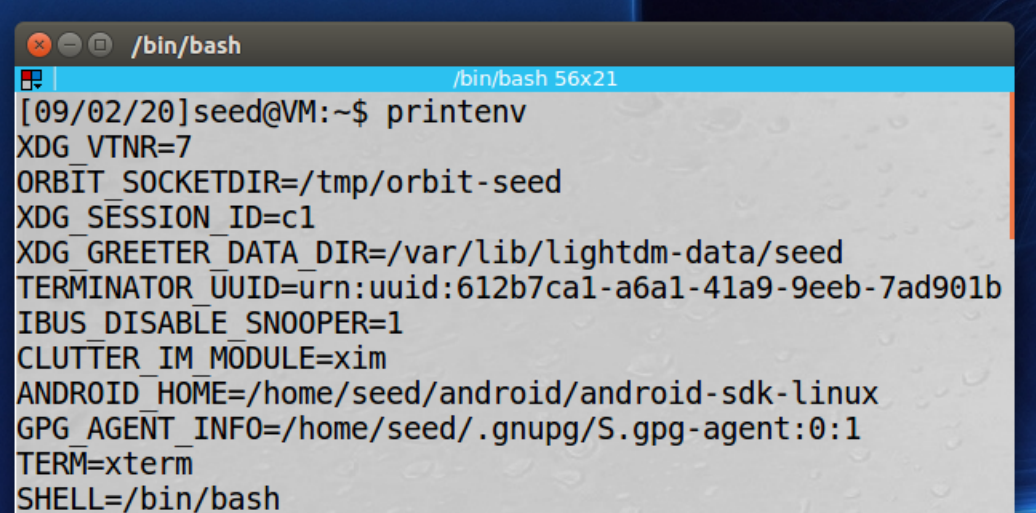
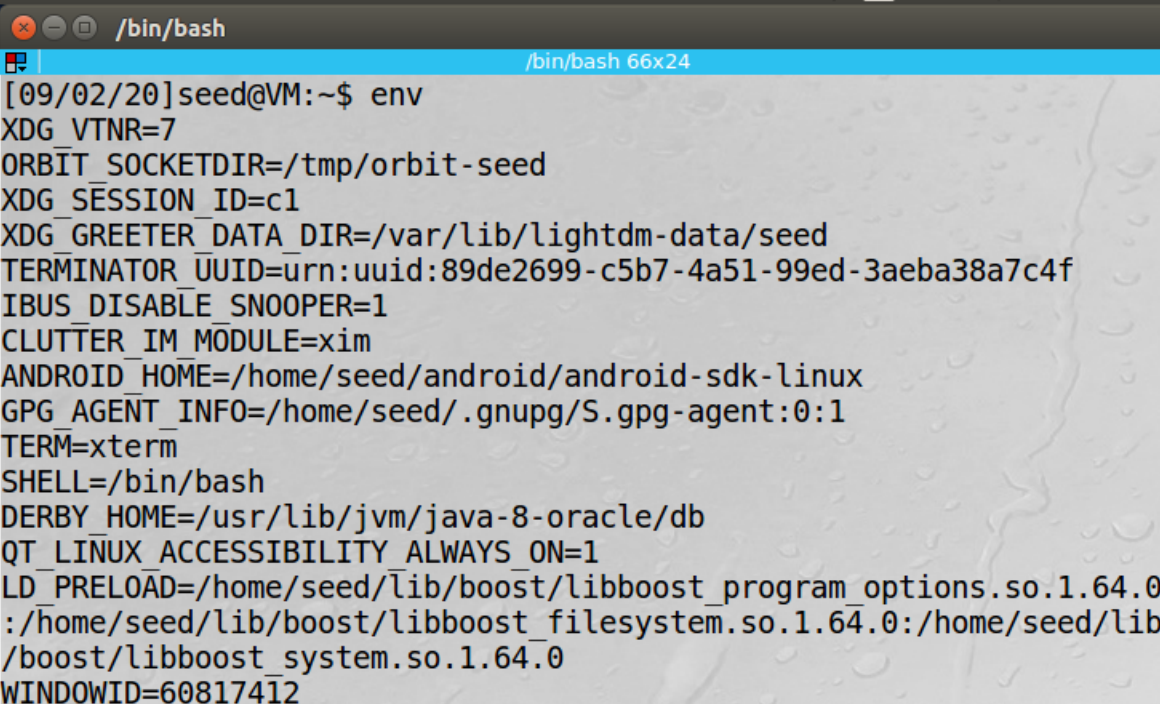
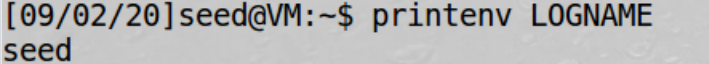
## Task 1: Manipulating Environment Variables

