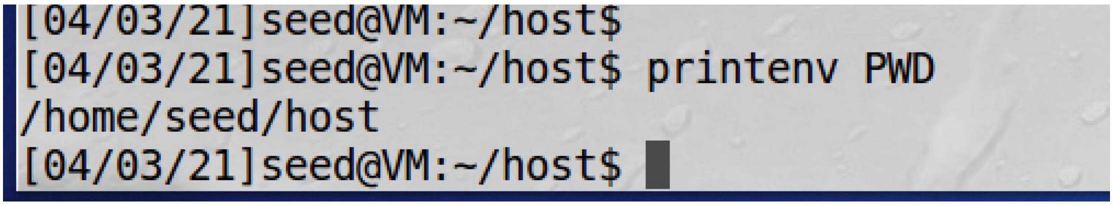
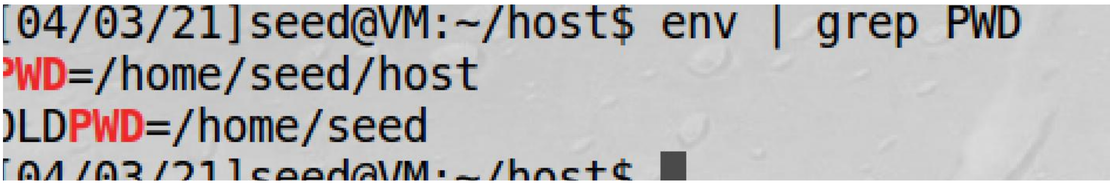
信安实验 1

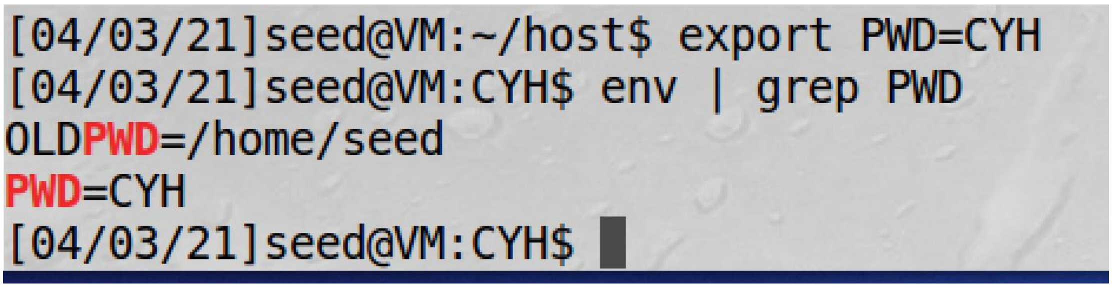
1.Manipulating Environment Variables



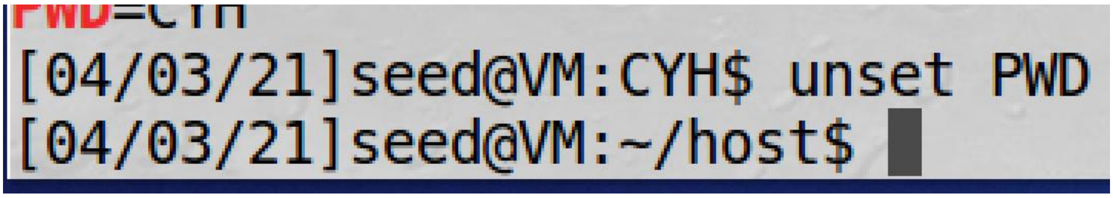
```
[04/03/21]seed@VM:~/host$
[04/03/21]seed@VM:~/host$ printenv PWD
/home/seed/host
[04/03/21]seed@VM:~/host$ █
```



```
[04/03/21]seed@VM:~/host$ env | grep PWD
PWD=/home/seed/host
OLDPWD=/home/seed
[04/03/21]seed@VM:~/host$ █
```



```
[04/03/21]seed@VM:~/host$ export PWD=CYH
[04/03/21]seed@VM:CYH$ env | grep PWD
OLDPWD=/home/seed
PWD=CYH
[04/03/21]seed@VM:CYH$ █
```

