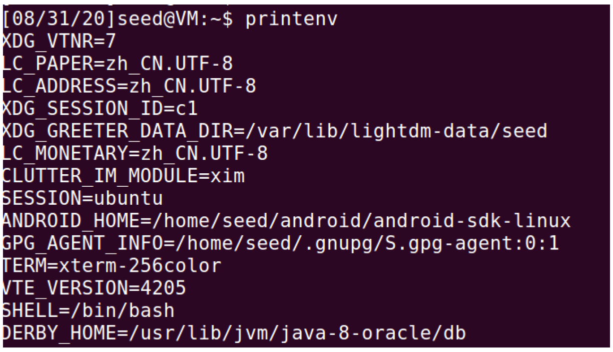
lab1-report

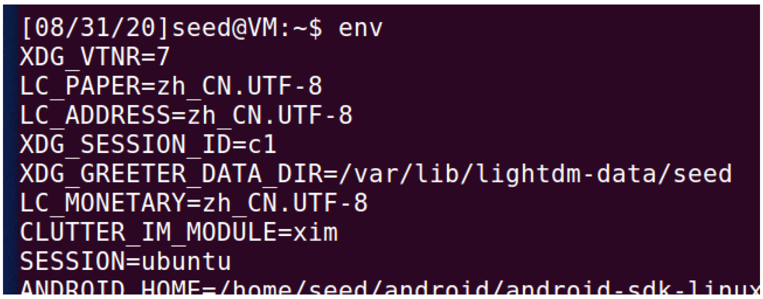
57117227 邵长捷

Task 1 Manipulating Environment Variables

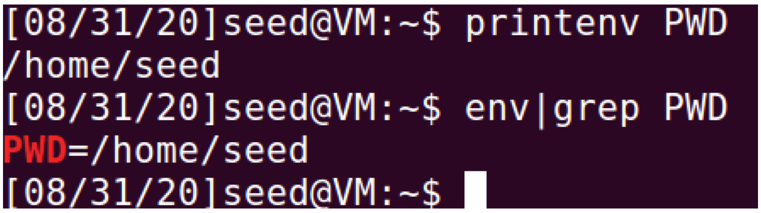
Step 1 Try env and printenv



```
[08/31/20]seed@VM:~$ printenv
XDG_VTNR=7
LC_PAPER=zh_CN.UTF-8
LC_ADDRESS=zh_CN.UTF-8
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
LC_MONETARY=zh_CN.UTF-8
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
```

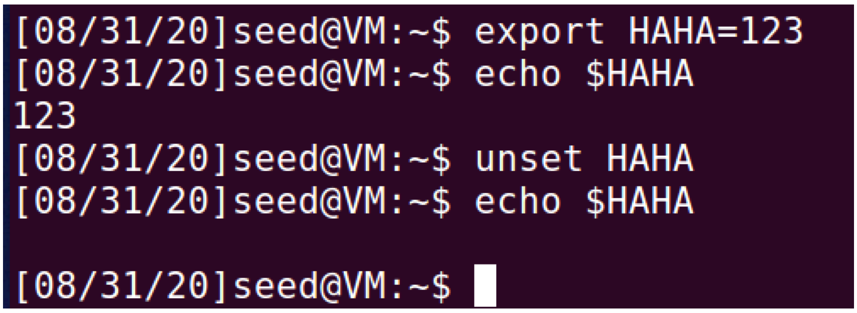


```
[08/31/20]seed@VM:~$ env
XDG_VTNR=7
LC_PAPER=zh_CN.UTF-8
LC_ADDRESS=zh_CN.UTF-8
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
LC_MONETARY=zh_CN.UTF-8
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
ANDROID_HOME=/home/seed/android/android-sdk-linux
```



```
[08/31/20]seed@VM:~$ printenv PWD
/home/seed
[08/31/20]seed@VM:~$ env|grep PWD
PWD=/home/seed
[08/31/20]seed@VM:~$ 
```

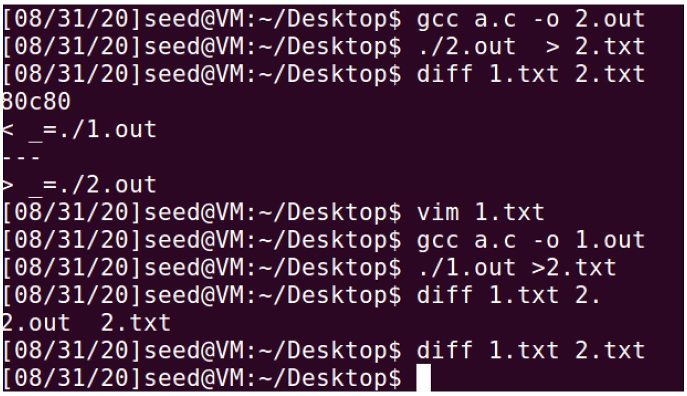
Step 2 Use export and unset



```
[08/31/20]seed@VM:~$ export HAHA=123
[08/31/20]seed@VM:~$ echo $HAHA
123
[08/31/20]seed@VM:~$ unset HAHA
[08/31/20]seed@VM:~$ echo $HAHA

[08/31/20]seed@VM:~$ 
```

Task 2 Passing Environment Variables from Parent Process to Child Process

Conclusion



```
[08/31/20]seed@VM:~/Desktop$ gcc a.c -o 2.out
[08/31/20]seed@VM:~/Desktop$ ./2.out  > 2.txt
[08/31/20]seed@VM:~/Desktop$ diff 1.txt 2.txt
80c80
< _=./1.out
---
> _=./2.out
[08/31/20]seed@VM:~/Desktop$ vim 1.txt
[08/31/20]seed@VM:~/Desktop$ gcc a.c -o 1.out
[08/31/20]seed@VM:~/Desktop$ ./1.out >2.txt
[08/31/20]seed@VM:~/Desktop$ diff 1.txt 2.
2.out  2.txt
[08/31/20]seed@VM:~/Desktop$ diff 1.txt 2.txt
[08/31/20]seed@VM:~/Desktop$
```

