**信息安全实验报告**

**Lab 4 Buffer Overflow Vulnerability**

**孙铁**

**SA20225414**

**实验开始之前，需要将针对缓冲区溢出攻击的相关防御机制关闭：**

1. 关闭地址空间随机化；

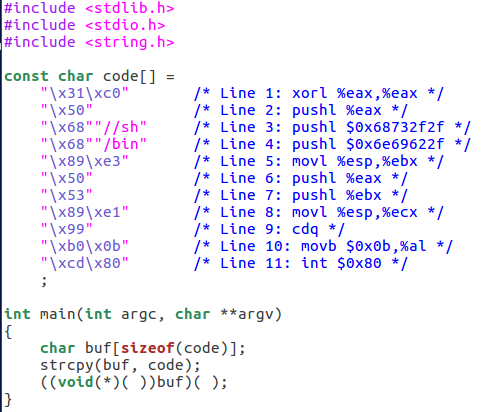


1. 关闭“Stack Guard”机制，在编译相关程序时设置对应参数；
2. 允许栈执行，在编译相关程序时设置对应参数；
3. 将 /bin/sh 链接到 /bin/zsh。

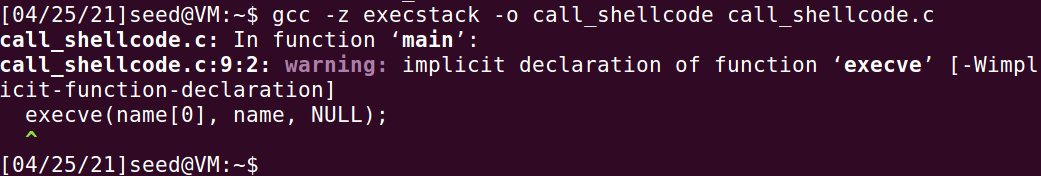


**Task 1**

创建文件call\_shellcode.c：



编译并设置允许栈运行：

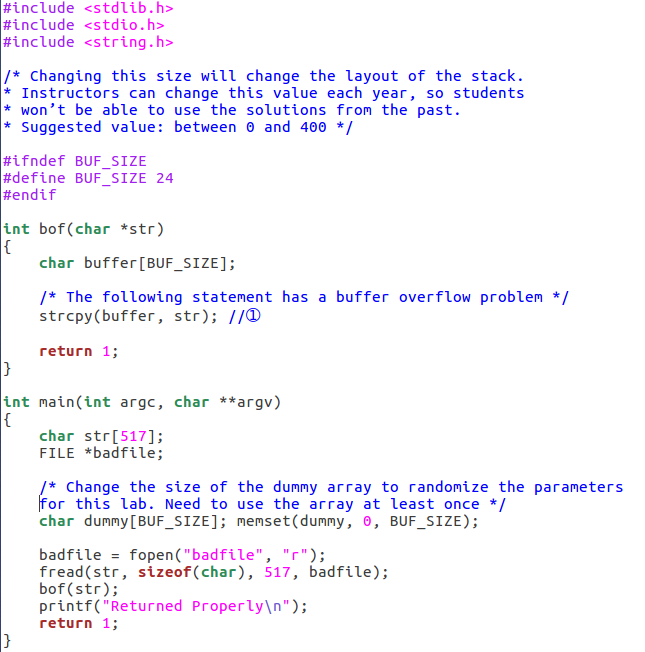


运行call\_shellcode开启了一个shell：



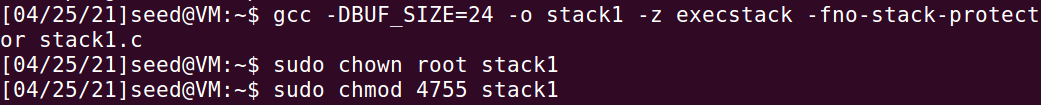
接下来使用缓冲区溢出攻击方式来尝试启动shell：

创建文件stack1.c：



可以看出本代码的作用是使用bof函数读取badfile的内容，并将其复制到缓冲区。

关闭防御机制编译stack1.c为可执行文件stack1，并将其设为root的Set-UID程序：

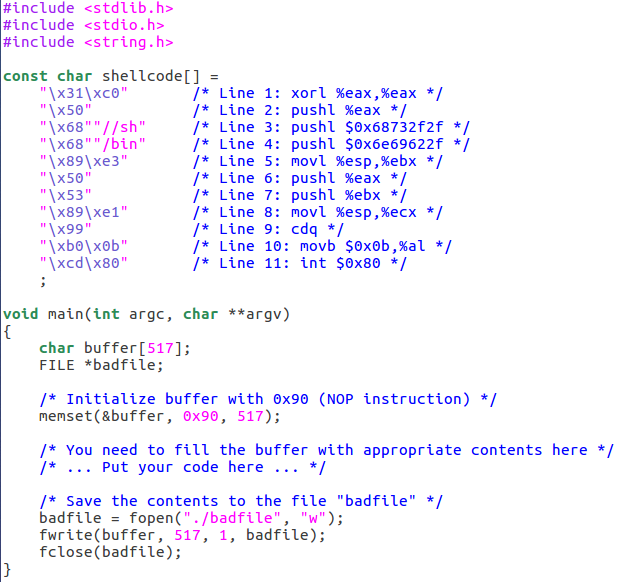


运行stack1，系统提示发生了段访问错误，这是由于还没有构建badfile文件。



**Task 2**

创建文件exploit.c：



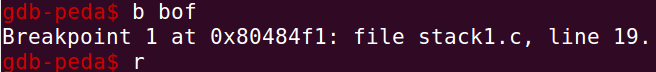
需要在红框部分加入代码，修改buffer数组特定位置的数据，将包含恶意代码的buffer数组传入badfile，这样运行stack1程序时就可以将恶意代码读入缓冲区，从而使得缓冲区溢出攻击成功。

