# Lab2-report

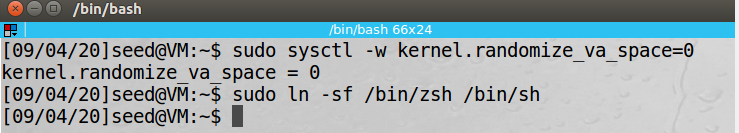
**57117201王培丽**

#### Buffer Overﬂow Vulnerability Lab

选择BUF\_SIZE

The BUF\_SIZE value for this lab is: 100

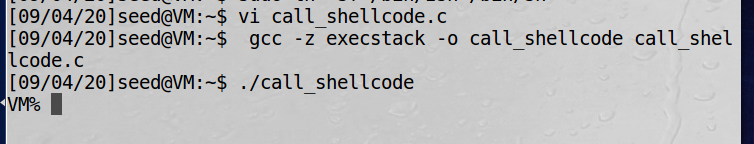
实验前准备（关闭地址随机化、配置/bin/sh等）



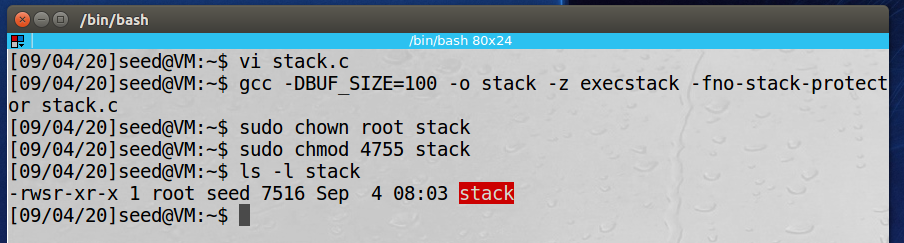
* **Task1: Running Shellcode**

编译运行程序call\_shellcode.c（关闭不可执行保护）

运行call\_shellcode，发现开启了一个shell



编译漏洞程序stack.c（攻击目标），编译时设置关闭栈保护和不可执行保护，生成stack并将其设置为Set-UID root



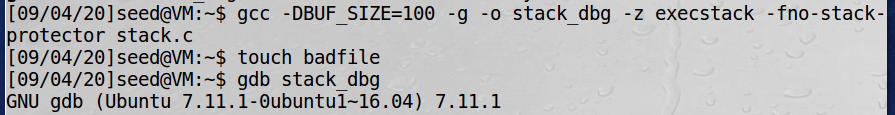
* **Task2: Exploiting the Vulnerability**