由图可见，fork 出的子进程的环境变量由父进程的环境变量复制而来。

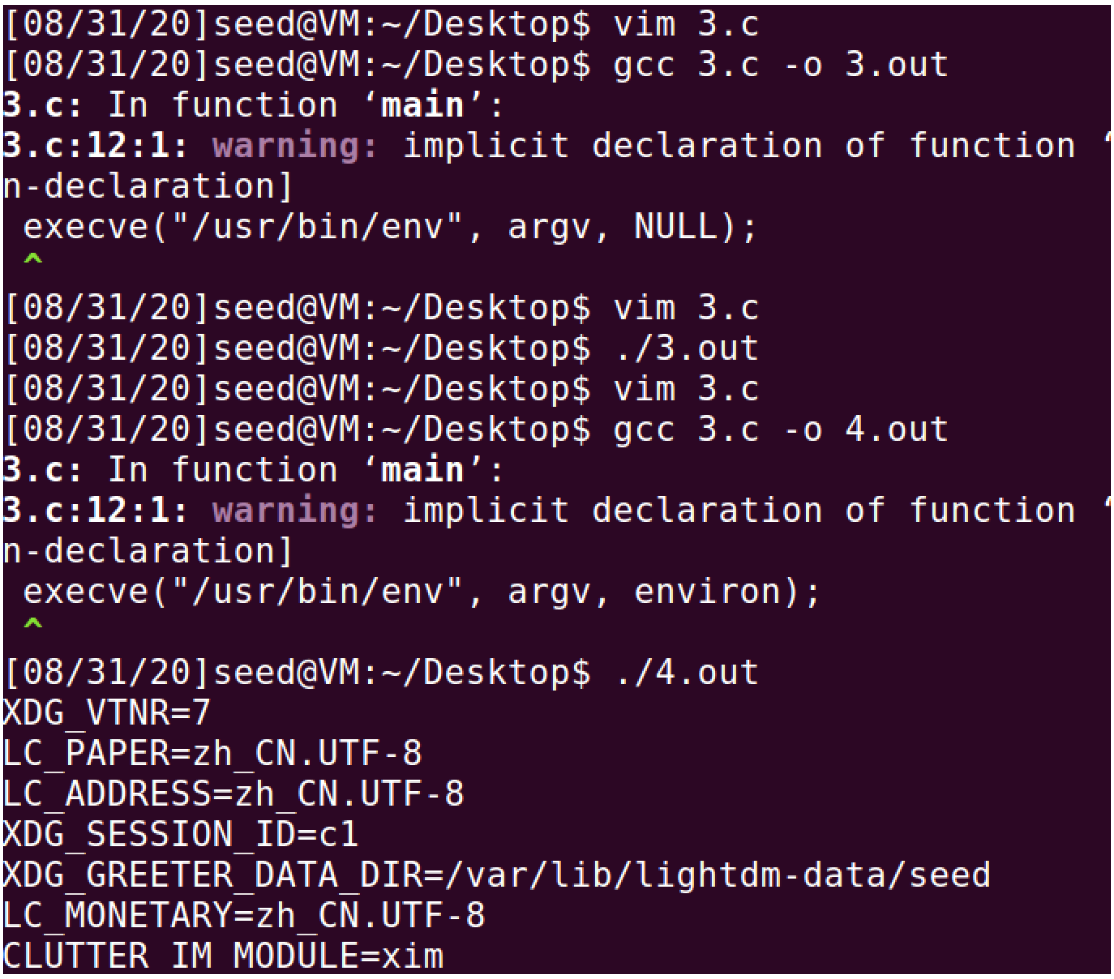
当两个编译生成的程序文件名分别为 1.out 和 2.out 时，在两个输出的 txt 文件的第 80 行会有差别，因为该行和程序文件名相关。

如果编译生成的程序名字一致时，二者的环境变量没有差别。

Task 3 Environment Variables and execve()

