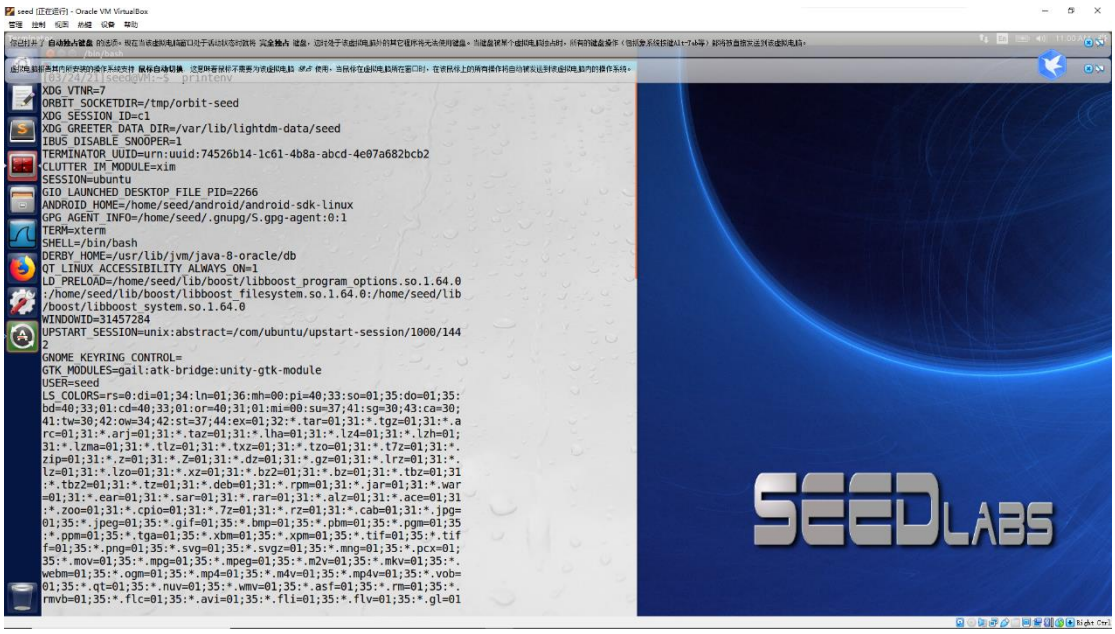
SEED 实验

李津全 57119121 2021/7/10

Overview The learning objective of this lab is for students to understand how environment variables affect program and system behaviors. Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are used by most operating systems, since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve that is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities. In this lab, students will understand how environment variables work, how they are propagated from parent process to child, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs. This lab covers the following topics: • Environment variables • Set-UID programs • Securely invoke external programs • Capability leaking • Dynamic loader/linker Readings and videos. Detailed coverage of the Set-UID mechanism, environment variables, and their related security problems can be found in the following: • Chapters 1 and 2 of the SEED Book, Computer & Internet Security: A Hands-on Approach, 2nd Edition, by Wenliang Du. See details at https://www.handsonsecurity.net. • Section 2 of the SEED Lecture at Udemy, Computer Security: A Hands-on Approach, by Wenliang Du. See details at https://www.handsonsecurity.net/video.html. Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

