从内核在 WOW64 进程中执行用户态 shellcode

By xSpy /vxjump.net
2014-07-30

首先说说 KeUserModeCallback

之前做某个项目时,用到了 KeUserModeCallback 的方式执行用户态的 shellcode.
当时遗留了一个问题,关于 wow64 下,KeUserModeCallback 的分发没有搞清楚.
导致只能执行原生的 shellcode.
普通的 32 位和 64 位模式下原生的 KeUserModeCallback 执行 ring3 代码大家都不陌生了.
详情可见俄国人的 ring0MsgBox.

但是如果内核所在的进程空间是一个 wow64 的进程,就有点麻烦了.
在内核 nt! KeUserModeCallback
-> nt! KiCallUserMode
进入 Ring3 之后,首先到达的分发函数是 ntdll64!KiUserCallbackDispatcher
虽然是 WOW64 进程,但是当前的 CPU 模式是 x64!是没法直接执行我们的 x86 shellcode 的.

```
.text:0000000078E9FDD6                  ; NTSTATUS __stdcall KiUserCallbackDispatcher(ULONG Index, PVOID Argument,
ULONG ArgumentLength)
```

```
.text:0000000078E9FDD6                                  public KiUserCallbackDispatcher
.text:0000000078E9FDD6                  KiUserCallbackDispatcher proc near      ; DATA XREF: .rdata:off_78F58128o
.text:0000000078E9FDD6
.text:0000000078E9FDD6                  arg_18          = qword ptr  20h
.text:0000000078E9FDD6                  arg_20          = dword ptr  28h
.text:0000000078E9FDD6                  arg_24          = dword ptr  2Ch
.text:0000000078E9FDD6
.text:0000000078E9FDD6 48 8B 4C 24 20                     mov     rcx, [rsp+arg_18]
.text:0000000078E9FDDB 8B 54 24 28                        mov     edx, [rsp+arg_20]
.text:0000000078E9FDDF 44 8B 44 24 2C                     mov     r8d, [rsp+arg_24]
.text:0000000078E9FDE4 65 48 8B 04 25 60 00 00+             mov     rax, gs:60h     ; rax = PEB
.text:0000000078E9FDED 4C 8B 48 58                        mov     r9, [rax+58h]   ; r9 = KernelCallbackTable
.text:0000000078E9FDF1 43 FF 14 C1                        call    qword ptr [r9+r8*8]
.text:0000000078E9FDF1                  KiUserCallbackDispatcher endp ; sp-analysis failed
```

但是系统的自带的功能是可以正常切换到 wow64 模式的,比如 user32!_ClientLoadLibrary.
简单 IDA 之后,发现系统是存在一个模式切换的过程.

以 ClientLoadLibrary 为例.
首先,到达 ntdll64!KiUserCallbackDispatcher 之后, `call qword ptr [r9+r8*8]` 这里的地址是 wow64win !whcbClientLoadLibrary 这里都是 x64 的代码.

```
.text:0000000078BA92C3 41 B9 28 00 00 00                  mov     r9d, 28h                ;参数的大小
.text:0000000078BA92C9 45 8B C2                           mov     r8d, r10d               ;参数的地址
.text:0000000078BA92CC 41 8D 51 19                        lea     edx, [r9+19h]           ; Wow64模式的ApiIndex
.text:0000000078BA92D0 48 8D 4C 24 30                     lea     rcx, [rsp+9F8h+pContext] ;线程的上下文
```

```
.text:0000000078BA92D5 E8 2C 69 00 00                    call    Wow64KiUserCallbackDispatcher
```

wow64win !whcbClientLoadLibrary 内部继续分发了 wow64!Wow64KiUserCallbackDispatcher

```
    void Wow64KiUserCallbackDispatcher(
OUT PCONTEXT Context,    //线程上下文,需要外部分配内存
IN LONG ApiIndex,    //wow64的KernelCallbackTable的ApiIndex
IN PVOID pParam,
IN ULONG ParamSize
);
```