Step 1　execve()第三个参数为 NULL

Step 2　execve()第三个参数为 environ

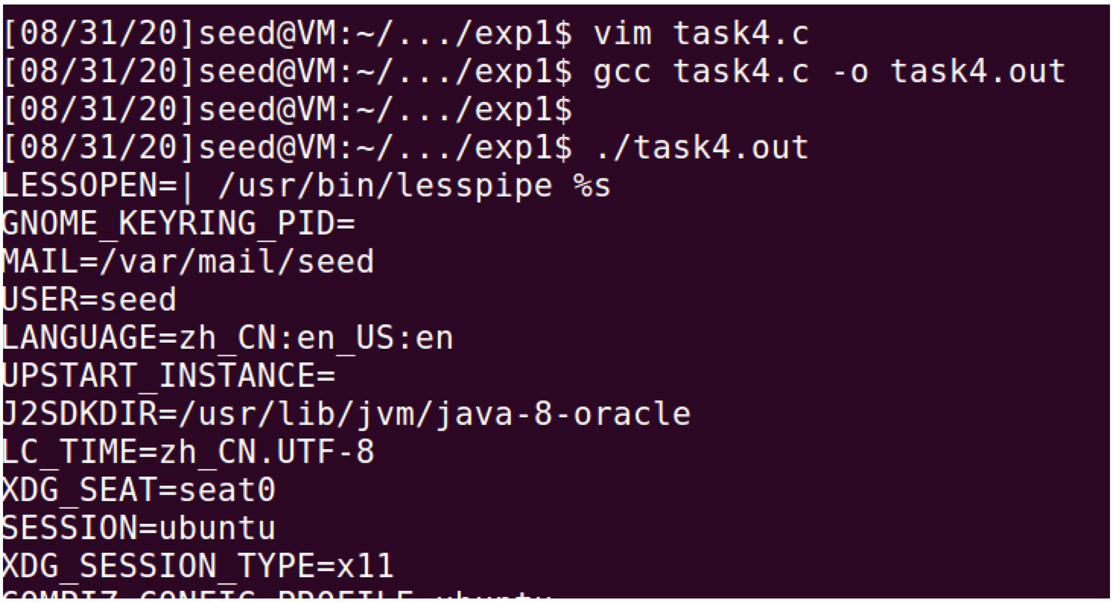


```
[08/31/20]seed@VM:~/Desktop$ vim 3.c
[08/31/20]seed@VM:~/Desktop$ gcc 3.c -o 3.out
3.c: In function ‘main’:
3.c:12:1: warning: implicit declaration of function ‘
n-declaration]
 execve("/usr/bin/env", argv, NULL);
 ^
[08/31/20]seed@VM:~/Desktop$ vim 3.c
[08/31/20]seed@VM:~/Desktop$ ./3.out
[08/31/20]seed@VM:~/Desktop$ vim 3.c
[08/31/20]seed@VM:~/Desktop$ gcc 3.c -o 4.out
3.c: In function ‘main’:
3.c:12:1: warning: implicit declaration of function ‘
n-declaration]
 execve("/usr/bin/env", argv, environ);
 ^
[08/31/20]seed@VM:~/Desktop$ ./4.out
XDG_VTNR=7
LC_PAPER=zh_CN.UTF-8
LC_ADDRESS=zh_CN.UTF-8
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
LC_MONETARY=zh_CN.UTF-8
CLUTTER_IM_MODULE=xim
```

Conclusion

在使用 execve()函数时，若不指定第三个参数，则运行的新程序不会继承该进程的环境变量，当传入全局变量 environ 时，运行的新程序则会继承原先的环境变量。

Task 4 Environment Variables and system()



```
[08/31/20]seed@VM:~/.../exp1$ vim task4.c
[08/31/20]seed@VM:~/.../exp1$ gcc task4.c -o task4.out
[08/31/20]seed@VM:~/.../exp1$
[08/31/20]seed@VM:~/.../exp1$ ./task4.out
LESSOPEN=| /usr/bin/lesspipe %s
GNOME_KEYRING_PID=
MAIL=/var/mail/seed
USER=seed
LANGUAGE=zh_CN:en_US:en
UPSTART_INSTANCE=
J2SDKDIR=/usr/lib/jvm/java-8-oracle
LC_TIME=zh_CN.UTF-8
XDG_SEAT=seat0
SESSION=ubuntu
XDG_SESSION_TYPE=x11
```

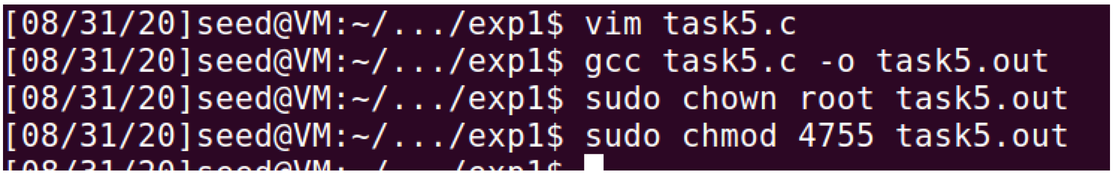
Conclusion

由图可见，编译并运行 task4.out 后，程序运行了 env 命令的功能，因而证明了 system()函数会新建 shell 并运行指令的功能。

Task 5 Environment Variable and Set-UID Programs

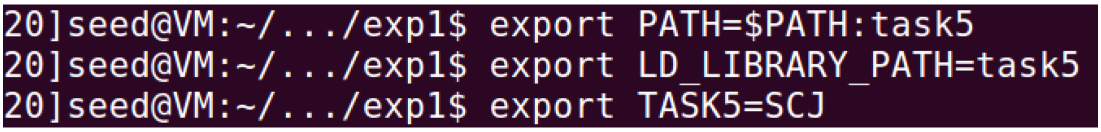
Step 1　编译 task5.c

Step 2　更改程序拥有者并指定为 Set-UID 程序



```
[08/31/20]seed@VM:~/.../exp1$ vim task5.c
[08/31/20]seed@VM:~/.../exp1$ gcc task5.c -o task5.out
[08/31/20]seed@VM:~/.../exp1$ sudo chown root task5.out
[08/31/20]seed@VM:~/.../exp1$ sudo chmod 4755 task5.out
```

Step 3　设置环境变量并检查

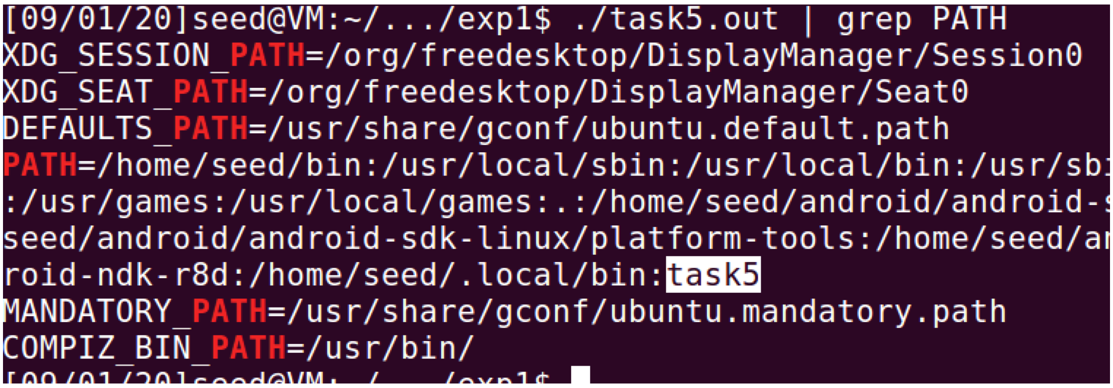


```
20]seed@VM:~/.../exp1$ export PATH=$PATH:task5
20]seed@VM:~/.../exp1$ export LD_LIBRARY_PATH=task5
20]seed@VM:~/.../exp1$ export TASK5=SCJ
```