1. 补全exploit.py代码：

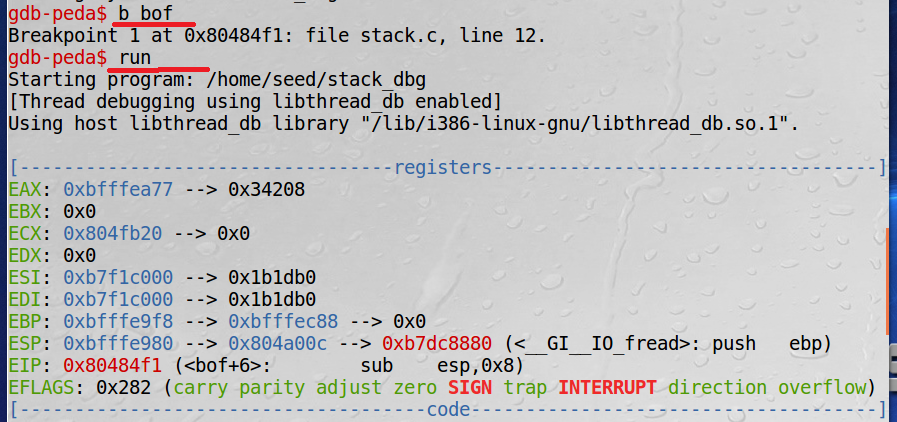
首先生成一个用于调试的stack\_dbg

新建一个空的badfile

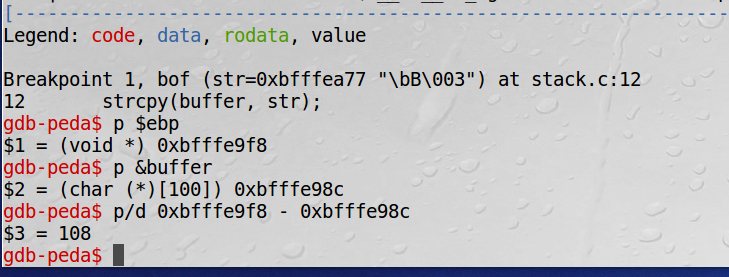
利用gdb调试stack\_dbg程序



在关键函数bof()处设置断点



在程序停在strcpy(buffer, str)的位置时，查看此时的ebp指针，以及buffer的值，计算差值



$ebp = 0xbfffe9f8

&buffer = 0xbfffe98c

两者相差108

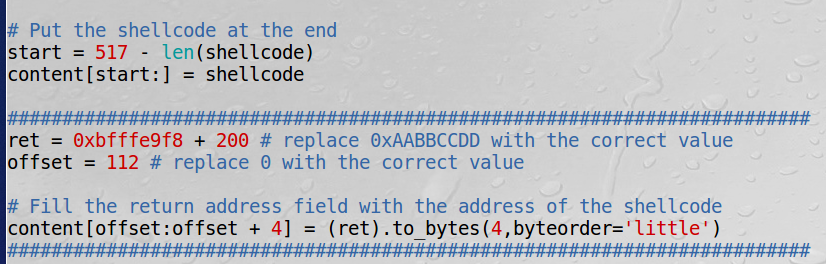
所以可得buffer+108+4即为Return Address的位置。

大概取517的一半（200）左右的偏移量作为Return Address所指向的地址

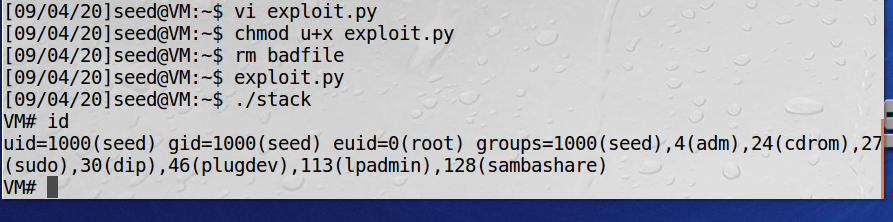
所以Return Address = ebp + 200 = 0xbfffe9f8 + 200

而Return Address在buffer + 108 + 4的位置，也就是offset =112

所以exploit.py中的对应代码如下：

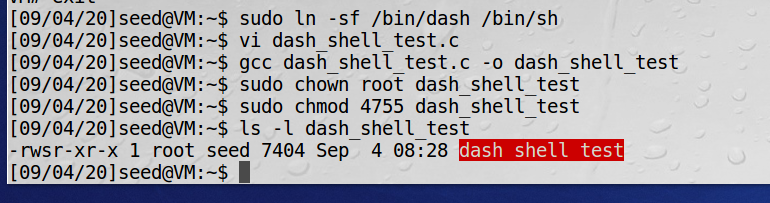


1. 编译exploit.py来生成badfile，再运行task1中的漏洞程序stack，可以看到获得了root权限（euid=0）



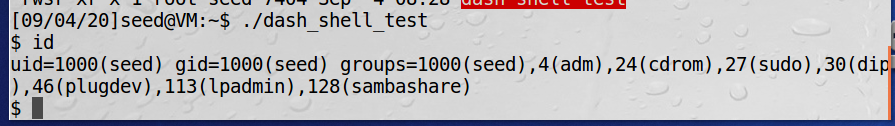
* **Task3: Defeating dash’s Countermeasure**