为了构建badfile，我们需要获得stack1程序运行时main函数调用bof函数的返回地址以及shellcode数组（既main函数中char类型数组str）的起始地址。

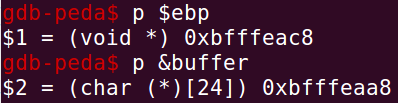
使用gdb对stack1进行调试：



对bof函数打上断点：

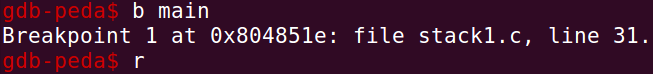


查看ebp与buffer地址：



可以看到ebp与buffer之间相差0x20，由于返回地址在ebp上4字节处，由此可得到返回地址区域到buffer起始位置的距离为0x24。

对main函数打上断点：



查看str的起始地址：



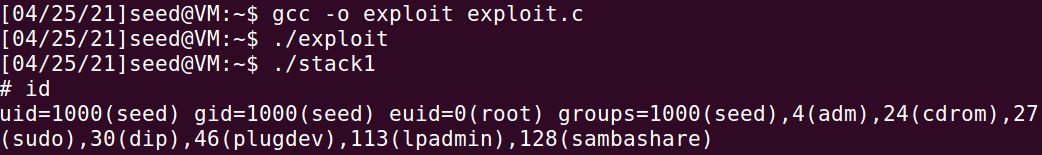
0xbfffeb07加上预定的shellcode偏移量100（0x64）即得到新的返回地址：

0xbfffeb6a。

在exploit.c中红框部分加入如下代码：



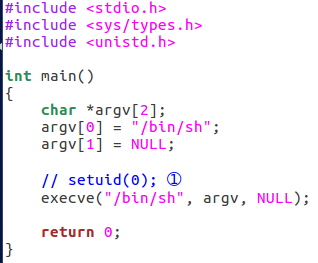
编译并运行exploit，然后运行stack1：



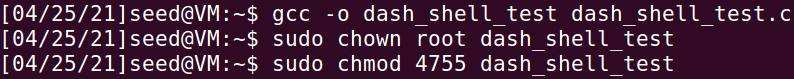
成功开启shell，验证为root权限（euid为root）。

**Task 3**

创建文件dash\_shell\_test.c：



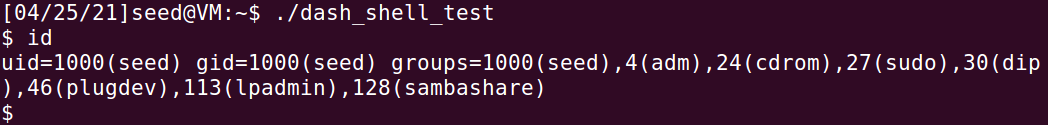
编译并将其设置为root的Set-UID程序：



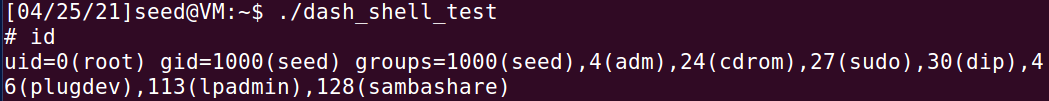
更改/bin/sh链接为/bin/dash：



运行dash\_shell\_test：



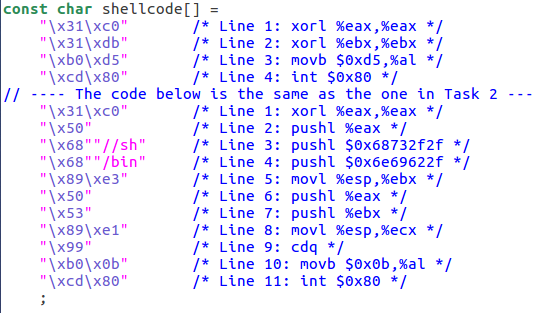
将dash\_shell\_test.c文件中setuid(0)注释取消，重新编译设置运行：



