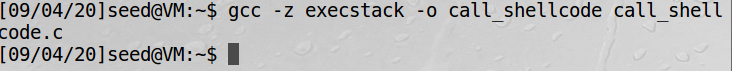
**Lab2**

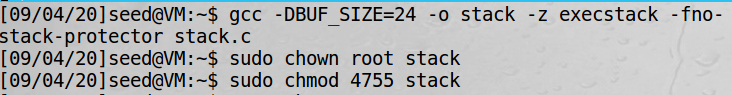
**57118111 王雯**

**实验目的：**

本实验室的学习目的是获得关于缓冲区溢出漏洞的经验，通过从阶级中学到的关于脆弱性的知识付诸行动。缓冲区溢出是定义为程序试图写入超出预分配边界的数据的一种情况固定长度的缓冲区。这个漏洞可以被恶意用户用来改变程序的流控制，导致恶意代码的执行。此漏洞是由于混合存储而产生的。通过一个具有缓冲区溢出漏洞的程序，来掌握这些，

**Task 1: Running Shellcode**

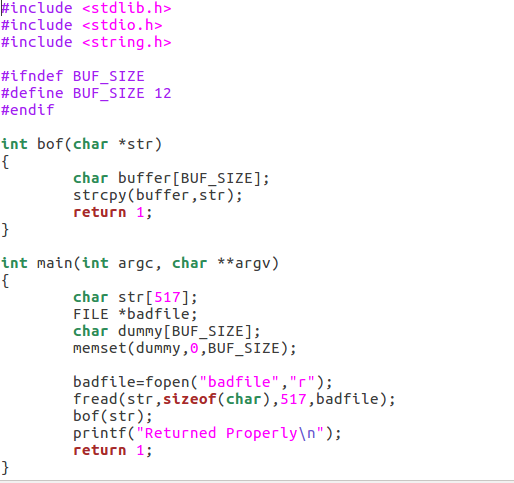


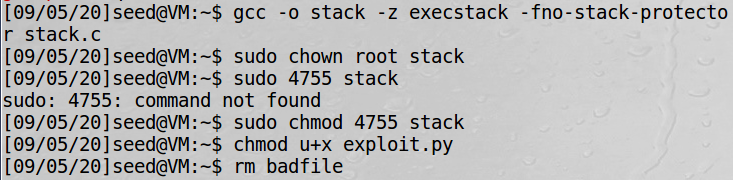


**Task 2: Exploiting the Vulnerability**

首先：我们在task1的基础上，编译stack.c文件，我们要使该文件成为易受攻击的文件，要使用-fno-stack-protector和-z execstack关闭StackGuard和non-executable stack protections，然后使程序成为root的Set-UID程序。

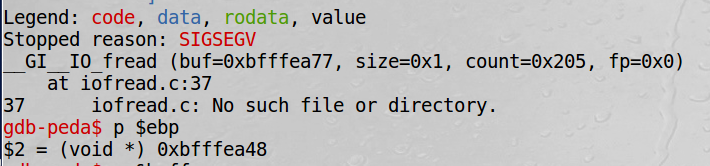
由于原本的stack.c文件具有缓冲区溢出漏洞，它首先从一个名为badfile的文件中读取输入，然后将该输入传递到bof()函数中的另一个缓冲区。原始输入的最大长度可以是517字节，但是bof()中的缓冲区只有BUF\_SIZE字节长，小于517。因为strcpy()不检查边界，所以会发生缓冲区溢出。由于这个程序是root拥有的Set-UID程序，如果普通用户利用这个缓冲区溢出漏洞可能能够获得一个root shell，程序从一个名为badfile的文件中获取输入，此文件由用户控制。

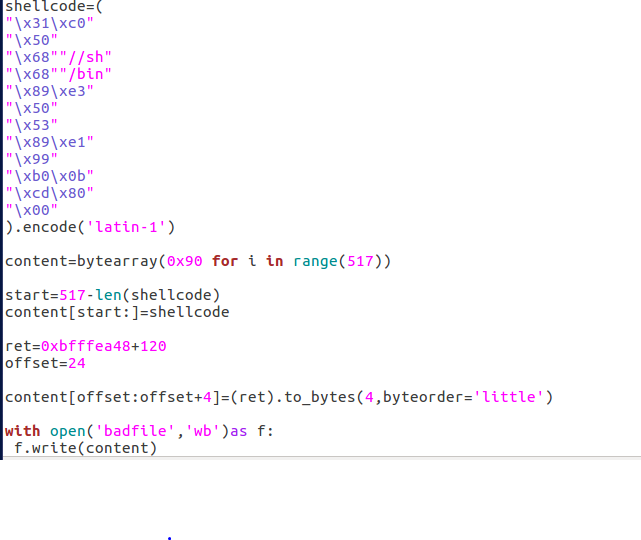




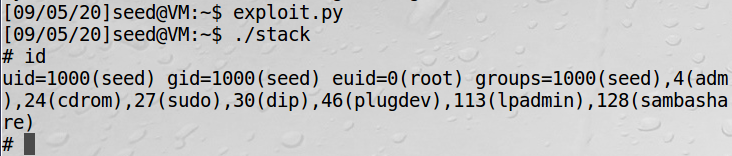
其次：我们将我们写好的攻击程序exploit.py编译并运行，这将会生成badfile的内容，然后运行易受攻击程序堆栈，发现得到rootshell，进入了特权模式。

此中，我们要找到buffer，ebp的地址，计算出距离。然后我们计算shellcode的地址，写入我们exploit.py的程序里，这是我们利用缓冲区溢出所要的覆盖的部分。



