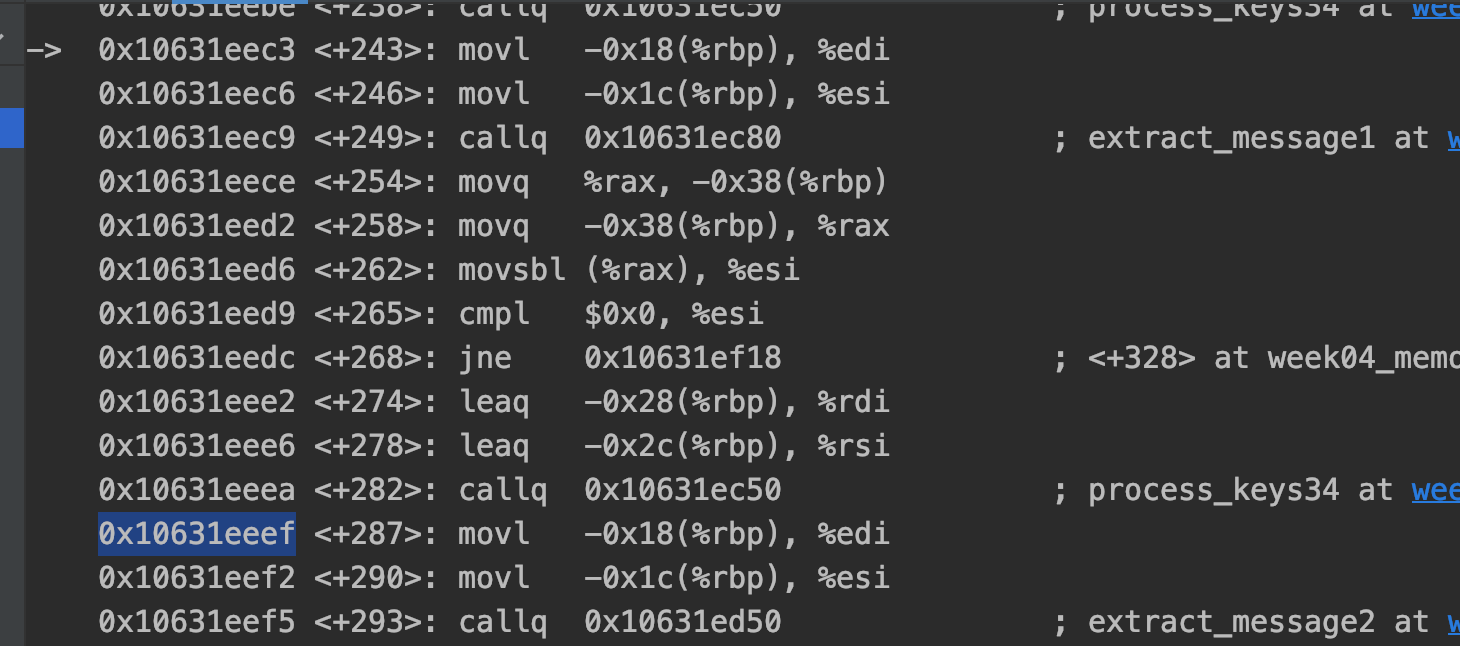
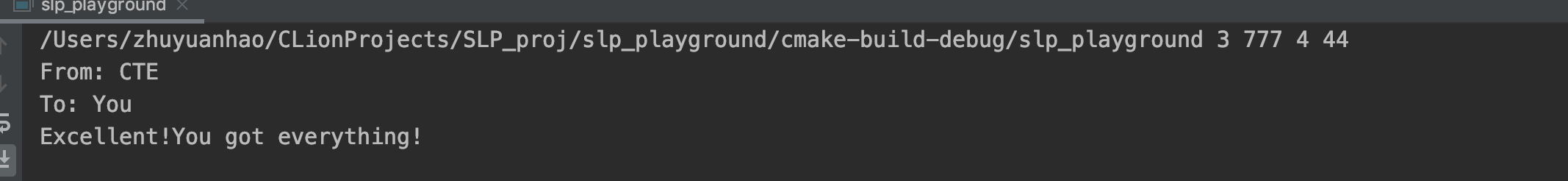
1. key3 key4 解体思路，首先看到processkey34这个函数，我们可以知道这个函数是将key3这个传入的指针的地址向地址高位移动key3指针取值的值这么多个单位的整形指针偏移量，然后将这个地址中的值加上key4取值的值，我们从stackframe的知识中可以知道，一个函数的栈最开头的地方是caller函数给压入的callee函数执行完成后回到caller函数执行的地址，所以自然想到通过更改processkey34函数栈中回到main的执行地址，直接跳过if语句而去执行extractinfo2，而我们在process34函数栈中可以看到，key3 距离返回main函数的地址10631eec3正好是三个整形的距离，所以key3是3，