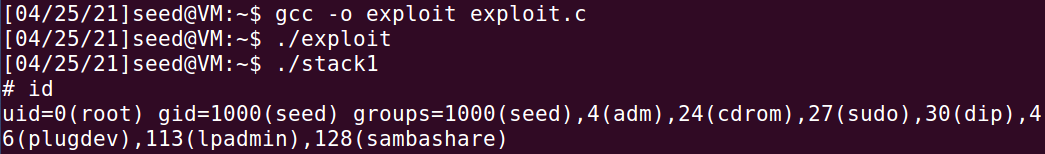
可以看到运行用户(uid)变为了root。

使用缓冲区溢出攻击进行同样操作：

将exploit.c中的shellcode数组改为：

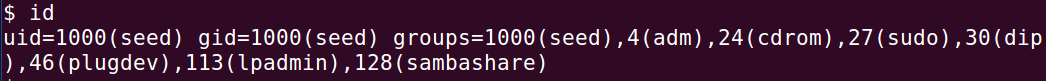


编译并运行exploit，然后运行stack1：



获得了root运行的shell。

将前四句注释掉，恢复task2的操作：



发现虽然stack1为root用户拥有的Set-UID程序，本次攻击却只获得了一个普通用户的shell。

这是因为/bin/sh链接改为了/bin/dash，dash和bash拥有权限保护机制，当dash发现有效用户和运行用户不一样时，它们会把有效用户变成运行用户，主动放弃特权。而setuid(0)则可以穿过dash的保护机制，直接进行系统调用将运行用户更改为root。被注释掉的前四条指令可以执行setuid(0)系统调用，从而获得root权限。

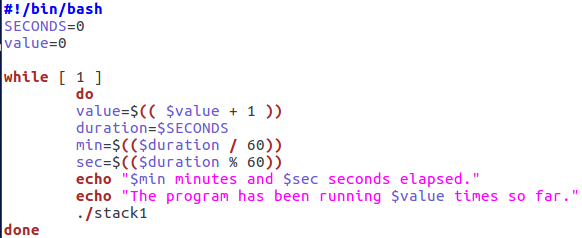
**Task 4**

在32位的linux系统中，栈空间拥有19bit的熵，这意味着栈地址最多有219=524288种可能性，这个数目可以轻易地通过暴力方法来遍历。也就是说，可以使用暴力攻击方法来破解地址随机化。

打开 Ubuntu 的地址随机化：

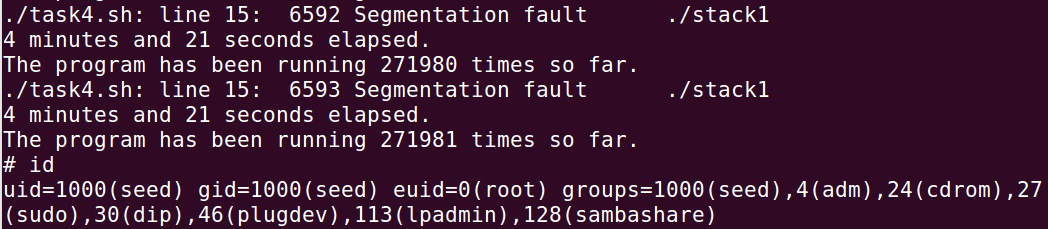


创建文件task4.sh：



由于地址随机化，注入badfile中的地址可能是错误的，而此脚本的作用是不断循环进行task2中的缓冲区溢出攻击，直到badfile中的地址正确。

赋予权限并运行task4.sh：



运行脚本进行了271981次尝试后，恶意代码得到了执行，成功启动了root权限的shell。

**Task 5**

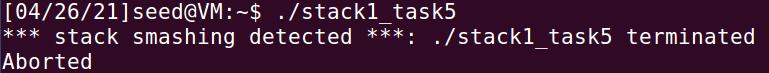
尝试在StackGuard下进行缓冲区溢出攻击，即在编译程序时不带-fno stack protector选项：



为了控制唯一变量，将地址随机化关闭：



将stack1\_task5设置为root用户拥有的Set-UID程序，运行：



StackGuard检测到缓冲区溢出，输出“stack smashing detected”并终止程序。

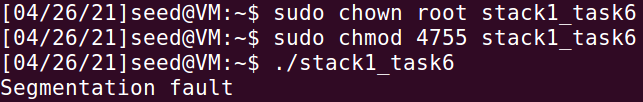
由于缓冲区溢出攻击修改返回地址时，所有处于缓冲区和返回地址之间的数据内容都会被修改，而StackGuard机制会在缓冲区与返回地址之间放置一个不可预测的数值哨兵。在被调函数返回之前，StackGuard会检测哨兵是否被修改，如果哨兵被修改，则说明发生了缓冲区溢出问题。

**Task 6**

尝试在不允许栈运行的情况下进行缓冲区溢出攻击，即编译程序时使用-z noexecstack选项（同样将地址随机化关闭）：



将stack1\_task6设置为root用户拥有的Set-UID程序，运行：



系统提示发生段错误，攻击失败。

缓冲区溢出攻击的关键在于执行保存在栈中的shellcode，将栈设置为不可执行，能够将代码和数据分离开来，处理器就会拒绝运行被被标记为不可执行内存区域中的任何代码，这样缓冲区溢出攻击就会无法成功。