KeUserModeCallback用法详解

转载

cosmoslife 于 2013-11-08 22:38:49 发布 634 收藏
分类专栏： 驱动开发学习


驱动开发学习 专栏收录该内容
467 篇文章 6 订阅
订阅专栏

- 标 题：KeUserModeCallback用法详解
- 作 者：achillis
- 时 间：2010-01-10 19:29:43
- 链 接：http://bbs.pediy.com/showthread.php?t=104918

ring0调用ring3早已不是什么新鲜事，除了APC，我们知道还有KeUserModeCallback.其原型如下：

代码:

```
NTSTATUS
KeUserModeCallback (
    IN ULONG ApiNumber,
    IN PVOID InputBuffer,
    IN ULONG InputLength,
    OUT PVOID *OutputBuffer,
    IN PULONG OutputLength
    );
```

但是对于这个函数怎么用，很多人还不是很清楚，因为它未文档化，参数意义又比较晦涩，但是我们除了wrk中的源码可以参考，还有一些实例可供调试研究。所以，要搞清楚它的调用过程并不困难，但是需要一些基本的知识，比如系统调用和返回的过程(KiFastCallEntry,KiServiceExit等)。本文尽量不多谈这种过程转换的细节，而把主要注意力集中在传入的参数及调用过程，以灰盒的方式来分析它。
调试的例子是一个俄国佬的Ring0MessageBox，我自己也写过一份(其实就是逆的)，放在看雪了，几乎完全一样。代码地址放在文后了，这东西以前发过一次，不过没讲原理.
有兴趣的可以下载回来扔到Vmware里，拿起调试器，一起来Debug~
一、KeUserModeCallback的基础知识
我们都知道一个普通的系统调用，它的过程大概是这样的(以OpenProcess为例)：
kernel32!OpenProcess -> ntdll!ZwOpenProcess -> ntdll!KiFastSystemCall -> sysenter -> nt!KiFastCallEntry -> nt!NtOpenProcess -> nt!KiFastCallEntry(返回后)-> nt!KiServiceExit -> sysexit -> ntdll!KiFastSystemCallRet -> kernel32!OpenProcess(返回后)
这是一个ring3->ring0->ring3的过程。
而KeUserModeCallback的过程是这样的
nt!KeUserModeCallback -> nt!KiCallUserMode -> nt!KiServiceExit -> ntdll!KiUserCallbackDispatcher -> 回调函数 -> int2B -> nt!KiCallbackReturn -> nt!KeUserModeCallback(调用后)
这是一个ring0->ring3->ring0的过程，在堆栈准备完毕后，借用KiServiceExit的力量回到了ring3，它的着陆点是KiUserCallbackDispatcher，然后KiUserCallbackDispatcher从PEB中取出KernelCallbackTable的基址，再以ApiIndex作为索引在这个表中查找对应的回调函数并调用，调用完之后再int2B触发nt!KiCallbackReturn再次进入内核，修正堆栈后跳回KeUserModeCallback，完成调用。
大致的流程是这样的，只要搞清楚传入的参数是如何使用并如何影响到最后的执行结果，就知道怎么去调用了。
需要注意的地方是：
(1) ring3的指令只能访问ring3的内存地址，所以如果需要访问数据，必须放在ring3可以访问到的内存中。所以在准备参数的过程中，通常是使用ZwAllocateVirtualMemory来申请ring3内存，其它方法如使用已有的内存，或放在栈中(_ClientLoadLibrary的方法)等都是可以的，只是访问数据时小有差别罢了。

(2)KeUserModeCallback的调用需要用户态栈的参与，因此内核线程无法调用KeUserModeCallback，必须在某个ring3线程的上下文中才可以。
下面进行具体的调试观察，以事实说话。

二、数据准备

调试环境:Vmware+WinXPSP2+Windbg
加载Callback.sys驱动，然后执行ring3部分的CallBackClient.exe，触发CallMessageBox调用~
这个函数先是一些准备工作，先获取当前进程的PEB并从中找到user32.dll的地址，然后GetProcAddress取得MessageBoxA的地址，这个没什么好说的。
这里取到的数据是:
user32.dll Base=0x77D10000
Address of MessageBoxA=0x77D5058A
KernelCallbackTable=0x77D12970
然后申请一块用户态内存，放入shellcode和需要的数据（这里是字符串），这样在ring3就可以访问了。
Alloced Buffer=0x00370000 //申请到的内存地址
然后把我们在ring3要执行的代码和数据放到申请的buffer里.