wow64!Wow64KiUserCallbackDispatcher 中会复制参数到之前的 context 对应的栈上,
这个 context 已经是将来要使用的 wow64 模式 的 context

```
.text:0000000078BD8A29 65 48 8B 04 25 30 00 00+          mov     rax, gs:30h
.text:0000000078BD8A32 48 05 00 20 00 00                 add     rax, 2000h
.text:0000000078BD8A38 48 89 44 24 30                    mov     [rsp+348h+var_318], rax
.text:0000000078BD8A3D 48 8B 44 24 30                    mov     rax, [rsp+348h+var_318]
.text:0000000078BD8A42 8B 00                             mov     eax, [rax]
.text:0000000078BD8A44 89 84 24 18 03 00 00              mov     [rsp+348h+var_30], eax
.text:0000000078BD8A4B E8 D8 46 00 00                    call    RunCpuSimulation
```

这里的 wow64!RunCpuSimulation 只是一个stub,会跳转到 wow64cpu!CpuSimulate

```
.text:0000000078B625B0                                   public CpuSimulate
```

```
.text:0000000078B625B0                  CpuSimulate     proc near               ; DATA XREF: .text:off_78B63168o
.text:0000000078B625B0                                                          ; .pdata:0000000078B650B4o
.text:0000000078B625B0
.text:0000000078B625B0                  var_B8          = qword ptr -0B8h
.text:0000000078B625B0                  var_B0          = word ptr -0B0h
.text:0000000078B625B0                  var_A8          = dword ptr -0A8h
.text:0000000078B625B0                  var_A0          = qword ptr -0A0h
.text:0000000078B625B0                  var_98          = word ptr -98h
.text:0000000078B625B0                  var_48          = byte ptr -48h
.text:0000000078B625B0                  var_40          = qword ptr -40h
.text:0000000078B625B0                  var_38          = qword ptr -38h
.text:0000000078B625B0                  var_30          = qword ptr -30h
.text:0000000078B625B0                  var_28          = qword ptr -28h
.text:0000000078B625B0                  var_20          = qword ptr -20h
.text:0000000078B625B0                  var_18          = qword ptr -18h
.text:0000000078B625B0                  var_10          = qword ptr -10h
.text:0000000078B625B0                  var_8           = qword ptr -8
.text:0000000078B625B0
.text:0000000078B625B0 48 81 EC B8 00 00 00             sub     rsp, 0B8h
.text:0000000078B625B7 48 89 6C 24 78                   mov     [rsp+0B8h+var_40], rbp
.text:0000000078B625BC 48 89 BC 24 80 00 00 00          mov     [rsp+0B8h+var_38], rdi
.text:0000000078B625C4 48 89 B4 24 88 00 00 00          mov     [rsp+0B8h+var_30], rsi
.text:0000000078B625CC 48 89 9C 24 90 00 00 00          mov     [rsp+0B8h+var_28], rbx
.text:0000000078B625D4 4C 89 A4 24 98 00 00 00          mov     [rsp+0B8h+var_20], r12
.text:0000000078B625DC 4C 89 AC 24 A0 00 00 00          mov     [rsp+0B8h+var_18], r13
.text:0000000078B625E4 4C 89 B4 24 A8 00 00 00          mov     [rsp+0B8h+var_10], r14
```

```
.text:0000000078B625EC 4C 89 BC 24 B0 00 00 00              mov     [rsp+0B8h+var_8], r15
.text:0000000078B625F4 4C 8D 74 24 70                      lea     r14, [rsp+0B8h+var_48]
.text:0000000078B625F9 65 4C 8B 24 25 30 00 00 00+         mov     r12, gs:30h
.text:0000000078B62602 4C 8D 3D 47 FE FF FF                lea     r15, off_78B62450
.text:0000000078B62609 4D 8B AC 24 88 14 00 00             mov     r13, [r12+1488h]
.text:0000000078B62611
.text:0000000078B62611                          loc_78B62611:                           ; CODE XREF:
   ;TurboDispatchJumpAddressEnd+34j
.text:0000000078B62611 41 83 A5 D0 02 00 00 01             and     dword ptr [r13+2D0h], 1
.text:0000000078B62619 0F 84 AF 00 00 00                   jz      loc_78B626CE