一、打印环境变量





二、打印具体环境变量





三、新增或修改一个环境变量

[root@VM\_0\_12\_centos ~]# export var1=value1　　#定义一个环境变量

[root@VM\_0\_12\_centos ~]# env|grep var

var1=value1

MAIL=/var/spool/mail/root

[root@VM\_0\_12\_centos ~]# export var1=value11111　　#修改上面的环境变量

[root@VM\_0\_12\_centos ~]# env|grep var

var1=value11111

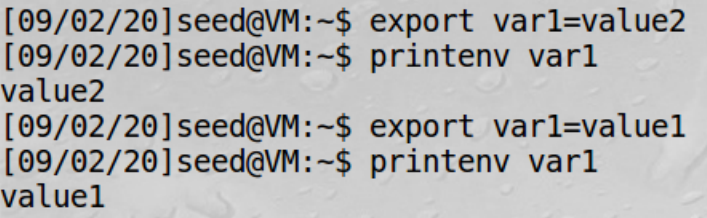
MAIL=/var/spool/mail/root

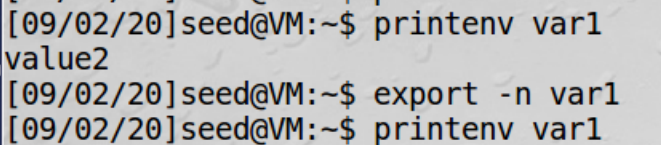
[root@VM\_0\_12\_centos ~]# export -n var1　　#删除上面的环境变量，只是从环境变量中删除，shell变量仍然存在