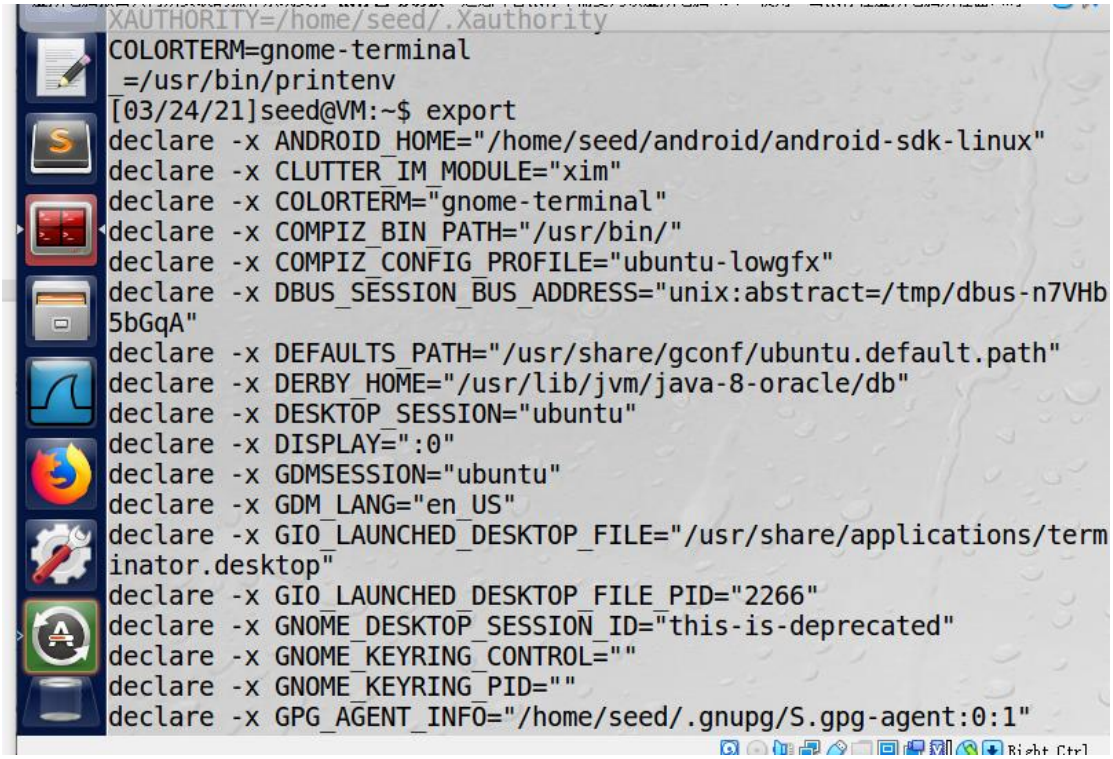
Task1

使用 printenv 命令输出环境变量输入命令 printenv，得到如下结果（截取部分）



使用 export 和 unset 设置或删除环境变量

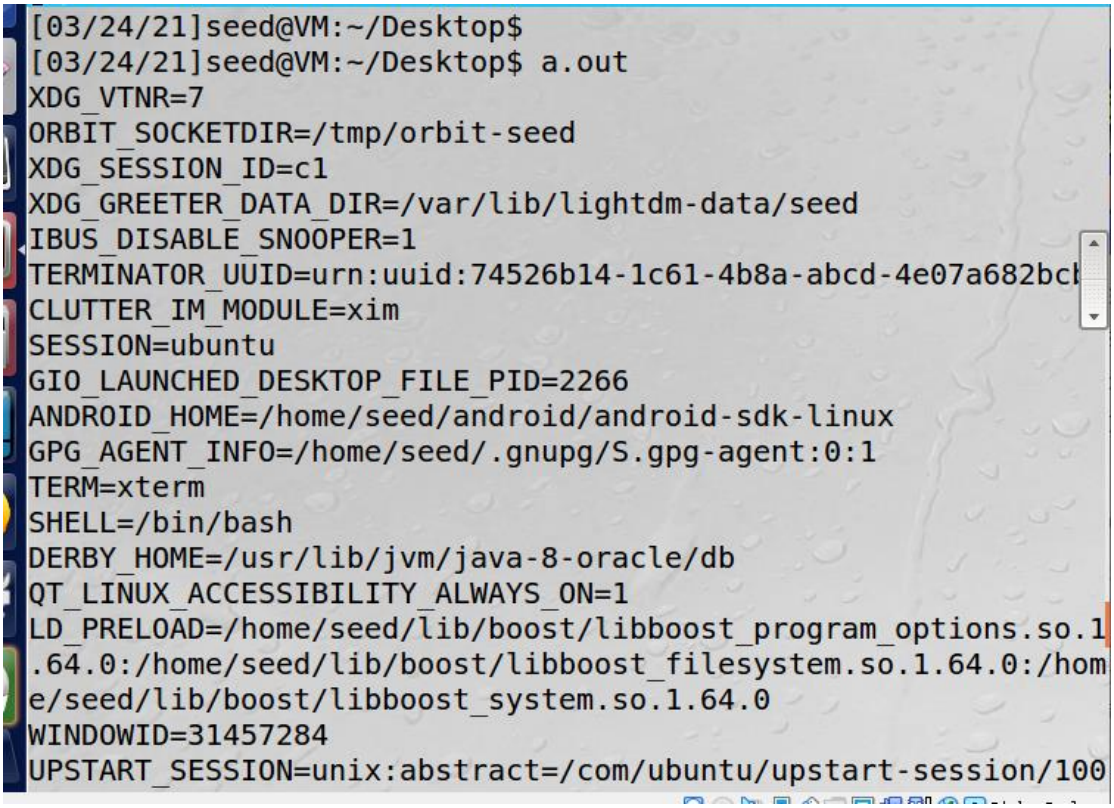
使用 export 设置环境变量，使用 echo 显示，$符号实际作用是将变量转换成字符，方便输出 。