.text:0000000078B6261F 41 0F 28 85 70 01 00 00             movaps  xmm0, xmmword ptr [r13+170h]      ;x64模式!
.text:0000000078B62627 41 0F 28 8D 80 01 00 00             movaps  xmm1, xmmword ptr [r13+180h]
.text:0000000078B6262F 41 0F 28 95 90 01 00 00             movaps  xmm2, xmmword ptr [r13+190h]
.text:0000000078B62637 41 0F 28 9D A0 01 00 00             movaps  xmm3, xmmword ptr [r13+1A0h]
.text:0000000078B6263F 41 0F 28 A5 B0 01 00 00             movaps  xmm4, xmmword ptr [r13+1B0h]
.text:0000000078B62647 41 0F 28 AD C0 01 00 00             movaps  xmm5, xmmword ptr [r13+1C0h]
.text:0000000078B6264F 41 8B 8D B0 00 00 00                mov     ecx, [r13+0B0h]
.text:0000000078B62656 41 8B 95 AC 00 00 00                mov     edx, [r13+0ACh]
.text:0000000078B6265D 41 83 A5 D0 02 00 00 FE             and     dword ptr [r13+2D0h], 0FFFFFFFEh
.text:0000000078B62665 41 8B BD A0 00 00 00                mov     edi, [r13+0A0h]
.text:0000000078B6266C 41 8B B5 A4 00 00 00                mov     esi, [r13+0A4h]
.text:0000000078B62673 41 8B 9D A8 00 00 00                mov     ebx, [r13+0A8h]
.text:0000000078B6267A 41 8B AD B8 00 00 00                mov     ebp, [r13+0B8h]
.text:0000000078B62681 41 8B 85 B4 00 00 00                mov     eax, [r13+0B4h]
.text:0000000078B62688 49 89 A4 24 80 14 00 00             mov     [r12+1480h], rsp
.text:0000000078B62690 66 C7 44 24 08 23 00                mov     [rsp+0B8h+var_B0], 23h
```

```
.text:0000000078B62697 66 C7 44 24 20 2B 00                mov     [rsp+0B8h+var_98], 2Bh
.text:0000000078B6269E 45 8B 85 C4 00 00 00                mov     r8d, [r13+0C4h]
.text:0000000078B626A5 41 81 A5 C4 00 00 00 FF+            and     dword ptr [r13+0C4h], 0FFFFFEFFh
.text:0000000078B626B0 44 89 44 24 10                     mov     [rsp+0B8h+var_A8], r8d
.text:0000000078B626B5 45 8B 85 C8 00 00 00                mov     r8d, [r13+0C8h]
.text:0000000078B626BC 4C 89 44 24 18                     mov     [rsp+0B8h+var_A0], r8
.text:0000000078B626C1 45 8B 85 BC 00 00 00                mov     r8d, [r13+0BCh]
.text:0000000078B626C8 4C 89 04 24                        mov     [rsp+0B8h+var_B8], r8
.text:0000000078B626CC 48 CF                              iretq           ;返回到x64
.text:0000000078B626CE                          ;
-------------------------------------------------------------------------------
.text:0000000078B626CE
.text:0000000078B626CE                          loc_78B626CE:                           ;x86模式
.text:0000000078B626CE 41 8B BD A0 00 00 00                mov     edi, [r13+0A0h]
.text:0000000078B626D5 41 8B B5 A4 00 00 00                mov     esi, [r13+0A4h]
.text:0000000078B626DC 41 8B 9D A8 00 00 00                mov     ebx, [r13+0A8h]
.text:0000000078B626E3 41 8B AD B8 00 00 00                mov     ebp, [r13+0B8h]
.text:0000000078B626EA 41 8B 85 B4 00 00 00                mov     eax, [r13+0B4h]
.text:0000000078B626F1 49 89 A4 24 80 14 00 00             mov     [r12+1480h], rsp
.text:0000000078B626F9 41 C7 46 04 23 00 00 00             mov     dword ptr [r14+4], 23h
.text:0000000078B62701 41 B8 2B 00 00 00                   mov     r8d, 2Bh
.text:0000000078B62707 41 8E D0                           mov     ss, r8w
.text:0000000078B6270A                                    assume ss:nothing
.text:0000000078B6270A 41 8B A5 C8 00 00 00                mov     esp, [r13+0C8h]
.text:0000000078B62711 45 8B 8D BC 00 00 00                mov     r9d, [r13+0BCh]
.text:0000000078B62718 45 89 0E                           mov     [r14], r9d
```

```
.text:0000000078B6271B 41 FF 2E                    jmp     fword ptr [r14] ;这个大跳会切换到x86模式!
```

