**Lab2实验报告**

57117235李娜

**Buffer Overﬂow Vulnerability Lab**

一、Turning Off Countermeasures

该步骤是关闭系统和编译器对缓冲区溢出攻击的防御机制，以最简单的方式模拟攻击。

1、关闭操作系统的地址空间随机化：



编译时取消StackGuard保护机制和栈不可执行机制。

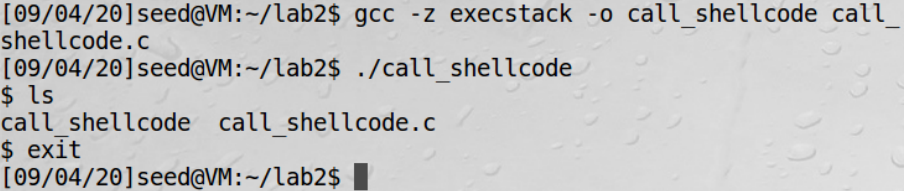
2、在Ubuntu 16.04中，将shell链接到/bin/zsh（否则dash shell有自动降权限的保护机制）：



二、Task1: Running Shellcode

本实验实现用shellcode调用shell，为后面的攻击提供可行性。

1、编译并运行call\_shellcode.c代码：

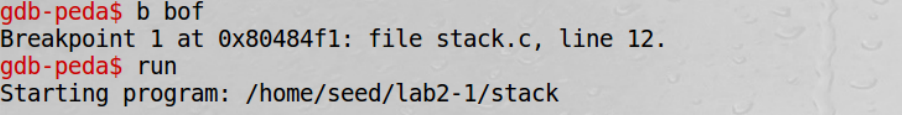


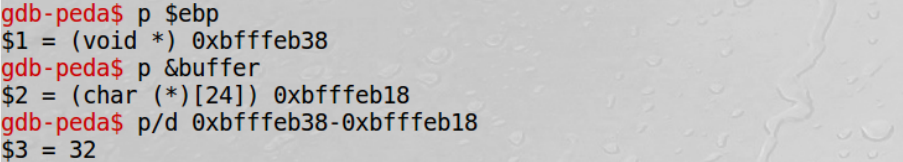
程序执行后，将buf转化为函数指针，并执行其中代码；然后调用系统调用函数execve()执行/bin/sh进入到shell界面，可以执行用户输入的命令。

三、Task2: Exploiting the Vulnerability

本实验实现缓冲区溢出攻击，漏洞程序stack.c将256字节的字符串拷贝到长为24字节的缓冲区中，可能会导致缓冲区泄露。

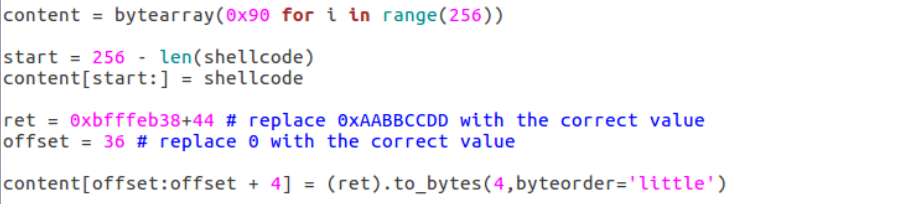
1、用gdb调试stack.c程序，在bof()函数中设置断点，找到ebp和buffer之间的距离为32字节：





则buffer与Return Address之间的距离为32+4=36字节。

2、据此修改并运行exploit.py程序，填充badfile的内容，使缓冲区溢出时正好能实现事先写好的恶意代码：

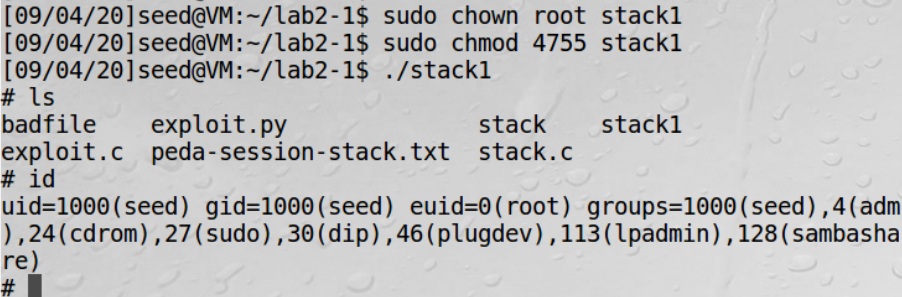


将原函数栈中的Return Address覆盖成比RA低4字节的地址，则函数调用结束后将返回到该地址，并执行NOP指令，一直执行到shellcode指令，实现调用具有root权限的shell。