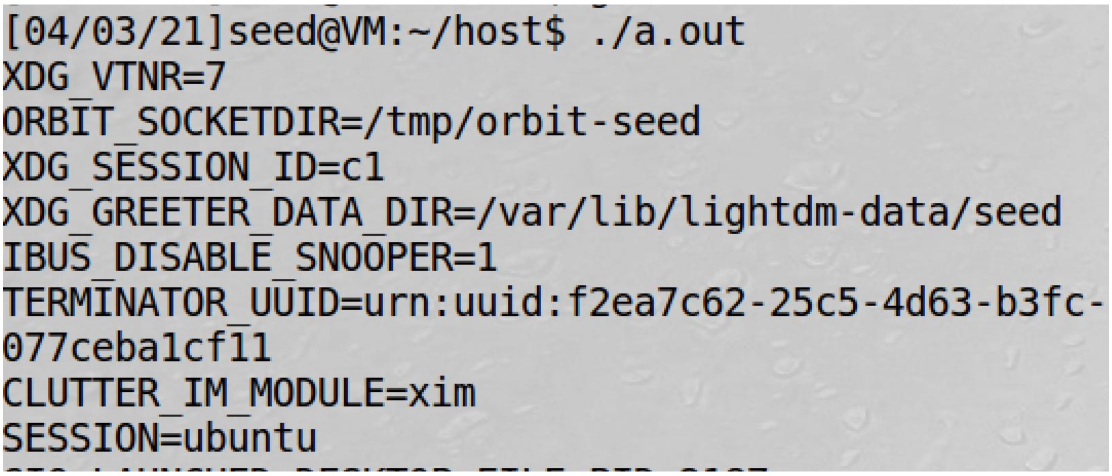


```
PWD=CYH
[04/03/21]seed@VM:CYH$ unset PWD
[04/03/21]seed@VM:~/host$ █
```

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

Step 1.

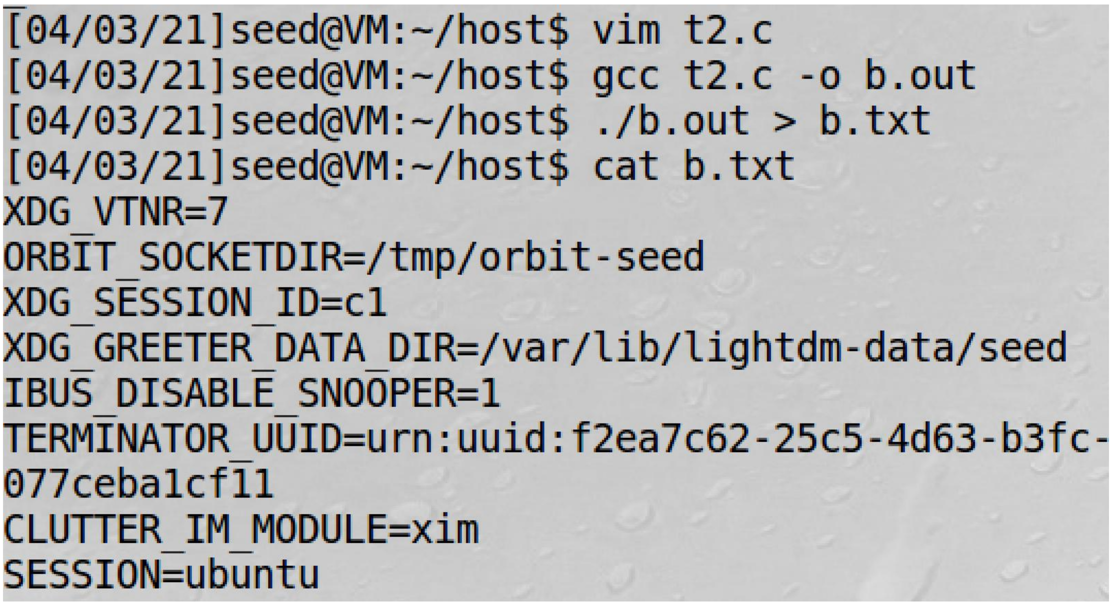
打印出了所有的环境变量



```
[04/03/21]seed@VM:~/host$ ./a.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
TERMINATOR_UUID=urn:uuid:f2ea7c62-25c5-4d63-b3fc-
077ceba1cf11
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
```

Step2

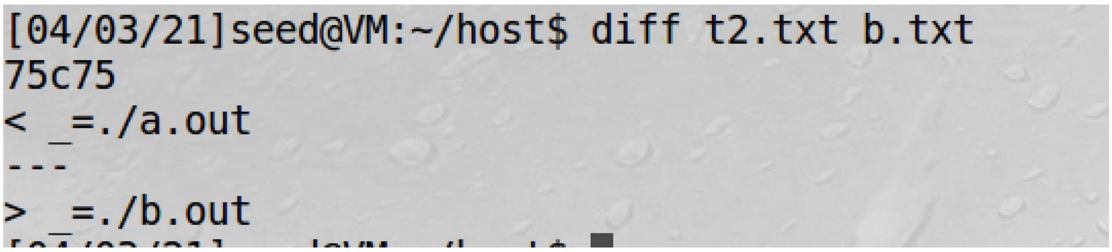
打印出了所有的环境变量



```
[04/03/21]seed@VM:~/host$ vim t2.c
[04/03/21]seed@VM:~/host$ gcc t2.c -o b.out
[04/03/21]seed@VM:~/host$ ./b.out > b.txt
[04/03/21]seed@VM:~/host$ cat b.txt
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
TERMINATOR_UUID=urn:uuid:f2ea7c62-25c5-4d63-b3fc-
077ceba1cf11
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
```

