Analysis of Uroburos, using WinDbg

gdatasoftware.com/blog/2014/06/23953-analysis-of-uroburos-using-windbg

Uroburos was already described as a very sophisticated and highly complex malware in our G DATA Red Paper, where we had a look at the malware's behavior. This malware belongs to a specific type called rootkit. The general purpose of a rootkit is to modify the behavior of the system and, especially, to hide its activity. Generally, a rootkit resides in the kernel. To analyze this kind of malicious software, analysts need to use specific tools, such as WinDbg, to debug the Microsoft Windows kernel. WinDbg is a debugger provided by Microsoft. One can use this tool to debug user mode applications and kernel mode applications (for example the drivers).

Today, we would like to give you an understanding of how analysts work their way through malware and give you some insights into the code of one of the most sophisticated digital threats. In this current example case, we decided to work with a memory dump (crash dump) of a system infected with Uroburos. To facilitate the analysis, we added an extension to add the support of python, called: PyKd. WinDbg has its own script language, but it is not easy to understand. One can download this python extension here, for free: http://pykd.codeplex.com/.

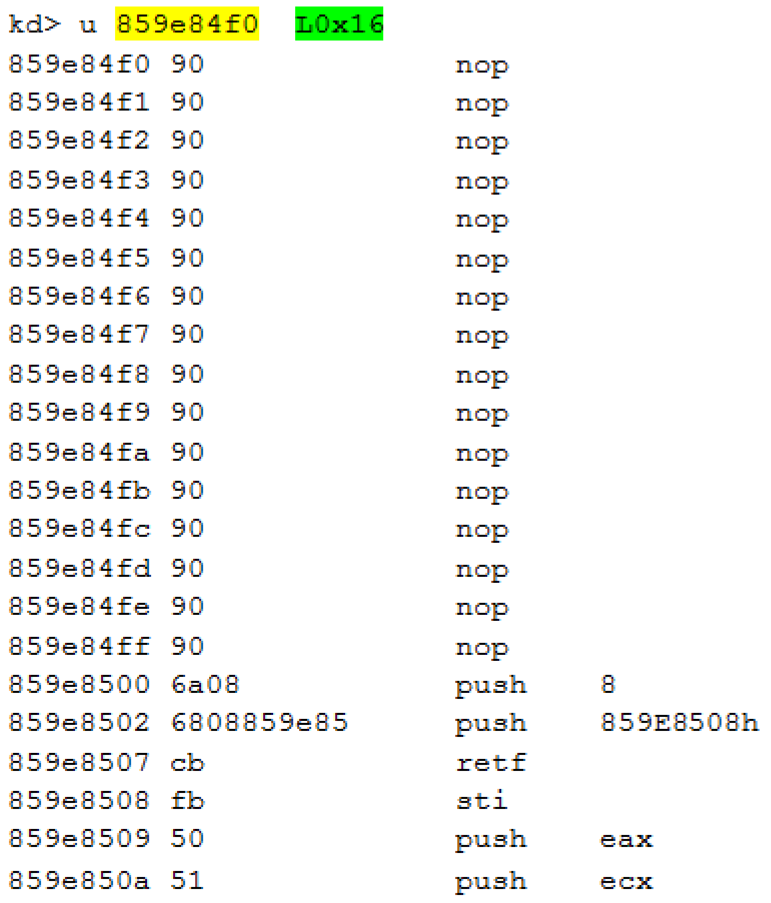
To realize this article, the machine was infected by the Uroburos dropper with the following md5: 626576e5f0f85d77c460a322a92bb267.

Visualization of the hooks

The Uroburos rootkit adds several hooks to hide its activity. In our specific case, the hooking is a technique used to alter the behavior of specific system functions; the rootkit fakes the output of the Microsoft Windows API. For example, it hides registry entries, files and more. To perform this task, the rootkit developers decided to use interrupts. We can display the Interrupt Descriptor Table (IDT), as shown below. The IDT table stores pointers to ISR (Interrupt Service Routines), which are called when an interrupt is triggered.

