**信息安全实验报告**

**Lab 6 Return to Libc**

**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；

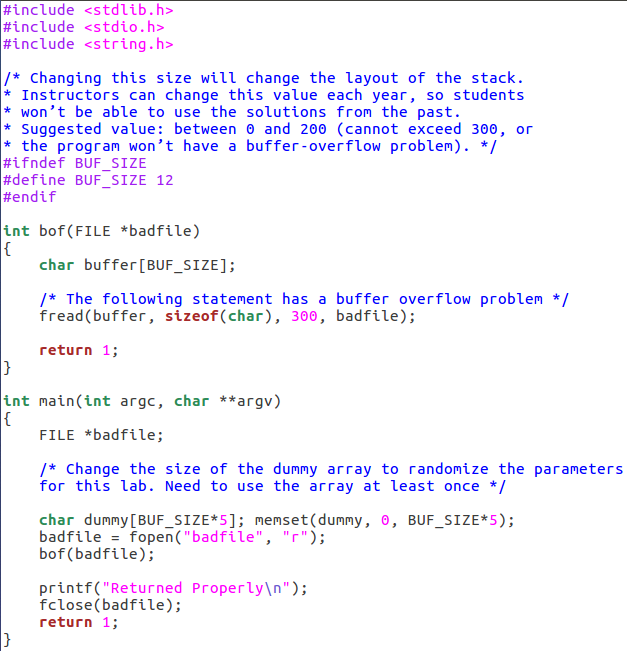


1. 关闭“Stack Guard”机制，在编译相关程序时设置对应参数；
2. 与普通的缓冲区溢出不同，本次实验要设置不允许栈执行，在编译相关程序时设置对应参数；
3. 将 /bin/sh 链接到 /bin/zsh。



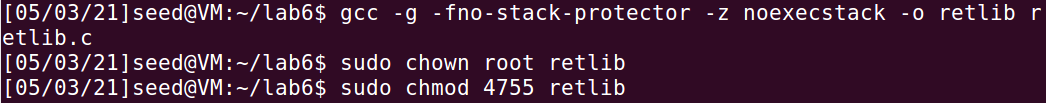
**Task 1**

创建文件retlib.c：

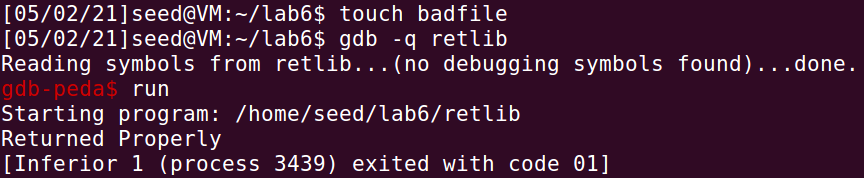


本代码的作用是使用bof函数读取badfile的内容，并将其复制到缓冲区。

编译并设置关闭“Stack Guard”机制，不允许栈运行。不设置-DBUF\_SIZE字段，令BUF\_SIZE使用默认的值12。将产生的可执行文件retlib设置为root的Set-UID程序：

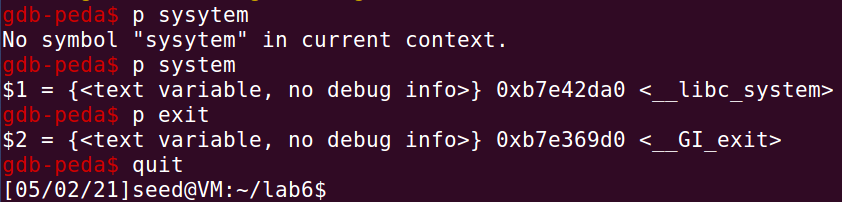


使用gdb来对retlib进行调试跟踪：



这里需要用“run”指令来执行，不然libc函数库不会被加载到内存中。

打印system函数以及exit函数地址：

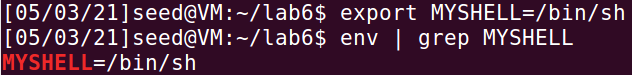


**Task 2**

考虑如何令system函数运行“/bin/sh”命令来获取一个shell。

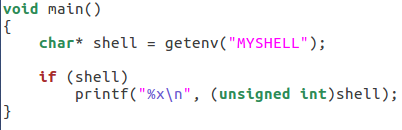
首先需要让内存中有一个字符串“/bin/sh”，然后将字符串地址作为参数传给system函数。利用环境变量传递给子进程的机制就可以让字符串进入程序内存。