Step3

两个文件只有在最后一行不同，即当前执行的文件，如果两个可执行文件命名相同，则两个输出文件完全一致，所以子进程继承了父进程的环境变量。



```
[04/03/21]seed@VM:~/host$ diff t2.txt b.txt
75c75
< _=./a.out
---
> _=./b.out
```

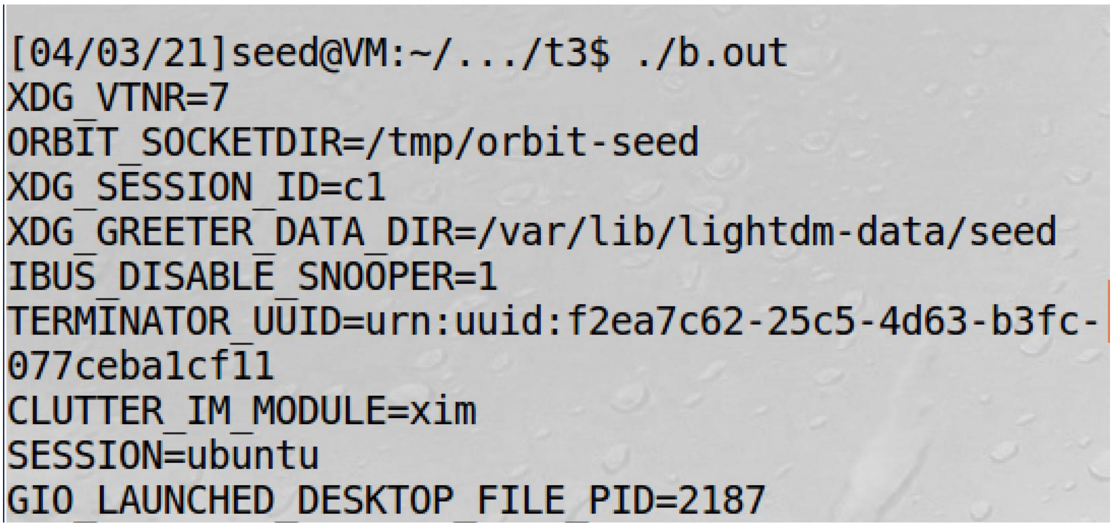
Task 3: Environment Variables and execve()

Step 1.

什么也没打印

Step 2.

打印出了所有的环境变量



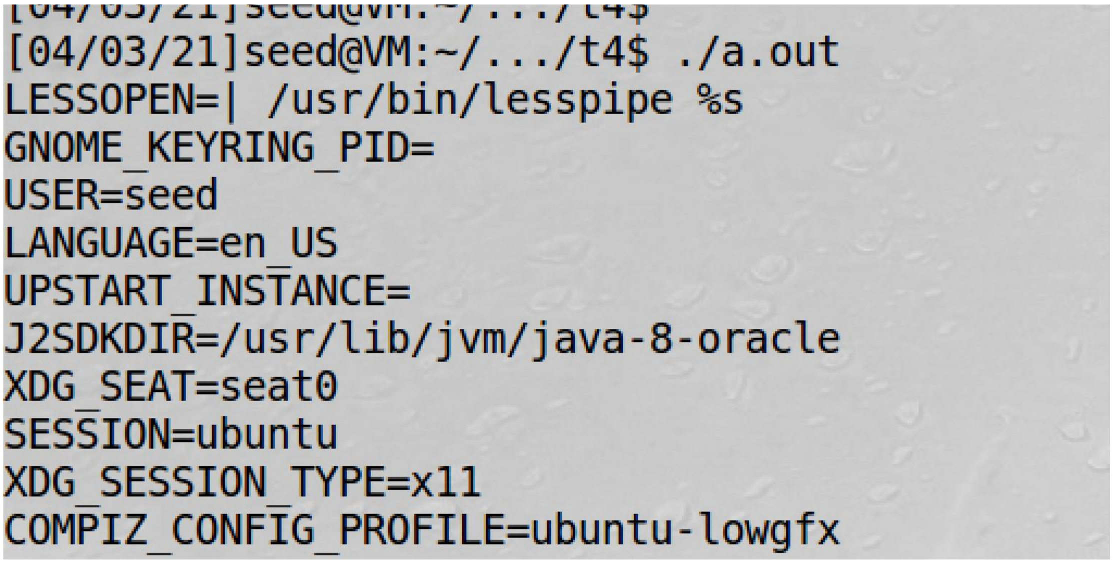
```
[04/03/21]seed@VM:~/.../t3$ ./b.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
TERMINATOR_UUID=urn:uuid:f2ea7c62-25c5-4d63-b3fc-
077ceba1cf11
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=2187
```