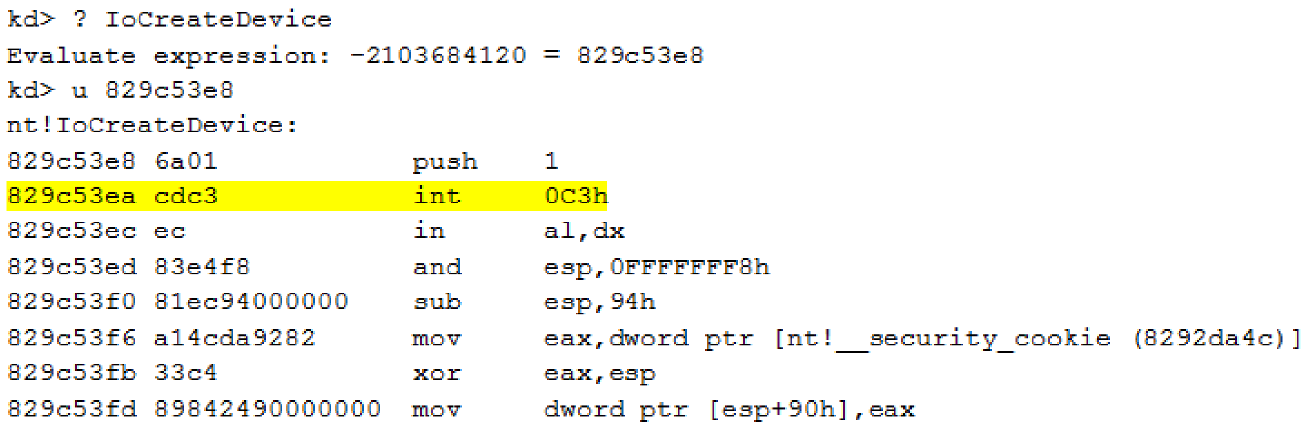
```
kd> !idt

Dumping IDT: 80b95400

3194895000000030:      82c27ca4 hal!Halp8254ClockInterrupt
3194895000000031:      8486b058 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8486b000)
3194895000000038:      82c18c6c hal!HalpRtcProfileInterrupt
3194895000000039:      8486bcd8 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 8486bc80)
319489500000003a:      85afd7d8 ndis!ndisMiniportIsr (KINTERRUPT 85afd780)
319489500000003b:      8486b558 ataport!IdePortInterrupt (KINTERRUPT 8486b500)
319489500000003c:      85afdcd8 i8042prt!I8042MouseInterruptService (KINTERRUPT 85afdc80)
319489500000003e:      8486ba58 ataport!IdePortInterrupt (KINTERRUPT 8486ba00)
319489500000003f:      8486b7d8 ataport!IdePortInterrupt (KINTERRUPT 8486b780)
31948950000000c3:      859e84f0
```

One of the pointers (0x859e84f0) is unknown and cannot be resolved. All other pointers have a function name, following the address. The last digits of the first column are the ID of the interrupt (in our case 0xC3). We can disassemble the code available at this address:



```
kd> u 859e84f0  L0x16
859e84f0 90              nop
859e84f1 90              nop
859e84f2 90              nop
859e84f3 90              nop
859e84f4 90              nop
859e84f5 90              nop
859e84f6 90              nop
859e84f7 90              nop
859e84f8 90              nop
859e84f9 90              nop
859e84fa 90              nop
859e84fb 90              nop
859e84fc 90              nop
859e84fd 90              nop
859e84fe 90              nop
859e84ff 90              nop
859e8500 6a08            push    8
859e8502 6808859e85      push    859E8508h
859e8507 cb              retf
859e8508 fb              sti
859e8509 50              push    eax
859e850a 51              push    ecx
```

The last argument of the WinDbg command is the length (L0x16) to disassemble. The function starts by a series of NOP. The interrupt 0xC3 is used by the malware, the next step is to identify how and when this interrupt is triggered. Here is the code of the beginning of the function IoCreateDevice():



```
kd> ? IoCreateDevice
Evaluate expression: -2103684120 = 829c53e8
kd> u 829c53e8
nt!IoCreateDevice:
829c53e8 6a01            push    1
829c53ea cdc3            int     0C3h
829c53ec ec              in      al,dx
829c53ed 83e4f8          and     esp,0FFFFFFF8h
829c53f0 81ec94000000    sub     esp,94h
829c53f6 a14cda9282      mov     eax,dword ptr [nt!__security_cookie (8292da4c)]
829c53fb 33c4            xor     eax,esp
829c53fd 8984249000000   mov     dword ptr [esp+90h],eax
```

We can see that the second instruction is int 0xC3 (interrupt 0xC3). Thanks to the PyKd extension, we can easily create a python script to detect every function with this interrupt:

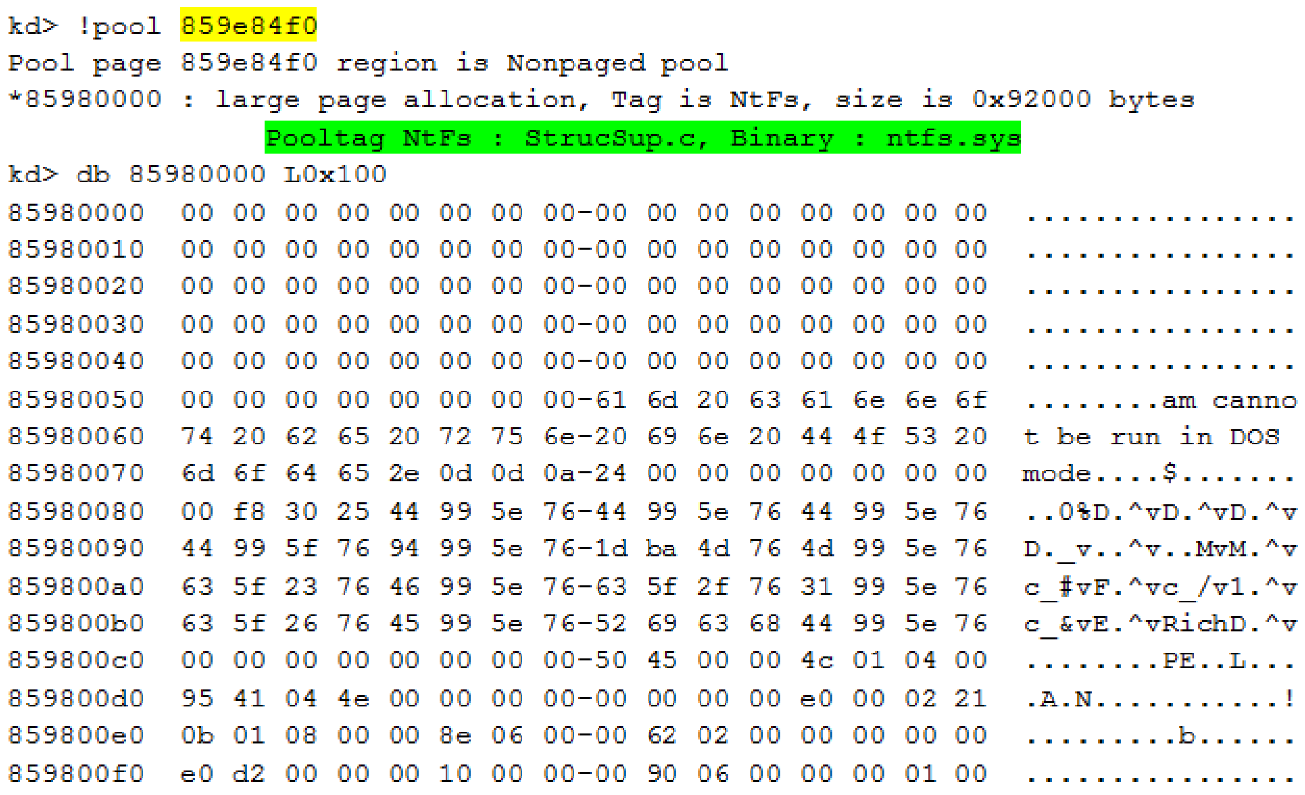
```
import pykd

output = pykd.dbgCommand("x nt!*").split("\n")
for i in output:
  if i != "":
    addr=i.split()[0]
    name=i.split()[1]
    opcode=pykd.dbgCommand("db %(addr)s+2 L2" % {'addr': addr}).split()
    if (opcode[1] == "cd") and (opcode[2] == "c3"):
      print "Hook: "+name
```

This script starts to list each exported function in ntoskrnl.exe. Secondly, for each function it checks if the second instruction is int 0xC3 (cdc3). If it is the case, the exported function’s name is displayed. Here is the output of the script regarding the current analysis:

```
kd> !py c:\hook.py
Hook:  nt!NtCreateKey
Hook:  nt!NtQueryInformationProcess
Hook:  nt!NtQuerySystemInformation
Hook:  nt!ObOpenObjectByName
Hook:  nt!NtClose
Hook:  nt!IoCreateDevice
Hook:  nt!NtEnumerateKey
Hook:  nt!NtShutdownSystem
Hook:  nt!NtTerminateProcess
Hook:  nt!IofCallDriver
Hook:  nt!NtQueryKey
Hook:  nt!NtCreateUserProcess
Hook:  nt!NtCreateThread
Hook:  nt!NtSaveKey
Hook:  nt!NtReadFile
```

We could use the function: !chkimg to easily identify the hook. However, it was a good exercise to play with PyKd.

