**Lab2 Report**

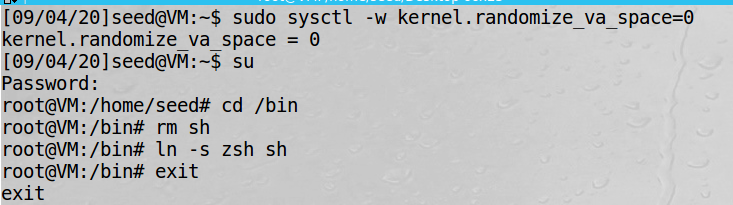
**Buffer Overﬂow Vulnerability and Return-to-libc Attack**

57117107 赵笛言

**Buffer Overflow Vulnerability Lab**

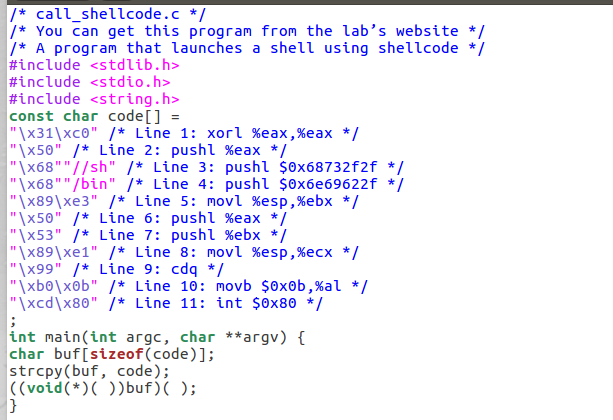
**Task1:** Turning Off Counter Measures and Running Shellcode

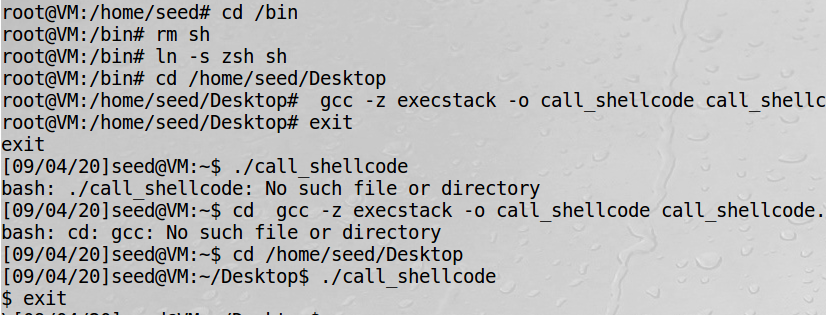
Step1：Turning Off Counter Measures

由于Ubuntu使用地址空间随机化来随机堆栈的初始地址，这让直接计算出精确的内存地址变得十分困难。因此实验前，我们需要关闭这一功能；为了进一步防范缓冲区溢出攻击及其它利用shell程序的攻击，在/bin/bash中设置有防御措施，即使能使一个Set-UID程序调用一个shell，也不能在这个shell中保持root权限，为了实现最简单的攻击，使用另一个shell程序（zsh）代替/bin/bash。

Step2：Running Shellcode

Please compile and run the following code, and see whether a shell is invoked.