根据要求设置三个环境变量。

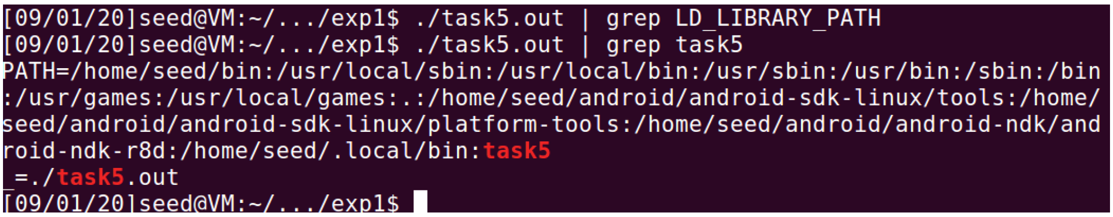
运行并查看相应的环境变量是否更新或生成。

PATH 成功更新：



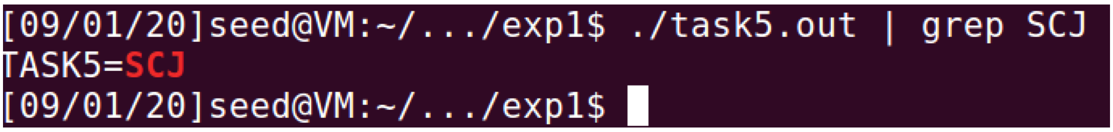
```
[09/01/20]seed@VM:~/.../exp1$ ./task5.out | grep PATH
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sb
:/usr/games:/usr/local/games:.:/home/seed/android/android-s
seed/android/android-sdk-linux/platform-tools:/home/seed/ar
roid-ndk-r8d:/home/seed/.local/bin:task5
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[09/01/20]seed@VM:~/.../exp1$
```

LD_LIBRARY_PATH 并没有更新：



```
[09/01/20]seed@VM:~/.../exp1$ ./task5.out | grep LD_LIBRARY_PATH
[09/01/20]seed@VM:~/.../exp1$ ./task5.out | grep task5
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
:/usr/games:/usr/local/games:.:/home/seed/android/android-sdk-linux/tools:/home/
seed/android/android-sdk-linux/platform-tools:/home/seed/android/android-ndk/and
roid-ndk-r8d:/home/seed/.local/bin:task5
_=./task5.out
[09/01/20]seed@VM:~/.../exp1$
```

自定义的变量 TASK5 成功生成：

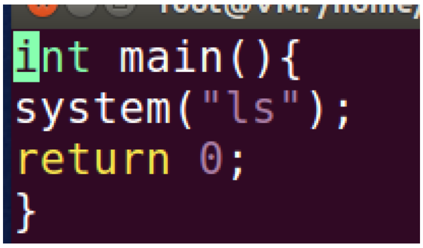


```
[09/01/20]seed@VM:~/.../exp1$ ./task5.out | grep SCJ
TASK5=SCJ
[09/01/20]seed@VM:~/.../exp1$
```

Conclusion

为了防止特殊环境变量对特权程序的动态链接产生任何影响，在运行特权程序 task5.out 的进程中并不包含 LD_LIBRARY_PATH。

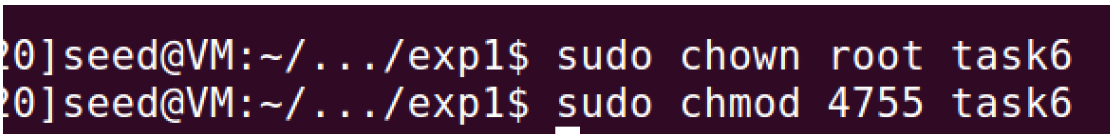
Task 6 The PATH Environment Variable and Set-UID Programs



```
int main(){
system("ls");
return 0;
}
```

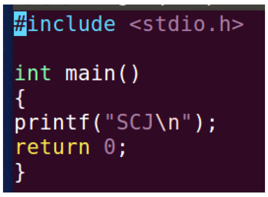
Step 1　编译上图程序为 task6

Step 2　更改程序拥有者并指定为 Set-UID 程序



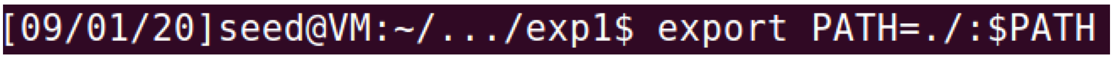
```
20]seed@VM:~/.../exp1$ sudo chown root task6
20]seed@VM:~/.../exp1$ sudo chmod 4755 task6
```