Another interesting step is to dump the code of the driver. To perform this task, we first need to find the beginning of the PE. We can find the address thanks to the address of the code executed when an interrupt is triggered:


```
kd> !pool 859e84f0
Pool page 859e84f0 region is Nonpaged pool
*85980000 : large page allocation, Tag is NtFs, size is 0x92000 bytes
                Pooltag NtFs : StrucSup.c, Binary : ntfs.sys
kd> db 85980000 L0x100
85980000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
85980010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
85980020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
85980030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
85980040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
85980050  00 00 00 00 00 00 00 00-61 6d 20 63 61 6e 6e 6f  ........am canno
85980060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS 
85980070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
85980080  00 f8 30 25 44 99 5e 76-44 99 5e 76 44 99 5e 76  ..0%D.^vD.^vD.^v
85980090  44 99 5f 76 94 99 5e 76-1d ba 4d 76 4d 99 5e 76  D._v..^v..MvM.^v
859800a0  63 5f 23 76 46 99 5e 76-63 5f 2f 76 31 99 5e 76  c_#vF.^vc_/v1.^v
859800b0  63 5f 26 76 45 99 5e 76-52 69 63 68 44 99 5e 76  c_&vE.^vRichD.^v
859800c0  00 00 00 00 00 00 00 00-50 45 00 00 4c 01 04 00  ........PE..L...
859800d0  95 41 04 4e 00 00 00 00-00 00 00 00 e0 00 02 21  .A.N...........!
859800e0  0b 01 08 00 00 8e 06 00-00 62 02 00 00 00 00 00  .........b......
859800f0  e0 d2 00 00 00 10 00 00-00 90 06 00 00 00 01 00  ................
```
This output shows us two remarkable things:

- First, the driver uses a well-known Windows kernel memory pool tag called "NtFs". The Windows components mark allocated memory block with a unique tag. But the rootkit uses the same tag as the legitimate ntfs.sys driver. This choice was made to hide the rootkit and dupe the analyst.
- Secondly, the output looks like the beginning of a PE. But this PE is broken: the MZ is not available and some information is missing. For example, the value of the SizeOfImage (85980000+0x140) is null...

The rootkit alters the beginning of the PE to hide itself. Some tools parse the memory and look for the MZ string to identify the beginning of a PE. In our current case, if we used these tools looking for a PE file, we would never identify our malware using this automation. Manual analysis is needed here. To dump our driver we need to reconstruct the PE but we don't know the size of the binary, as mentioned above, so we need to make a large dump, to be sure to not forget a part of the binary.

Modules, drivers and devices

We can now display the loaded (and unloaded) modules with WinDbg:

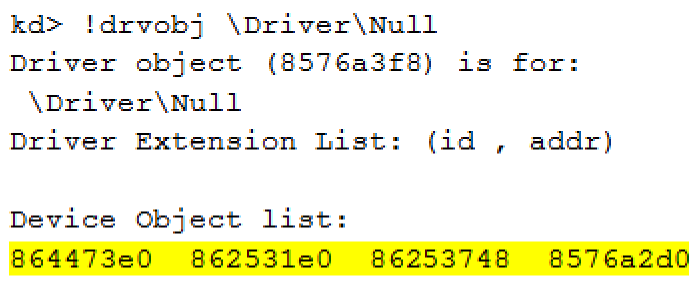
```
kd> lm f
start    end        module name
00400000 0041a000   win32dd_400000 C:\Users\user1\Desktop\win32dd.exe
737f0000 73815000   POWRPROF C:\Windows\system32\POWRPROF.dll
74ff0000 75017000   CFGMGR32 C:\Windows\system32\CFGMGR32.dll
75020000 7506b000   KERNELBASE C:\Windows\system32\KERNELBASE.dll
75070000 75082000   DEVOBJ   C:\Windows\system32\DEVOBJ.dll
751c0000 7520e000   GDI32    C:\Windows\system32\GDI32.dll
75210000 75229000   sechost  C:\Windows\SYSTEM32\sechost.dll
752f0000 753c4000   kernel32 C:\Windows\system32\kernel32.dll
[...]
993e8000 993f5000   tcpipreg \SystemRoot\System32\drivers\tcpipreg.sys

Unloaded modules:
8da09000 8dbf7000   fdisk.sys
```