Step 3.

execve()不会获取调用进程的环境变量，而是需要通过传参的方式，才能把环境变量传给被调用的进程。

Task 4: Environment Variables and system()

环境变量被传递给了被调用进程

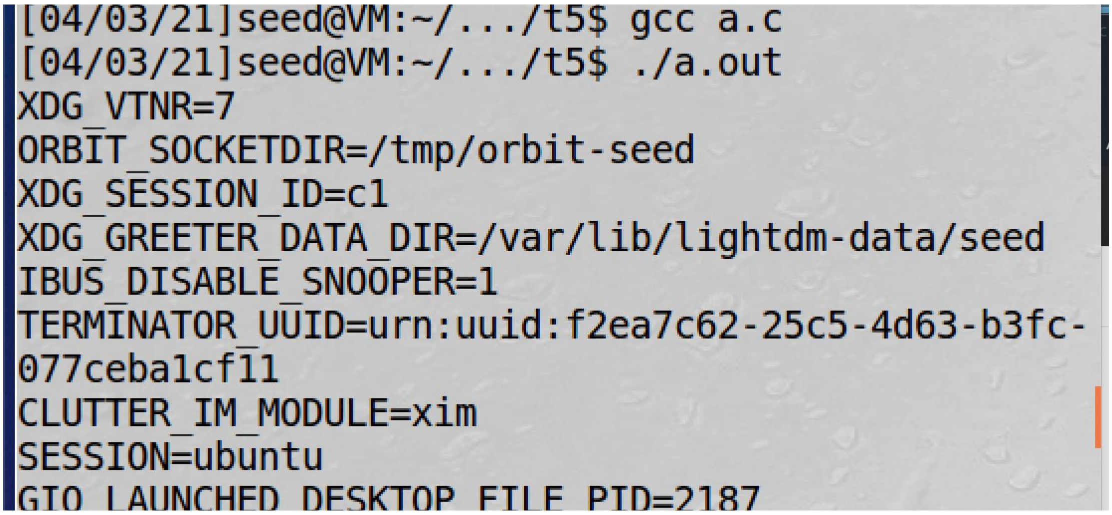


```
[04/03/21]seed@VM:~/.../t4$
[04/03/21]seed@VM:~/.../t4$ ./a.out
LESSOPEN=| /usr/bin/lesspipe %s
GNOME_KEYRING_PID=
USER=seed
LANGUAGE=en_US
UPSTART_INSTANCE=
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_SEAT=seat0
SESSION=ubuntu
XDG_SESSION_TYPE=x11
COMPIZ_CONFIG_PROFILE=ubuntu-lowgfx
```

Task 5: Environment Variable and Set-UID Programs

Step 1.