然后调出main函数的反汇编可以看到我们想要跳到的extractmessage2函数所在的指令地址是0x10631eeef ef-c3=2c 换算成十进制也就是44，所以key4是44



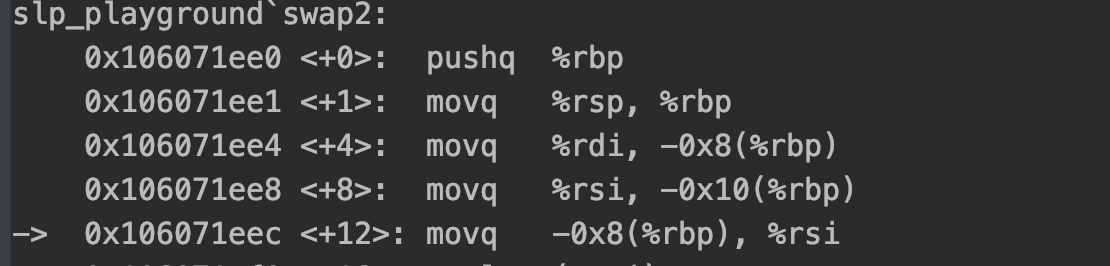
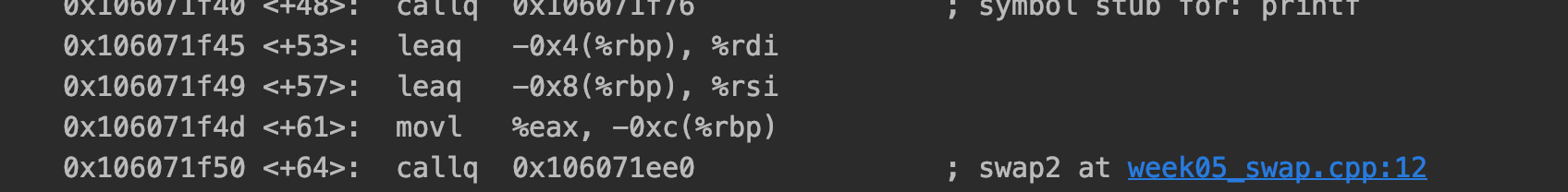
最后运行结果：



## 当程序执行swap2中的temp=\*x;指令时程序的stackframe结构

操作平台：macOS 编译器信息





首先查看反汇编代码，发现main函数在调用函数的时候不保存寄存器，并且传递参数的方式是直接通过寄存器保存参数然后被调函数去寄存器中直接获取，与vc6.0的push寄存器到栈中有稍微的不同。最后由此得到在执行temp = \*x指令时程序的栈帧状态结构为如下