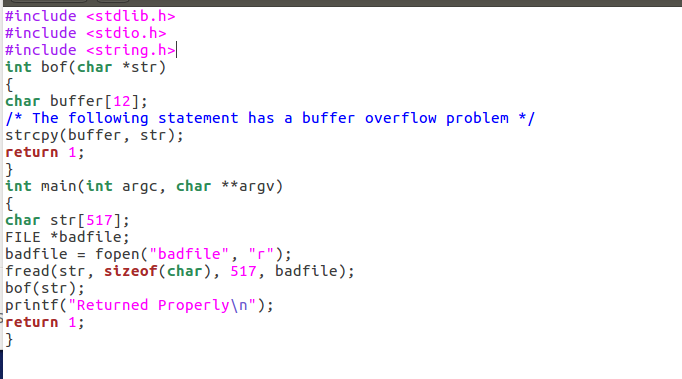


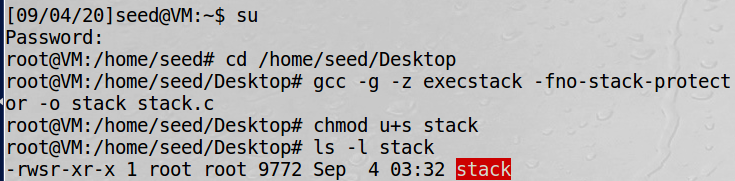


可以看出成功创建了一个shell，但并非root权限。

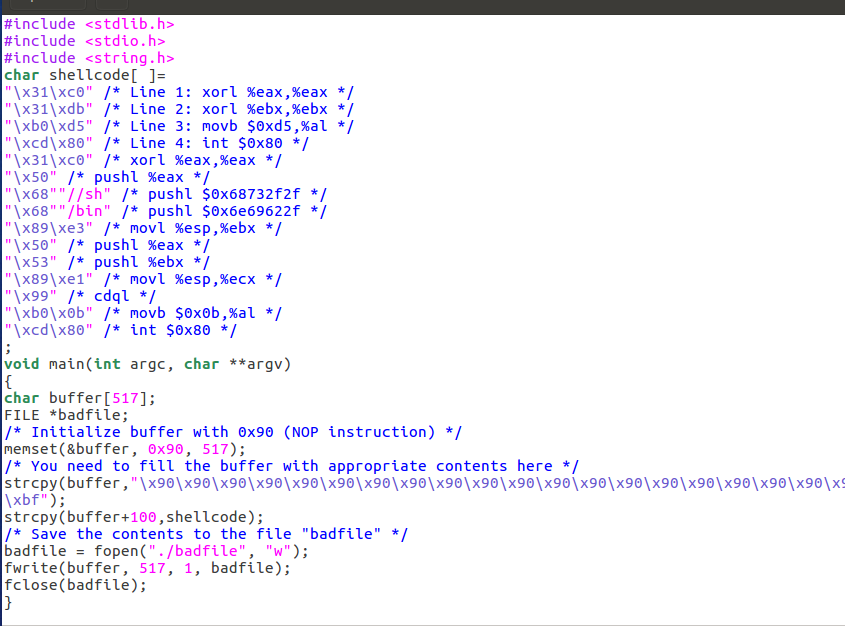
**Task2:** Exploiting the Vulnerability

Step1：编译漏洞程序，并setuid，设置读写等权限。有一种栈保护机制来阻止缓冲区溢出，所以需要用 –fno-stack-protector 关闭这种机制。并使用-z execstack 用于允许执行栈。



