptr [rax+18h], STATUS_NOT_FOUND
mov     rcx, [rsp+38h+arg_0]
call    FltSetCallbackDataDirty
mov     [rsp+38h+var_18], 4
```

Overview of the minifilter structure

The Windows kernel frequently uses linked lists. The file system minifilters uses circular doubly linked list. Here is the definition of this kind of list:

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY  *Flink;
  struct _LIST_ENTRY  *Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

The schema of the linked filters can be seen below:



Dynamic analysis with WinDBG

Thanks to the previous schema, we can analyse and identify the file-system minifilters with WinDBG. The `fltkd` extension allows to list the registered filter (and get the address of the `ListHead` in red) and the configuration of these filters:

```
kd> !fltkd.filters

Filter List: ffffe001b9e730c0 "Frame 0"
   FLT_FILTER: ffffe001b9e8ba90 "WdFilter" "328010"
      FLT_INSTANCE: ffffe001ba3cd780 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffe001ba47f010 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffe001ba8c8640 "WdFilter Instance" "328010"
      FLT_INSTANCE: ffffe001ba982640 "WdFilter Instance" "328010"
   FLT_FILTER: ffffe001b9b42010 "FsFlt" "262100"
      FLT_INSTANCE: ffffe001ba8366f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba8306f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba83c6f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba8416f0 "minifilter Instance" "262100"
   FLT_FILTER: ffffe001b9240320 "storqosflt" "244000"
   FLT_FILTER: ffffe001b954a5e0 "FileCrypt" "141100"
   FLT_FILTER: ffffe001bb694720 "luafv" "135000"
      FLT_INSTANCE: ffffe001bb69b010 "luafv" "135000"
   FLT_FILTER: ffffe001ba4b2cb0 "npsvctrig" "46000"
      FLT_INSTANCE: ffffe001ba98c710 "npsvctrig" "46000"
   FLT_FILTER: ffffe001b9e85010 "FileInfo" "45000"
      FLT_INSTANCE: ffffe001ba3cdb40 "FileInfo" "45000"
      FLT_INSTANCE: ffffe001ba46e420 "FileInfo" "45000"
      FLT_INSTANCE: ffffe001baa45310 "FileInfo" "45000"
      FLT_INSTANCE: ffffe001ba721b40 "FileInfo" "45000"
   FLT_FILTER: ffffe001b9e88580 "Wof" "40700"
      FLT_INSTANCE: ffffe001ba462b90 "Wof Instance" "40700"
      FLT_INSTANCE:        ffffe001ba73c640        "Wof        Instance"        "40700"

kd> !fltkd.filter ffffe001b9b42010

FLT_FILTER: ffffe001b9b42010 "FsFlt" "262100"
   FLT_OBJECT: ffffe001b9b42010  [02000000] Filter
      RundownRef               : 0x000000000000000a (5)
      PointerCount             : 0x00000001
      PrimaryLink              : [ffffe001b9240330-ffffe001b9e8baa0]
   Frame                    : ffffe001b9e73010 "Frame 0"
   Flags                    : [00000002] FilteringInitiated
```

```
   DriverObject              : ffffe001bc5768d0
   FilterLink                : [ffffe001b9240330-ffffe001b9e8baa0]
   PreVolumeMount            : 0000000000000000  (null)
   PostVolumeMount           : 0000000000000000  (null)
   FilterUnload              : fffff80198af13f0  fsflt+0x13f0
   InstanceSetup             : fffff80198af14b0  fsflt+0x14b0
   InstanceQueryTeardown     : fffff80198af14d0  fsflt+0x14d0
   InstanceTeardownStart     : fffff80198af14f0  fsflt+0x14f0
   InstanceTeardownComplete  : fffff80198af14f0  fsflt+0x14f0
   ActiveOpens               : (ffffe001b9b421c8)  mCount=0
   Communication Port List   : (ffffe001b9b42218)  mCount=0
   Client Port List          : (ffffe001b9b42268)  mCount=0
   VerifierExtension         : 0000000000000000
   Operations                : ffffe001b9b422c0
   OldDriverUnload           : 0000000000000000  (null)
   SupportedContexts         : (ffffe001b9b42140)
      VolumeContexts            : (ffffe001b9b42140)
      InstanceContexts          : (ffffe001b9b42140)
      FileContexts              : (ffffe001b9b42140)
      StreamContexts            : (ffffe001b9b42140)
      StreamHandleContexts      : (ffffe001b9b42140)
      TransactionContext        : (ffffe001b9b42140)
      (null)                    : (ffffe001b9b42140)
   InstanceList              : (ffffe001b9b42078)
      FLT_INSTANCE: ffffe001ba8366f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba8306f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba83c6f0 "minifilter Instance" "262100"
      FLT_INSTANCE: ffffe001ba8416f0 "minifilter Instance" "262100"
```

We can see few callbacks to the `FsFlt` driver. Sadly, the `fltkd` extension hides few elements and we cannot find the `PreOperation()` function. We used a trick to get it by parsing the kernel structures as described in the previous schema:

```
kd> dt _FLT_FILTER ffffe001b9b42010
FLTMGR!_FLT_FILTER
   +0x000 Base             : _FLT_OBJECT
   +0x030 Frame            : 0xffffe001`b9e73010 _FLTP_FRAME
   +0x038 Name             : _UNICODE_STRING "FsFlt"
   +0x048 DefaultAltitude  : _UNICODE_STRING "262100"
   +0x058 Flags            : 2 ( FLTFL_FILTERING_INITIATED )
   +0x060 DriverObject     : 0xffffe001`bc5768d0 _DRIVER_OBJECT
   +0x068 InstanceList     : _FLT_RESOURCE_LIST_HEAD
   +0x0e8 VerifierExtension : (null)
```

```
   +0x0f0 VerifiedFiltersLink    :    _LIST_ENTRY  [   0x00000000`00000000  -
0x00000000`00000000 ]
   +0x100 FilterUnload     : 0xfffff801`98af13f0     long  +0
   +0x108 InstanceSetup    : 0xfffff801`98af14b0     long  +0
   +0x110 InstanceQueryTeardown : 0xfffff801`98af14d0     long  +0
   +0x118 InstanceTeardownStart : 0xfffff801`98af14f0     void  +0
   +0x120 InstanceTeardownComplete : 0xfffff801`98af14f0     void  +0
   +0x128 SupportedContextsListHead : (null)
   +0x130 SupportedContexts : [7] (null)
   +0x168 PreVolumeMount   : (null)
   +0x170 PostVolumeMount  : (null)
   +0x178 GenerateFileName : (null)
   +0x180 NormalizeNameComponent : (null)
   +0x188 NormalizeNameComponentEx : (null)
   +0x190 NormalizeContextCleanup : (null)
   +0x198 KtmNotification  : (null)
   +0x1a0 SectionNotification : (null)
   +0x1a8 Operations       : 0xffffe001`b9b422c0 _FLT_OPERATION_REGISTRATION
   +0x1b0 OldDriverUnload  : (null)
   +0x1b8 ActiveOpens      : _FLT_MUTEX_LIST_HEAD
   +0x208 ConnectionList   : _FLT_MUTEX_LIST_HEAD
   +0x258 PortList         : _FLT_MUTEX_LIST_HEAD
   +0x2a8 PortLock         : _EX_PUSH_LOCK

kd> dt _FLT_OPERATION_REGISTRATION 0xffffe001`b9b422c0
FLTMGR!_FLT_OPERATION_REGISTRATION
   +0x000 MajorFunction    : 0 ''
   +0x004 Flags            : 0
   +0x008 PreOperation     : 0xfffff801`98af4100     _FLT_PREOP_CALLBACK_STATUS  +0
   +0x010 PostOperation    : (null)
   +0x018 Reserved1        : (null)