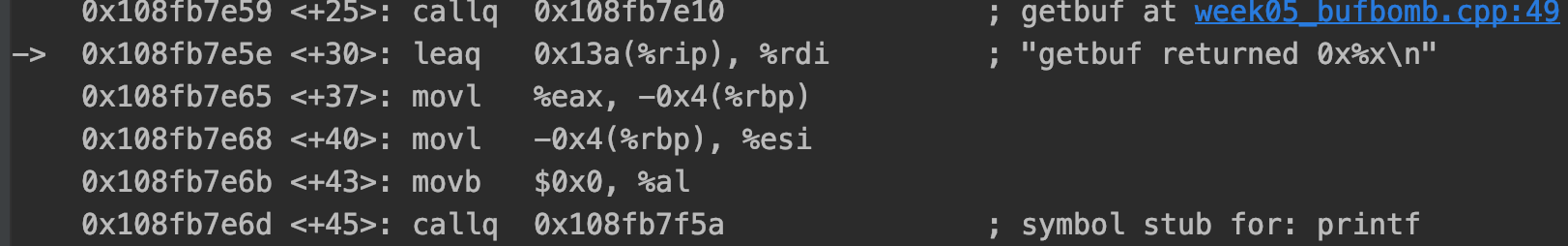
（下方是高地址）

|  |
| --- |
| swap函数的局部变量：传入参数指针x和指针y，以及声明的本地变量temp |
| swap函数保存的bp：c0e5b8e9 fe7f0000  在main函数中时，我们查看rbp的值可以发现就是上述这个地址 |
| main函数的返回地址：551f0706 01000000  从反汇编可以看到这个地址对应的是： |
| main函数的局部变量：a，b 以及空闲一些空闲空间，因为在开辟栈的时候为main函数开辟了16个字节的局部变量空间 |
| main函数保存的bp：d8e5b8e9fe7f0000 |

## BufBomb

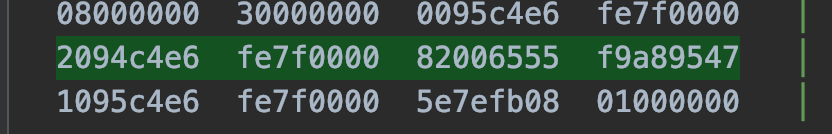
攻击思路：通过更改printf的传入参数来使得打印的内容变成deadbeef而不是0x1

首先观察test的反汇编

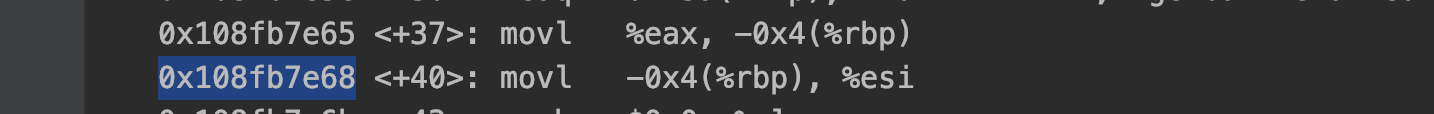


这一段当中在getbuf返回之后把eax的值先是给了val，也就是movl %eax，-0x4(%rbp)然后才将val的值传给printf，也就是movl -0x4(%rbp)，%esi，所以我们可以通过更改getbuf的返回地址的方式让他直接跳过将getbuf的返回值给到val这个步骤而直接将val的值传递给printf，同时更改val的值为deadbeef即可达到攻击效果。

首先我们从栈的结构得知，buf这个数组的前面还有getbuf保存的ebp，然后才到返回地址，而ebp对于每一次程序运行来说不同，所以每一次的攻击输入的字符要通过寄存器查看ebp的值得到

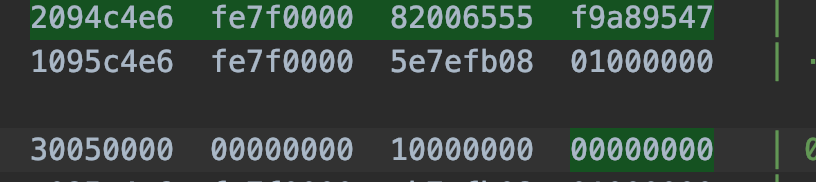


在这个地方我们可以看到返回地址是108fb7e5e,ebp的值为7ffee6c49510,我们再查看test函数的反汇编代码可以看到我们需要跳转到的地方是



所以我们输入的字符串从33位开始应该是1095c4e6fe7f0000687efb0801000000

接着我们要改变val的值为deafbeef



我们可以看到val的值离getbuf的栈有12个字节，而这12个字节的值是可以任意填充的，于是接下来的字符串应该为000000000000efbeadde

宗上 我们得到的密码为

000000000000000000000000000000001095c4e6fe7f0000687efb0801000000000000000000efbeadde

dbq没做完这方法有缺陷