In our case, the rootkit's module is fdisk.sys. According to the code shown above, it seems to be unloaded, but as we analyzed before, the code is really present on the infected system. So, the developers found a way to unload the modules while the malicious code is still running!

We can list the drive

```
kd> !object \driver\
Object: 8985ea70  Type: (84841e90) Directory
    ObjectHeader: 8985ea58 (new version)
    HandleCount: 0  PointerCount: 92
    Directory Object: 89805e28  Name: Driver

    Hash Address  Type          Name
    ---- -------  ----          ----
     00  85ae0530 Driver        rdpbus
         8576a1d8 Driver        Beep
         855b74b0 Driver        NDIS
         [...]
         85a3d310 Driver        Wanarpv6
     28  85a51030 Driver        discache
         8576a3f8 Driver        Null
     29  85a7aa38 Driver        VBoxVideo
         [...]
         855e6610 Driver        rdyboost
         8487e780 Driver        intelide
```

The driver used by our module is \driver\Null. All other modules are legitimate modules used by Windows. We can display the devices associated to the driver we are focusing on:



```
kd> !drvobj \Driver\Null
Driver object (8576a3f8) is for:
 \Driver\Null
Driver Extension List: (id , addr)

Device Object list:
864473e0  862531e0  86253748  8576a2d0
```

The device objects associated to our driver are:

- 0x864473e0
- 0x862531e0
- 0x86253748
- 0x8576a2d0

Furthermore, we can see the description of those devic

```
kd> !devobj 864473e0
Device object (864473e0) is for:
 FWPMCALLOUT \Driver\Null DriverObject 8576a3f8
Current Irp 00000000 RefCount 0 Type 00000000 Flags 000000c0
Dacl 8985aaf0 DevExt 00000000 DevObjExt 86447498
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
Device queue is not busy.

kd> !devobj 0x862531e0
Device object (862531e0) is for:
 RawDisk2 \Driver\Null DriverObject 8576a3f8
Current Irp 00000000 RefCount 0 Type 00000007 Flags 00000050
Vpb 86253158 DevExt 00000000 DevObjExt 86253298 Dope 86257008
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000001)  FILE_REMOVABLE_MEDIA
Device queue is not busy.

kd> !devobj 86253748
Device object (86253748) is for:
 RawDisk1 \Driver\Null DriverObject 8576a3f8
Current Irp 00000000 RefCount 22 Type 00000007 Flags 00000050
Vpb 862536c0 DevExt 00000000 DevObjExt 86253800 Dope 86253678
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000001)  FILE_REMOVABLE_MEDIA
Device queue is not busy.

kd> !devobj 8576a2d0
Device object (8576a2d0) is for:
 Null \Driver\Null DriverObject 8576a3f8
Current Irp 00000000 RefCount 0 Type 00000015 Flags 00000040
Dacl 8985aaf0 DevExt 00000000 DevObjExt 8576a388
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
Device queue is not busy.
```
Two objects are particularly interesting: FWPMCALLOUT and RawDisk1

WFP callout

This first device is FWPMCALLOUT. Thanks to the name of the device we can guess that the rootkit registers a callout for Windows Filtering Platform (WFP). The WFP is a set of API and system services which provides a platform for creating network filtering applications. In our case, the rootkit uses this technology to perform Deep Packet Inspection (DPI) and modifications of the network flow. The purpose of this device is to intercept relevant data as soon as a connection to the Command & Control server or other local infected machines used as relay is established and to receive commands.

As there is no command to simply list the WFP callouts, we have to extract the information needed using different steps:

First, the netio!gWfpGlobal variable contains the starting point for the WFP data structures:

```
kd> dp netio!gWfpGlobal L1
88b95260  84845008
```

A global table stores the number of callouts and the array of the corresponding callout structures.