Step 3　在当前目录自定义 ls()函数并编译



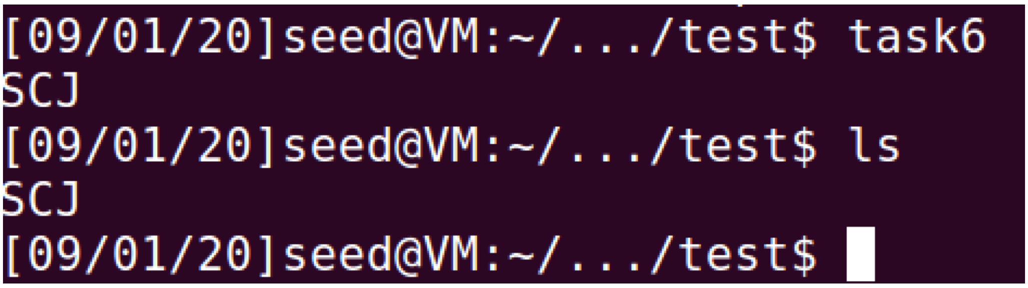
```
#include <stdio.h>

int main()
{
printf("SCJ\n");
return 0;
}
```

Step 4　更改 PATH 环境变量



```
[09/01/20]seed@VM:~/.../exp1$ export PATH=./:$PATH
```

Step 5　运行 task6



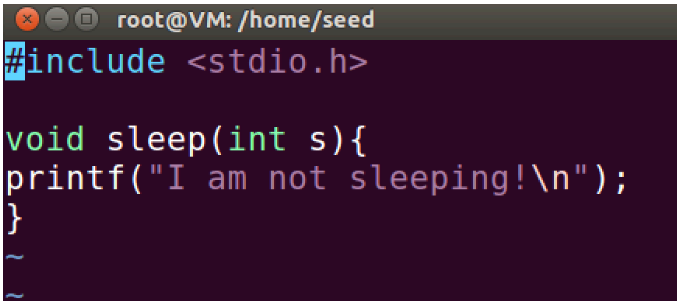
```
[09/01/20]seed@VM:~/.../test$ task6
SCJ
[09/01/20]seed@VM:~/.../test$ ls
SCJ
[09/01/20]seed@VM:~/.../test$ 
```

Conclusion

实验结果表明，在 PATH 中加入当前目录时，将优先搜索当前目录中的可执行文件 ls，从而实现对自定义 ls 的调用。

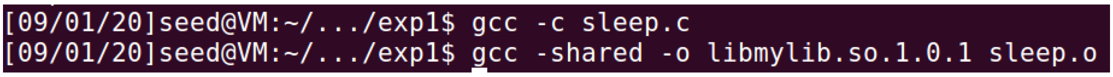
Task 7 The LD PRELOAD Environment Variable and Set-UID Programs

Step 1　重写 sleep()函数



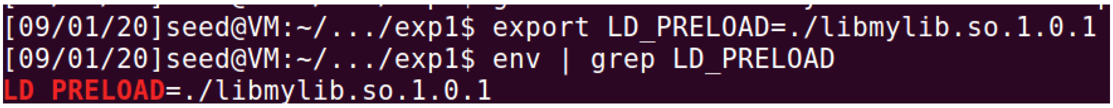
```
#include <stdio.h>

void sleep(int s){
printf("I am not sleeping!\n");
}
~
~
```

Step 2　编译并创建新的共享库



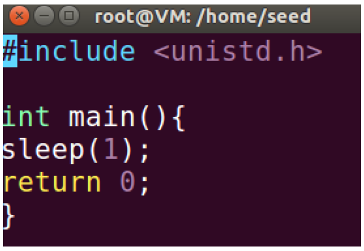
```
[09/01/20]seed@VM:~/.../exp1$ gcc -c sleep.c
[09/01/20]seed@VM:~/.../exp1$ gcc -shared -o libmylib.so.1.0.1 sleep.o
```

Step 3　设置 LD_PRELOAD 环境变量



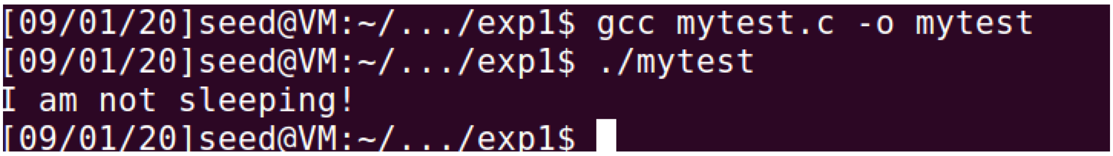
```
[09/01/20]seed@VM:~/.../exp1$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/01/20]seed@VM:~/.../exp1$ env | grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
```

Step 4　定义调用函数 mytest()



```
#include <unistd.h>

int main(){
sleep(1);
return 0;
}
```

Step 5 编译并运行 mytest



```
[09/01/20]seed@VM:~/.../exp1$ gcc mytest.c -o mytest
[09/01/20]seed@VM:~/.../exp1$ ./mytest
I am not sleeping!
[09/01/20]seed@VM:~/.../exp1$
```

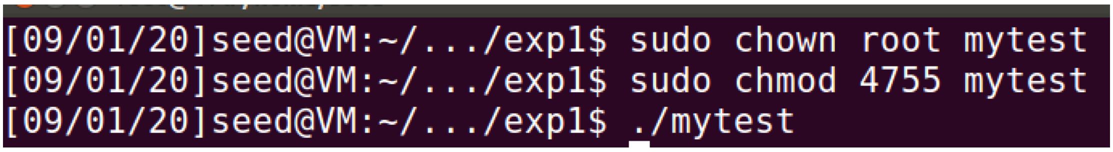
成功调用了自定义的 sleep()函数。

Step 6 Make mytest a regular program, and run it as a normal user.

上述 1-5 步骤即为普通用户运行非特权程序 mytest 的情况。

Step 7 Make mytest a Set-UID root program, and run it as a normal user.