代码:

```
//开始填充缓冲区
*(ULONG*)pBuf=(ULONG)pBuf+sizeof(ULONG);//这里放的是一个shellcode的指针,shellcode将会被放在pBuf+4的位置
RtlCopyMemory(pBuf+4,(char*)CallBackStub,CALLSTUBLEN);//pBuf+4的位置开始是shellcode
RtlCopyMemory(pBuf+32,szText,strlen(szText));//从bufer+32开始拷贝字符参数
RtlCopyMemory(pBuf+32+strlen(szText)+1,szCaption,strlen(szCaption));//注意加1以便留出一个0x00作为sxText的结束符
缓冲区的填充并没有什么特别的要求。这里我排列好的样子是:
kd> db 0x00370000
00370000  04 00 37 00 8b 44 24 04-ff 70 10 ff 70 0c ff 70  ..7..D$..p..p..p
00370010  08 ff 70 04 ff 10 c2 08-00 00 00 00 00 00 00 00  ..p.............
00370020  4b 65 55 73 65 72 4d 6f-64 65 43 61 6c 6c 62 61  KeUserModeCallba
00370030  63 6b 3a 20 48 65 6c 6c-6f 20 66 72 6f 6d 20 72  ck: Hello from r
00370040  69 6e 67 30 20 21 21 21-00 52 69 6e 67 30 00 00  ing0 !!!.Ring0..
00370050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00370060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
00370070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

pBuf开头，存放的是shellcode的地址
pBuf+4， 是我们准备的一小段shellcode，地址为0x00370004
pBuf+0x20,是MessageBoxA的szText参数
pBuf+0x4A,是MessageBoxA的szCaption参数

三、参数准备

然后才是最关键的KeUserModeCallback的参数填写:
参数一:ApiIndex

代码:

```
ApiIndex=((ULONG)pBuf - KernelCallbackTable)/4;
```

为什么这么计算呢？因为ApiIndex本来的意义就是像它的名字所描述的那样，是在一个表中的索引。这个表就是KernelCallbackTable，KiUserCallbackDispatcher是这样取得回调函数的地址的:
CallbackFunPointer=KernelCallbackTable[ApiIndex]
所以我们计算Index的时候刚好相反，减去KernelCallbackTable，再除以4(也就是机器字长，指针的长度)，但是注意pBuf必须是按字长对齐的，否则经过逆运算之后无法得到原始值。
所以，ApiIndex=(0x00370000-0x77D12970)/4=0x221975a4
然后:

代码:

```
Arguments[0]=addrMessageBoxA;
Arguments[1]=0;//hWnd=NULL
Arguments[2]=(ULONG)pBuf+32;//szText
Arguments[3]=(ULONG)pBuf+32+strlen(szText)+1;//szCaption
Arguments[4]=0;//MB_OK
```

Arguments是我构造的一个数组，其第一个元素是MessageBoxA的地址，后面依次是MessageBoxA的各个参数,这此数据将被会shellcode所使用。
然后调用KeUserModeCallback，传入的ApiIndex即刚才计算得到的数据，InputBuffer就是构造的Arguments数组，InputBufferLen就是传入的数组的大小。
OutputBuffer和OutputBuffer似乎并没有被使用，只要传入有效的值就可以了。
具体数据：ApiIndex=0x221975a4,InputBufferLength=0x14

四、调用细节

接下来，我们进入KeUserModeCallback的内部过程(参考了wrk)，由于后面涉及的调用大多都不是标准的调用过程，所以必须密切关注栈的变化。

代码:

```
KeUserModeCallback()
{
  //只分析关键代码
  //从KPCR->CurrentThread->TrapFrame->HardwareEsp取得UserStack的地址，这里为0012fef0
  UserStack = KiGetUserModeStackAddress (); //注意该函数返回的是一个指向UserStack的指针
  //保存原始的UserStack，后面还要恢复
    OldStack = *UserStack;
    //接下来要往栈里放数据，计算新的栈顶,OldStack - InputLength就是新的栈顶了，再对齐一下，这里值为0012fedc
    NewStack = (OldStack - InputLength) & ~(__alignof(EXCEPTION_REGISTRATION_RECORD) - 1);
    //计算EXCEPTION_REGISTRATION_RECORD需要的Length，这里为KiCallUserMode做准备(xp和win2003此处略有不同,xp下
Length固定为0x10)
    Length = 4*sizeof(ULONG) + sizeof(EXCEPTION_REGISTRATION_RECORD);
    //从NewStack再向上Length大小的位置开始，验证是否可写，验证的地址为0012fecc 长度为
Length + InputLength= 0x10 + 0x14 = 0x24
    ProbeForWrite ((PCHAR)(NewStack - Length), Length + InputLength, sizeof(CHAR));
    //没有异常，将传入的数据拷入栈中，此时NewStack=OldStack - InputLength
  kd> r esi,edi,ecx
  esi=f6299bdc edi=0012fedc ecx=5
  kd> dd esi
  f6299bdc  77d5058a 00000000 00370020 00370049
  f6299bec  00000000 00001000 f6299b34 00000346
  kd> dd edi //NewStack
  0012fedc  00000018 00000000 0012ff1c 00000040
  0012feec  00000000 7c92d8ef 7c801671 000000a4
    RtlCopyMemory ((PVOID)NewStack, InputBuffer, InputLength);
    //
    // Push arguments onto user stack. Note space remains for the exception
    // registration record following the callback function arguments.
    //
    //0012fedc再减0x10，为0012fecc
    NewStack -= Length;
    *((PULONG)NewStack) = 0;
    *(((PULONG)NewStack) + 1) = ApiNumber;
   *(((PULONG)NewStack) + 2) = (ULONG)(NewStack+Length);
    *(((PULONG)NewStack) + 3) = (ULONG)InputLength;
  //保存新设置的NewStack到TrapFrame中
    *UserStack = NewStack;
    //调用KiCallUserMode
    Status = KiCallUserMode(OutputBuffer, OutputLength);
    //调用KiCallUserMode时用户栈的内容是这样的(esp=0x0012fecc):
    kd> dd 0012fecc
  0012fecc  00000000 221975a4 0012fedc 00000014
  0012fedc  77d5058a 00000000 00370020 00370049
  0012feec  00000000 7c92d8ef 7c801671 0000009c
  //KiCallUserMode的细节不多讲，下面省略
}
```

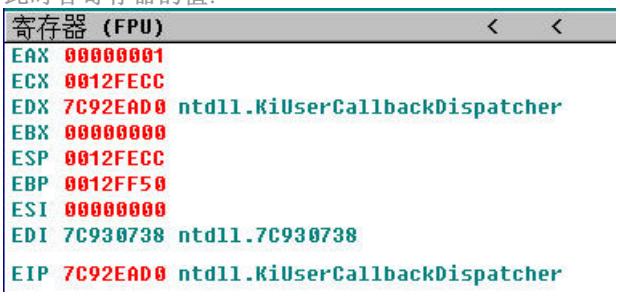
五、回到ring3继续执行

现在回到ring3了，ring3下还是习惯用OD，在KiUserCallbackDispatcher处下断(注意使用条件断点，否则无数飞向user32的调用将会把你淹没)。

此时栈的情况是这样的:

地址	数值	注释
0012FECC	00000000	
0012FED0	221975A4	
0012FED4	0012FEDC	
0012FED8	00000014	
0012FEDC	77D5058A	USER32.MessageBoxA
0012FEE0	00000000	
0012FEE4	00370020	ASCII "KeUserModeCallback: Hell
0012FEE8	00370049	ASCII "Ring0"
0012FEEC	00000000	
0012FEF0	7C92D8EF	返回到 ntdll.7C92D8EF
0012FEF4	7C801671	返回到 kernel32.7C801671 来自 n
0012FEF8	000000A4	
0012FEFC	00000000	

此时各寄存器的值:



寄存器 (FPU)

```
EAX 00000001
ECX 0012FECC
EDX 7C92EAD0 ntdll.KiUserCallbackDispatcher
EBX 00000000
ESP 0012FECC
EBP 0012FF50
ESI 00000000
EDI 7C930738 ntdll.7C930738

EIP 7C92EAD0 ntdll.KiUserCallbackDispatcher
```

可以看到，这时esp=0x12fecc，正是KiCallUserMode之前UserESP的值，栈中的数据和KiCallUserMode时用户栈的内容是一样的.

代码:

```
ntdll!KiUserCallbackDispatcher
7C92EAD0 >  83C4 04               add esp,4  //跳过栈中第一个数据，然后栈顶就是ApiIndex了
7C92EAD3    5A                    pop edx //ApiIndex到edx中
7C92EAD4    64:A1 18000000        mov eax,dword ptr fs:[18] //TEB->NtTib.Self，指向TEB本身，放到eax中
7C92EADA    8B40 30               mov eax,dword ptr ds:[eax+30] //TEB偏移0x30处即PEB，放到eax中
7C92EADD    8B40 2C               mov eax,dword ptr ds:[eax+2C] //PEB偏移0x2C处即KernelCallbackTable，放
到eax中
7C92EAE0    FF1490                call dword ptr ds:[eax+edx*4] //查表并调用
7C92EAE3    33C9                  xor ecx,ecx
7C92EAE5    33D2                  xor edx,edx
7C92EAE7    CD 2B                 int 2B
```

在这里：call dword ptr ds:[eax+edx*4]
eax即KernelCallbackTable,edx即ApiIndex，这只是一个简单的查表动作
这个KernelCallbackTable是给use32.dll用的，可以观察一下(省略了很多内容):

代码:

```
77d12970  77d3f534 USER32!__fnCOPYDATA
77d12974  77d583ac USER32!__fnCOPYGLOBALDATA
...
77d12a5c  77d3f5cb USER32!__ClientFreeLibrary
77d12a60  77d3a3fc USER32!__ClientGetCharsetInfo
77d12a64  77d58a7c USER32!__ClientGetDDEFlags
77d12a68  77d58bd5 USER32!__ClientGetDDEHookData
77d12a6c  77d4f715 USER32!__ClientGetListboxString
77d12a70  77d365aa USER32!__ClientGetMessageMPH
77d12a74  77d3aa6d USER32!__ClientLoadImage
77d12a78  77d3dc84 USER32!__ClientLoadLibrary
...
77d12af0  77d590c5 USER32!__fnOUTLPCOMBOBOXINFO
77d12af4  77d59105 USER32!__fnOUTLPSCROLLBARINFO
```

这里面就有大家比较熟悉的全局钩子相关的USER32!__ClientLoadLibrary和USER32!__ClientFreeLibrary
也就是说，KiUserCallbackDispatcher仅仅是按照索引从这个表中取出对应的函数的地址并调用，这就是参数名称为什么叫做ApiIndex.
我们前面计算ApiIndex的过程就是这个过程的逆过程，所以计算结果为:
FunPointer=77d12970 + 0x221975a4 * 4 = 0x00370000 //(实际上高位发生了溢出，但是并不影响)
所以这里实际上就是call dword ptr ds:[0x00370000]
0x00370000是什么呢？就是我们前面申请的内存的地址

而0x00370000里的值是0x00370004，也就是我们放置的shellcode的地址,这样就call到我们的shellcode去了~~
如图:

```
ds:[00370000]=00370004
```

地址	十六进制	ASCII
00370000	04 00 37 00 8B 44 24 04 FF 70 10 FF 70 0C FF 70	♦.7.嫦$ ÿp►ÿp.ÿp
00370010	08 FF 70 04 FF 10 C2 08 00 00 00 00 00 00 00 00	◘ÿp ÿ►?........
00370020	4B 65 55 73 65 72 4D 6F 64 65 43 61 6C 6C 62 61	KeUserModeCallba
00370030	63 6B 3A 20 48 65 6C 6C 6F 20 66 72 6F 6D 20 72	ck: Hello from r
00370040	69 6E 67 30 20 21 21 21 00 52 69 6E 67 30 00 00	ing0 !!!.Ring0..

shellcode:

地址	十六进制	反汇编
00370004	8B4424 04	mov eax,dword ptr ss:[esp+4]
00370008	FF70 10	push dword ptr ds:[eax+10]
0037000B	FF70 0C	push dword ptr ds:[eax+C]
0037000E	FF70 08	push dword ptr ds:[eax+8]
00370011	FF70 04	push dword ptr ds:[eax+4]
00370014	FF10	call dword ptr ds:[eax]
00370016	C2 0800	retn 8

现在对于前面为何那样放置数据比较清楚了吧~

六、执行指定代码

下面是准备执行shellcode的时候的栈中的数据：

地址	数值	注释
0012FED0	7C92EAE3	返回到 ntdll.7C92EAE3
0012FED4	0012FEDC	
0012FED8	00000014	
0012FEDC	77D5058A	USER32.MessageBoxA
0012FEE0	00000000	
0012FEE4	00370020	ASCII "KeUserModeCallback: Hell
0012FEE8	00370049	ASCII "Ring0"
0012FEEC	00000000	
0012FEF0	7C92D8EF	返回到 ntdll.7C92D8EF
0012FEF4	7C801671	返回到 kernel32.7C801671 来自 n
0012FEF8	000000A4	

简单分析：此时栈顶是返回地址，然后依次存放的是复制参数时的UserStack,然后是InputBufferLength,再往下就是我们传入的Arguments数组的内容了，此时栈中的数据已经非常明了，所以shellcode为什么这样写一点也不用奇怪了~~

代码:

```
//[esp]是返回地址，返回到KiUserCallbackDispatcher中
00370004    8B4424 04               mov eax,dword ptr ss:[esp+4] //eax就是KeUserModeCallback中NewStack的
值，指向我们传入的InputBuffer
00370008    FF70 10                 push dword ptr ds:[eax+10] //第四个参数入栈
0037000B    FF70 0C                 push dword ptr ds:[eax+C]  //第三个参数入栈
0037000E    FF70 08                 push dword ptr ds:[eax+8]  //第二个参数入栈
00370011    FF70 04                 push dword ptr ds:[eax+4]  //第一个参数入栈
00370014    FF10                    call dword ptr ds:[eax]    //[eax]就是MessageBoxA，执行到这
里，MessageBox弹出来了~
00370016    C2 0800                 retn 8 //这个ret 8是固定的
```

为什么要有shellcode?
对于win32k发起的KeUserModeCallback，因为它在ring3有人接头(KernelCallbackTable中的各个函数)，所以只需要传入ApiIndex和InputBuffer相关的就可以了。而我们自己发起的KeUserModeCallback则没有接头人，所以必须自己操办一切，在ring0找好接头人，然后回调过去让它办事。
shellcode一定要这么写吗？
不是的。事实上这个shellcode的写法并不好，只能处理固定的四个参数，若参数不是四个将会出现错误。具体shellcode如何写，取决于栈中的数据排列和数据的意义。只要能正确使用数据并返回，shellcode怎么写随你的便~~甚至你可以直接修改KiUserCallbackDispatcher的代码，那么怎么调用你说了算(比如发现是全局钩子回调节器的ApiIndex,也就是0x42就咔嚓掉啊~~).

七、返回内核

执行完回调函数(在这里就是我们的shellcode)返回到KiUserCallbackDispatcher中,ecx和edx清零后，int2B返回到内核nt!KiCallbackReturn处，随后根据之前保存的环境和数据返回到KeUserModeCallback，KeUserModeCallback再恢复TrapFrame->HardwareEsp，于是就像什么都没有发生过一样~~
实际上由于call之后的操作很简单，所以我们也可以自己实现
add esp,8
int 2B
这样就回到内核了，中间的细节不多讲了，不是今天主题~~

关于回调函数的返回值：
在int2B时，回调函数的返回值在eax中，刚回到KiCallBackReturn时，eax不变，这个返回值一值被保存至返回到

KeUserModeCallBack返回，所以KeUserModeCallback的返回值也就是回调函数的返回值。即如果你回调的是OpenProcess，那么KeUserModeCallback的返回值就是ProcessHandle~~

一句话总结： KeUserModeCallback就是把参数放在栈中然后借用KiServiceExit回到ring3的KiUserCallbackDispatcher后得出回调函数地址并执行(参数在栈中)，然后又int2B回到ring0。

其它事项：
(1)KeUserModeCallback的调用是一个非常“精密”的过程，所以必须小心填写参数.
(2)由于KernelCallbackTable是专为User32.dll使用的，所以如果一个进程未加载user32.dll，那么PEB->KernelCallbackTable将为NULL.此时上面的代码将无法工作，但是我们可以自己构造一个表并填充这个域~
(3)纯内核线程无法调用KeUserModeCallback，因为它没有UserStack~
所以，想在DriverEntry中Attach到别的进程使用KeUserModeCallback是不可能成功的，但是我们可以Hook某个地方，然后等目标进程到来的时候，借用它的线程来实现调用~

上传的附件

Ring0MessageBox_Src.rar