kd> u 0xfffff801`98af4100
fsflt+0x4100:
fffff801`98af4100 4c89442418      mov     qword ptr [rsp+18h],r8
fffff801`98af4105 4889542410      mov     qword ptr [rsp+10h],rdx
fffff801`98af410a 48894c2408      mov     qword ptr [rsp+8],rcx
fffff801`98af410f 4883ec38        sub     rsp,38h
fffff801`98af4113 c744242000000000 mov     dword ptr [rsp+20h],0
fffff801`98af411b 48c744242800000000 mov    qword ptr [rsp+28h],0
fffff801`98af4124 488b442440      mov     rax,qword ptr [rsp+40h]
fffff801`98af4129 488b4010        mov     rax,qword ptr [rax+10h]
```

We can confirm that the defined `PreOperation()` function is at `FsFlt+0x4100` as mentioned previously.

Limitations

The implementation made by the rootkit developer has several limitations. The most interesting is the fact that files and directories implementing hiding feature is performed using their full volume paths (for example: `\Device\HarddiskVolume1\Sekoia\ApplicationDummy.dll`). By changing the volume name, we bypass the protection and we can access to the hidden artefacts. An easy way to change the volume name is to create a Shadow Copy of the drive. The volume path pattern of a Shadow Copy is `\Device\HarddiskVolumeShadowCopyX`. By mounting it, we are able to access to the protection files and directories. If you use live forensics tools with Shadow Copy feature to retrieve the artefacts (such as FastIR Collector: https://github.com/SekoiaLab/Fastir_Collector) the rootkit is simply inefficient.

REGISTRY CALLBACKS

Static analysis

A registry access callback is basically registered and started with one function:

- `CmRegisterCallbackEx()` (MSDN documentation: https://msdn.microsoft.com/en-us/library/windows/hardware/ff541921(v=vs.85).aspx)

From a reverse engineering point of view, the first argument of this function is the most interesting. It contains the function to be executed by the callback:

```
loc_1453B:
mov     rax, cs:P
add     rax, 38h
mov     rdx, cs:P
add     rdx, 10h
mov     [rsp+48h+var_20], 0
mov     [rsp+48h+var_28], rax
mov     r9, cs:P
mov     r8, [rsp+48h+arg_0]
lea     rcx, RegCallBacksFunction
call    cs:CmRegisterCallbackEx
mov     [rsp+48h+var_18], eax
xor     eax, eax
test    eax, eax
jnz     loc_144B1
```

The callback function is `RegCallBacksFunction()` (0x4870). This function checks if the accessed registry name matches the value to be hidden. If the result is successful, the function performs a second check on the process path that tries to access to this registry:

```
call    cs:PsGetCurrentProcessId
mov     rcx, rax
call    GetProcessName
mov     [rsp+88h+var_18], rax
cmp     [rsp+88h+var_18], 0
jnz     short loc_149FD
```

```
loc_149FD:                      ; "*\\services.exe"
lea     rdx, aServices_exe
lea     rcx, [rsp+88h+DestinationString] ; DestinationString
call    cs:RtlInitUnicodeString
mov     cs:byte_19170, 1
cmp     cs:dword_19110, 5
jl      short loc_14A3C
```

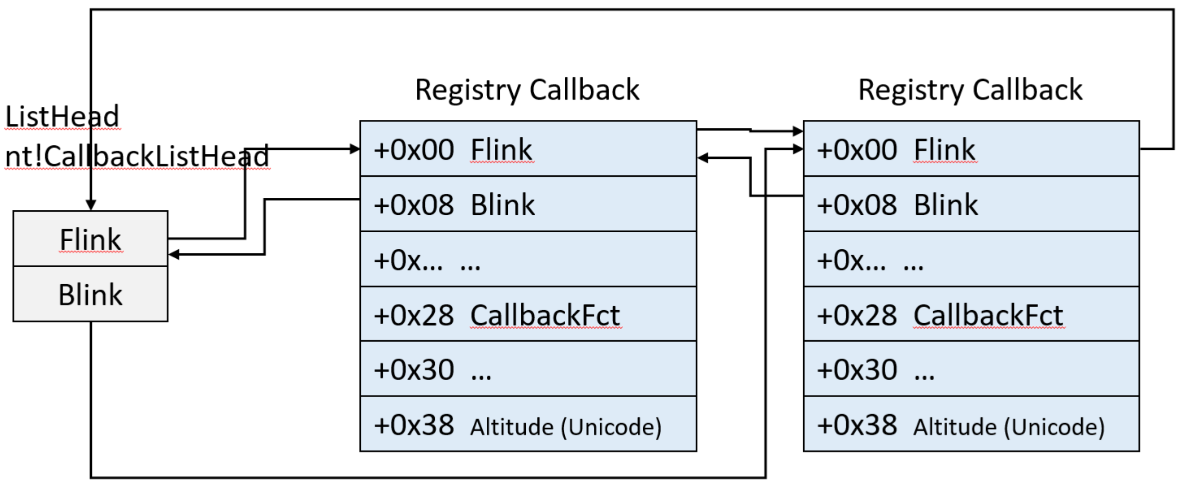
```
loc_14A3C:
xor     r9d, r9d
xor     r8d, r8d
mov     rdx, [rsp+88h+var_18]
lea     rcx, [rsp+88h+DestinationString]
call    cs:FsRtlIsNameInExpression
movzx   eax, al
test    eax, eax
jnz     short loc_14A6D
```

If the process path that ends with `services.exe` is available to the hidden registry, the rootkit does not hide it. However, the callback function changes the contents of the callback data structure to `STATUS_NOT_FOUND`:

```
mov     rax, [rsp+88h+var_48]
mov     dword ptr [rax+18h], STATUS_NOT_FOUND
mov     [rsp+88h+var_58], 0C0000503h ; STATUS_CALLBACK_BYPASS
```

Overview of the registry callbacks structure

The schema of the registry callback linked list looks like this:



Dynamic analysis with WinDBG

We can list the registry callbacks thanks to WinDBG. The first step is to get the number of defined callbacks:

```
kd> dd nt!CmpCallBackCount L1
fffff803`6db02be0  00000002
```

On our virtual machine, we have 2 registry callbacks listed into a `_LIST_ENTRY` list:

```
kd> dps nt!CallbackListHead L2
fffff803`6dafa700  ffffc000`92eb3bb0
fffff803`6dafa708  ffffc000`9605c710
kd> dt nt!_LIST_ENTRY fffff803`6dafa700
 [ 0xffffc000`92eb3bb0 - 0xffffc000`9605c710 ]
   +0x000 Flink            : 0xffffc000`92eb3bb0 _LIST_ENTRY [ 0xffffc000`9605c710 - 0xfffff803`6dafa700 ]
   +0x008 Blink            : 0xffffc000`9605c710 _LIST_ENTRY [ 0xfffff803`6dafa700 - 0xffffc000`92eb3bb0 ]

kd> dps 0xffffc000`9605c710 L8
ffffc000`9605c710  fffff803`6dafa700 nt!CallbackListHead
ffffc000`9605c718  ffffc000`92eb3bb0
ffffc000`9605c720  0069006e`00000000
ffffc000`9605c728  01d1d67d`e400c771
ffffc000`9605c730  ffffc000`952b4a80
ffffc000`9605c738  fffff801`98af4870 fsflt+0x4870
ffffc000`9605c740  00650065`000c000c
ffffc000`9605c748  ffffc000`9498ca70
```

We can see that one of the callbacks refers to `FsFlt+0x4870`. It matches the offset mentioned previously.

Limitations

The implementation by the rootkit developer has several limitations. By simply copying `regedit.exe` on the desktop and by renaming it to `services.exe`, the user can execute it and see the hidden registry key because the executable file path will end with `services.exe` and the rootkit will think that it's the real `services.exe` process.

PROCESS CREATION CALLBACKS

Static analysis

A process creation callback is basically registered and started with one function:

- `PsSetCreateProcessNotifyRoutine()` (MSDN documentation: https://msdn.microsoft.com/en-us/library/windows/hardware/ff559951(v=vs.85).aspx)

Once again, from a reverse engineering point of view, the first argument of the function is the most interesting since it contains the function name to be executed when a process is created or deleted:

```
loc_15112:
xor     edx, edx
lea     rcx, ProcessCallBacks
call    cs:PsSetCreateProcessNotifyRoutine
mov     [rsp+158h+var_18], eax
cmp     [rsp+158h+var_18], 0
jge     short loc_15166
```

In our sample, the function is `ProcessCallbacks()` (0x15280). This function checks if the process name is `explorer.exe`. If so, the rootkit sets an event in order to inject the library (.dll) configured in the registry.

Dynamic analysis with WinDBG

We can list the process creation and deletion callbacks thanks to WinDBG. The first step is to get the number of callbacks:

```
kd> dd nt!PspCreateProcessNotifyRoutineCount L1
fffff803`6defadcc  00000006
kd> dd nt!PspCreateProcessNotifyRoutineExCount L1
fffff803`6defadc8  00000002
```

On the system, we have 8 process callbacks. With a small script, we can list them all:

```
kd> .for (r $t0=0; $t0 < 8; r $t0=$t0+1) { r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine); .if ($t1 == 0) { .continue }; r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0; dps $t1+8 L1;}
ffffe001`b9248688  fffff803`6d8db7e0 nt!ViCreateProcessCallback
ffffe001`b9239348  fffff801`96687290 cng!CngCreateProcessNotifyRoutine
ffffe001`b95f7df8  fffff801`96c970a0 WdFilter!MpCreateProcessNotifyRoutineEx
ffffe001`b9e8e1f8  fffff801`96488da0 ksecdd!KsecCreateProcessNotifyRoutine
ffffe001`ba312748  fffff801`96fc30d0 tcpip!CreateProcessNotifyRoutineEx
ffffe001`ba372f58  fffff801`9633d7b0 CI!I_PEProcessNotify
```

```
ffffe001`b92cb2e8  fffff801`988daba0 peauth+0x2aba0
ffffe001`bada52d8  fffff801`98af5280 fsflt+0x5280
```

On the results above, the last callback points to the `ProcessCallbacks()` function mentioned previously. As WinDBG scripting language is not really user-friendly, here is the explanation:

```
.for (r $t0=0; $t0 < 8; r $t0=$t0+1)  #a loop of 8 iterations (our 8 callbacks)
{
  r $t1=poi($t0 * 8 + nt!PspCreateProcessNotifyRoutine);
  #For each callback, get the pointer to the process callback structure
  .if ($t1 == 0) {
    .continue
  };
  r $t1 = $t1 & 0xFFFFFFFFFFFFFFF0;  #Apply a mask on the pointer
  dps $t1+8 L1; #the callback function is at the offset 0x8
}
```

PAYLOAD INJECTION

To perform the library injection in the process explorer.exe, the rootkit uses the APC (Asynchronous Procedure Calls). You can find a document on the feature on MSDN: https://msdn.microsoft.com/fr-fr/library/windows/desktop/ms681951(v=vs.85).Aspx.

The technique was used in the well-known rootkit TLD3/TLD4. The driver uses the API `KeStackAttachProcess()` and `KeUnstackDetachProcess()` in order to attach the current driver thread to the address space of the `explorer.exe` process. The driver gets the base address of `kernel32.dll` and particularly the address of `LoadLibrary()` in `explorer.exe`. Then, the driver allocates memory in the process to copy the library path. Finally, the rootkit executes `KeInitializeApc()` and `KeInsertQueueAPC()` to execute `LoadLibrary()` in order to load and execute the library.

As a tutorial, you can read the source code of TLD3 in order to understand the techniques in use: http://pastebin.com/UpvGUw19.

CHAPTER ABSTRACT

With one driver, we browsed explanations on file-system, registry callbacks, process creation callbacks and code injection via APC. These techniques are really common in rootkit analysis. The biggest missing piece concerns the network capabilities. This rootkit does not implement NDIS Filter or WFP (Windows Filtering Platform). The network capabilities are really common; for example, this mechanism was implemented in the Turla rootkit. In order to be as complete as possible, here is the way to investigate and to list the WPF callbacks with WinDBG:

```
kd> dp netio!gWfpGlobal L1
fffff801`96a63258  ffffe001`b9e025b0

kd> u netio!FeInitCalloutTable L10
NETIO!FeInitCalloutTable:
fffff801`96a22490 4053             push    rbx
fffff801`96a22492 4883ec20         sub     rsp,20h
fffff801`96a22496 488b05bb0d0400   mov     rax,qword ptr [NETIO!gWfpGlobal (fffff801`96a63258)]
fffff801`96a2249d 33c9             xor     ecx,ecx
fffff801`96a2249f ba57667043       mov     edx,43706657h
fffff801`96a224a4 48898848010000   mov     qword ptr [rax+148h],rcx
fffff801`96a224ab 48898850010000   mov     qword ptr [rax+150h],rcx
fffff801`96a224b2 b900400100       mov     ecx,14000h
fffff801`96a224b7 4c8b059a0d0400   mov     r8,qword ptr [NETIO!gWfpGlobal (fffff801`96a63258)]
fffff801`96a224be 4981c050010000   add     r8,150h
fffff801`96a224c5 e8223dfeff       call    NETIO!WfpPoolAllocNonPaged (fffff801`96a061ec)
fffff801`96a224ca 488bd8           mov     rbx,rax

kd> dps ffffe001`b9e025b0+0x150 L1
ffffe001`b9e02700  ffffe001`b9e07000

kd> !pool ffffe001`b9e07000
Pool page ffffe001b9e07000 region is Nonpaged pool
*ffffe001b9e07000 : large page allocation, tag is WfpC, size is 0x14000 bytes
          Pooltag WfpC : WFP callouts, Binary : netio.sys

kd> u NETIO!InitDefaultCallout
NETIO!InitDefaultCallout:
fffff801`96a2251c 4053             push    rbx
fffff801`96a2251e 4883ec20         sub     rsp,20h
fffff801`96a22522 4c8d051f150400   lea     r8,[NETIO!gFeCallout (fffff801`96a63a48)]
fffff801`96a22529 ba57667043       mov     edx,43706657h
fffff801`96a2252e b950000000       mov     ecx,50h
fffff801`96a22533 e8b43cfeff       call    NETIO!WfpPoolAllocNonPaged
```

```
fffff801`96a22538 488bd8          mov     rbx,rax
fffff801`96a2253b 4885c0          test    rax,rax

kd> r $t0=ffffe001b9e07000; .for( r $t1=0; @$t1 < 0x30; r $t1=@$t1+1) {dps @$t0+2*@$ptrsize L2; r $t0=@$t0+0x50;}
ffffe001`b9e07010  00000000`00000000
ffffe001`b9e07018  00000000`00000000
ffffe001`b9e07060  fffff801`971ab5c0 tcpip!IPSecInboundTransportFilterCalloutClassifyV4
ffffe001`b9e07068  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
ffffe001`b9e070b0  fffff801`971ab700 tcpip!IPSecInboundTransportFilterCalloutClassifyV6
ffffe001`b9e070b8  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
ffffe001`b9e07100  fffff801`971aaf70 tcpip!IPSecOutboundTransportFilterCalloutClassifyV4
ffffe001`b9e07108  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
ffffe001`b9e07150  fffff801`971b30d0 tcpip!IPSecOutboundTransportFilterCalloutClassifyV6
ffffe001`b9e07158  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
ffffe001`b9e071a0  fffff801`971b2990 tcpip!IPSecInboundTunnelFilterCalloutClassifyV4
ffffe001`b9e071a8  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
ffffe001`b9e071f0  fffff801`971b2a50 tcpip!IPSecInboundTunnelFilterCalloutClassifyV6
ffffe001`b9e071f8  fffff801`9712b060 tcpip!IPSecAleConnectCalloutNotify
[…]
ffffe001`b9e07c90  fffff801`97037500 tcpip!WfpAlepSetOptionsCalloutClassify
ffffe001`b9e07c98  fffff801`9707ce80 tcpip!FllAddGroup
ffffe001`b9e07ce0  00000000`00000000
```

BONUS: DIFFERENCE BETWEEN THE X86 & THE X64 VERSIONS

CONTEXT

We decided to add a small chapter concerning the x86 version of HIDEDRV. This version contains 2 major difference with the x64 version:

- The driver creates a device and a symbolic link;
- Instead of using file system minifilters to hide elements, the driver defines 3 SSDT (System Service Dispatch Table) hooks.

As we wrote, the SSDT hooks are not possible in Windows x64 (except when bypassing Patch Guard). This constraint does not exist in x86. This approach is not popular anymore but it's interesting to keep it in mind and be able to analyse it.

For those interested in this x86 sample, the associated hash is: `b1900cb7d1216d1dbc19b4c6c8567d48215148034a41913cc6e59958445aebde`

DEVICE AND SYMBOLIC LINK

On the x86 version of the rootkit, the developer created a driver device and a symbolic link:

```
loc_105E5:                   ; "\\Device\\dfsflt"
push    offset aDeviceDfsflt
lea     eax, [ebp+DestinationString]
push    eax                  ; DestinationString
call    ds:RtlInitUnicodeString
push    offset SourceString ; "\\DosDevices\\dfsflt"
lea     ecx, [ebp+SymbolicLinkName]
push    ecx                  ; DestinationString
call    ds:RtlInitUnicodeString
```

```
loc_10639:
lea     edx, [ebp+DeviceObject]
push    edx             ; DeviceObject
push    0               ; Exclusive
push    100h            ; DeviceCharacteristics
push    22h             ; DeviceType
lea     eax, [ebp+DestinationString]
push    eax             ; DeviceName
push    0               ; DeviceExtensionSize
mov     ecx, [ebp+DriverObject]
push    ecx             ; DriverObject
call    ds:IoCreateDevice
mov     [ebp+var_18], eax
cmp     [ebp+var_18], 0
jge     short loc_10687
```

```
loc_10704:
lea     eax, [ebp+DestinationString]
push    eax             ; DeviceName
lea     ecx, [ebp+SymbolicLinkName]
push    ecx             ; SymbolicLinkName
call    ds:IoCreateSymbolicLink
mov     [ebp+var_18], eax
cmp     [ebp+var_18], 0
jge     short loc_10750
```

In the analysed sample, the device and symbolic link are not used. The symbolic links are usually created in order to receive notification (IOCTL) from the user space via the `DeviceIoControl()` API.

We can list the device thanks to WinDBG:

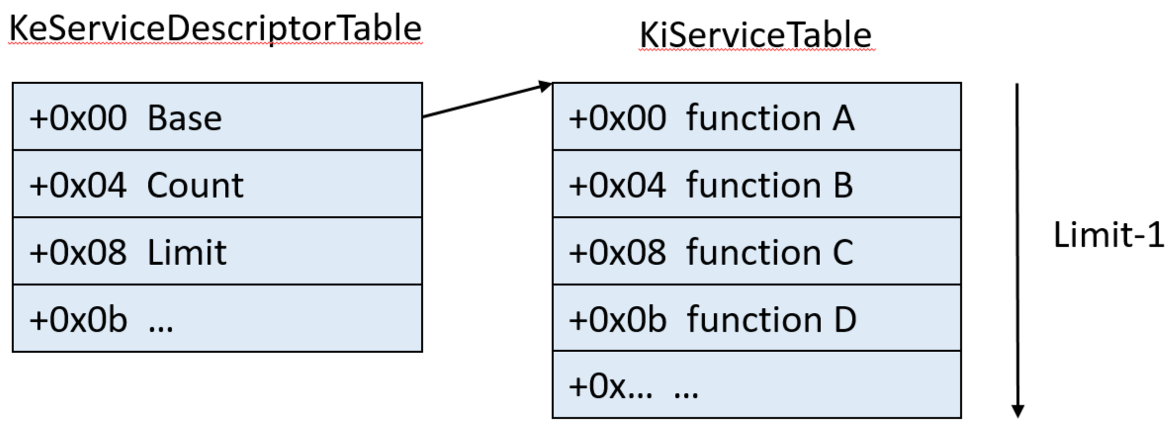
```
kd> !object \Device
Object: 88e0f030  Type: (85253e90) Directory
    ObjectHeader: 88e0f018 (new version)
    HandleCount: 0  PointerCount: 232
    Directory Object: 88e010e8  Name: Device