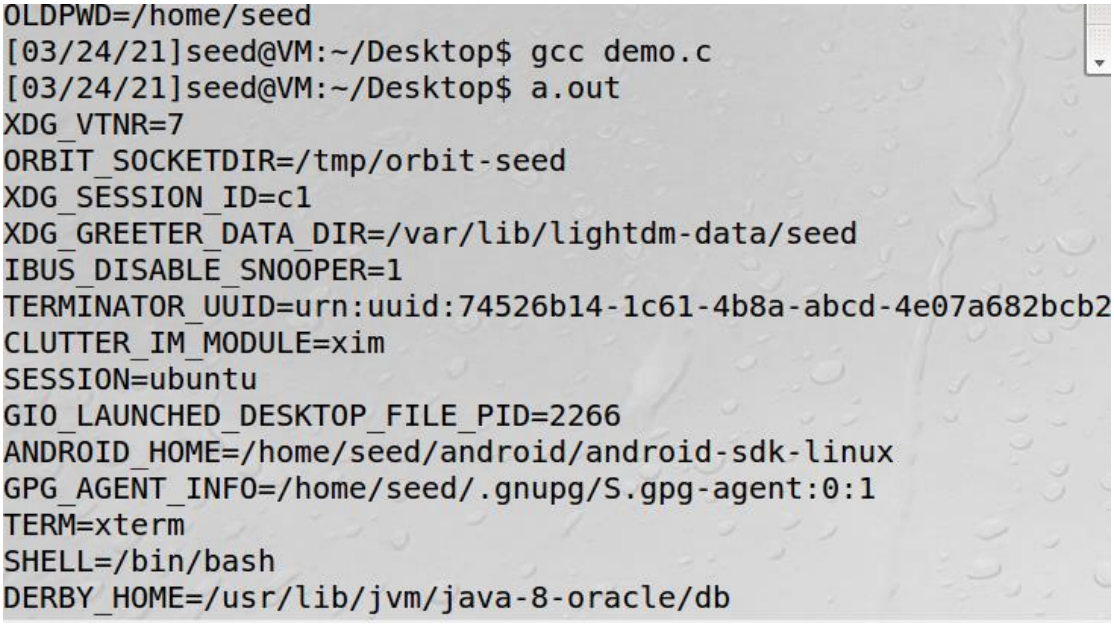
Task2

编译 C 文件，将结果保存为 a.out 文件将代码保存为 demo.c 文件并放在桌面。进入桌面路径，编译 C 文件。

执行保存结果的 a.out 文件，查看代码的运行结果，发现为各个环境变量的值（截取部分）。



按题意，将 child process 中 printenv()注释，将 process 中 parent printenv()取消注释，重新保存编译 C 文件。