Here is a method to find suitable offsets:

```
kd> u netio!FeInitCalloutTable L10
NETIO!FeInitCalloutTable:
88b7859e 8bff            mov     edi,edi
88b785a0 56              push    esi
88b785a1 57              push    edi
88b785a2 8b3d6052b988    mov     edi,dword ptr [NETIO!gWfpGlobal (88b95260)]
88b785a8 33c0            xor     eax,eax
88b785aa 81c7d8020000    add     edi,2D8h
88b785b0 ab              stos    dword ptr es:[edi]
88b785b1 ab              stos    dword ptr es:[edi]
88b785b2 a16052b988      mov     eax,dword ptr [NETIO!gWfpGlobal (88b95260)]
88b785b7 05dc020000      add     eax,2DCh
88b785bc 50              push    eax
88b785bd 6857667043      push    43706657h
88b785c2 be90010000      mov     esi,190h
88b785c7 56              push    esi
88b785c8 e8e4f8feff      call    NETIO!WfpPoolAllocNonPaged (88b67eb1)
88b785cd 8bf8            mov     edi,eax
```

The first number is the offset that contains the number of total callouts made, in hex, of course:

```
kd> dps 84845008+2D8 L1
848452e0  0000011e
```

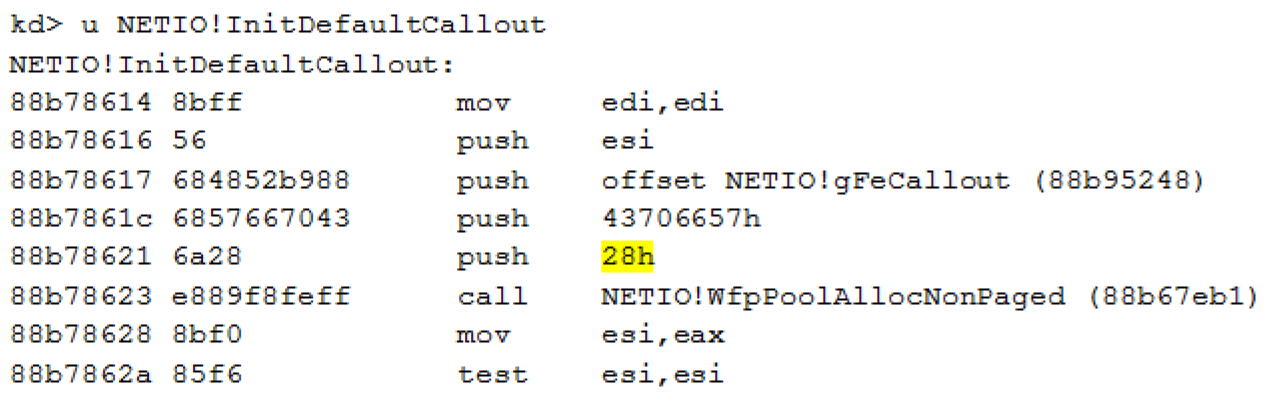
The second number is the offset that contains the array in which the callout structure is stored:

```
kd> dps 84845008+2DC L1
848452e4  86233000
```

The pool tag of this address confirms our findings so far and proves that we have found the right track:

```
kd> !pool 86233000
Pool page 86233000 region is Nonpaged pool
*86233000 : large page allocation, Tag is WfpC, size is 0x2cb8 bytes
                Pooltag WfpC : WFP callouts, Binary : netio.sys
```

We can now extract the size of each structure stored within the array. As it is not documented by Microsoft, we identify the size by disassembling the function InitDefaultCallout():



```
kd> u NETIO!InitDefaultCallout
NETIO!InitDefaultCallout:
88b78614 8bff            mov     edi,edi
88b78616 56              push    esi
88b78617 684852b988      push    offset NETIO!gFeCallout (88b95248)
88b7861c 6857667043      push    43706657h
88b78621 6a28            push    28h
88b78623 e889f8feff      call    NETIO!WfpPoolAllocNonPaged (88b67eb1)
88b78628 8bf0            mov     esi,eax
88b7862a 85f6            test    esi,esi
```