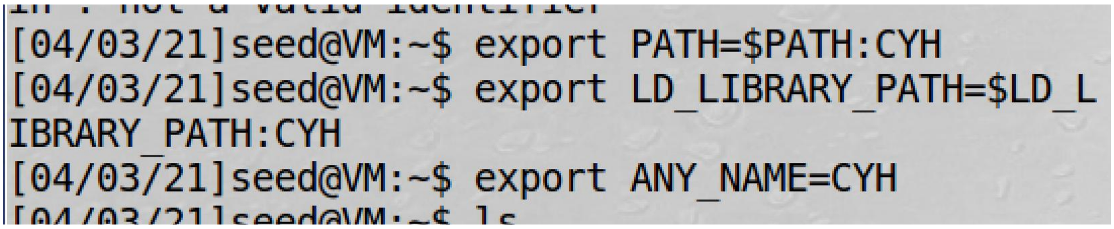
打印出了环境变量



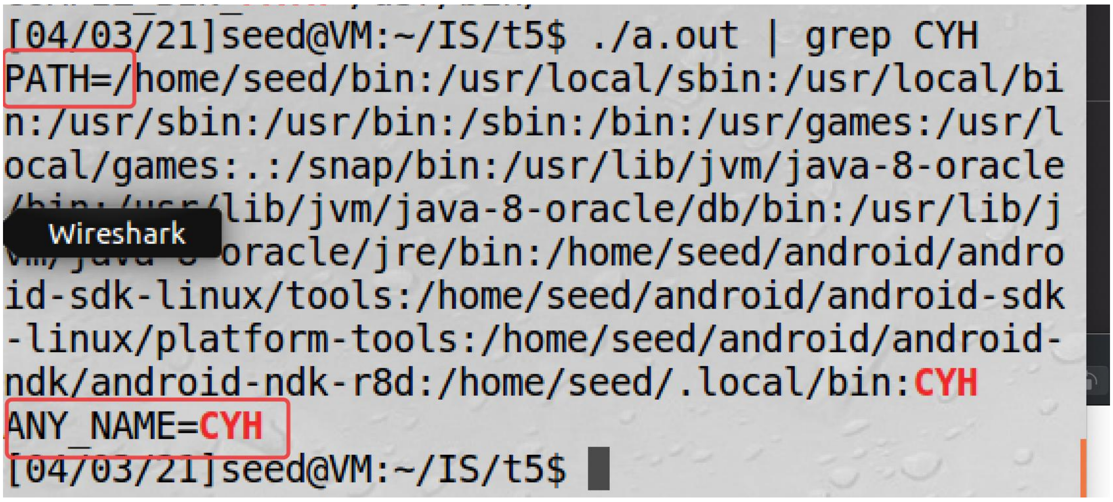
```
[04/03/21]seed@VM:~/.../t5$ gcc a.c
[04/03/21]seed@VM:~/.../t5$ ./a.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
TERMINATOR_UUID=urn:uuid:f2ea7c62-25c5-4d63-b3fc-
077ceba1cf11
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=2187
```

Step2,Step3

没有打印出 LD_LIBRARY_PATH 中的值，这是因为动态链接器需要从 LD_LIBRARY_PATH 查找程序所用的库，并且动态链接器实施了一些防御策略，当进程的真实用户 ID 和有效用户 ID 不一样时，即当前程序是一个 Set-Uid 程序，进程将忽略 LD_LIBRARY_PATH 环境变量的值。PATH 和自定义环境变量被正常传递。



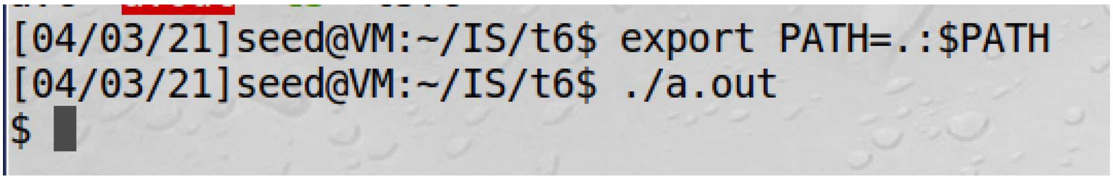
```
in' : not a valid identifier
[04/03/21]seed@VM:~$ export PATH=$PATH:CYH
[04/03/21]seed@VM:~$ export LD_LIBRARY_PATH=$LD_L
IBRARY_PATH:CYH
[04/03/21]seed@VM:~$ export ANY_NAME=CYH
[04/03/21]seed@VM:~$ ls
```



```
[04/03/21]seed@VM:~/IS/t5$ ./a.out | grep CYH
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bi
n:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
ocal/games:.:/snap/bin:/usr/lib/jvm/java-8-oracle
/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/j
vm/java-8-oracle/jre/bin:/home/seed/android/andro
id-sdk-linux/tools:/home/seed/android/android-sdk
-linux/platform-tools:/home/seed/android/android-
ndk/android-ndk-r8d:/home/seed/.local/bin:CYH
ANY_NAME=CYH
[04/03/21]seed@VM:~/IS/t5$
```

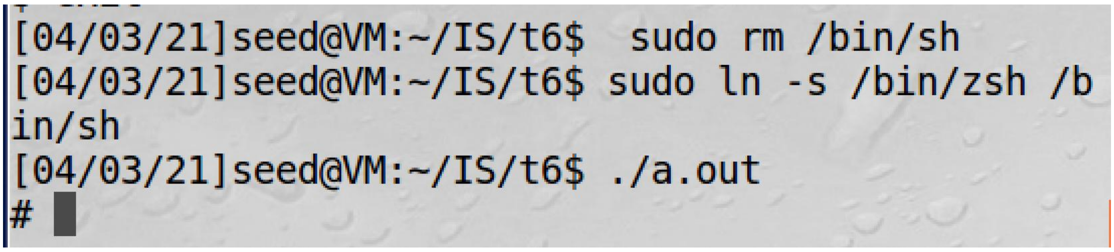
Task 6: The PATH Environment Variable and Set-UID Programs