[root@VM\_0\_12\_centos ~]# env|grep var

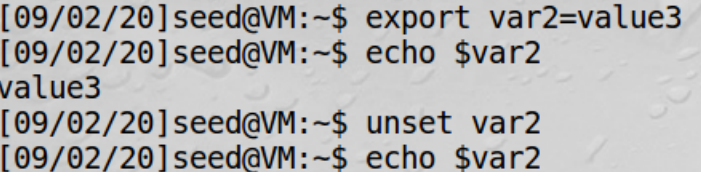
MAIL=/var/spool/mail/root

[root@VM\_0\_12\_centos ~]#





四、删除环境变量



## Task 2: Passing Environment Variables from Parent Process to Child Process

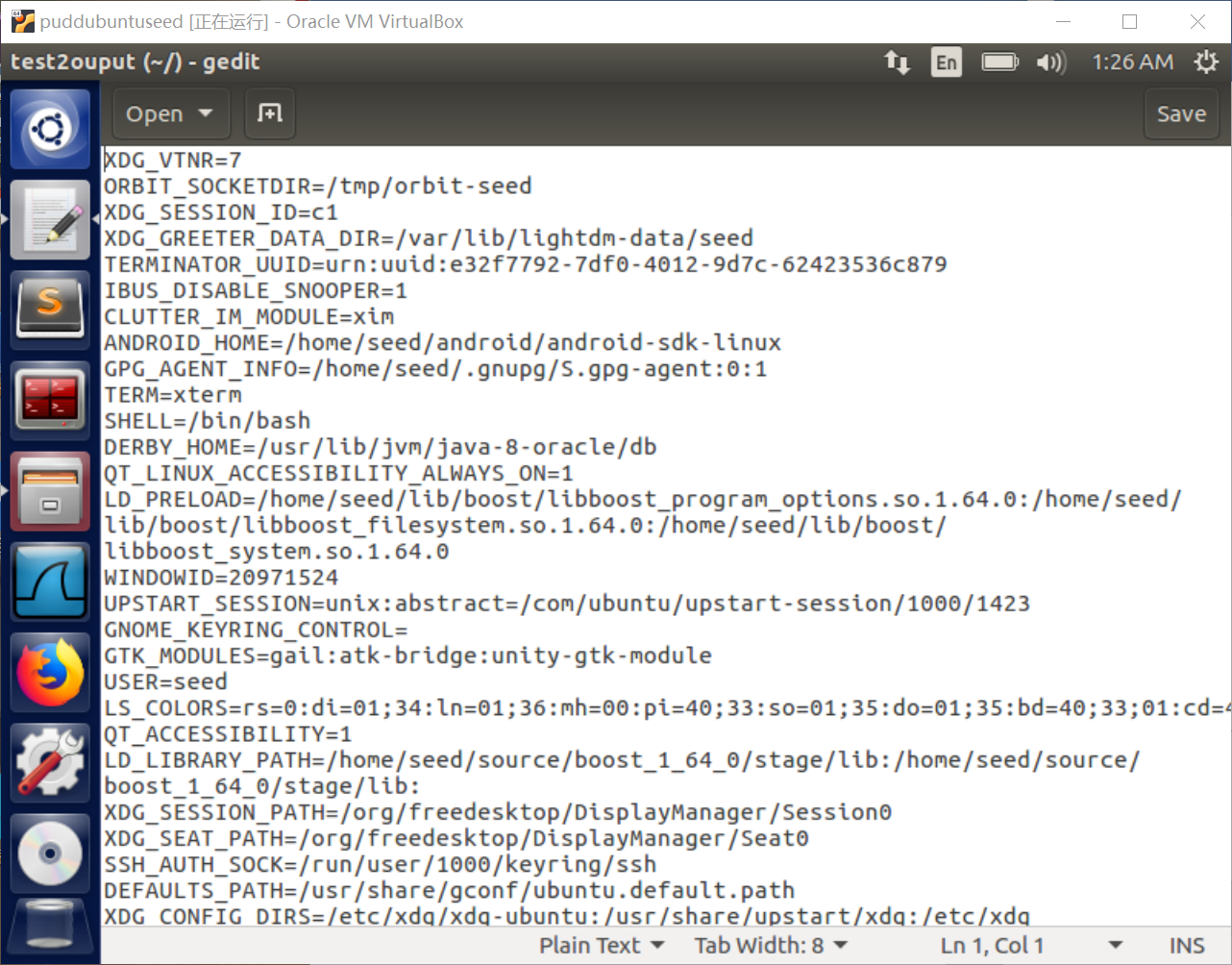
vim test2.c