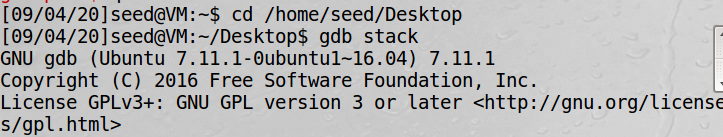


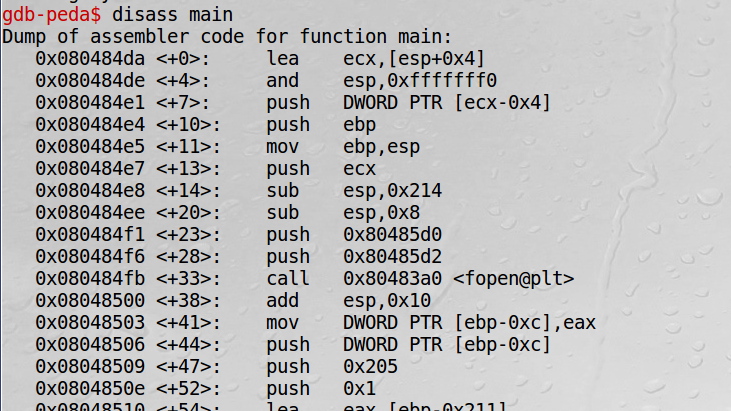
Step2：编写攻击程序。（需要第三步中的地址来将代码补充完整）



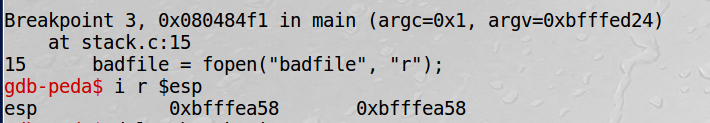
Step3：寻找用以获得root权限的地址。

这一步是缓冲区溢出攻击中最为关键的一步，利用gdb进行。



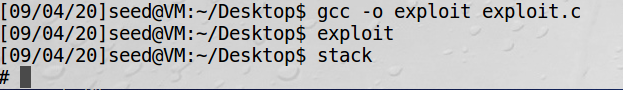


在0x080484f1处设置断点。

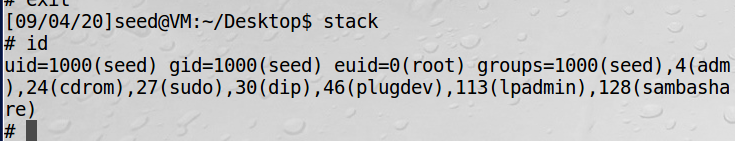


得到的地址为0xbfffea58，再在此地址上加上100（十进制），得出地址0xbfffeabc，即在攻击程序中填入\xbc\xea\xff\xbf。

Step4：进行缓冲区溢出攻击。



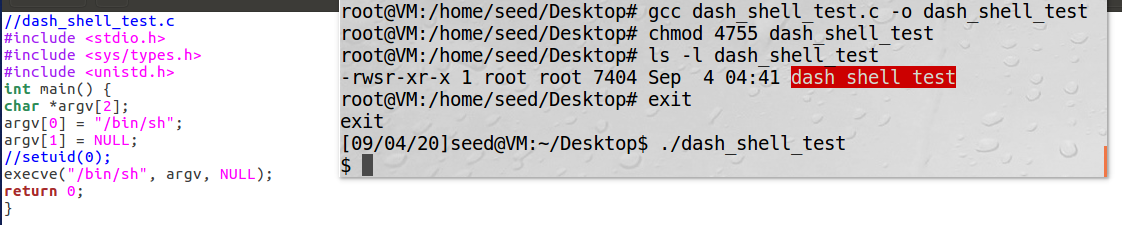
攻击成功！Check id：

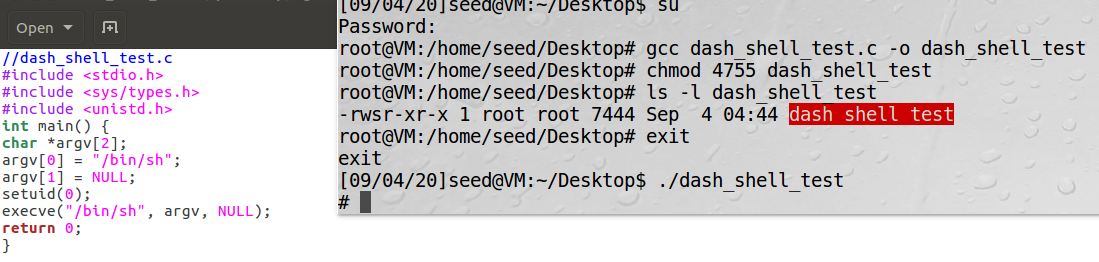


**Task3**：Defeating dash’s Countermeasure

Step1：将bin/sh指向bin/dash （使用命令ln -s）

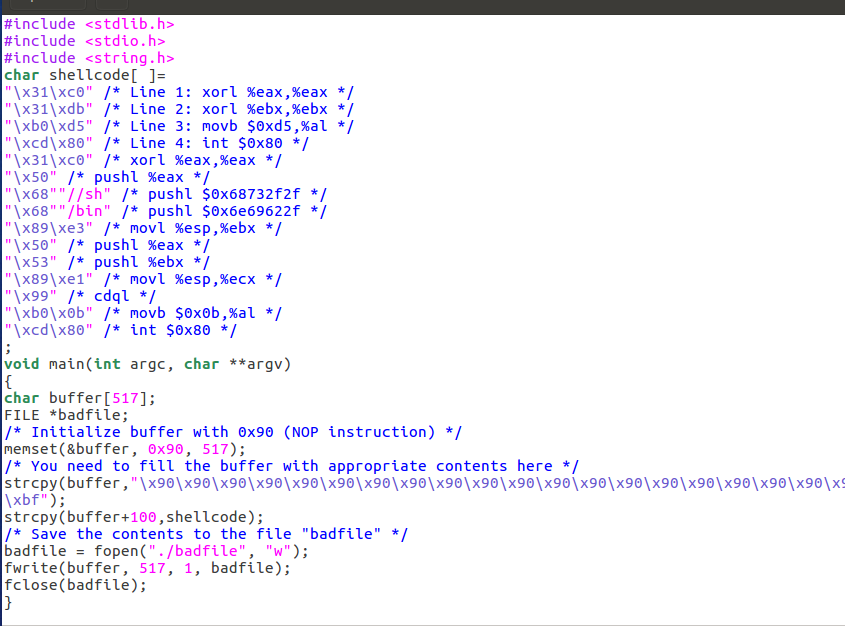
Step2：编译并运行dash\_shell\_test.c，比较有无setuid(0)命令时运行结果的区别。