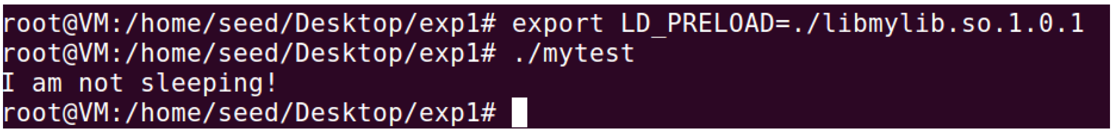
将 mytest 更改为特权程序后，运行时调用系统函数 sleep()。



```
[09/01/20]seed@VM:~/.../exp1$ sudo chown root mytest
[09/01/20]seed@VM:~/.../exp1$ sudo chmod 4755 mytest
[09/01/20]seed@VM:~/.../exp1$ ./mytest
```

Step 8 Make mytest a Set-UID root program, export the LD_PRELOAD environment variable again in the root account and run it.

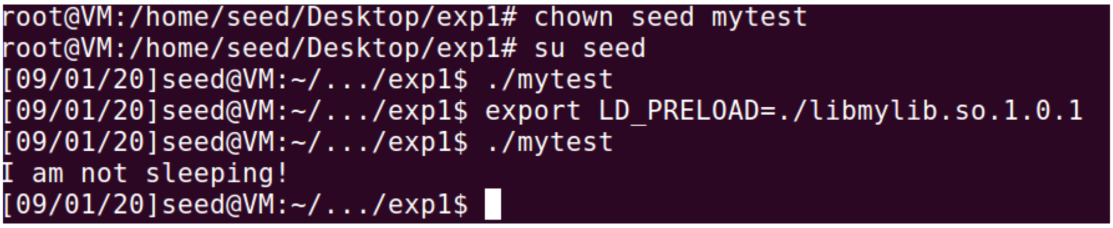
此时可以调用自定义的 sleep()函数。



```
root@VM:/home/seed/Desktop/exp1# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Desktop/exp1# ./mytest
I am not sleeping!
root@VM:/home/seed/Desktop/exp1#
```

Step 9 Make mytest a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user's account (not-root user) and run

此时可以调用自定义的 sleep()函数。

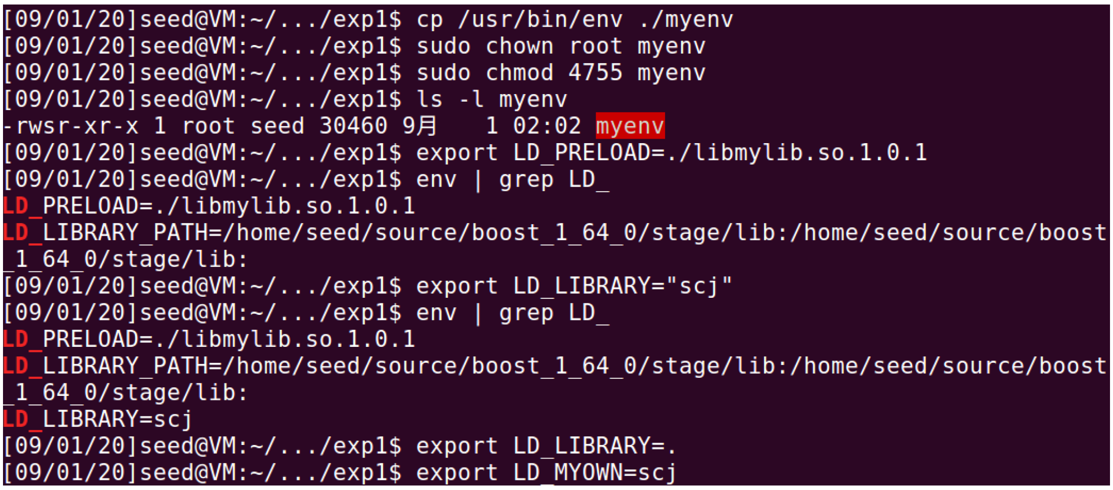


```
root@VM:/home/seed/Desktop/exp1# chown seed mytest
root@VM:/home/seed/Desktop/exp1# su seed
[09/01/20]seed@VM:~/.../exp1$ ./mytest
[09/01/20]seed@VM:~/.../exp1$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/01/20]seed@VM:~/.../exp1$ ./mytest
I am not sleeping!
[09/01/20]seed@VM:~/.../exp1$
```

Conclusion

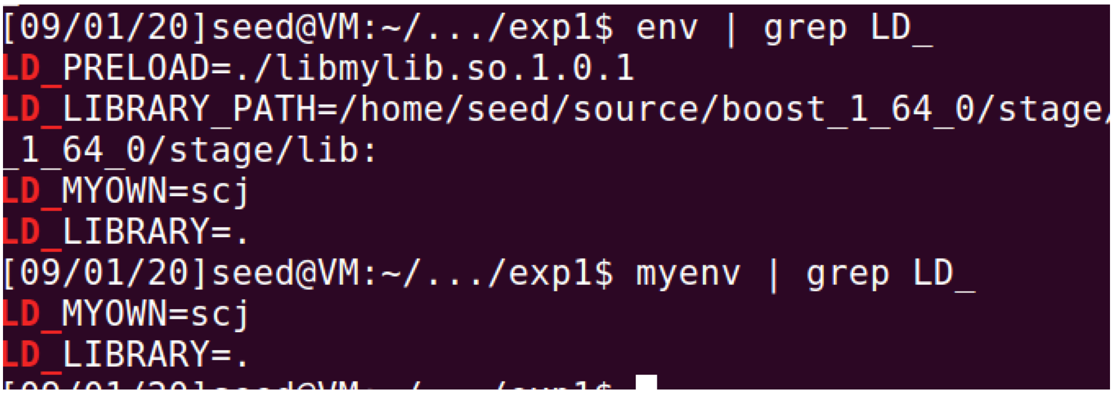
实验结果表明，当进程的真实用户 ID 和有效用户 ID 不一样时，进程将忽略 LD_PRELOAD 环境变量。当设置 Set-UID 为后，有效用户等于文件的所有者，因而 Step8 和 Step9 可以成功运行自定义 sleep()函数。