1. 更改/bin/sh符号链接，使其指向/bin/dash
2. 编译程序dash\_shell\_test.c，将其设置为Set-UID root



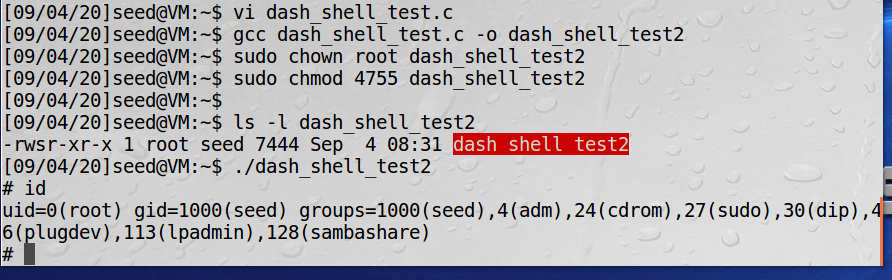
程序中注释掉setuid(0)时的运行结果：

此时没有获得root权限



取消setuid(0)的注释后再次编译运行：

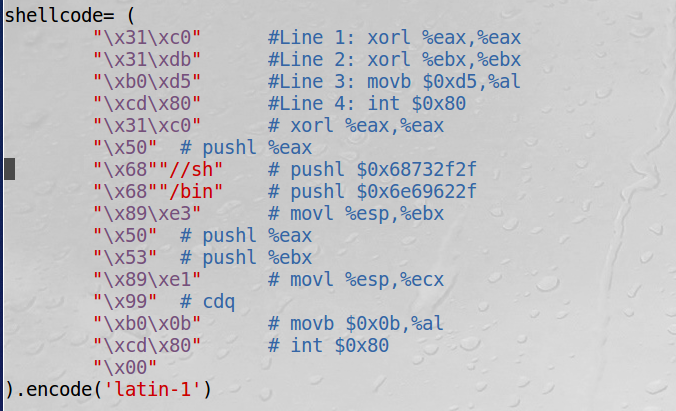
uid=0，获得了root权限



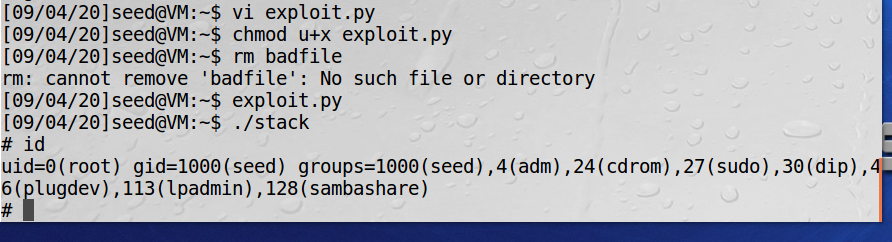
由上可以看出在运行Set-UID程序时，dash会检查real UID和effective UID是否一致，如果不一致会取消特权，而通过先调用setuid(0)将real UID改为0，就可以抵御dash的检查。

1. 更改task2中的exploit.py代码，在shellcode[]中加上对setuid(0)的调用：

前4行为新增（第7行/zsh要改回//sh）



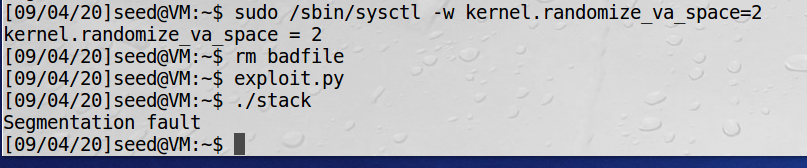
1. 重复task2中的步骤（运行exploit.py来生成badfile，再运行漏洞程序stack）。



此时，在/bin/sh链接向/bin/dash的情况下，仍能获得root权限，因为shellcode[]中，在调用execve()之前先调用了setuid(0)，从而抵御了dash的检查