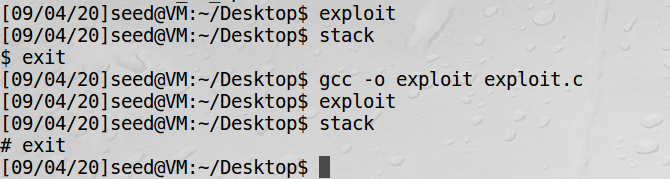


可以看出，在没有setuid=0的情况下，即使成功创建了一个shell也无法获取root高权限，这就是dash的保护机制，当uid不等于euid，自动取消特权。

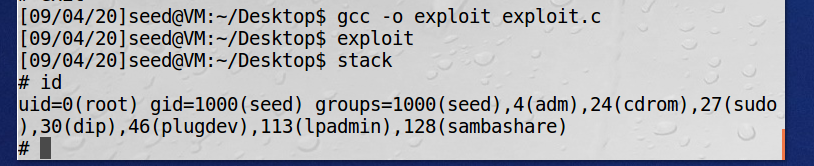
Step3：改写exploit.c 再次尝试Task2



可以对比一下当sh指向dash，没有如上更改以及更改后的情况：



再对第二种情况check id：

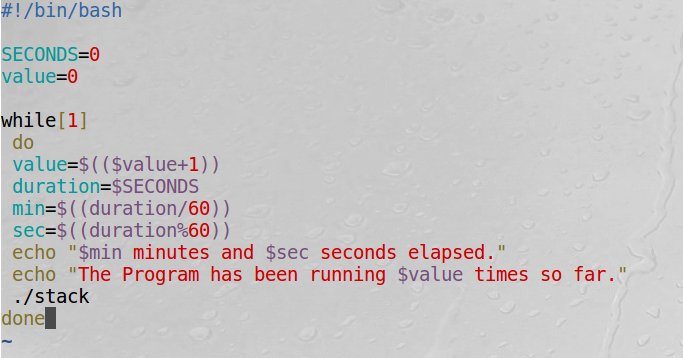


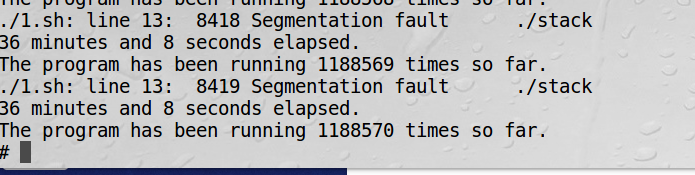
**Task4**: Defeating Address Randomization

Step1：将地址随机化选项开启，重新编译执行stack，发现会出现段错误。



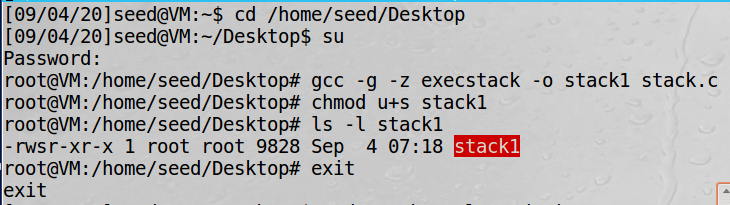
Step2：编写shell脚本循环运行./stack，可以攻击成功，但耗时不定。

