**Lab1实验报告**

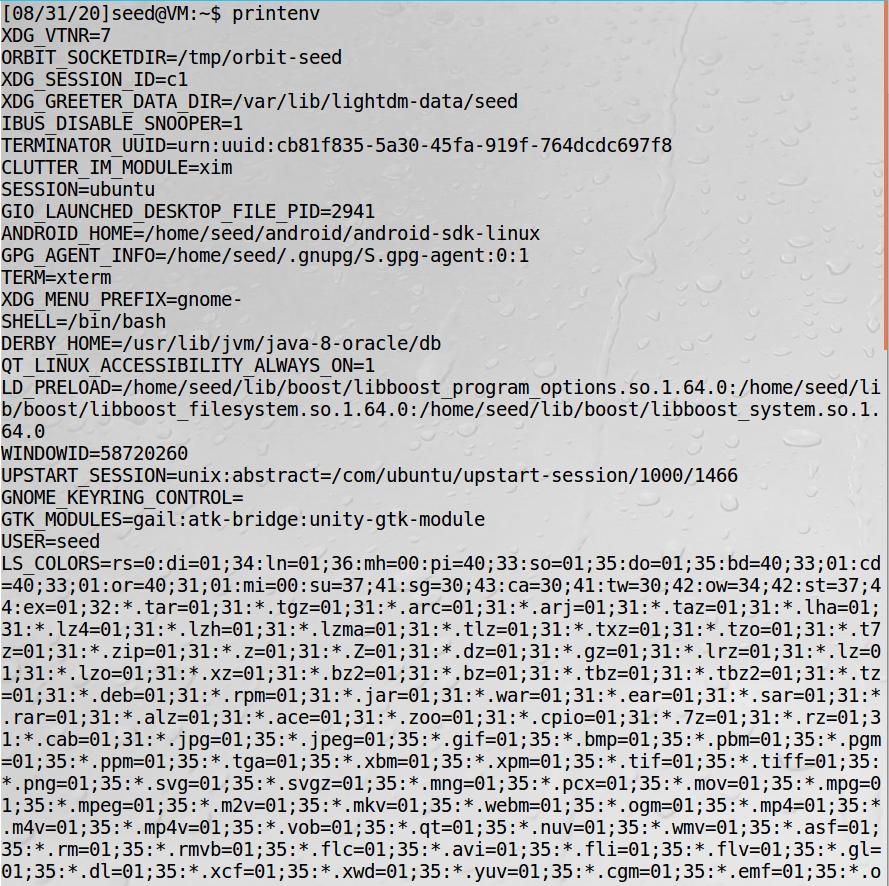
**Environment Variable and Set-UID Program**

57117138 吴伊杰

# Task 1: Manipulating Environment Variables

1、使用printenv或env打印出环境变量，得到如下结果：

（内容较多，只截取部分）



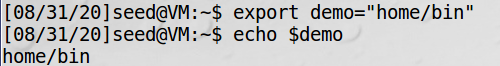
使用env命令有相同的效果，但是打印指定名称的环境变量时，两者用法不同。

2、使用printenv命令查看PATH环境变量：

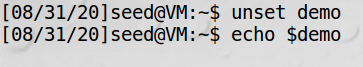


3、使用export和unset设置或删除环境变量

使用export设置环境变量，使用echo显示，$符号实际作用是将变量转换成字符，方便输出。



使用unset删除环境变量。

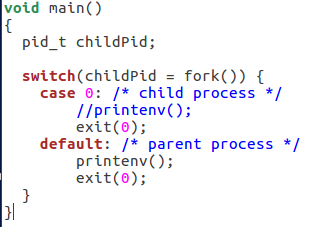


# Task 2: Passing Environment Variables from Parent Process to Child Process

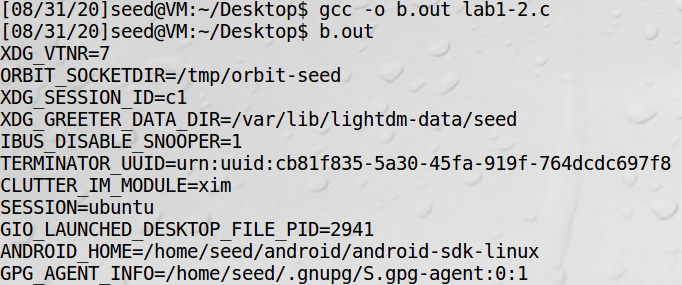
1、将手册所给的程序编译并执行，生成a.out文件，查看其内容，为各个环境变量的值



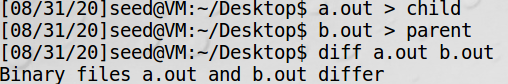
2、将程序中child process下的printenv()注释，将parent process下的printenv()取消注释