* **Task4: Defeating Address Randomization**

1. 首先打开地址随机化
2. 再次进行task2中的攻击步骤（运行exploit.py来生成badfile，再运行漏洞程序stack）

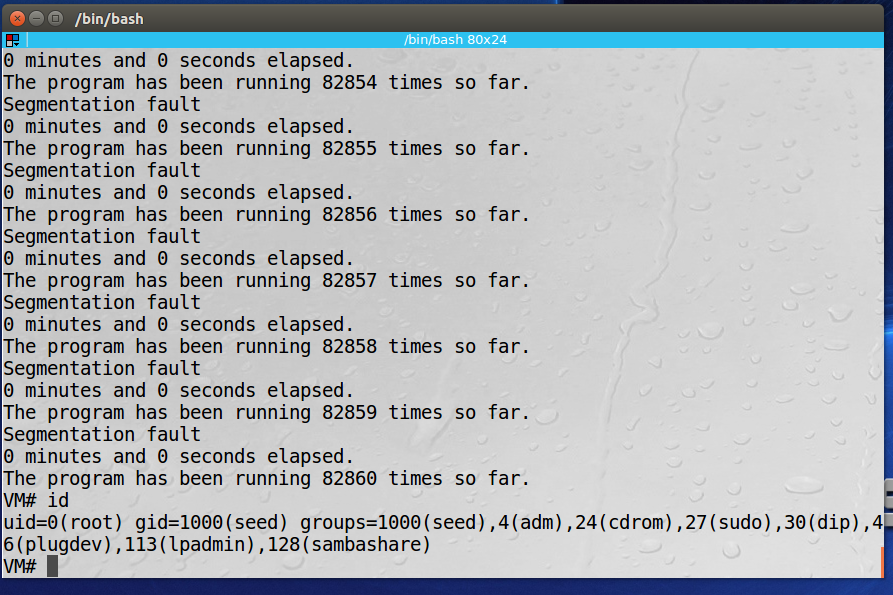


发现此时没有运行成功，因为地址随机化的保护，使栈溢出攻击失败。

1. 运行shell脚本来暴力破解

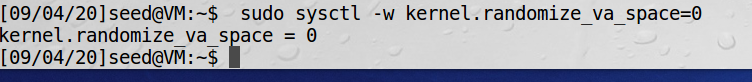


最终获得了root权限



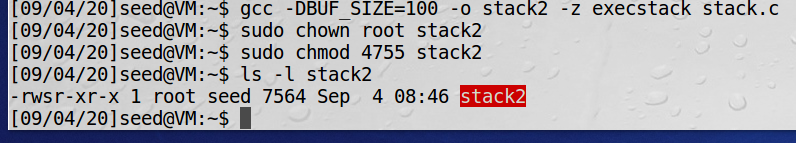
* **Task5: Turn on the Stack Guard Protection**

1. 首先关闭地址随机化

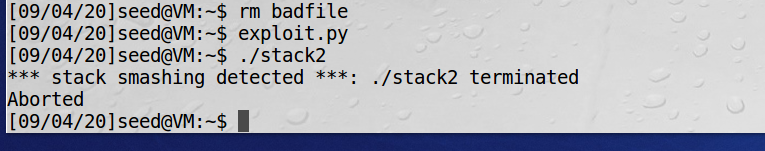


1. 在开启StackGuard的条件下（但不可执行保护仍关闭）重新编译stack.c，生成stack2（区别于stack）并设置为Set-UID root

（不带参数-fno-stack-protector，使可执行-z execstack）



1. 再次进行task2中的攻击步骤（运行exploit.py来生成badfile，再运行漏洞程序stack2）

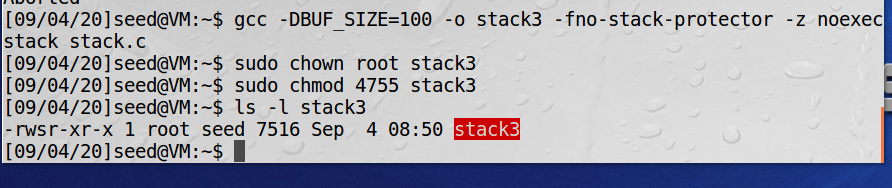


发现在开启栈保护时，攻击不成功

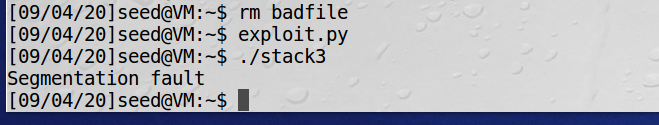
* **Task6: Turn on the Non-executable Stack Protection**