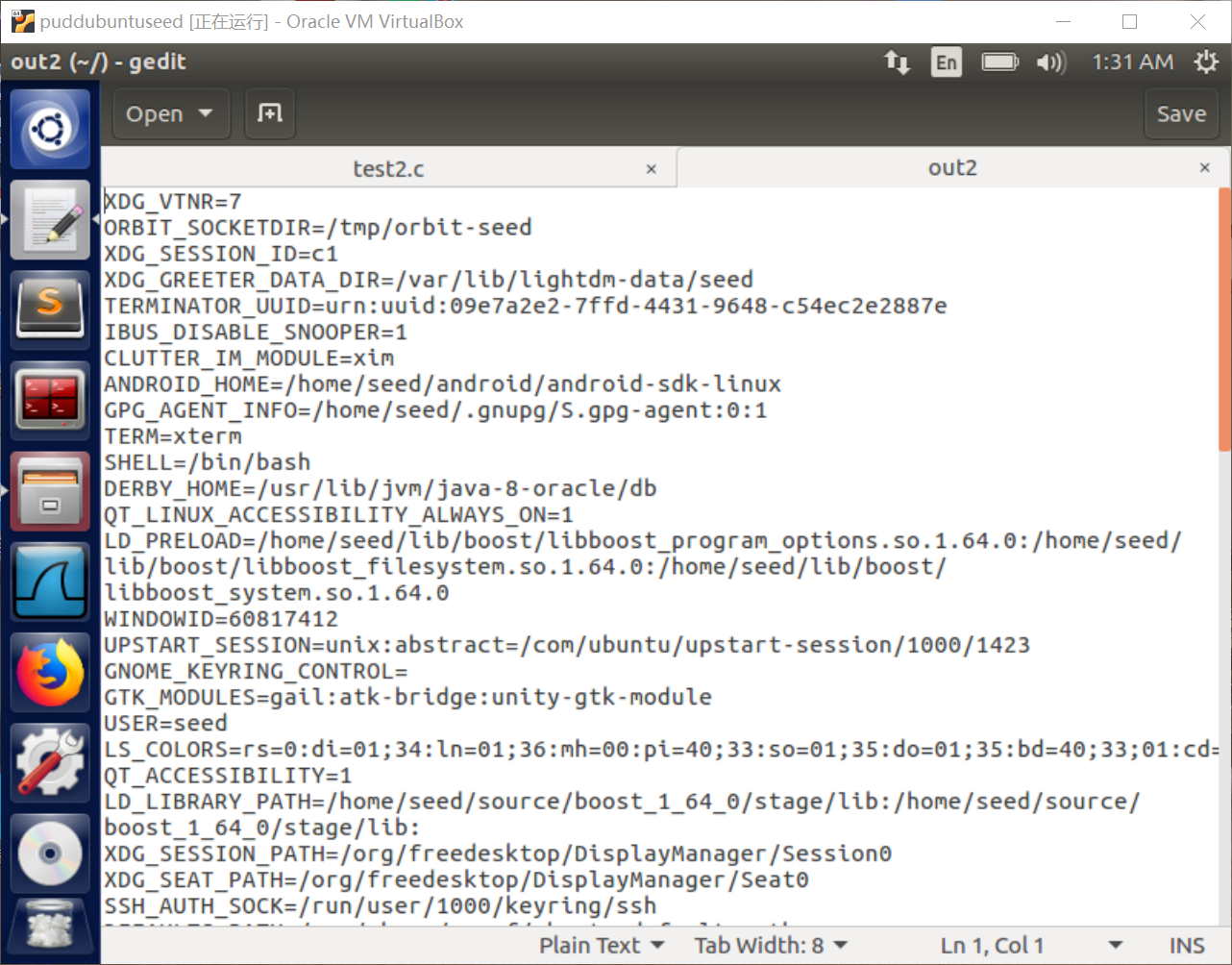
gcc test2.c -o test2

./test2

test2 > test2ouput

修改后：test2>out2





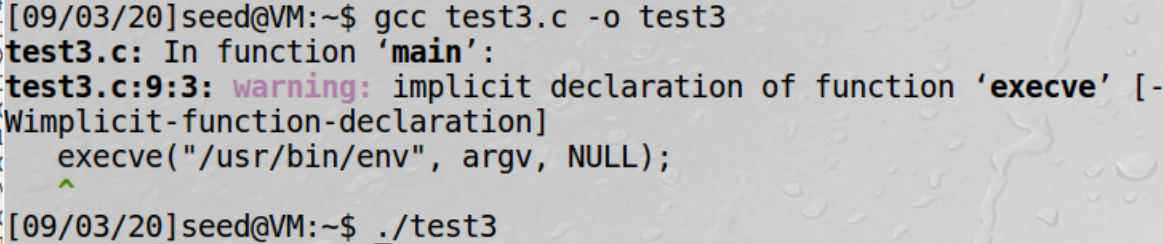


子进程环境变量完全继承。

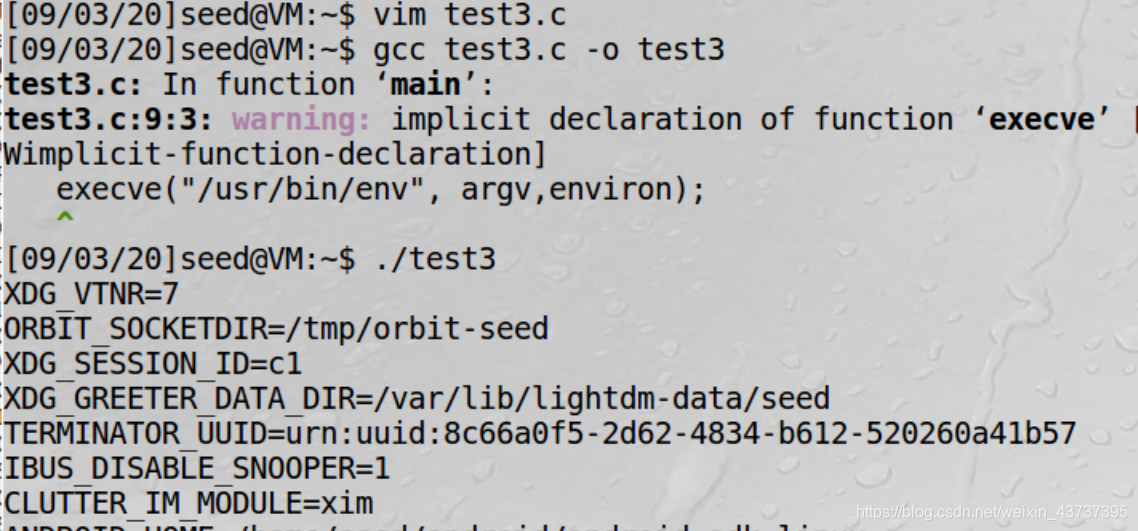
### Task 3: Environment Variables and execve()

execve()：比system()安全的。

第三个参数为空时



啥也没有。说明没传环境变量。