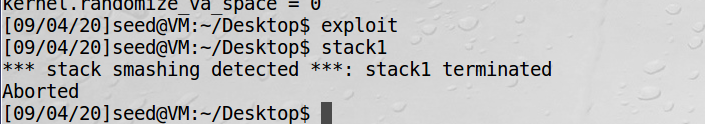




**Task5:** Turn on the Stack Guard Protection

即在编译stack文件时不使用–fno-stack-protector命令。



运行stack1结果：

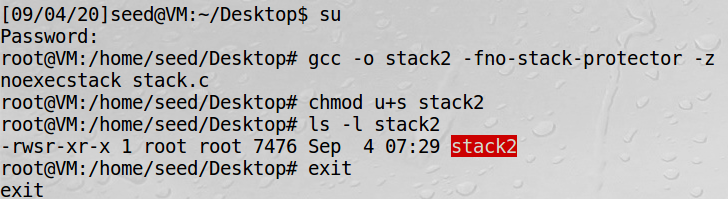
该机制阻止缓冲区溢出，攻击失败。

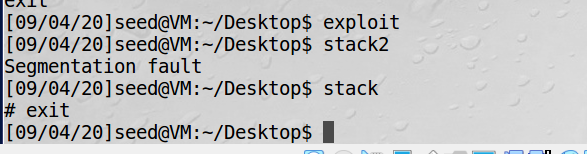
**Task6:** Turn on the Non-executable Stack Protection

允许或禁止栈执行的命令如下：

允许：gcc -z execstack -o test test.c

禁止：gcc -z noexecstack -o test test.c





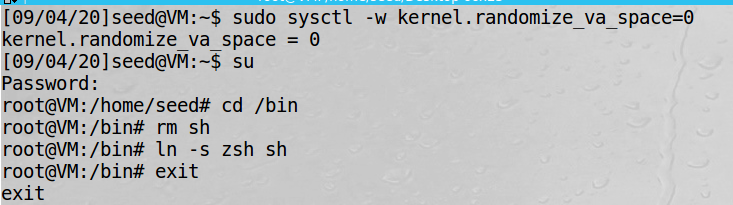
无法执行时会直接判断为段错误而不是执行恶意代码。

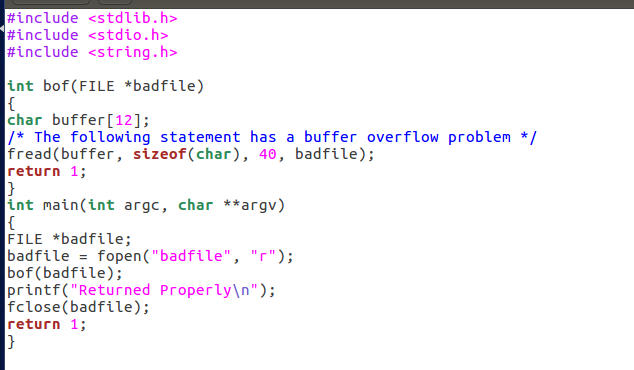
**Submission1：**该实验的难点主要在于task2的地址确定，虽然知道应用了很方便的gdb，但判断具体按哪个地址作为断点、找到断点后如何确定shellcode放置的位置以及恶意程序如何根据地址编写都存在一定的难度。我个人还在使用ln -sf命令时出现了问题，形成了无法控制的循环，按照实验手册时，我直接将sh指向zsh，后期想改回时想当然的又将zsh指回sh，最终导致整个虚拟机出现问题，后期重新设置了虚拟机才得以解决。于是再次实验室，一步步按照rm sh再将ln -s zsh sh这样操作，幸好没有再出现难以解决的问题。