验证实验



```
[09/01/20]seed@VM:~/.../exp1$ cp /usr/bin/env ./myenv
[09/01/20]seed@VM:~/.../exp1$ sudo chown root myenv
[09/01/20]seed@VM:~/.../exp1$ sudo chmod 4755 myenv
[09/01/20]seed@VM:~/.../exp1$ ls -l myenv
-rwsr-xr-x 1 root seed 30460 9月   1 02:02 myenv
[09/01/20]seed@VM:~/.../exp1$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/01/20]seed@VM:~/.../exp1$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
[09/01/20]seed@VM:~/.../exp1$ export LD_LIBRARY="scj"
[09/01/20]seed@VM:~/.../exp1$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
LD_LIBRARY=scj
[09/01/20]seed@VM:~/.../exp1$ export LD_LIBRARY=.
[09/01/20]seed@VM:~/.../exp1$ export LD_MYOWN=scj
```

复制一份 env 程序并设置为特权程序，更改 LD_PRELOAD 和 LD_LIBRARY_PATH 环境变量，新增自定义的环境变量 LD_MYOWN 和 LD_LIBRARY。



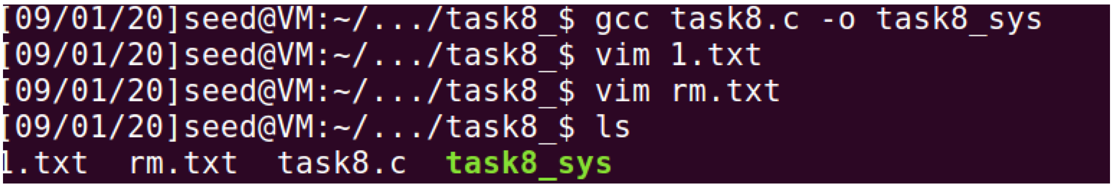
```
[09/01/20]seed@VM:~/.../exp1$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/
_1_64_0/stage/lib:
LD_MYOWN=scj
LD_LIBRARY=.
[09/01/20]seed@VM:~/.../exp1$ myenv | grep LD_
LD_MYOWN=scj
LD_LIBRARY=.
[09/01/20]seed@VM:~/.../exp1$
```

可见，非特权程序 env 不会屏蔽 LD_PRELOAD 和 LD_LIBRARY_PATH 环境变量，而特权程序则会屏蔽二者，只显示了自定义的环境变量。

上述实验验证了 Set_UID 进程会屏蔽 LD_PRELOAD 和 LD_LIBRARY_PATH 环境变量，以防其对特权程序的动态链接产生任何影响。

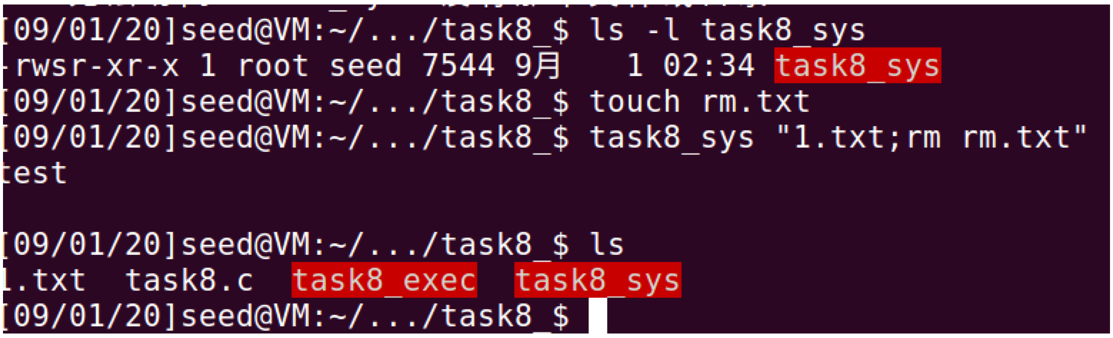
Task 8 Invoking External Programs Using system() versus

Step 1 编译并设置 task8_sys 为特权程序



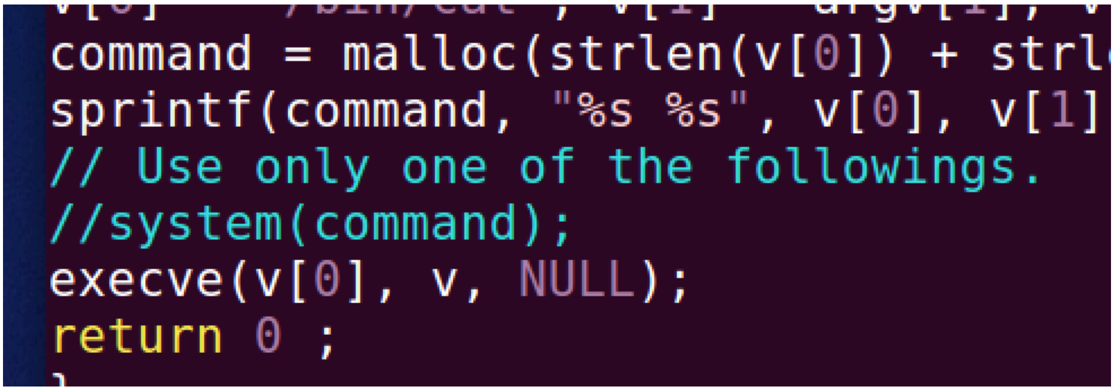
```
[09/01/20]seed@VM:~/.../task8_$ gcc task8.c -o task8_sys
[09/01/20]seed@VM:~/.../task8_$ vim 1.txt
[09/01/20]seed@VM:~/.../task8_$ vim rm.txt
[09/01/20]seed@VM:~/.../task8_$ ls
1.txt  rm.txt  task8.c  task8_sys
```