    Hash Address  Type                      Name
    ---- -------  ----                      ----
     00  85fb9738 Device                    KsecDD
         862092d8 Device                    Beep
         86004388 Device                    Ndis
     […]
     28  854762e8 Device                    dfsflt
         86214578 Device                    Null
         852d77f0 Device                    00000010
         852c7030 Device                    00000003
     […]
```

SSDT HOOKS

Overview of the SSDT

The SSDT is the table that contains the addresses of the functions to be executed when a syscall is made. Here is the schema of the table:



To understand how it works, we can look the assembly code of the function `NtQueryKey()`:

```
kd> u ntdll!NtQueryKey
ntdll!ZwQueryKey:
77ca60e8 b8f4000000      mov     eax,0F4h
77ca60ed ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub
77ca60f2 ff12            call    dword ptr [edx]
77ca60f4 c21400          ret     14h
77ca60f7 90              nop
```

The function executes a system call with the argument 0xF4. We can get the function executed when this system call is performed:

```
kd> dps KiServiceTable+0xf4*4 L1
826b816c  82886cae nt!NtQueryKey
```

Note: a second table (`KeServiceDescriptorTableShadow`) exists. The table contains a pointer to `KiServiceTable` (the same as previously) and to `W32perviceTable` (syscall for the GUI threads).

Static analysis

In the function `SSDT_Hook()` (0x13490), the rootkit replaces 3 functions addresses in the `KiServiceTable`. These functions are used to read registry value, get file information and get directory information:

```
loc_134FD:
mov     edx, [ebp+var_4]
push    edx
push    offset dword_15BDC
push    offset Hook_ZwQueryDirectoryFile
push    offset aZwquerydirecto ; "ZwQueryDirectoryFile"
call    Set_Hook
mov     eax, [ebp+var_4]
push    eax
push    offset dword_15BD4
push    offset Hook_ZwSetInformationFile
push    offset aZwsetinformati ; "ZwSetInformationFile"
call    Set_Hook
mov     ecx, [ebp+var_4]
push    ecx
push    offset dword_15BE0
push    offset Hook_ZwEnumerateKey
push    offset aZwenumeratekey ; "ZwEnumerateKey"
call    Set_Hook
mov     eax, [ebp+var_8]
```

The malicious code functions have the same purpose as the callbacks previously described:

- if the accessed file, directory or registry must be hidden, the rootkit returns that the element does not exist;
- if the accessed file, directory or registry is not in the list, the rootkit executes the original function of the SSDT (previously saved in a global variable).
-

Dynamic analysis with WinDBG

The SSDT hook can be directly identified with WinDBG:

```
kd> dps nt!KeServiceDescriptorTable L3
827a39c0  826b7d9c nt!KiServiceTable
827a39c4  00000000
827a39c8  00000191
kd> .shell -ci "dps nt!KiServiceTable L0x191" find "FsFlt"
826b7f6c  92af9160 FsFlt+0x2160
826b8118  92afac00 FsFlt+0x3c00
826b82c0  92afa9c0 FsFlt+0x39c0
.shell: Process exited
```

CHAPTER ABSTRACT

The SSDT hooks are becoming rarer. The same approach can be performed in the IDT (Interrupt Description Table). The table is used to identify the function address to execute when an interrupt is called. We can display the table thanks to WinDBG:

```
kd> !idt -a

Dumping IDT: 80b95400

3255d61800000000:82677fc0 nt!KiTrap00
3255d61800000001:82678150 nt!KiTrap01
3255d61800000002:Task Selector = 0x0058
3255d61800000003:826785c0 nt!KiTrap03
3255d61800000004:82678748 nt!KiTrap04
[…]
```

As explained previously, this approach was popular a few years ago on x86 platform but it tends to become rarer today due to Patch Guard.

CONCLUSION

This document has been written as a “hands on” for reverse engineering beginners and willing to leverage his experience on rootkits.

The use case of the document is a very interesting case study for basic rootkit techniques. Using different tools and tricks, we overviewed the main features of the rootkit such as filesystem manipulation, registry and process callbacks, code injection and even network manipulation.