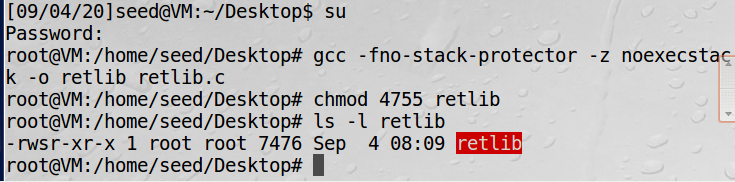
**Return-to-libc Attack Lab**

**Task1：** Finding out the addresses of libc functions

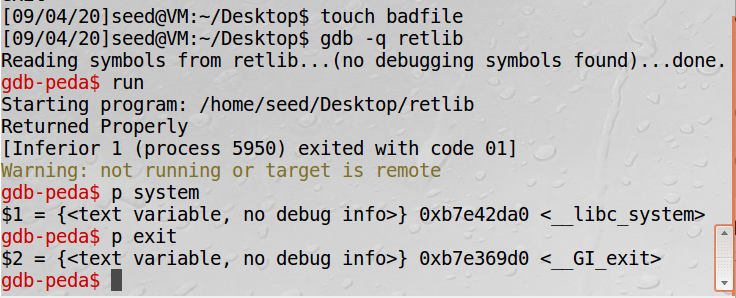
Step1：关闭保护措施并编译漏洞程序，setuid以及读写等权限。







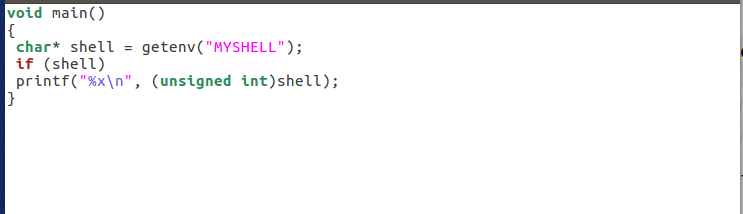
Step2：利用gdb找到system以及exit地址，如下图所示：



**Task2：** Putting the shell string in the memory

这次的task的主要目的是找到/bin/sh的地址用以完善下一task的攻击程序。

Step1：构造新的shell变量MYSHELL

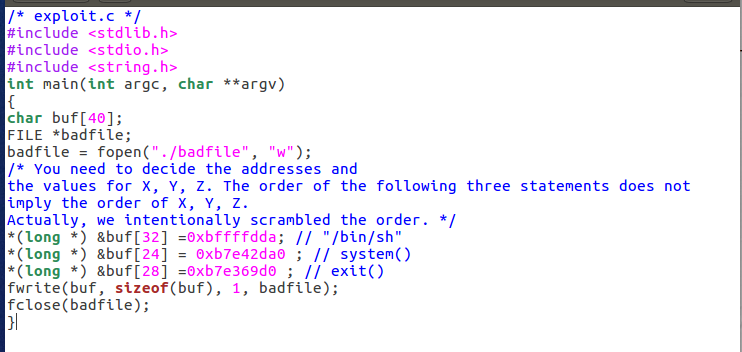