3、编译stack1.c，同时设置无栈保护机制和栈可执行：



4、再将stack1设为特权程序并执行：



成功进入到root权限的shell界面，即实现了缓冲区溢出攻击。

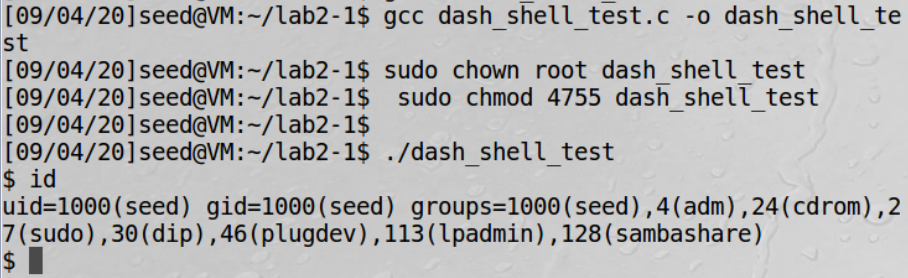
四、Task3: Defeating dash’s Countermeasure

dash shell在被调用时若发现EUID不等于用户的真实UID时，将会自动将root权限降为普通用户权限，则攻击者即使实现了缓冲区溢出攻击，也不能获得root权限。本实验验证了一种对抗这种保护机制的缓冲区溢出攻击。

1、将/bin/sh链接到/bin/dash：

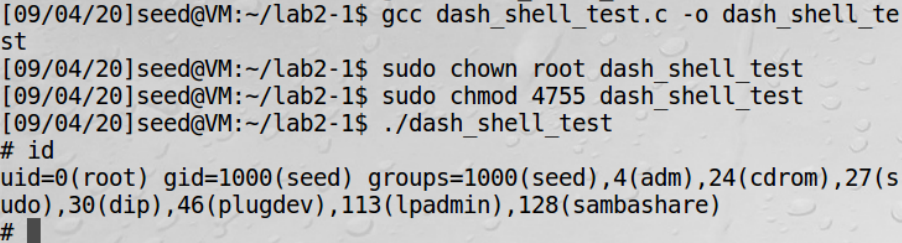


2、当第①行被注释时编译dash\_shell\_test并设置其为Set-UID程序，运行结果如下：



进入shell界面后，程序权限降为普通用户，不再有root权限，这是dash shell的保护机制。

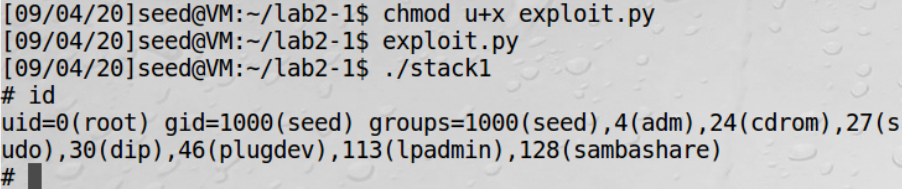
3、取消第①行的注释后，再编译dash\_shell\_test并设置其为Set-UID程序，结果如下：