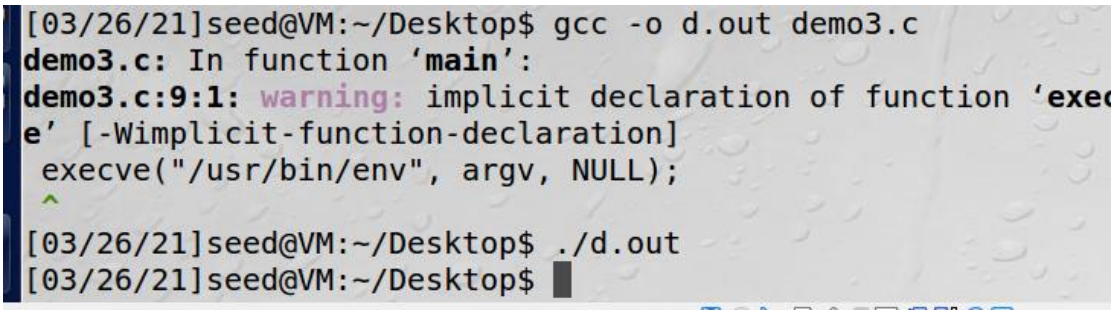
比较两者结果

子进程环境变量会继承父环境变量。子进程自父进程继承到进程的资格、环境、堆栈、内存等，但子进程所独有的是不同的父进程号、自己的文件描述符和目录流的拷贝、在 tms 结构中的系统时间、不继承异步输入和输出

Task3

编译并运行以下程序。描述观察到的实验结果。该程序简单地调用了/usr/bin/env,该系统调用能够打印出当前进程的环境变量。

重新保存和编译文件，发现执行结果为空。



把 execve（）的调用改为以下内容，观察结果

将原语句换为：execve("/usr/bin/env", argv, environ)；重新保存和编译文件，得到如下结果。

```
[03/26/21]seed@VM:~/Desktop$ gcc -o c.out demo.c
[03/26/21]seed@VM:~/Desktop$ ./c.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:74526b14-1c61-4b8a-abcd-4e07a682bcb2
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=2266
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1
.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/hom
```

Task4

重 新 保 存 和 编 译 文 件

```
[03/26/21]seed@VM:~/Desktop$ gcc -o e.out demo4.c
[03/26/21]seed@VM:~/Desktop$ ./e.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
ORBIT_SOCKETDIR=/tmp/orbit-seed
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/hom
e/seed/source/boost_1_64_0/stage/lib:
SHLVL=1
LIBGL_ALWAYS_SOFTWARE=1
J2REDIR=/usr/lib/jvm/java-8-oracle/jre
HOME=/home/seed
QT4_IM_MODULE=xim
OLDPWD=/home/seed
DESKTOP_SESSION=ubuntu
```

查阅资料得 system（）的调用格式如下：

int system（const char * string）

system()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，此命>令执行完后随即返回原调用的进程。在调用 system()期间 SIGCHLD 信号会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略。

具体个描述为这样三个步骤：调用 fork（）函数新建一个子进程；在子进程中调用 exec 函数去执行 command；在父进程中调用 wait 去等待子进程结束。

返回值 =-1:出现错误 =0:调用成功但是没有出现子进程 >0:成功退出的子进程的 id 如果 system()在调用/bin/sh 时失败则返回 127，其他失败原因返回-1。若参数 string 为空指针(NULL)，则返回非零值>。如果 system()调用成功则最后会返回执行 shell 命令后的返回值，但是此返回值也有可能为 system()调用/bin/sh 失败所返回的 127，因此最好能再检查 errno 来确认执行成功。