重新编译并执行，结果保存为b.out文件，查看其内容，与a.out很相似



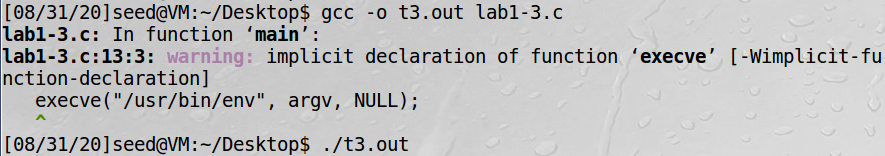
3、为了将两者进行比较，分别将a.out、b.out文件的结果分别保存为child和parent，再使用diff命令比较，两者的内容是完全相同的。



这说明子进程环境变量会继承父环境变量。

# Task 3: Environment Variables and execve()

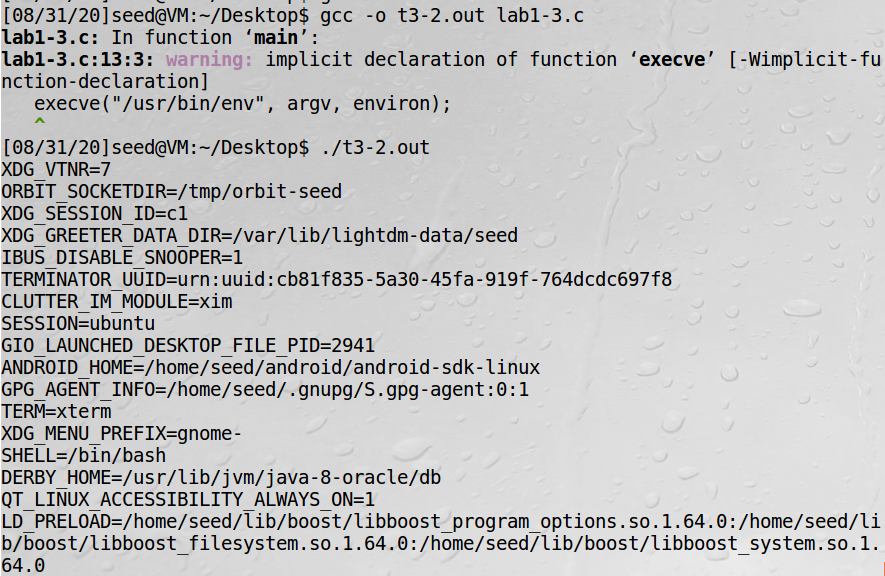
1、将所给程序编译并运行，该程序调用了/usr/bin/env，可以打印出当前进程的环境变量。运行后，发现执行结果为空。



2、把execve()的调用改为以下内容

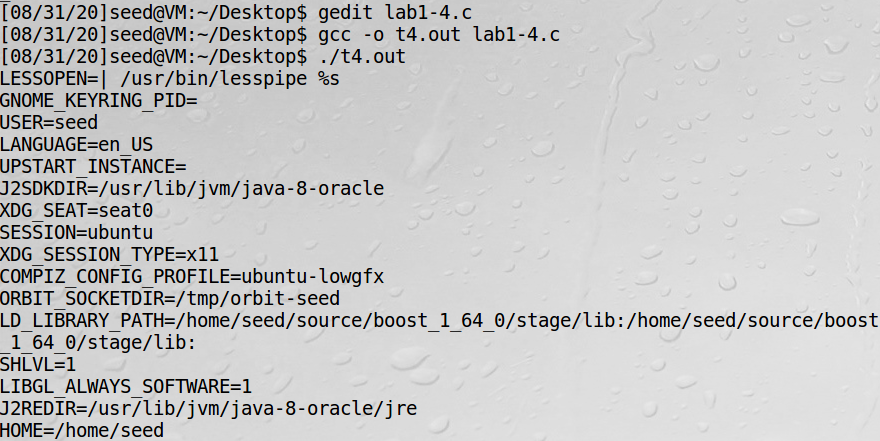


因为execve第三个参数为传递给执行文件的新环境变量数，第一步我们赋予新进程的环境变量为空，所以印出环境变量结果为空，现在修改参数后，可以看到结果不为空：



# Task 4: Environment Variables and system()

编译并运行所给程序，执行结果：



原理：

system函数的调用格式为int system（const char \* string），该函数会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，其执行过程可以归纳为三步：（1）fork一个子进程；（2）在子进程中调用exec函数去执行command；（3）在父进程中调用wait去等待子进程结束。

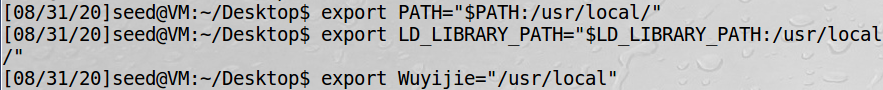
若fork失败，system（）函数返回-1。如果exec执行成功，即命令顺利执行完毕，则返回命令通过exit或return返回的值。如果exec执行失败，即命令没有顺利执行，比如信号被中断，或者命令根本不存在，system（）函数返回127，如果命令为NULL，则system（）函数返回值非0，一般为1。