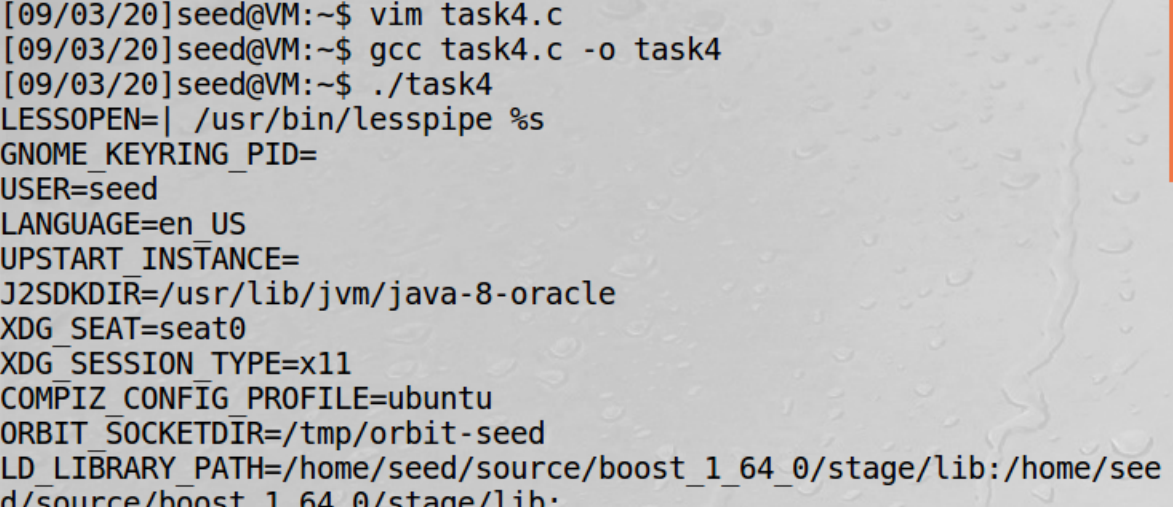


结论：

execve函数接收3个参数，第一个是可执行文件的路径pathname，第二个是参数的指针数组argv，第三个是环境变量的指针数组envp。最后一个参数envp就是新进程或子进程继承的环境变量。第三个参数为空就不传环境变量。

## Task 4: Environment Variables and system()

证明using system(), the environment variables of the calling process is passed to the new program /bin/sh.

