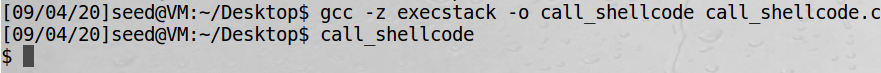
**实验1：Buffer Overﬂow Vulnerability Lab**

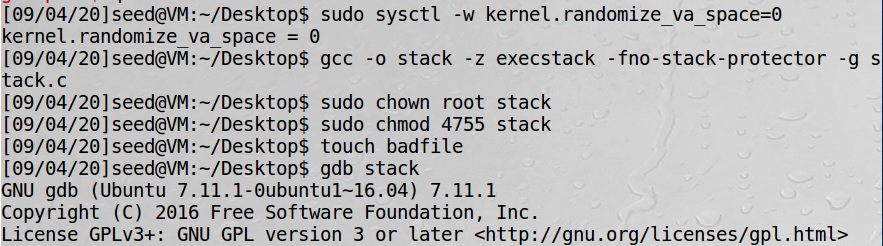
**Task 1: Running Shellcode**

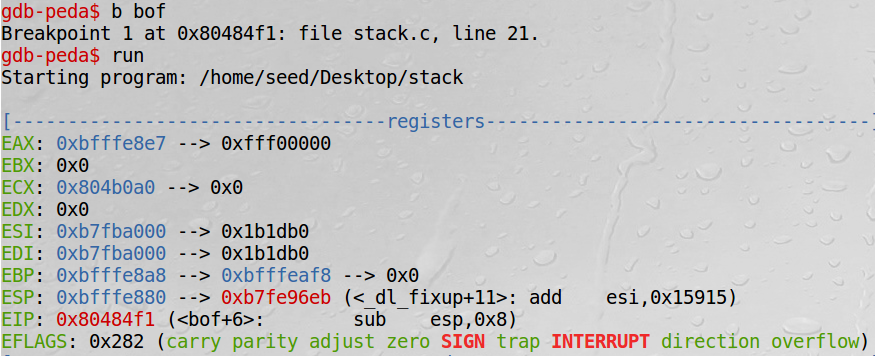


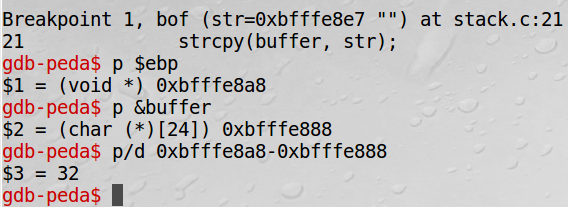
shellcode执行，成功调出了shell

**Task 2: Exploiting the Vulnerability**

编译stack.c，gdb调试

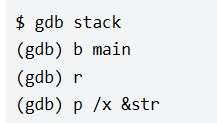


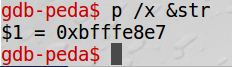




得到buffer到return address的距离为32+4=36

接下来获取shellcode的地址



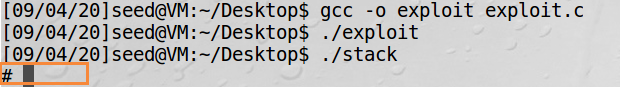


str的地址为0xbfffe8e7，计算上shellcode偏移量100（0x64）,则shellcode地址为0xbfffe94b

exploit.c的完整代码如下：

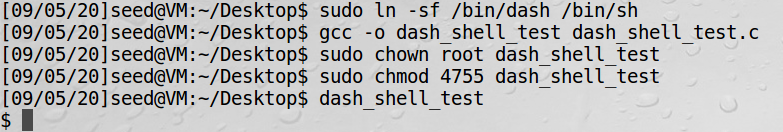


编译并执行exploit.c，得到badfile，再执行stack，得到了一个root shell，证明执行成功

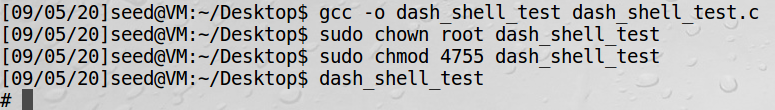


**Task 3: Defeating dash’s Countermeasure**

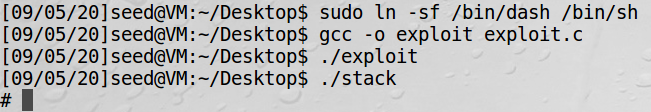
首先注释①，并且将程序作为Set-uid程序运行（程序的拥有者是root），获取的是用户权限的shell