Step2：编写程序输出地址

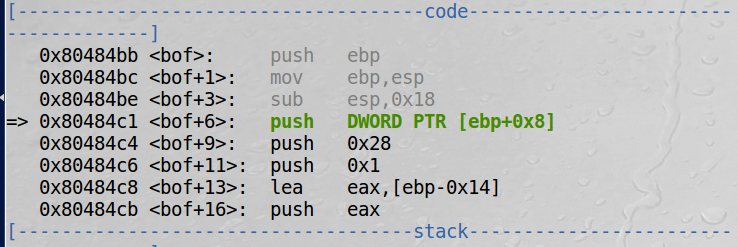


**Task3：**Exploiting the buffer-overflow vulnerability

Step1：编写exploit.c程序，写入前俩个task得到的三个地址



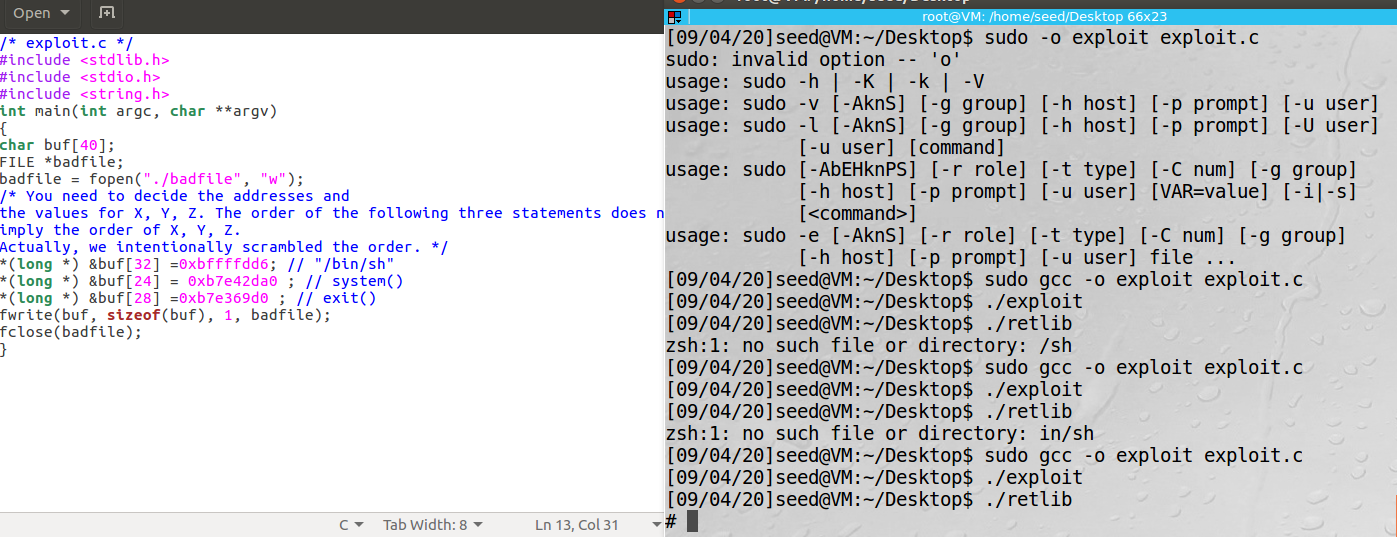
关于程序中buf后的数据应该如何选取，可以通过bof函数判断：



buffer距离ebp的距离为0x14，所以buffer距离return address的距离为0x18，在exploit.c中将system()在buffer数组中的下标改为24（10进制的0x18），函数调用时在函数自身ebp下是return address 和参数，故将exit()在buffer数组中的下标置为28，参数/bin/sh在buffer数组中的下标置为32，从而完成exploit程序。