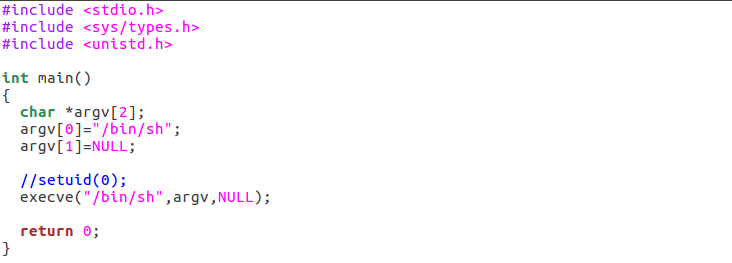


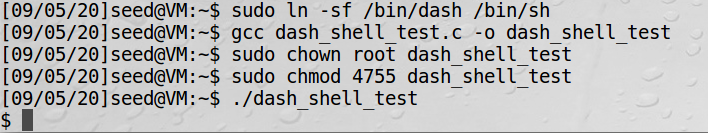
进入特权模式，漏洞被正确实现：



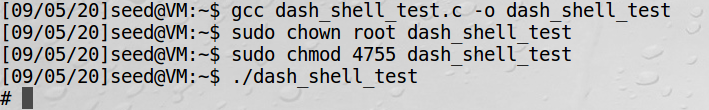
**Task 3: Defeating dash’s Countermeasure**

Task3在调用dash程序之前将受害进程的真实用户ID更改为零，可以通过在shell代码中执行execve()之前调用setuid(0)来实现Ubuntu 16.04中的dash shell会删除特权的程序。第一次执行：



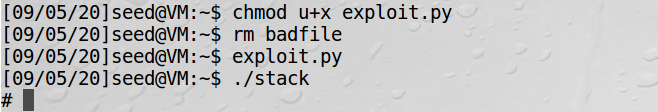


然后我们取消注释并再次运行程序，发现进入特权模式：



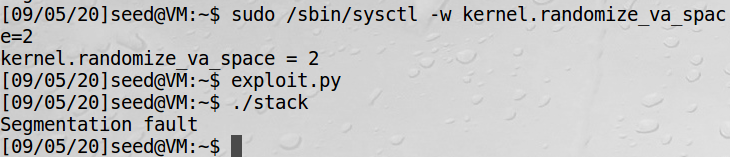
然后在exploit.py中添加4条指令，再次重复task2中的攻击，进入特权模式：





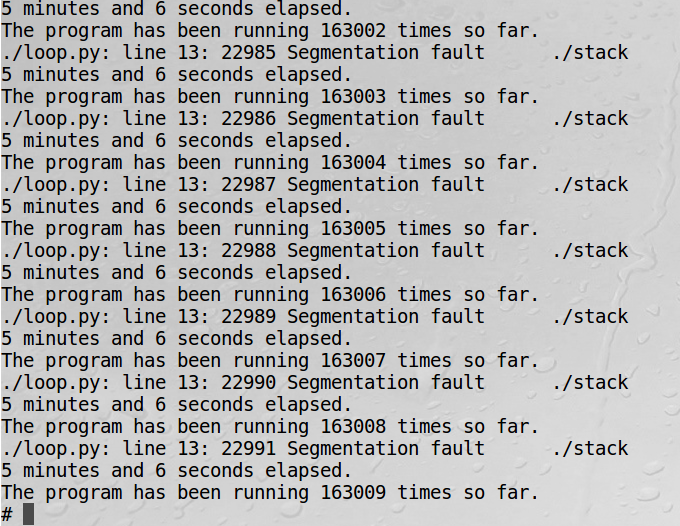
**Task4 Defeating Address Randomization**

首先：使用下面的命令打开Ubuntu的地址随机化：



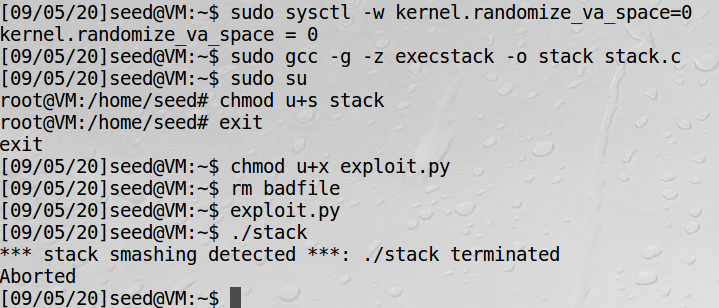
然后使用重复攻击易受攻击的程序，希望放入badfile的地址最终可以是正确的。使用shell脚本在无限循环中运行易受攻击程序。攻击成功，脚本将停止：

循环5min 6sec，执行了163009次后命中地址，进入特权模式



**Task5： Turn on the** **StackGuard Protection**

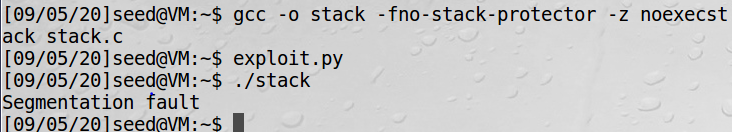
首先关闭地址随机化，在实验中，编译程序时不使用-fno-stack-protector。发现再次运行task2就不会进入特权模式，而是出现了缓冲区溢出错误。



**Task6 Turn on the** **Non-executable Stack Protection**

首先关闭地址随机化，实验中使用noexecstack重新编译：

发现不能得到shell，而是显示Segmentationfault。



**实验感想**：

在本次实验中，通过对易攻击的程序对缓冲区漏洞的攻击，并由此获得root权限。对缓冲区溢出的问题有了更为深刻的理解，也知道了在编写程序的时候，堆栈问题的重要性。本次实验中由于前期对于堆栈中的return address，以及shellcode的地址的理解不到位，导致在task2的exploit.py文件中，我们攻击程序需要获取的地址存在很多疑问。通过积极问助教和同学，将此问题解决。