# Task 5: Environment Variable and Set-UID Programs

1、根据所给程序，该程序可以在当前进程中打印出所有的环境变量。然后切换为root权限，将上述程序的所有权改为root，并使它成为一个Set-UID程序



2、切换为普通用户，使用export命令设置环境变量：PATH、LD\_LIBRARY\_PATH、ANY\_NAME

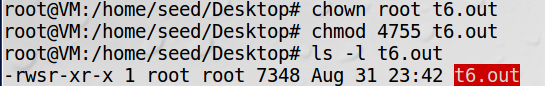


可以看到以上三个被定义的环境变量全部被包括在shell中

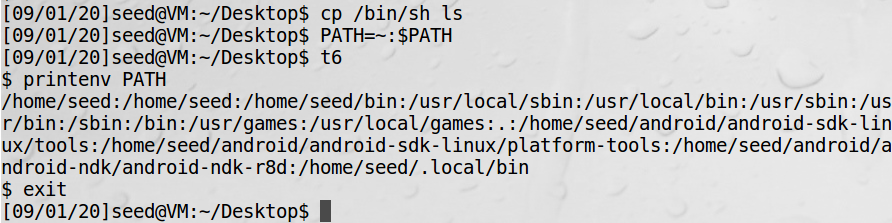


# Task 6: The PATH Environment Variable and Set-UID Programs

1、将所给程序编译，将执行文件设置为set-UID文件，使用ls -l命令可以查看设置成功



2、将/bin/sh复制到程序当前目录，命令为ls，设置环境变量，



PATH环境变量的命令找寻顺序是先找寻当前目录，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，直到我们主动退出这个程序，我们才会回到原来的权限。

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

新建一个动态链接库，手册所给程序mylib.c，内有sleep函数。按照所给命令编译mylib.c，然后将LD\_PRELOAD设置为我们编译的动态链接库。

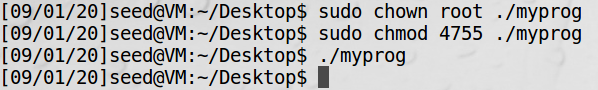
在链接库的相同目录下编译myprog程序，并在不同的情形下运行myprog程序：

（1）普通用户运行程序



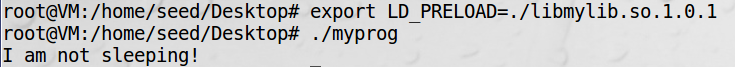
有输出I am not sleeping! 说明sleep函数为链接库的

（2）普通用户运行set-UID的root程序



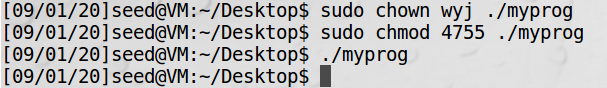
有sleep，无输出 说明sleep函数为系统库的

（3）root用户运行设置过LD\_PRELOAD的root程序



有输出

（4）用户1运行其他用户（我这里用户为wyj）的程序

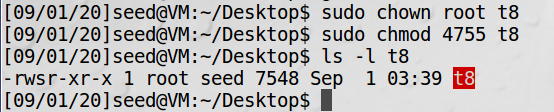


无输出

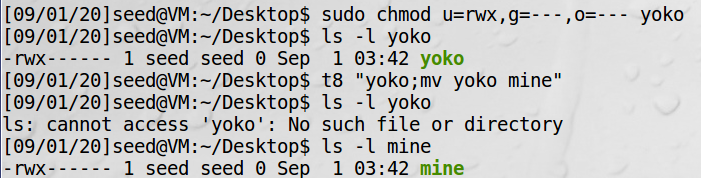
总结：只有用户自己创建的程序自己去运行，才会使用LD\_PRELOAD环境变量，否则忽略LD\_PRELOAD环境变量。

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

将所给程序编译，将其设置为set-UID程序



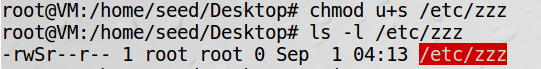
新建一个文件yoko，将其权限设置为仅限root用户可读写执行，然后执行上述程序，发现要求root权限的yoko文件已经被删除了，被更名为mine



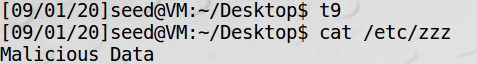
注释掉system(command)语句，并取消execve（）语句，则该程序将使用execve（）来调用该命令。然后重新按照上述做法，发现yoko文件不会被删除，原来的攻击方法已经失效了，因为execve()函数会把file; mv file file\_new 看成是一个文件名，系统会提示不存在这个文件，system()则不会。

# Task 9: Capability Leaking

先创建/etc/zzz文件，并将其设置权限



然后编译所给程序并执行之，这时再查看文件，会发现被写入了信息



原因：在取消权限前并没有关闭文件，导致seed用户仍可以进行root用户才可以执行的写入操作