定义一个环境变量MYSHELL=“/bin/sh”：



这样执行漏洞程序时，MYSHELL环境变量就会出现在程序的内存中，接下来需要找到MYSHELL在内存中的地址。

创建文件envaddr.c：



代码作用是打印出此进程内存中MYSHELL环境变量的地址。

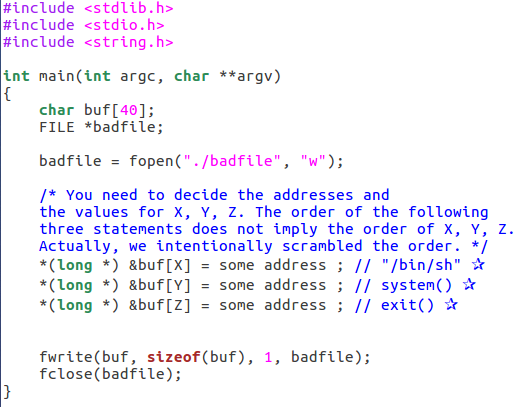
编译运行envaddr.c，得到环境变量MYSHELL即字符串“/bin/sh”地址。



需要注意的是，在环境变量入栈之前，需要将程序名称压入栈，这就导致程序名称的长度会影响此程序内存中环境变量的地址。

**Task 3**

创建文件exploit.c：



此代码作用是将char类型数组buf写入badfile文件。为了使攻击成功，需要修改buf数组的特定位置，在buf[X]放入system参数的地址，在buf[Y]放入system函数地址，在buf[Z]放入exit函数地址。

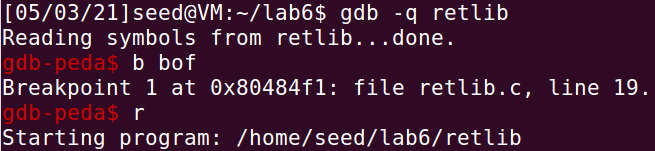
将envaddr.c编译为与漏洞程序retlib名字长度相同的可执行文件env555：



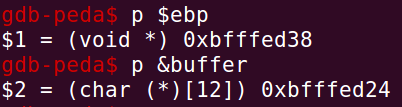
得到retlib执行时字符串“/bin/sh”的地址：



跟踪调试retlib，在被调函数bof处打上断点，运行到bof函数：

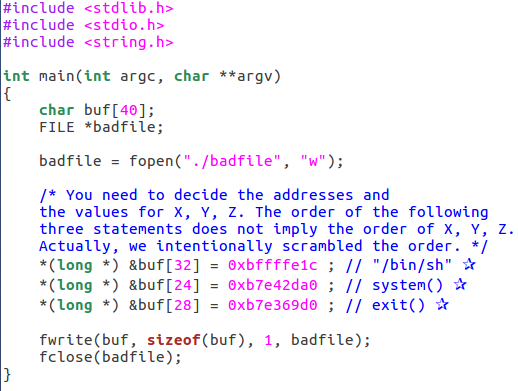


查看此时ebp与buffer的地址：

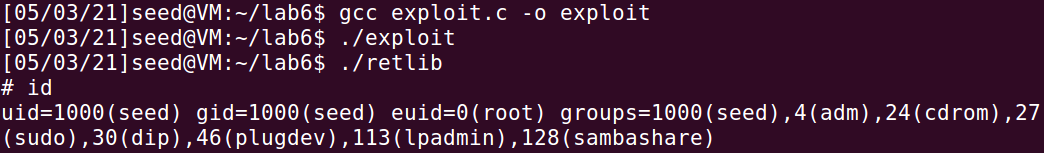


可以看到ebp与buffer地址差值为0x14，也就是说此时ebp放在buf[20]上，bof的函数调用栈的返回地址在buf[24]。此时将system函数地址放在buf[24]就可以让system函数调用取代函数返回操作。在此基础上可以推算得到exit函数地址要放在buf[28]，system函数参数“/bin/sh”的地址要放在buf[32]。