这个函数主要就是从之前获取到的 context 里初始化各个通用寄存器.然后根据想要模拟的 CPU 环境进行跳转.

OK.到这里,整个大概的分发流程已经清楚了, 实现我们的需求就很简单了,模拟一次即可以了.

要执行 wow64 模式的 shellcode,
也需要构造这些 stub.

一共是 2 个 stub

1. PsGetProcessPeb 返回的原生的 PEB.
   添加 KeCallbackTable 的 entry 指向我们的 wow64 stub.

2. PsGetProcessWow64Process 获取到 wow64 模式的 PEB,
   然后定位到相应的 WOW64 KeCallbackTable ,添加 entry 指向需要执行的 x86 shellcode.

wow64 模式的 APC

相对来说就好多了.只有一个简单的变换

```
text:0000000078E9FCD0                              public KiUserApcDispatcher
```

```
.text:0000000078E9FCD0                  KiUserApcDispatcher:                ; CODE XREF: .text:0000000078E9FD07j
.text:0000000078E9FCD0                                                      ; DATA XREF: .rdata:off_78F58128o
.text:0000000078E9FCD0 48 8B 4C 24 18                mov     rcx, [rsp+18h]
.text:0000000078E9FCD5 48 8B C1                      mov     rax, rcx
.text:0000000078E9FCD8 4C 8B CC                      mov     r9, rsp
.text:0000000078E9FCDB 48 C1 F9 02                   sar     rcx, 2
.text:0000000078E9FCDF 48 8B 54 24 08                mov     rdx, [rsp+8]
.text:0000000078E9FCE4 48 F7 D9                      neg     rcx
.text:0000000078E9FCE7 4C 8B 44 24 10                mov     r8, [rsp+10h]
.text:0000000078E9FCEC 48 0F A4 C9 20                shld    rcx, rcx, 20h
.text:0000000078E9FCF1 85 C9                         test    ecx, ecx
.text:0000000078E9FCF3 74 22                         jz      short loc_78E9FD17
.text:0000000078E9FCF5 48 8B 0C 24                   mov     rcx, [rsp]
.text:0000000078E9FCF9 FF D0                         call    rax
```

红色标记的地方.,ntdll64! `KiUserApcDispatcher` 对应我们传递的 `Apc` 例程地址做了一个小变化,我们做逆变换即可.

```
#ifdef _WIN64
        if (bWow64)
        {
            //Neg rax
            //shl rax,2
            pFixStubAddr = (void*) NEG( (LONG)pFixStubAddr);
            pFixStubAddr = (void*) SHL( (LONG)pFixStubAddr,2);
        }
#endif
```