Finally, we use a one-liner command to list the elements of this array:

```
kd> r $t0=86233000 ;.for ( r $t1=0; @$t1 < 11e; r $t1=@$t1+1 ) {dps @$t0+2*@$ptrsize L2; r
$t0=@$t0+28;}
86233008  00000000
8623300c  00000000
[...]
86233238  00000000
862333a4  88cc1132 tcpip!WfpAlepSetOptionsCalloutClassify
862333c8  00000000
862333cc  88cc1132 tcpip!WfpAlepSetOptionsCalloutClassify
862333f0  00000000
862333f4  88d13ac7 tcpip!IPSecInboundTunnelAcceptAuthorizeCalloutClassify
86233418  00000000
8623341c  88d13ac7 tcpip!IPSecInboundTunnelAcceptAuthorizeCalloutClassify
86233440  00000000
86233444  88ce2e1d tcpip!FlpEdgeTraversalCalloutClassify
86233468  00000000
8623346c  88ce2e1d tcpip!FlpEdgeTraversalCalloutClassify
86233490  00000000
86233494  88ce2e1d tcpip!FlpEdgeTraversalCalloutClassify
862334b8  00000000
862334bc  88ce2e1d tcpip!FlpEdgeTraversalCalloutClassify
862334e0  00000000
862334e4  88d1f331 tcpip!IdpCalloutClassifyV6
86233508  00000000
8623350c  88d1f4d3 tcpip!IdpCalloutClassifyV4
86233530  00000000
[...]
86235808  00000000
8623580c  992ba750 mpsdrv!MpsGetFwpAuthData+0xff0
86235830  00000000
86235834  992ba750 mpsdrv!MpsGetFwpAuthData+0xff0
86235858  00000000
8623585c  992ba750 mpsdrv!MpsGetFwpAuthData+0xff0
86235880  00000000
86235884  992ba750 mpsdrv!MpsGetFwpAuthData+0xff0
86235b28  00000000
86235b2c  992baaf0 mpsdrv!MpsGetFwpAuthData+0x1390
86235b50  00000000
86235b54  992baaf0 mpsdrv!MpsGetFwpAuthData+0x1390
86235b78  00000000
86235b7c  859b5040
86235ba0  00000000
86235ba4  859b5520
86235bc8  00000000
86235bcc  00000000
```

The list of elements reminds us of the information we have seen in the IDT: two addresses are not resolved. Those two WFP callouts are: 0x859b5040 and 0x859b5520. WinDbg is not able to resolve these two addresses because the addresses are unknown. These are not addresses of a Microsoft. Now that we have the addresses, we can use the command !pool to validate that the addresses are in the same region as the code executed when an interrupt is triggered:

```
kd> !pool 859b5040
Pool page 859e84f0 region is Nonpaged pool
*85980000 : large page allocation, Tag is NtFs, size is 0x92000 bytes
                    Pooltag NtFs : StrucSup.c, Binary : ntfs.sys
kd> db 85980000 L0x100
```

Virtual file system

Previously, when looking at the device objects, we came across two devices with very similar names: RawDisk1 and RawDisk2. Let us have a detailed look at the first one:

```
kd> !devobj Rawdisk1
Device object (86253748) is for:
 RawDisk1 \Driver\Null DriverObject 8576a3f8
Current Irp 00000000 RefCount 22 Type 00000007 Flags 00000050
Vpb 862536c0 DevExt 00000000 DevObjExt 86253800 Dope 86253678
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000001)  FILE_REMOVABLE_MEDIA
Device queue is not busy.

kd> !vpb 862536c0
Vpb at 0x862536c0
Flags: 0x1 mounted
DeviceObject: 0x86259020
RealDevice:   0x86253748
RefCount: 22
Volume Label:

kd> !devobj 0x86259020
Device object (86259020) is for:
  \FileSystem\Ntfs DriverObject 8516e558
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00040000
DevExt 862590d8 DevObjExt 86259fb0
ExtensionFlags (0x00000800)  DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
AttachedDevice (Upper) 86253020 \FileSystem\FltMgr
Device queue is not busy.
```

As we can see, RawDisk1 device is in fact an NTFS file system; a virtual file system used by the rootkit to store its configuration, the exfiltrated data…