然后取消①的注释，再运行程序，获取的是root权限的shell



修改shellcode中的汇编代码，使得在连接到dash的情况下也可以获取root权限的shell



**Task 4: Defeating Address Randomization**

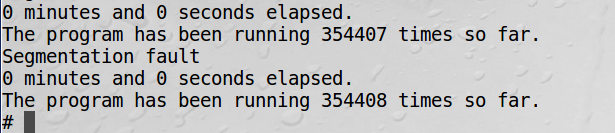
首先打开地址随机化



如果再次执行task 2中的攻击会发现攻击失败

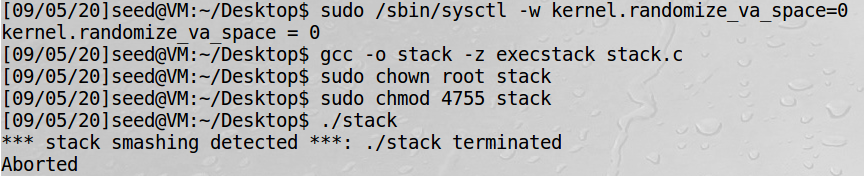


然后通过暴力破解的方法攻击，运行shell脚本，经过354408次后成功



**Task 5: Turn on the Stack Guard Protection**

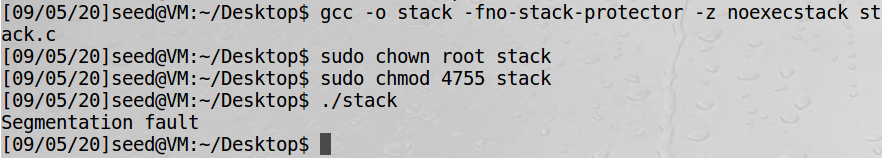
关闭地址随机化，编译时不关闭stackguard，重新执行task 2



在这种机制下发现stackguard部分被覆盖了，会终止程序执行

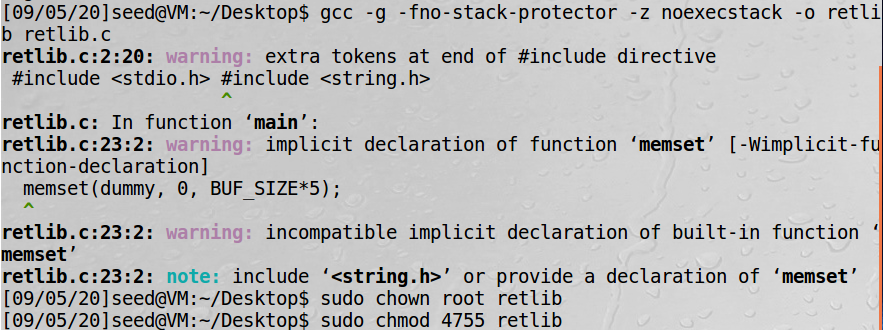
**Task 6: Turn on the Non-executable Stack Protection**