//附相关代码片段.

```
//支持WOW64的KeUserModeCallback , pStartAddress  是用户态地址
NTSTATUS UserModeCallback(IN void* pStartAddress)
{
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    PVOID       pKernelCallbackTable = NULL;
    PEB*        pPEB = NULL;
    UCHAR*          pEntryBuffer = NULL;
    ULONG       ulEntryBufferSize = 0;
    UCHAR*          pEntry = NULL;
    LONG        ApiIndex = 0;

    ULONG_PTR           Dummy_Input = 0;
    ULONG_PTR           Dummy_Output = 0;
    ULONG_PTR           Dummy_OutSize = sizeof(ULONG_PTR);


#ifdef _WIN64
    BOOLEAN         bWow64 = FALSE;
    PEB32*      pWow64_PEB = NULL;
    PVOID       pWow64_KernelCallbackTable = NULL;
    UCHAR*          pWow64Stub = NULL;
```

```
    ULONG       ulWow64StubSize = 0;
    UCHAR*          pWow64_EntryBuffer = NULL;
    UCHAR*          pWow64_Entry = NULL;
    LONG        nWow64_ApiIndex = 0;
#endif

    LARGE_INTEGER liStart = {0};
    LARGE_INTEGER liStop = {0};

    do
    {
        #ifdef _WIN64
            kIsWow64Process(PsGetCurrentProcess(),&bWow64);

            if (bWow64)
            {
                pWow64_PEB = (PEB32*)PsGetProcessWow64Process(PsGetCurrentProcess());
                if (NULL == pWow64_PEB)
                {
                    xDebugA(("[-] wow64 PEB is NULL ! \n"));
                    break;
                }
                pWow64_KernelCallbackTable = (PVOID)pWow64_PEB->KernelCallbackTable;
                if (NULL == pWow64_KernelCallbackTable)
                {
                    xDebugA(("[-] Wow64 KernelCallbackTable is NULL ! \n"));
```

```
        break;
    }

    ulWow64StubSize = sizeof(_KeUserModeCallback_wow64_stub);

    //这个stub这里没法释放
    Status = kVirtualAllocEx(PsGetCurrentProcess(),
        &pWow64Stub,
        ulWow64StubSize,
        MEM_COMMIT|MEM_RESERVE,
        PAGE_EXECUTE_READWRITE
    );

    if (!NT_SUCCESS(Status))
    {
        break;
    }

    RtlCopyMemory(pWow64Stub,
        _KeUserModeCallback_wow64_stub,
        ulWow64StubSize
    );

    ulEntryBufferSize = sizeof(ULONG_PTR) + MAX_INSTRUCTION + sizeof(ULONG_PTR);

    //靠近这个地址的
```

```
pWow64_EntryBuffer = pWow64_KernelCallbackTable;


KeQueryTickCount(&liStart);

//需要分配 靠近 Wow64_KernelCallbackTable 这个地址的用户态内存
//必须在2G范围内.
Status = kVirtualAllocNear(
    PsGetCurrentProcess(),
    &pWow64_EntryBuffer,
    ulEntryBufferSize,
    SIZE_GB,
    MEM_COMMIT|MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);

if (!NT_SUCCESS(Status))
{
    xDebugA(("[-] 找不到近地址! \n"));
    break;
}

KeQueryTickCount(&liStop);


xDebugA(("[*] 搜索耗时   %d ms ,在 0x%p 附近找到 0x%p \n" ,
```

```
        (LONG)(liStop.QuadPart - liStart.QuadPart) ,
        pWow64_KernelCallbackTable,
        pWow64_EntryBuffer

        )) ;


    //对齐
    pWow64_Entry = (UCHAR*)Align(pWow64_EntryBuffer, sizeof(ULONG) );

    //Entry  的地址设置为pStartAddress
    SETULONG32(pWow64_Entry,0,pStartAddress);

    //计算ApiIndex