如图，程序运行了自己生成的 ls 程序，但是在存在保护机制的情况下，没有获得 root 权限



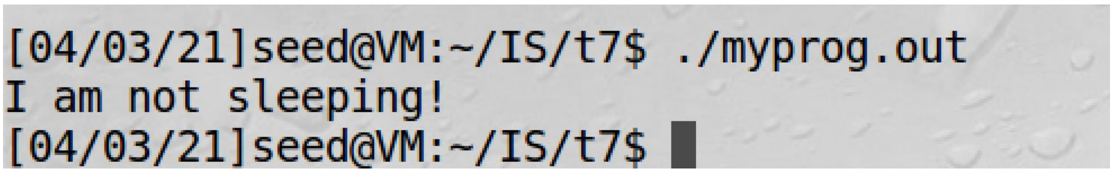
```
[04/03/21]seed@VM:~/IS/t6$ export PATH=.:$PATH
[04/03/21]seed@VM:~/IS/t6$ ./a.out
$ 
```

在去掉保护机制后，通过 system(“ls“);调用了自己生成的 ls 程序，并获得了 root 权限。



```
[04/03/21]seed@VM:~/IS/t6$  sudo rm /bin/sh
[04/03/21]seed@VM:~/IS/t6$ sudo ln -s /bin/zsh /b
in/sh
[04/03/21]seed@VM:~/IS/t6$ ./a.out
# 
```

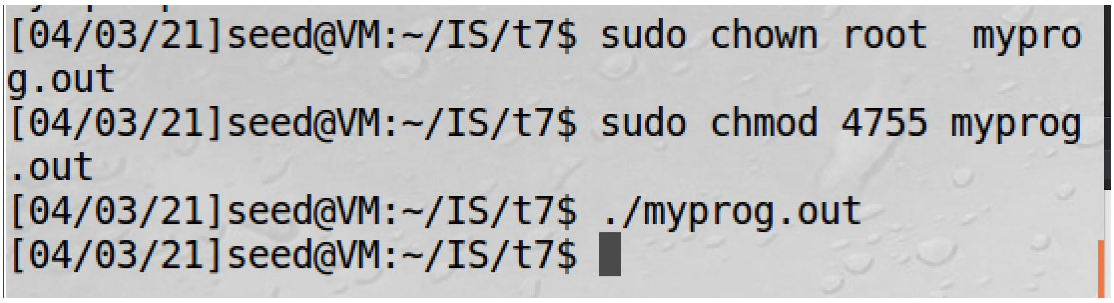
Task 7: The LD PRELOAD Environment Variable and Set-UID Programs
Step 1,Step 2.
Make myprog a regular program, and run it as a normal user.
程序调用了自己写的 sleep 方法，这意味着动态链接程序会从 LD PRELOAD 中去查找所需程序库的位置。



```
[04/03/21]seed@VM:~/IS/t7$ ./myprog.out
I am not sleeping!
[04/03/21]seed@VM:~/IS/t7$ 
```

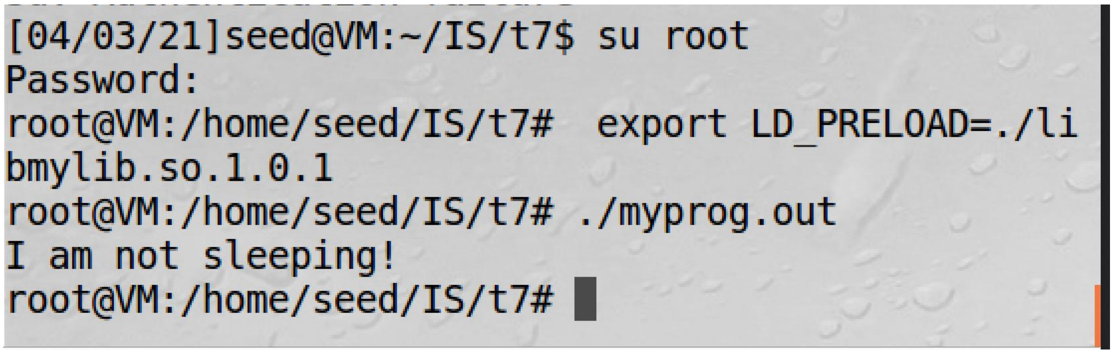
- Make myprog a Set-UID root program, and run it as a normal user.