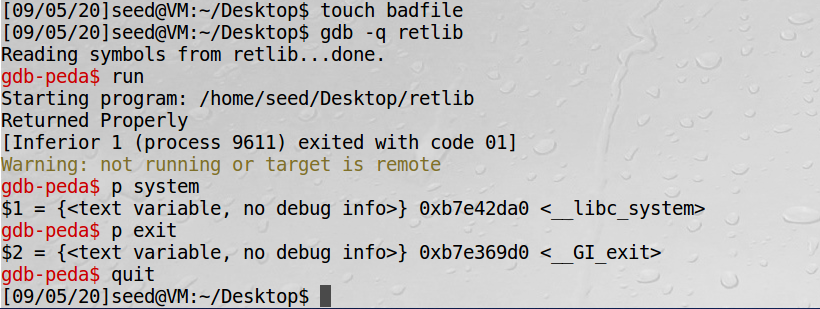
关闭地址随机化，编译时打开不可执行栈，重新执行task 2，攻击失败



**实验2：Buffer Overﬂow Vulnerability Lab**

**Task 1: Finding out the addresses of libc functions**

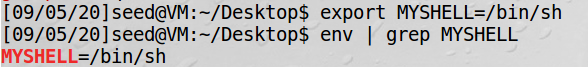




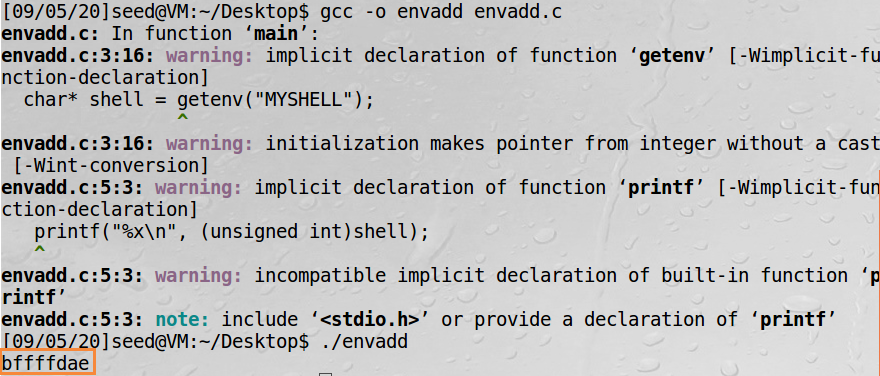
得到system()的地址为0xb7e42da0，exit()的地址为0xb7e369d0

**Task 2: Putting the shell string in the memory**

首先定义一个环境变量MYSHELL，值为/bin/sh



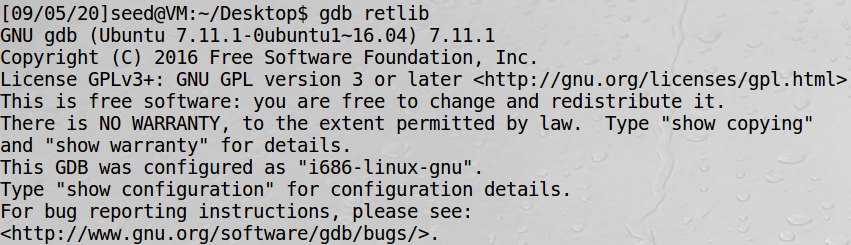
将环境变量的地址作为传递给system()系统调用的参数，要获取环境变量的地址

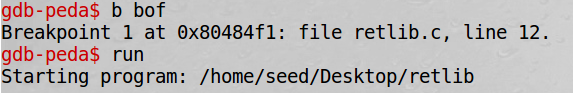


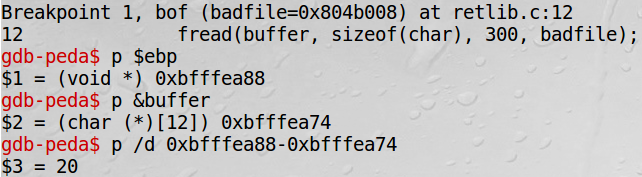
**Task 3: Exploiting the buffer-overﬂow vulnerability**

我们需要覆盖bof函数的返回地址，将其指向system()函数，并且将/bin/sh作为system()的参数，最后执行exit()函数范围，故我们必须知道bof函数的返回地址

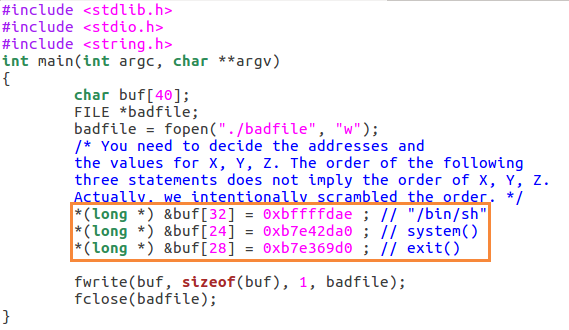
操作同上一个实验的task 2



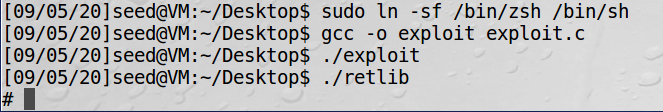




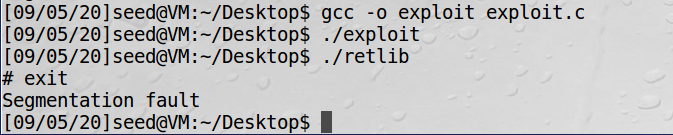
buffer到return address的距离为20+4=24，所以exploit.c中system()在buffer数组中的下标为24。可将exit()的地址作为system()的返回地址，所以exit()在buffer数组中的下标为28，参数/bin/sh在buffer数组中的下标为32。对应的地址前面两个task已经求过了。