    nWow64_ApiIndex = CalcKeUserModeCallbackApiIndex32(pWow64_Entry,pWow64_KernelCallbackTable);

    //填充nWow64_ApiIndex
    ReplaceULONG32(pWow64Stub,ulWow64StubSize,0x11111111,(ULONG)nWow64_ApiIndex,FALSE);

    xDebugA(("[*] pWow64Stub = 0x%p ,nWow64_ApiIndex = %d , pStartAddress = 0x%p \n",

        pWow64Stub,
        nWow64_ApiIndex,
        pStartAddress
        ));
}
```

```
#endif

//如果是WOW64模式,首先执行的是x64的,需要跳转
pPEB = PsGetProcessPeb(PsGetCurrentProcess());
if (NULL == pPEB)
{
	xDebugA(("[-] peb is null ! \n"));
	break;
}
pKernelCallbackTable = pPEB->KernelCallbackTable;

if ( (NULL == pKernelCallbackTable) || (!IS_VALID_USER_PTR(pKernelCallbackTable)) )
{
	xDebugA(("[-]获取  KernelCallbackTable  失败! \n"));
	break;
}

////////////////////////////////////////////////////////////////////////
ulEntryBufferSize = sizeof(ULONG_PTR) + MAX_INSTRUCTION + sizeof(ULONG_PTR);

//为添加的表向
pEntryBuffer = pKernelCallbackTable;

//pStub  这段内存这里没有机会释放.
liStart.QuadPart = 0;
```

```
KeQueryTickCount(&liStart);

//需要分配 靠近 KernelCallbackTable这个地址的用户态内存
//必须在2G范围内.
Status = kVirtualAllocNear(
    PsGetCurrentProcess(),
    &pEntryBuffer,
    ulEntryBufferSize,
    SIZE_GB,
    MEM_COMMIT|MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);

liStop.QuadPart = 0;
KeQueryTickCount(&liStop);
xDebugA(("[*] 搜索耗时   %d ms ,在  0x%p  附近找到  0x%p \n" ,
    (LONG)(liStop.QuadPart - liStart.QuadPart) ,
    pKernelCallbackTable,
    pEntryBuffer
    )) ;


if (!NT_SUCCESS(Status))
{
    xDebugA(("[-] 内存分配失败! \n"));
    break;
```

```
        }


        //对齐
        pEntry = (UCHAR*)Align(pEntryBuffer, sizeof(ULONG_PTR) );

        ApiIndex = CalcKeUserModeCallbackApiIndex(pEntry,pKernelCallbackTable);


#ifdef _WIN64

    if (bWow64)
    {
        //我们的Stub的地址
        SETULONG_PTR(pEntry,0,pWow64Stub);      //
    }
    else
    {
        //entry的值直接就是想要执行的地址
        SETULONG_PTR(pEntry,0,pStartAddress);
    }
#else

    //entry的值直接就是想要执行的地址里取一个地址
    SETULONG_PTR(pEntry,0,pStartAddress);
```

```
#endif

        __try
        {
            Dummy_OutSize = sizeof(ULONG_PTR);

            //shellcode返回了,这个函数才会返回
            KeUserModeCallback(ApiIndex,
                (PVOID*)&Dummy_Input,
                sizeof(Dummy_Input),
                (PVOID*)&Dummy_Output,
                &Dummy_OutSize
                );

            Status = STATUS_SUCCESS;
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            Status = STATUS_UNSUCCESSFUL;

            xDebugA(("[-] Call Ring3  发生异常! %d \n" , GetExceptionCode() ));
        }

    } while (FALSE);
```

```
    if (NULL != pEntryBuffer)
    {
        kVirtualFree(PsGetCurrentProcess(),pEntryBuffer);
        pEntryBuffer = NULL;
    }

#ifdef _WIN64

    if (NULL != pWow64_EntryBuffer)
    {
        kVirtualFree(PsGetCurrentProcess(),pWow64_EntryBuffer);
        pWow64_EntryBuffer = NULL;
    }

#endif

    return Status;
}
```

xSpy@BinVul.com
xSpy@Vxjump.net
20170724