Step 2 利用 system()进行多命令执行

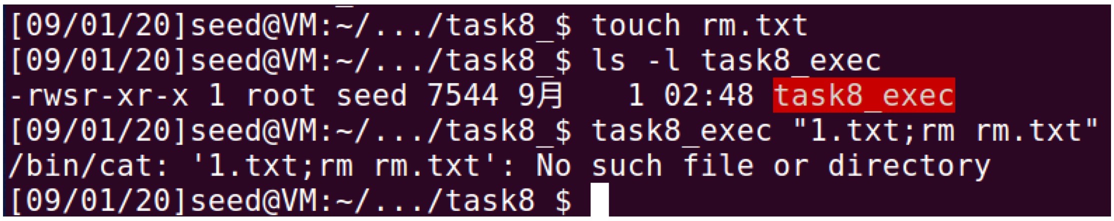


```
[09/01/20]seed@VM:~/.../task8_$ ls -l task8_sys
-rwsr-xr-x 1 root seed 7544 9月    1 02:34 task8_sys
[09/01/20]seed@VM:~/.../task8_$ touch rm.txt
[09/01/20]seed@VM:~/.../task8_$ task8_sys "1.txt;rm rm.txt"
test

[09/01/20]seed@VM:~/.../task8_$ ls
1.txt  task8.c  task8_exec  task8_sys
[09/01/20]seed@VM:~/.../task8_$ 
```

Step 3 换用 execve()函数



```
command = malloc(strlen(v[0]) + strl
sprintf(command, "%s %s", v[0], v[1]
// Use only one of the followings.
//system(command);
execve(v[0], v, NULL);
return 0 ;
```



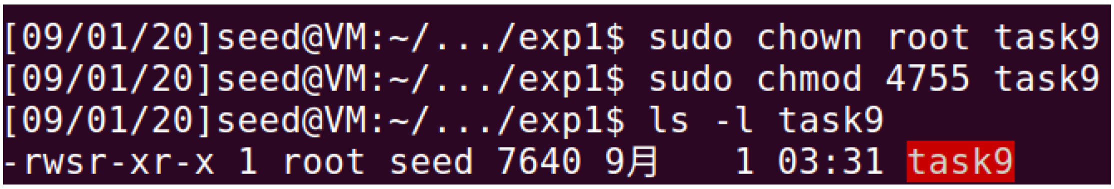
```
[09/01/20]seed@VM:~/.../task8_$ touch rm.txt
[09/01/20]seed@VM:~/.../task8_$ ls -l task8_exec
-rwsr-xr-x 1 root seed 7544 9月    1 02:48 task8_exec
[09/01/20]seed@VM:~/.../task8_$ task8_exec "1.txt;rm rm.txt"
/bin/cat: '1.txt;rm rm.txt': No such file or directory
[09/01/20]seed@VM:~/.../task8_$ 
```

Conclusion

由于 system()函数调用 shell，传入的参数可以用分号分割达到执行多条指令的效果；execve()函数会将参数作为一个整体运行，即将其作为一个文件名，找不到文件从而报错，因而不会产生调用 system()函数时的效果。

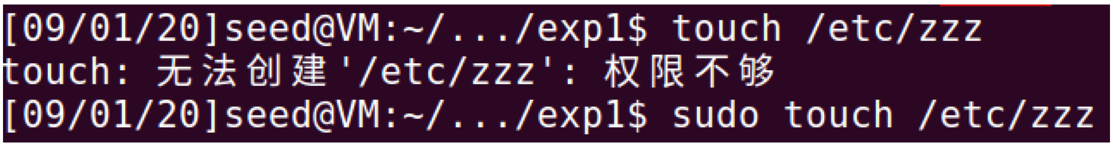
Task 9 Capability Leaking

Step 1 编译 task9 并设置为特权程序

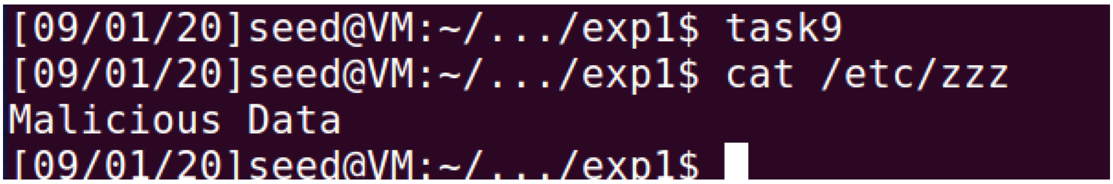


```
[09/01/20]seed@VM:~/.../exp1$ sudo chown root task9
[09/01/20]seed@VM:~/.../exp1$ sudo chmod 4755 task9
[09/01/20]seed@VM:~/.../exp1$ ls -l task9
-rwsr-xr-x 1 root seed 7640 9月    1 03:31 task9
```

Step 2 创建/etc/zzz 文件并运行 task9



```
[09/01/20]seed@VM:~/.../exp1$ touch /etc/zzz
touch: 无法创建'/etc/zzz': 权限不够
[09/01/20]seed@VM:~/.../exp1$ sudo touch /etc/zzz
```



```
[09/01/20]seed@VM:~/.../exp1$ task9
[09/01/20]seed@VM:~/.../exp1$ cat /etc/zzz
Malicious Data
[09/01/20]seed@VM:~/.../exp1$ 
```

Conclusion

fork 出的子进程仍然具有/etc/zzz 读写权限的文件描述符 fd，因而可以实现在/etc/zzz 文件中写入数据，是一种典型的权限泄露，应当在 fork 之前关闭文件描述符 fd，防止权限的恶意利用。