可以看到，和上次结果不同的是，这次的shell具有root权限。即我们可以找到一个方法——setuid(0)，对抗dash shell的保护机制。

4、在dash shell下尝试实现缓冲区溢出攻击

修改exploit.py程序中的shellcode，使其在调用execve()函数之前，先调用setuid(0)函数，运行漏洞程序stack1：

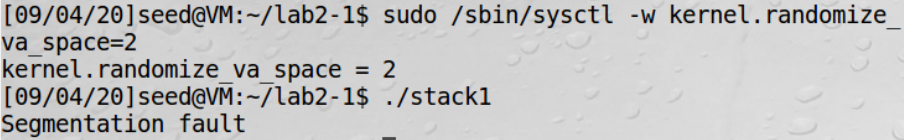


攻击成功，即使在dash shell的保护中，攻击者也能获得具有root权限的shell

五、Task4: Defeating Address Randomization

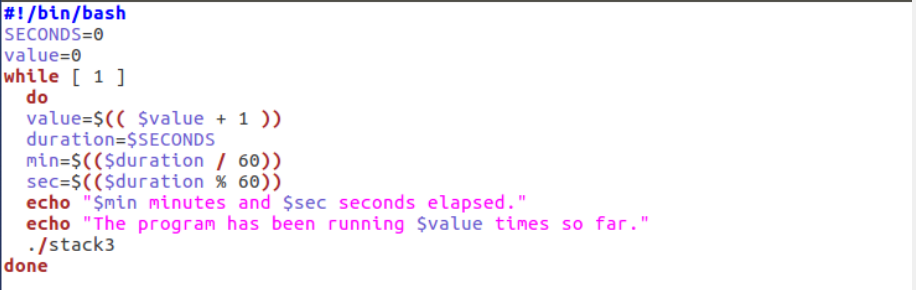
在操作系统实现地址随机化的情况下，本实验验证仍能通过暴力碰撞的方式实现缓冲区溢出攻击。

1、首先启动操作系统的地址随机化机制，验证Task2的攻击是否还有效：

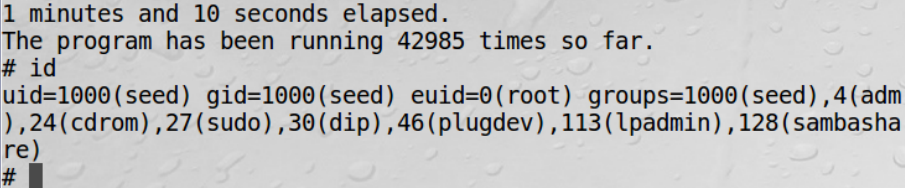


可以看到，由于地址随机，之前修改的RA等参数可能不再有效，导致攻击失败。

2、编写脚本script.sh，让其无限循环执行stack3，企图以暴力碰撞的方式碰到合适的地址：



运行脚本，得到结果如下：

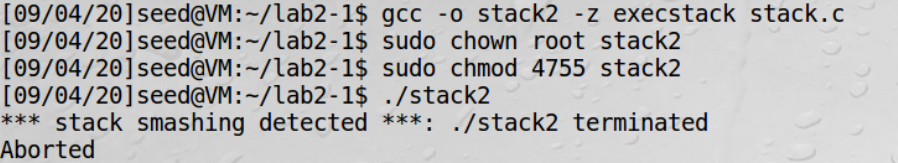


可以看到，程序执行了42985次，运行时间仅为1分10秒，就成功实现缓冲区溢出攻击。实验说明，即使开启了地址随机化，也无法完全抵挡缓冲区溢出攻击。

六、Task5: Turn on the StackGuard Protection

本实验在具有StackGuard保护机制的情况下，验证缓冲区溢出攻击是否可行。

1、关闭地址随机化，在StackGuard保护机制的情况下运行漏洞程序stack.c：

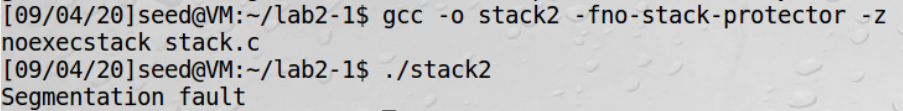


由于开启了StackGuard机制，编译器发现栈已被篡改，故不再执行程序，攻击失效。

七、Task6: Turn on the Non-executable Stack Protection

本实验在不可执行的栈保护的情况下，验证是否能实现缓冲区溢出攻击。

1、关闭地址随机化，在栈不可执行的情况下重新编译漏洞程序stack.c：



由于栈中shellcode代码不可执行，故攻击失败。

**Return-to-libc Attack Lab**

一、Turning off countermeasures

该步骤关闭一些保护机制，使得攻击更容易实现。

1、关闭地址空间随机化：



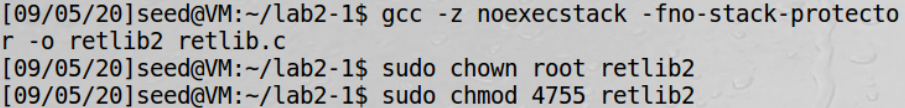
2、在编译时用-fno-stack-protector选项禁用StackGuard机制。

3、将shell链接到/bin/zsh，否则dash shell会有自动降权的保护机制:



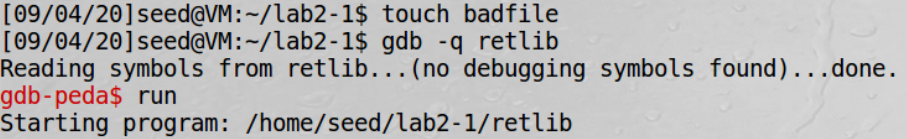
二、The Vulnerable Program

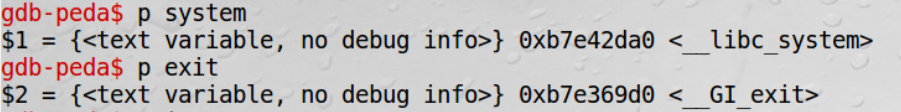
编译retlib.c，编译选项设为栈不执行，并更改其为Set-UID程序：



三、Task1: Finding out the addresses of libc functions

使用gdb找到libc中system()函数和exit()函数的地址：



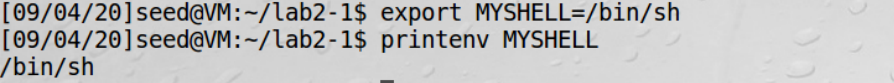


system()函数和exit()函数的地址分别是0xb7e42da0和0xb7e369d0。找到地址后，后续可以填进函数栈中的RA参数里，让system()函数被调用，从而获得一个shell。

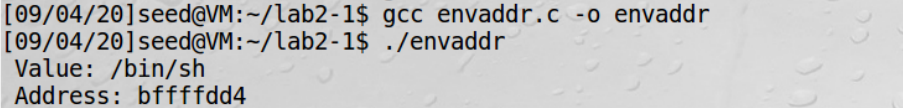
四、Task2: Putting the shell string in the memory

为了让system()函数执行/bin/sh，首先要把该字符串放进内存中，本实验使用环境变量的方式实现这一点。

1、定义一个新的环境变量MYSHELL，使其等于“/bin/sh”：



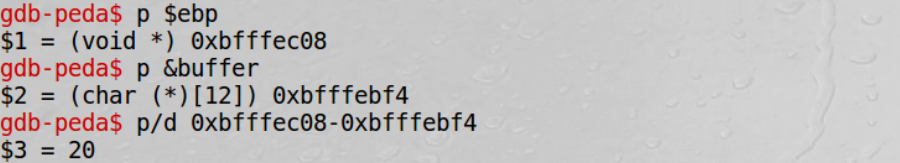
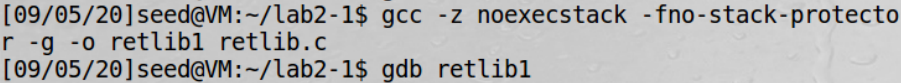
2、寻找该环境变量的地址，为0xbffffdd4：



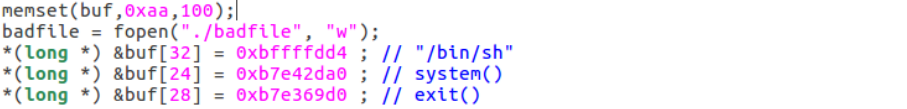
五、Task3: Exploiting the buffer-overﬂow vulnerability

本实验生成合适的badfile文件，当缓冲区溢出时，能够实现正确的代码覆盖，从而达成攻击。

1、用gdb调试，找到buffer和ebp之间的偏移为20：

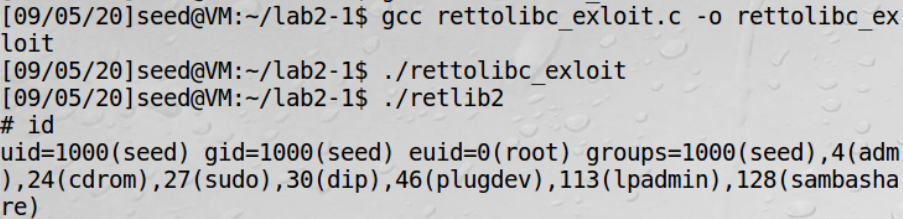


2、完善rettolibc\_exploit.c程序：



设offset为buffer和ebp之间的偏移，则offset+4为system()函数的地址，offset+8为exit()函数的地址，offset+12为环境变量“/bin/sh”的地址。

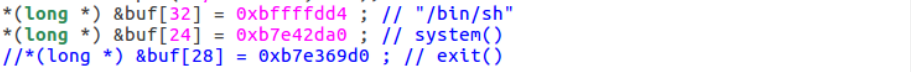
3、编译执行rettolibc\_exloit，生成badfile文件，执行retlib2，在栈不可执行的保护机制下实现缓冲区溢出攻击：