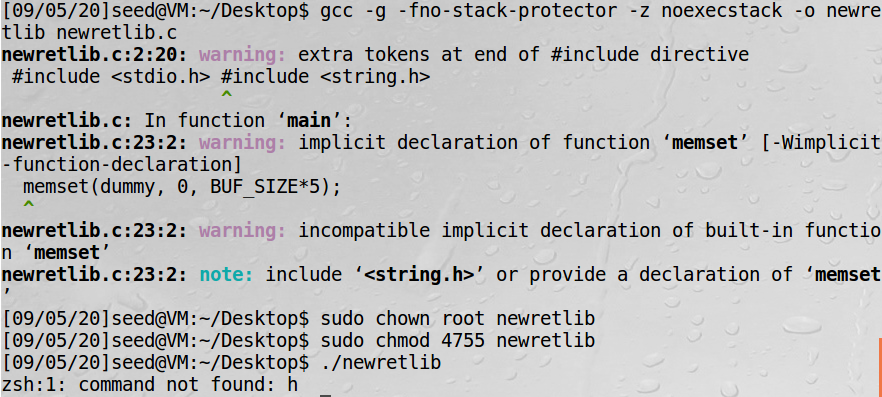
编译并执行代码，成功得到root shell



① exit()函数是必要的吗？是必要的，否则system()的返回地址就可能是任意值，当system()返回时会导致程序崩溃



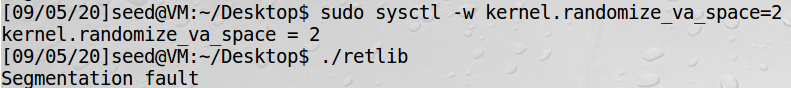
② 将retlib的名字改为newretlib，重新实验，实验失败了



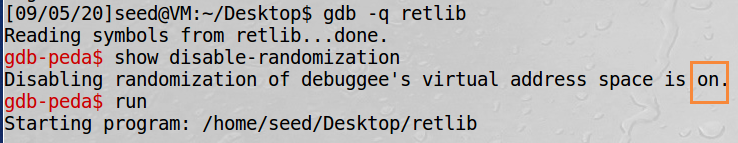
原因是环境变量的地址可能有所不同（比如地址会因为程序名字的改变而发生改变，主要是字符数）。程序名不同，传入栈的字符长度可能不同，从而导致环境变量的地址可能不同。

**Task 4: Turning on address randomization**

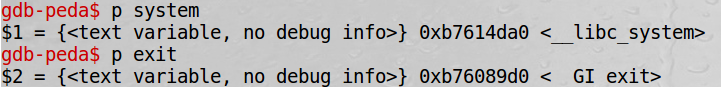
打开地址随机化，再次尝试之前的攻击，发现攻击失败

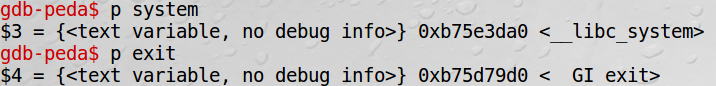


原因是地址随机化打开后，之前的三个地址就会发生变化



两次运行retlib，两次的system和exit的地址不同





**Task 5: Defeat Shell’s countermeasure**

首先将/bin/sh连接到/bin/dash

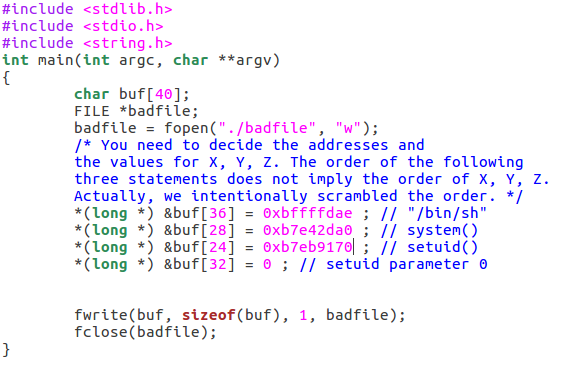
我们在调用system(“/bin/sh”)之前，先调用setuid(0)来提权

在bof的返回地址处buf[24]写入setuid()的地址，setuid的参数0写在buf[32]处。因为setuid()执行完毕之后，应转向system()，所以system放在buf[28]，system的参数/bin/shell放入buf[36]处。

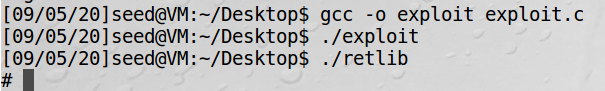


setuid的地址为0xb7eb9170

按上面的描述修改exploit.c



再次执行攻击，攻击成功



但是这种做法好像不太对，system的返回地址没有进行设置，退出的时候程序会崩溃