将对应地址填入exploit.c中：



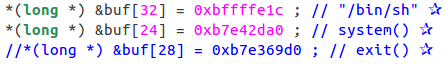
编译并运行exploit，然后运行retlib：



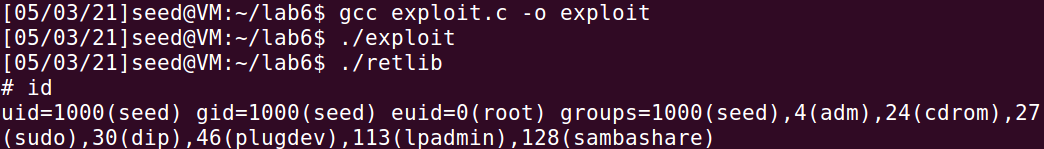
成功得到root权限的shell。

**变体1：**

将exit函数地址注释：



重新进行攻击

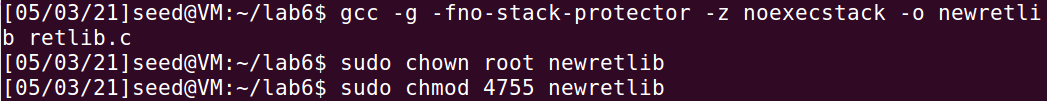


结果没有变化，说明在执行system函数之前，exit地址并不必要。

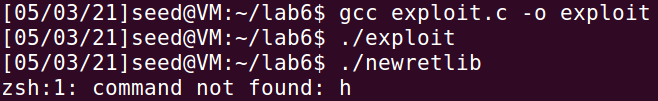
exit函数对应的是system函数的返回地址，如果不加以设置，当system函数返回时程序可能会崩溃。但是在调用system函数启动root权限shell时，exit地址设置不会产生影响。

变体2：

将retlib改名为newretlib：



重新进行攻击：

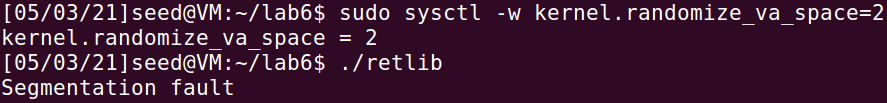


返回错误信息，“没有命令h”。

分析错误信息可知，这是由于system函数被传入了错误的参数。由于程序名变长，其内存中环境变量MYSHELL的地址也发生了变化，这就导致之前写入badfile中的字符串“/bin/sh”地址与真实地址不一致。此时需要修改env555的名称，使之与newretlib名字长度相同，输出新的地址，将其填入exploit.c中的“/bin/sh”地址即可。