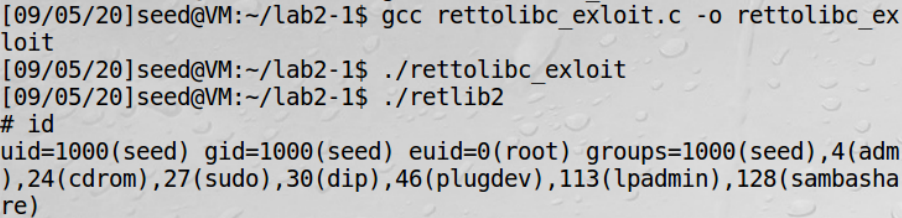
可以看到，成功以root权限获得了一个shell，攻击成功。

4、Attack variation1：验证无exit()函数地址时，攻击是否生效。

（1）将exit()行注释掉：



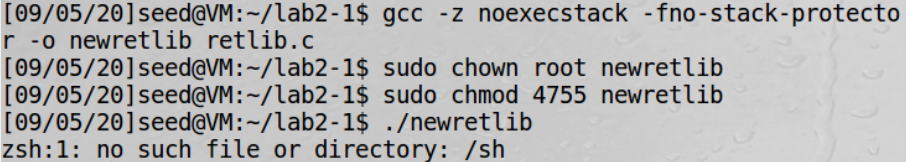
（2）重新编译执行rettolibc\_exloit，生成新的badfile文件，并尝试攻击：



获得root权限的shell，攻击成功，说明把exit()函数的地址写进函数栈不是必要的。

5、Attack variation2：修改漏洞程序的名字，验证攻击是否生效。

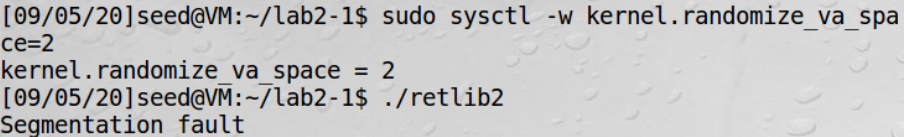
改变可执行程序名的长度，实施攻击：



未获得shell，攻击失败。环境变量MYSHELL的地址是对程序名长度敏感的，即程序名长度改变时，环境变量的地址也随之改变，则之前的地址不再适用于现在。即使调用了system()函数，也没有正确的输入字符串，导致攻击失效。

六、Task4: Turning on address randomization

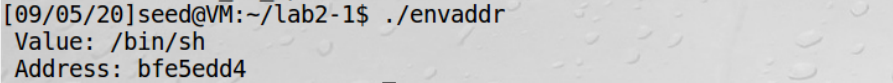
1、当开启地址随机化后，攻击失效：



2、编译器默认关闭地址随机化：



环境变量的地址改变：



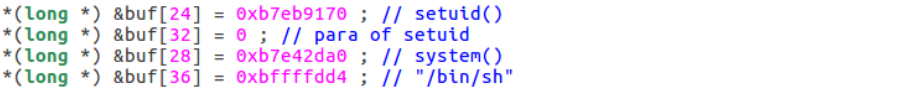
所以当开启地址随机化后，环境变量的地址受到了影响。

七、Task5: Defeat Shell’s countermeasure

1、用task1的方法找到setuid函数的地址，为0xb7eb9170：

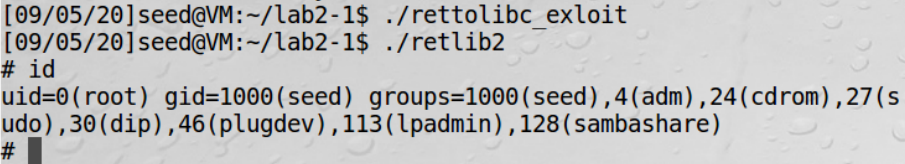


2、修改rettolibc\_exloit.c文件，把setuid()函数的地址和参数添加进去：



这样让程序执行时先调用setuid(0)函数，再调用system函数，setuid(0)函数将权限设为root，这样可以对抗dash的降权保护机制。

3、生成新的badfile文件，再执行攻击：



攻击成功。用户在/bin/dash的链接下，没有受到其保护机制的影响，仍获得了具有root权限的shell。