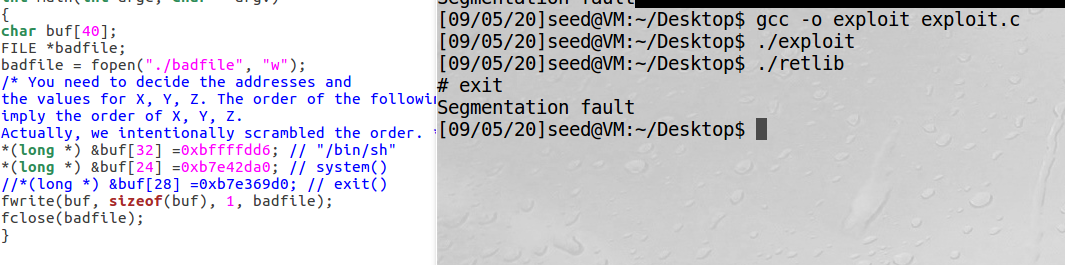
Step2：尝试攻击

由于/bin/shell的地址会有偏差，因此一边更改一边多次尝试攻击，最后得知其地址为0xbffffdd6，攻击成功。



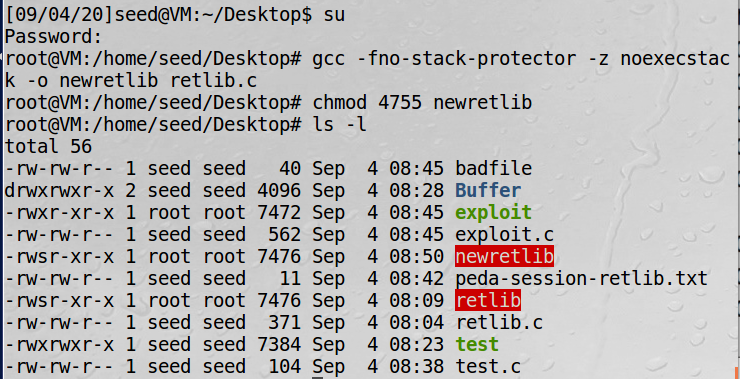
1. 关于exit（）函数

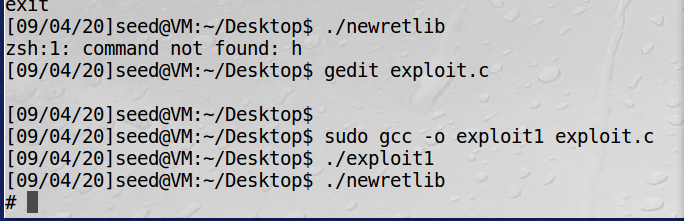
在exploit.c函数中注释exit()函数所在行，再次实验，依旧可以进入root权限，但是无法正常退出shell。



exit()用以填充在栈上的地址，是调用system()时的返回地址，如果没有exit()，原来填充exit()的数据可能是任何值，导致出现段错误。

1. 改变retlib文件名





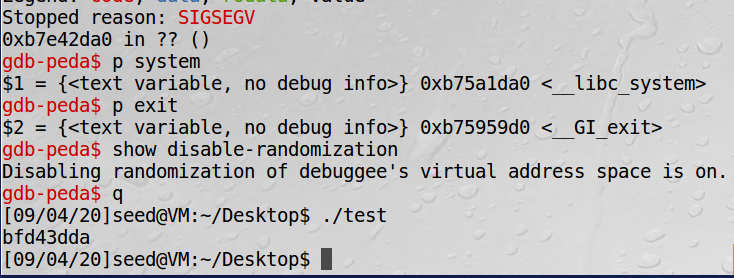
显示错误信息h，h相对于/bin/sh第一个字符/偏移为6，故减去。将bin/sh的地址改为0xbffffdd0，可实验成功。

**Task4：**Turning on address randomization

重复刚才的步骤，攻击失败。



使用gdb查看system和exit的地址；运行test查看bin/sh地址。



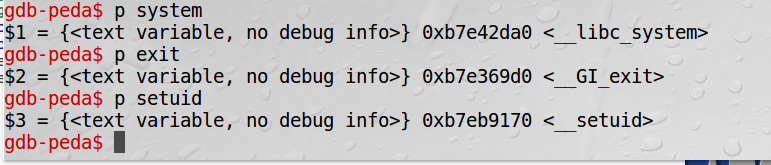
运行“ show disable-randomization”来查看随机化是否已关闭。

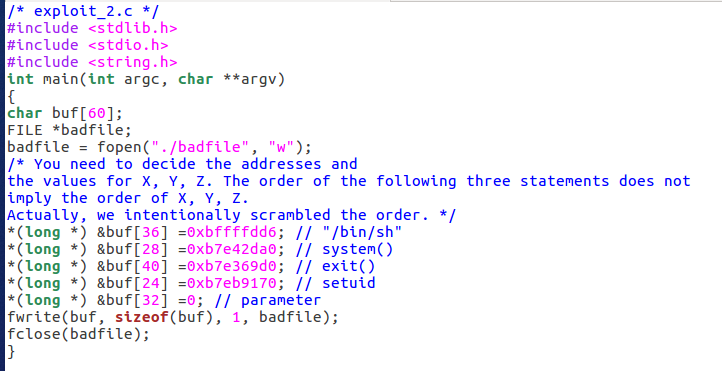
**Task5：**Defeat Shell’s countermeasure

在/bin/sh指向bash的情况下，根据上一个实验的提示可以考虑使用命令setuid（0）来完成对这种保护措施的攻击。

在bof的返回地址处（buf[20]）写入setuid()的地址，setuid的参数0写在与其入口地址相隔一个字的位置（即buf[28]）处。

如下图，求出setuid的地址后改写攻击程序：





随后可以攻击成功。



但由于为了放入setuid命令行及其参数，exit函数的栈中位置需要重新考虑，导致退出时会出现段错误。

**Submission2：**这个实验与实验1有着异曲同工的感觉，但地址相对更容易求一点，有些地址偏差可以直接通过边改边测试解决，没有像实验1卡壳，但task5和task6做了很久，而且task6还是没做出来，对rop还是很陌生。