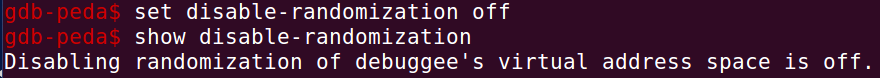
**Task 4**

将地址随机化开启，并重新进行攻击：

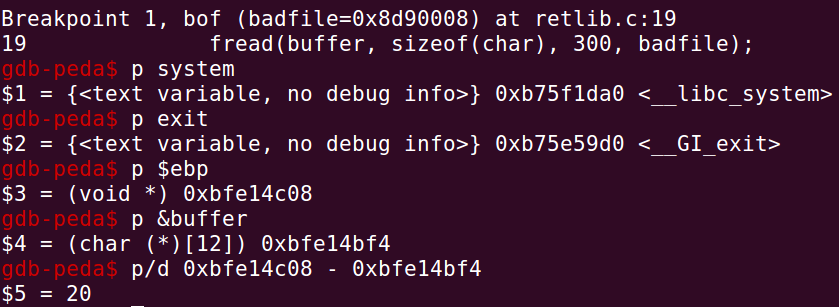


提示发生段错误。

gdb调试跟踪retlib，禁止随机化选项是默认打开的，将禁止随机化关闭：

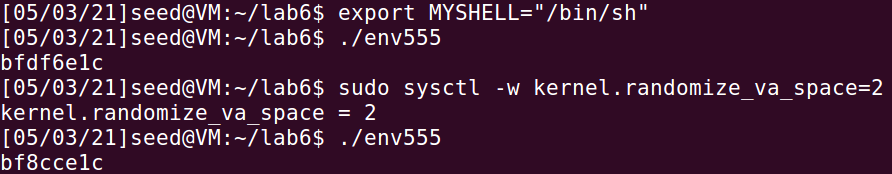


在函数bof处打上断点，运行至断点处:



可以看到buffer和ebp的地址差依然是20 (0X14)，说明由此地址差计算得到的buf数组下标X，Y，Z都没有发生变化，但是system地址和exit地址都发生了变化。

查看环境变量MYSHELL地址：



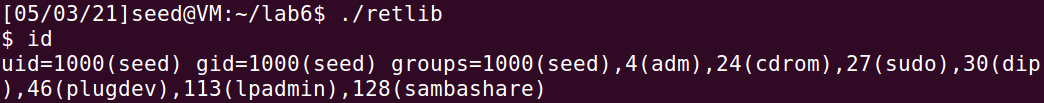
发现关闭地址随机化前后MYSHELL地址发生变化。

**Task 5**

尝试在dash下进行攻击，将shell连接至dash：



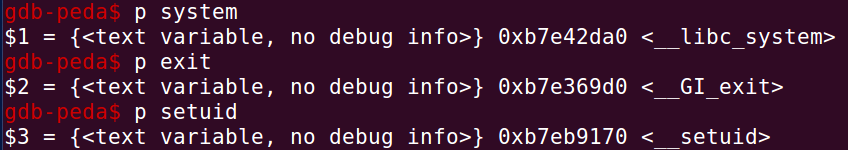
重新进行攻击：



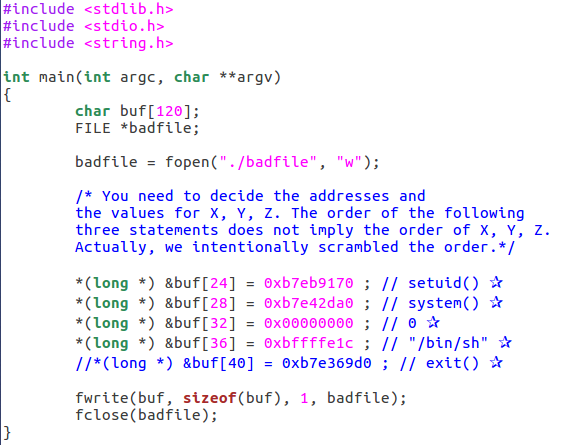
发现获得的只是普通权限的shell，这是因为dash拥有防御机制，当它运行在Set-UID进程时，会将有效ID变成实际ID，主动放弃特权。

为了获取root权限的shell，尝试在system函数之前调用setuid(0)来攻破dash的防御机制。

关闭地址随机化，调试跟踪retlib，打印system函数，exit函数，setuid函数地址：

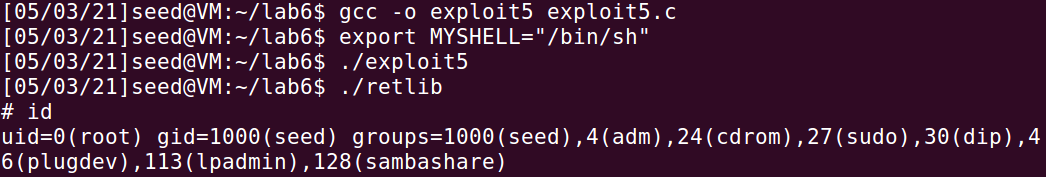


创建文件exploit5.c：



将setuid函数地址放在buf[24]，将函数参数0的地址放在buf[32]，在返回地址buf[28]放入system函数，再将system函数的参数“/bin/sh”放入buf[36]。

编译并运行exploit5.c，重新进行攻击：



成功攻破dash，得到root权限的shell。