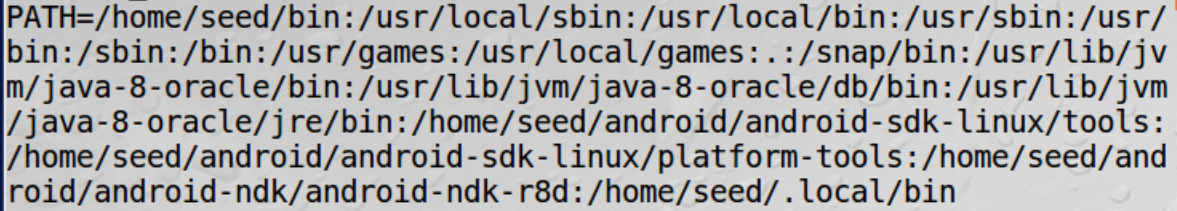
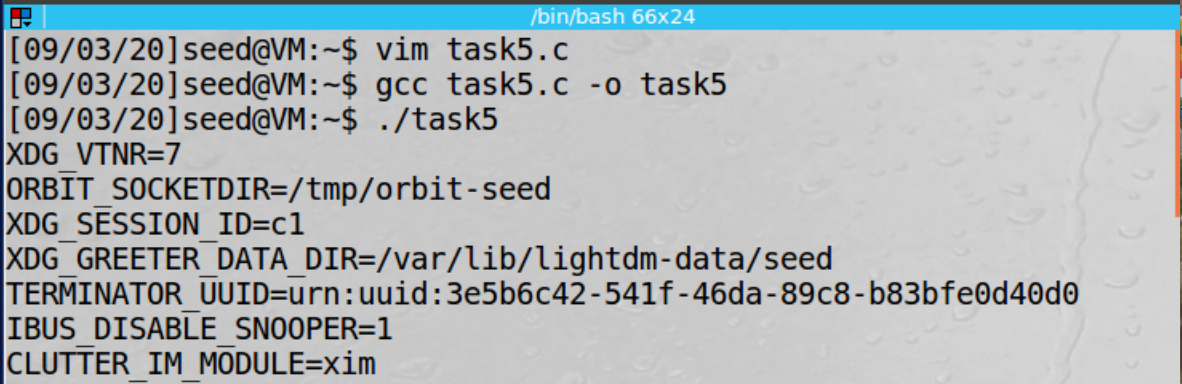


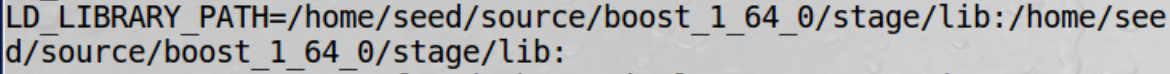
发现环境变量已经传递了。

## Task 5: Environment Variable and Set-UID Programs

此结果就是当前所有环境变量。

将上述程序的所有权改为root，并使它成为一个Set-UID程序  
先切换为root账户，使用chown root:root demo5.c将此c文件权限改为root权限