1. 开启不可执行（关闭StackGuard）的条件下重新编译stack.c，生成stack3并设置为Set-UID root

(带参数-fno-stack-protector，开启不可执行-z noexecstack）



1. 再次重复task2中的攻击步骤，攻击漏洞程序stack3



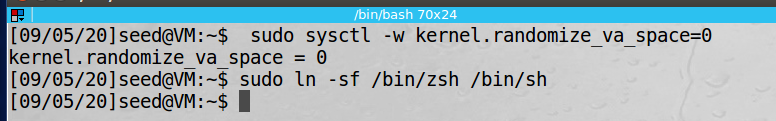
发现在开启不可执行时，攻击不成功：

#### Return-to-libc Attack Lab

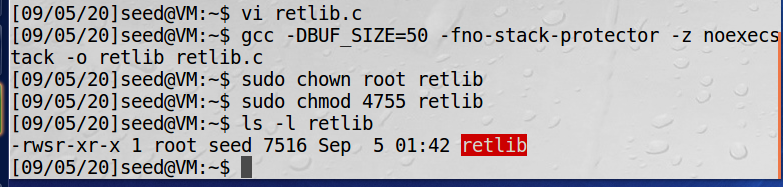
选择BUF\_SIZE

The BUF\_SIZE value for this lab is: 50

实验前准备（关闭地址随机化、配置/bin/sh等）

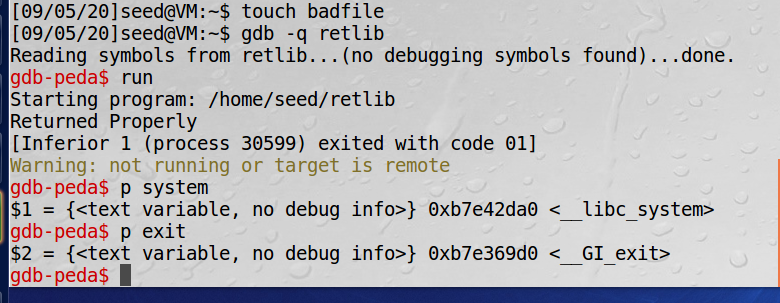


编译漏洞程序retlib.c，关闭StackGuard和开启不可执行，设置retlib为Set-UID root



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

利用gdb调试目标程序retlib，找到system()和exit()函数的位置：

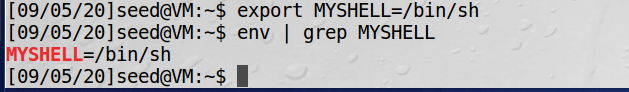


如上图，system()位置是0xb7e42da0

exit()位置是0xb7e369d0

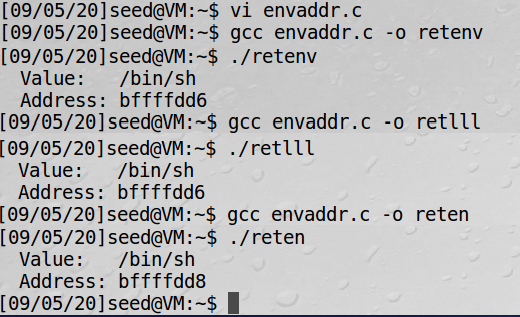
* **Task2: Putting the shell string in the memory**

1. 设置MYSHELL环境变量



1. 通过envaddr.c程序，找到MYSHELL环境变量的位置

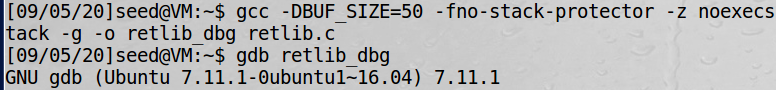
因为程序名字不同，环境变量的值也可能发生改变，为使得到的环境变量与运行retlib时的尽量一致，将其命名为retenv（和retlib长度一样）

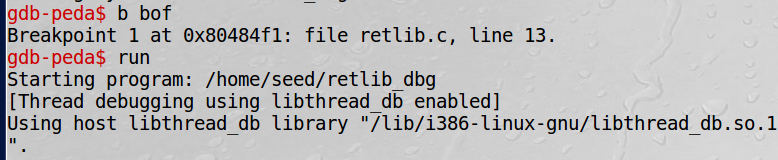


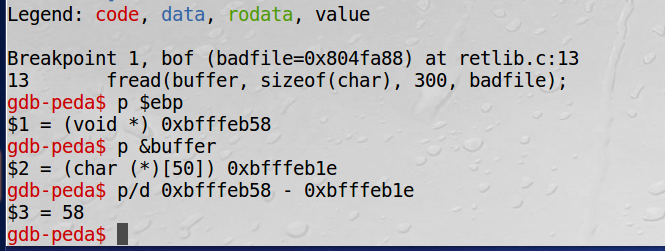
并且通过命名不同的名字：retenv、retlll、reten，发现程序名字长度一致时环境变量位置也是一致的，所以可以推测目标程序retlib的MYSHELL环境变量位置也是bffffdd6

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

1. 另外生成一个用于gdb调试目标程序retlib\_dbg，在关键函数bof处设置断点，查看ebp和buffer的值







由上可知

$ebp = 0xbfffeb58

&buffer = 0xbfffeb1e

由此算出ebp距离buffer的偏移量为58

1. 根据task1和task2中system()、exit()、/bin/sh的位置，以及ebp据buffer的偏移量补全exploit-retlib.c的代码：