如图，程序调用了 libc 中的 sleep()函数，之所以与之前的输出不一致，是因为动态链接器的实施了防御机制，当进程的真实用户 ID 和有效用户 ID 不一样，或者真实组 ID 和有效组 ID 不一样时，进程将忽略 LD_PRELOAD 环境变量。



```
[04/03/21]seed@VM:~/IS/t7$ sudo chown root  mypro
g.out
[04/03/21]seed@VM:~/IS/t7$ sudo chmod 4755 myprog
.out
[04/03/21]seed@VM:~/IS/t7$ ./myprog.out
[04/03/21]seed@VM:~/IS/t7$ 
```

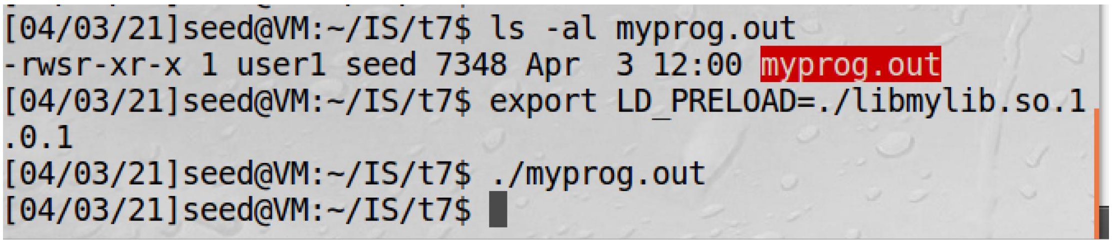
Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.
在切换到 root 后，发现调用了自定义的 sleep 方法，这是因为在 root 条件下，真实用户 ID 和有效 ID 一致，不会触发防御策略。



```
[04/03/21]seed@VM:~/IS/t7$ su root
Password:
root@VM:/home/seed/IS/t7#  export LD_PRELOAD=./li
bmylib.so.1.0.1
root@VM:/home/seed/IS/t7# ./myprog.out
I am not sleeping!
root@VM:/home/seed/IS/t7# 
```

Make myprog 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 it.

如图，程序调用了 libc 中的 sleep()函数,原因依然是当进程的真实用户 ID 和有效用户 ID 不一样，导致动态链接器的实施了防御机制，进程将忽略 LD_PRELOAD 环境变量。

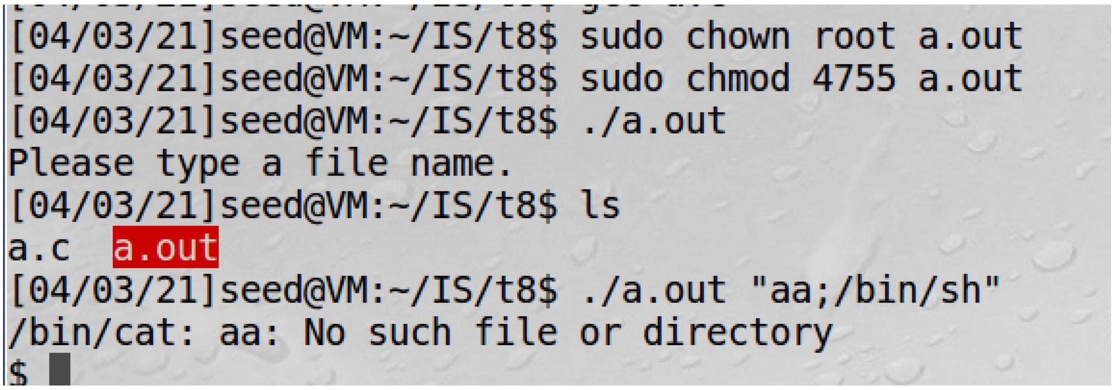


```
[04/03/21]seed@VM:~/IS/t7$ ls -al myprog.out
-rwsr-xr-x 1 user1 seed 7348 Apr  3 12:00 myprog.out
[04/03/21]seed@VM:~/IS/t7$ export LD_PRELOAD=./libmylib.so.1
.0.1
[04/03/21]seed@VM:~/IS/t7$ ./myprog.out
[04/03/21]seed@VM:~/IS/t7$ 
```