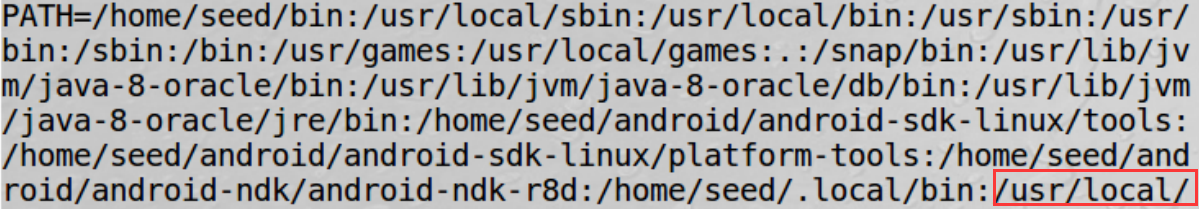


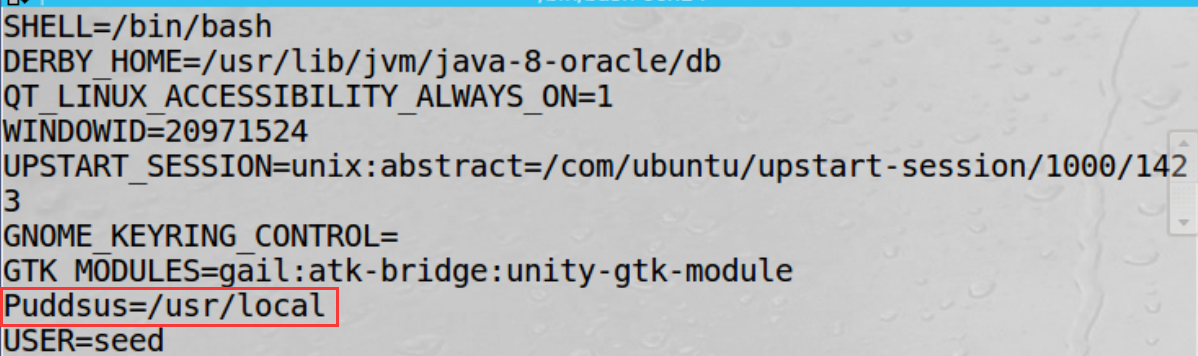


使用一般用户登录终端，使用export命令设置如下环境变量：PATH 、LD\_LIBRARY\_PATH、ANY\_NAME  
export PATH="$PATH:/usr/local/"  
export LD\_LIBRARY\_PATH="$LD\_LIBRARY\_PATH:/usr/local/"不继承  
export Puddsus="/usr/local"

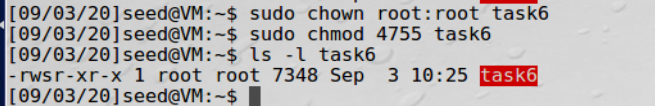
执行已经赋予Set-UID的task5程序，得到如下结果：

PATH增加，LD\_LIBRARY\_PATH不继承，Puddsus新增





Task 6: The PATH Environment Variable and Set-UID Programs



使用ls -l demo6语句查看文件的权限，验证操作确实成功完成，符合题设条件

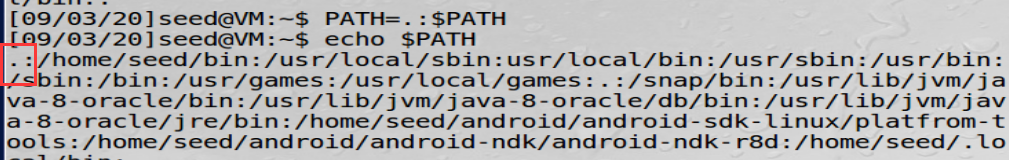
16.04防安全

“如果您有两个同名的可执行文件，位于两个不同的目录中，则 shell 将运行 $PATH 中的第一个所在的目录的程序。”

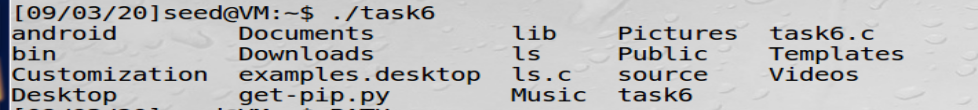
自己弄个ls文件放在当前目录，将当前目录路径放在原先path之前就行。

将bin/sh复制到当前目录并命名为ls，执行task6就会获得root权限。详细分析，先看一下PATH环境变量，它的命令找寻顺序是先找寻当前目录，而当前目录我们自己编造了一个ls，所以程序就会直接执行伪造的ls。sh原本的作用是创建一个新shell，在执行此命令后我们就会一直停留在子进程中，知道我们主动退出这个程序，我们才会回到原来的权限。