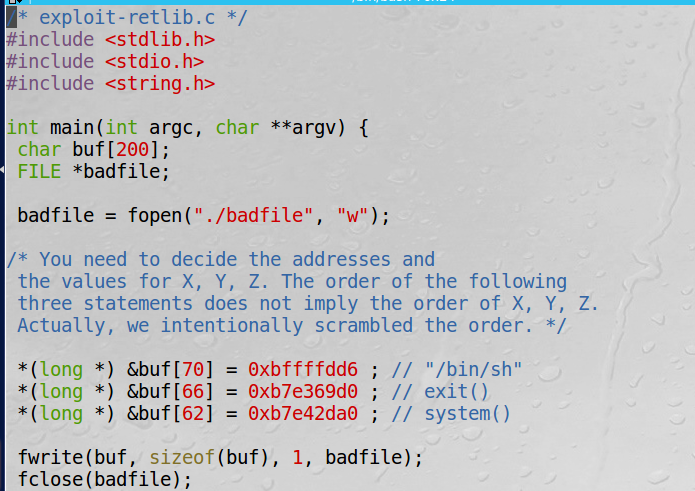
ebp+4=buffer+58+4 应为 system()的地址

ebp+8=buffer+58+8 应为 exit()的地址

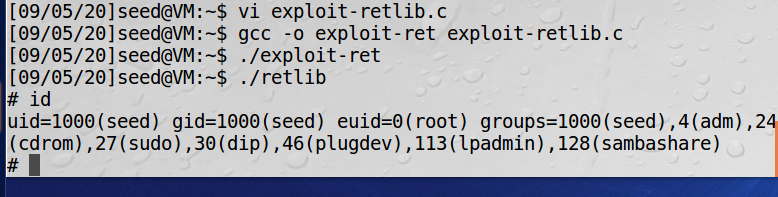
ebp+12=buffer+58+12 应为 “/bin/sh”的地址

所以代码如下：

(实验指导里的exploit.c定义的buf长度为40，明显不够，我改成了200)



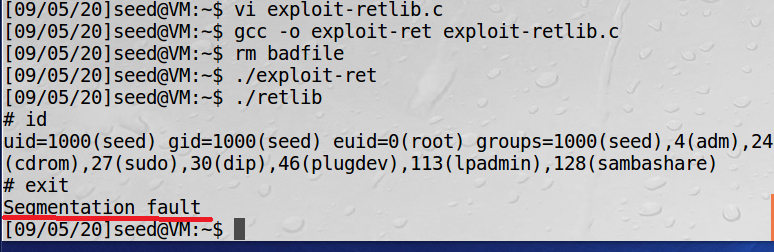
1. 编译运行exploit-retlib.c来生成badfile，再运行漏洞程序retlib，获得了root权限：



**Attack variation 1：**

修改exploit-retlib.c代码，不使用exit()函数，再次运行生成新的badfile，运行漏洞程序retlib，发现仍能获得root权限，但退出后程序出错：

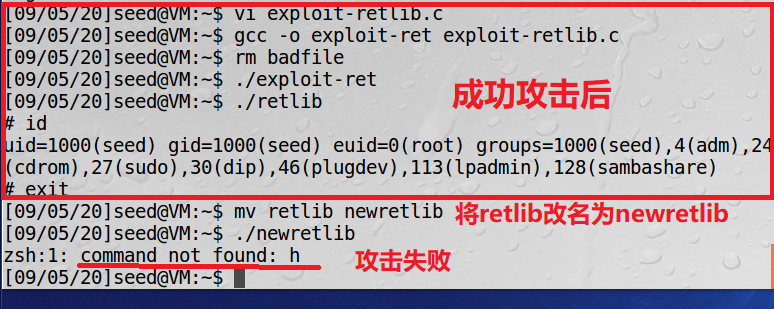




不加exit()时，程序运行完system(/bin/sh)后下一步转向地址ebp+8，若里面是乱码，程序会出错。

**Attack variation 2：**

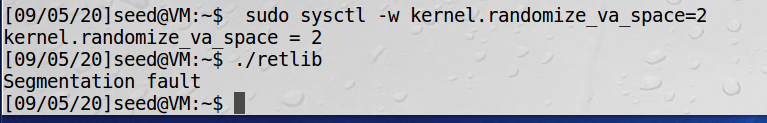
在一次成功的攻击后，将retlib改名为newretlib，（使用攻击成功时的badfile），运行newretlib，发现攻击失败：



更改retlib的名字，程序名字长度改变，会使环境变量的位置发生改变，“/bin/sh”已不是exploit-retlib.c代码中所填写的位置，传给system()的参数已经不是“/bin/sh”，所以会报错找不到command