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

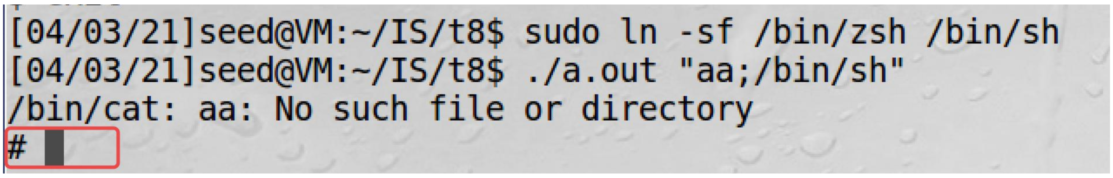
Step 1:

system 是通过调用/bin/sh -c command 命令来执行 command，外部命令不是直接执行，而是 shell 程序先执行，然后 shell 将 command 作为输入并解析它，因此可以通过一个分号分隔出两个命令，让 system 执行。如此一来可以轻易获取 root 权限的 shell，但是如图所示，由于 ubuntu16.04 中 dash 的保护机制，当它发现自己在一个 Set-Uid 的进程中运行时，会立刻把有效用户 ID 变成实际用户 ID，主动放弃特权。



```
[04/03/21]seed@VM:~/IS/t8$ sudo chown root a.out
[04/03/21]seed@VM:~/IS/t8$ sudo chmod 4755 a.out
[04/03/21]seed@VM:~/IS/t8$ ./a.out
Please type a file name.
[04/03/21]seed@VM:~/IS/t8$ ls
a.c  a.out
[04/03/21]seed@VM:~/IS/t8$ ./a.out "aa;/bin/sh"
/bin/cat: aa: No such file or directory
$ 
```

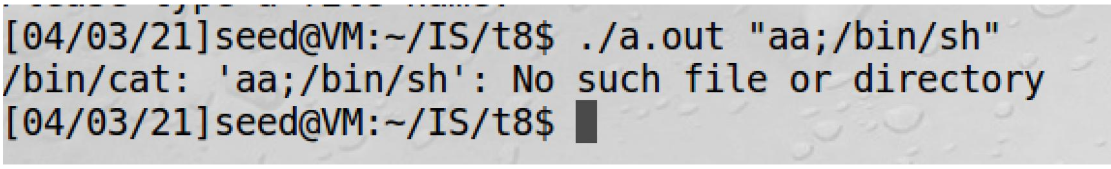
在使用一个没有实现保护机制的 zsh 后，则轻易获得了 root 权限。



```
[04/03/21]seed@VM:~/IS/t8$ sudo ln -sf /bin/zsh /bin/sh
[04/03/21]seed@VM:~/IS/t8$ ./a.out "aa;/bin/sh"
/bin/cat: aa: No such file or directory
# 
```

Step 2:

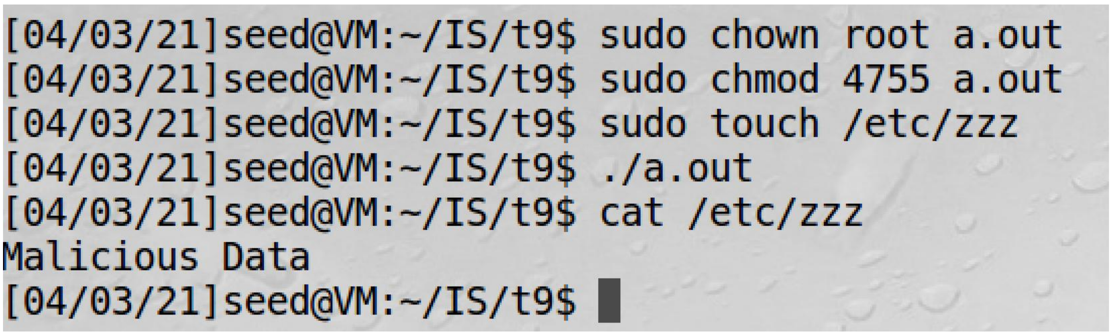
execve()是相对 system 更安全的方式，它去掉了 shell 这个中间人，直接请求操作系统执行指定的命令。所以攻击失败。



```
[04/03/21]seed@VM:~/IS/t8$ ./a.out "aa;/bin/sh"
/bin/cat: 'aa;/bin/sh': No such file or directory
[04/03/21]seed@VM:~/IS/t8$ 
```

Task 9: Capability Leaking

如图，/etc/zzz 文件被写入 Malicious Data，证明普通用户修改了 root 用户才能修改的文件，这是因为 a.out 在运行过程中提权到 root 级别，并且程序没有及时关闭持有的文件句柄，即使进行了降权操作，由于文件处于打开状态，普通用户依然能够访问该文件。



```
[04/03/21]seed@VM:~/IS/t9$ sudo chown root a.out
[04/03/21]seed@VM:~/IS/t9$ sudo chmod 4755 a.out
[04/03/21]seed@VM:~/IS/t9$ sudo touch /etc/zzz
[04/03/21]seed@VM:~/IS/t9$ ./a.out
[04/03/21]seed@VM:~/IS/t9$ cat /etc/zzz
Malicious Data
[04/03/21]seed@VM:~/IS/t9$ 
```