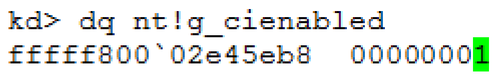
We can identify the used files (opened handles) within the file system, like \queue and \klog:

```
kd> !devhandles \device\Rawdisk1

Checking handle table for process 0x8483c8f0
Kernel handle table at 89801be0 with 411 entries in use

PROCESS 8483c8f0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 89801be0  HandleCount: 411.
    Image: System

02bc: Object: 8625b6e8  GrantedAccess: 0012019f Entry: 89803578
Object: 8625b6e8  Type: (848bd3f8) File
    ObjectHeader: 8625b6d0 (new version)
        HandleCount: 1  PointerCount: 2
        Directory Object: 00000000  Name: \$Extend\$RmMetadata\$TxfLog\$TxfLog.blf {RawDisk1}

[...]

PROCESS 8483c8f0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 89801be0  HandleCount: 411.
    Image: System

02f0: Object: 8626b6f0  GrantedAccess: 0012019f Entry: 898035e0
Object: 8626b6f0  Type: (848bd3f8) File
    ObjectHeader: 8626b6d8 (new version)
        HandleCount: 1  PointerCount: 10
        Directory Object: 00000000  Name: \queue {RawDisk1}

PROCESS 8483c8f0  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 89801be0  HandleCount: 411.
    Image: System

0344: Object: 8626f400  GrantedAccess: 00100004 Entry: 89803688
Object: 8626f400  Type: (848bd3f8) File
    ObjectHeader: 8626f3e8 (new version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \klog {RawDisk1}

[...]

PROCESS 86248a00  SessionId: 0  Cid: 01f0    Peb: 7ffd8000  ParentCid: 01a8
    DirBase: 7ec9b080  ObjectTable: 82374a98  HandleCount: 288.
    Image: services.exe

0340: Object: 86439038  GrantedAccess: 0012019f Entry: 8237b680
Object: 86439038  Type: (848bd3f8) File
    ObjectHeader: 86439020 (new version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \queue {RawDisk1}
[...]

Checking handle table for process 0x864bfa98
Handle table at 94997fc0 with 850 entries in use

PROCESS 864bfa98  SessionId: 1  Cid: 074c    Peb: 7ffdf000  ParentCid: 0540
    DirBase: 7ec9b1a0  ObjectTable: 94997fc0  HandleCount: 850.
    Image: explorer.exe

0c70: Object: 8649d350  GrantedAccess: 0012019f Entry: 9c01a8e0
Object: 8649d350  Type: (848bd3f8) File
    ObjectHeader: 8649d338 (new version)
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \queue {RawDisk1}
```

Thanks to this command we are able to list the files hidden from the operating system.

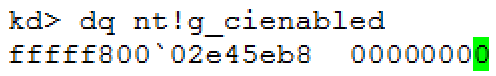
Digital signature enforcement

Microsoft created a Driver Signing Policy for its 64-bit versions of Windows Vista and later versions. To load a driver, the .sys file must be signed by a legitimate publisher. Developers may disable the Driver Signature Enforcement process during the development phase of a driver, which means a developer does not have to sign each compiled driver version during development phase. This mode is called “Test-mode”. In our case, the rootkit is not signed, which would usually mean that it had no chance to be accepted by Microsoft’s policy, but it disables the digital signature process to circumvent the checks. The status of this feature is stored in the global variable nt!g_cienabled. Compare the value of this variable on a clean system, without infection with the same information on an infected system:



```
kd> dq nt!g_cienabled
fffff800`02e45eb8  00000001
```

The code above shows that the value is set to 1



```
kd> dq nt!g_cienabled
fffff800`02e45eb8  00000000
```

We can clearly identify that the malware disabled the driver signature enforcement. Generally speaking, we could do the same by using bcdedit.exe -set TESTSIGNING OFF, to switch into testing mode to be able to load unsigned driver. The difference is: Using bcdedit.exe triggers a message window which is shown to the user, at the bottom of the desktop, and this is not very secretive. The action could be detected immediately.
More information about the malware’s circumvention of the driver signature enforcement can be found in our SecurityBlog article: Uroburos – Deeper travel into kernel protection mitigation

Conclusion

What you have seen now, is a very limited part of the extensive analysis of complicated malware and a very short introduction into WinDbg. Generally, it is very hard to apprehend such an extensive tool, but when working on such a case of kernel land analysis, researchers do not have a choice.

Processed like this, in an article with code snippets, the results seem logic and do make perfect sense. But, believe us, working with malware code costs a lot of extensive training, experience and time.

Related articles

2014/05/13 Uroburos rootkit: Belgian Foreign Ministry stricken

2014/03/07 Uroburos – Deeper travel into kernel protection mitigation

2014/02/28 Uroburos - highly complex espionage software with Russian roots

2014/02/28 G DATA RedPaper about Uroburos