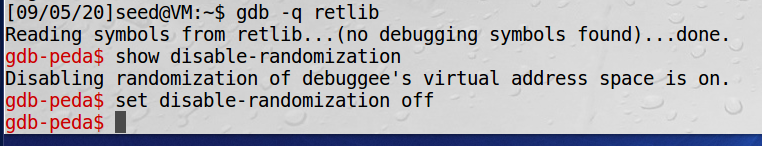
* **Task4: Turning on address randomization**

1. 打开地址随机化：
2. 重复攻击过程，发现攻击失败

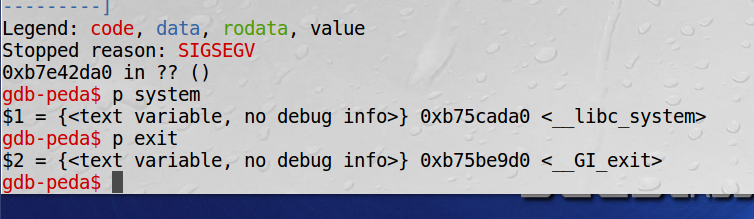


1. 用gdb调试retlib，查看system()和exit()的地址是否改变了

首先要在gdb里开启地址随机化：



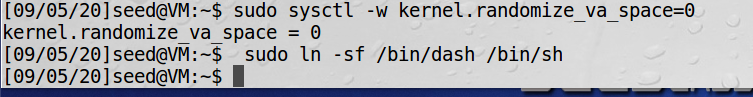
发现system()和exit()的地址的确不同于之前的，所以才会导致攻击失败



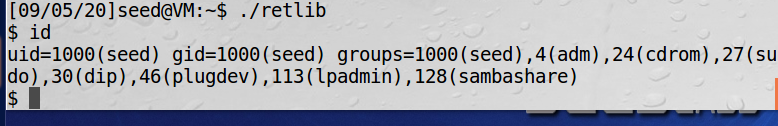
（未开启随机化时，system()位置是0xb7e42da0 ，exit()位置是0xb7e369d0 ）

* **Task5: Defeat Shell’s countermeasure**

1. 关闭地址随机化，更改符号链接，使/bin/sh指向/bin/dash

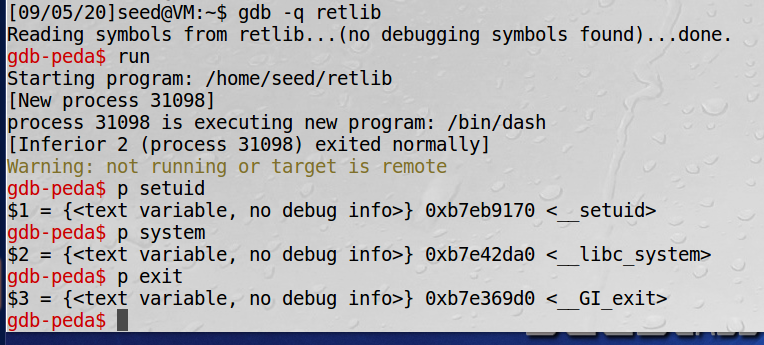


测试一下，特权被取消了



1. 为抵御dash的对RUID和EUID的检查，在使用system()调用前先调用setuid(0)，首先需要知道setuid()的位置

同理，利用gdb调试retlib，找到setuid的位置



如上，setuid()位置是0xb7eb9170

system()、exit()的位置没有改变

1. 更改exploit-retlib.c的代码，使程序先执行setuid(0)，再执行system(“/bin/sh”)

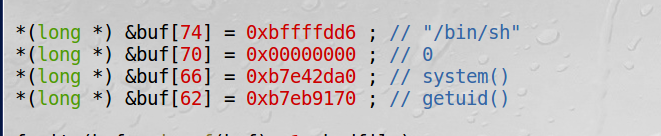
ebp+4=buffer+58+4 应为 setuid()的地址

ebp+8=buffer+58+8 应为 system()的地址

ebp+12=buffer+58+12 应为 setuid()的参数0

ebp+16=buffer+58+16 应为system()的参数“/bin/sh”

所以exploit-retlib.c的代码如下：



1. 编译exploit-retlib.c生成exploit-ret2，运行，再运行retlib，攻击成功，uid=0获得了root权限

(但调用完system()没有调用exit()来退出，退出shell后后程序会出错)