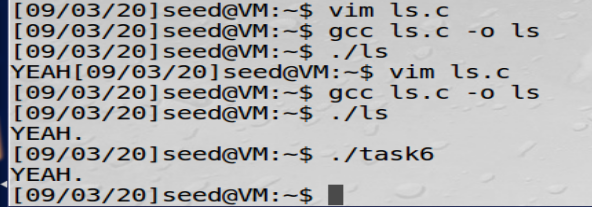
手动把当前目录添加在PATH之前



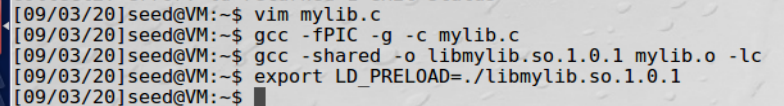
原本：



后来：



## Task 7: The LD PRELOAD Environment Variable and Set-UID Programs



在以下情况中运行myprog程序：

* 以普通用户的身份运行myprog程序。
* 以普通用户运行拥有root权限的myprog程序。
* 使myprog成为一个Set-UID user1程序，在user2用户（非root用户）中再次设置LD\_PRELOAD环境变量，运行myprog程序。

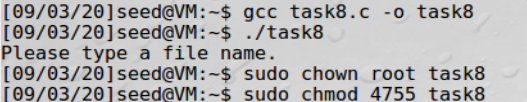
执行结果：

以普通用户的身份运行myprog程序时，输出：I am not sleeping!  
以普通用户运行拥有root权限的myprog程序时，无输出。  
在user2用户中再次设置LD\_PRELOAD环境变量并运行myprog程序时，输出：I am not sleeping!

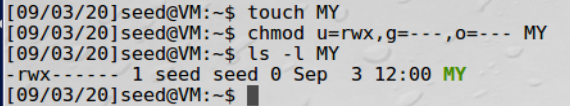
观察以上三次程序的执行结果，理解导致他们不同的原因。环境变量起了作用。设计实验证明主要因素，并解释第二步中行为的不同。

导致他们不同的原因就在于LD\_PRELOAD环境变量。LD\_PRELOAD环境变量是Unix动态链接库的世界中的一个环境变量，它可以影响程序的运行时的链接，它允许你定义在程序运行前优先加载的动态链接库。这个功能主要是用来有选择性的载入不同动态链接库中的相同函数。在该实验中，mylib.c通过sleep函数，生成了一个libmylib.so.1.0.1链接库。然后将该链接库添加到LD\_PRELOAD环境变量上。比较这三次实验，第一次和第三次实验myprog程序均具有seed用户权限，而在seed用户的LD\_PRELOAD环境变量中也添加了该链接库。

Task 8: Invoking External Programs Using system() versus execve()



新建一个名为MY的文件，并设置其权限为仅root用户可读、写、执行。

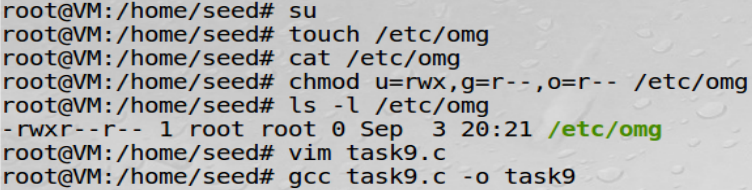






## Task 9: Capability Leaking

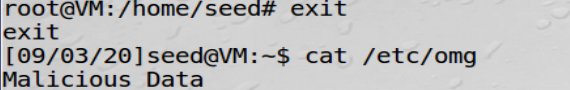
先切换为root用户，创建/etc/omg，设置其为只有root用户可以写入。保存代码为task9.c文件，将其编译为task9，并赋予SUID特殊权限。



加了一些头文件进去，不然有各种warning；把里面zzz改成omg。



exit成普通用户后，在seed下执行./task9，发现被写入信息



分析其原因是因为其在取消权限前并没有关闭文件，导致seed用户仍然可以进行root用户才可以执行的写入操作。