Task5

重新保存、编译和执行给出的代码，得到如下结果（截取部分），此结果就是当前所有环境变量：

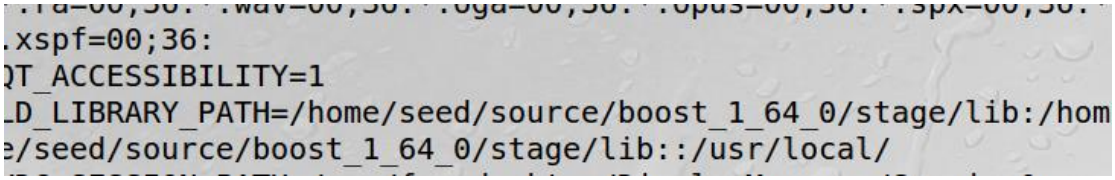
```
JOB=unity-settings-daemon
[03/26/21]seed@VM:~/Desktop$ gcc -o f.out demo5.c
[03/26/21]seed@VM:~/Desktop$ ./f.out
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
IBUS_DISABLE_SNOOPER=1
TERMINATOR_UUID=urn:uuid:74526b14-1c61-4b8a-abcd-4e07a682bcb2
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=2266
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1
.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/hom
```

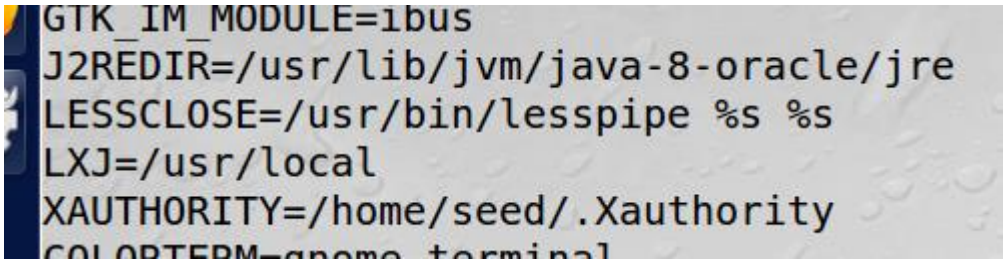
将上述程序的所有权改为 root，并使它成为一个 Set-UID 程序

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

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





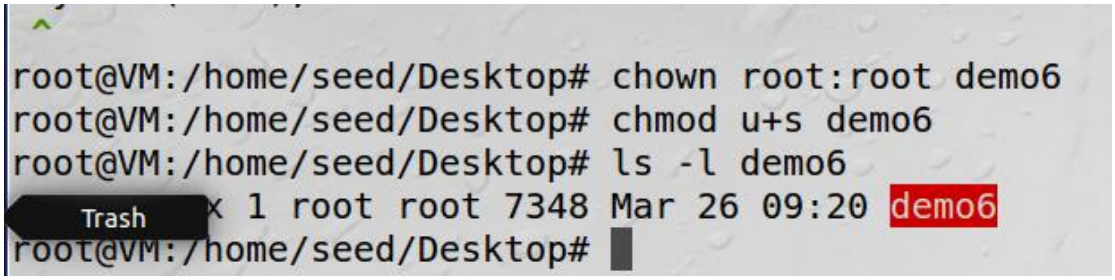
可以看到，以上三个被定义的环境变量全部被包括在 shell 中。

Task6

保存代码为 demo6.c 文件，切换为 root 用户，将其编译为 demo6，并设置其所有者为 root，赋予 SUID 特殊权限。

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

将 bin/sh 复制到当前目录并命名为 ls，执行 demo6 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。



```
-rwsr-xr-x 1 root root 7348 Mar 26 09:20 demo6
root@VM:/home/seed/Desktop# cp /bin/sh ls
root@VM:/home/seed/Desktop# exit
exit
[03/26/21]seed@VM:~/Desktop$ cp /bin/sh ls
cp: cannot create regular file 'ls': Permission denied
[03/26/21]seed@VM:~/Desktop$ demo6
a.out  c.out    demo4.c  demo6    demo.c  e.out  g.out
b.out  demo3.c  demo5.c  demo6.c  d.out   f.out  ls
[03/26/21]seed@VM:~/Desktop$
```

Task8

保存代码为 demo8.c 文件，切换为 root 用户，将其编译为 demo8，并设置其所有者为 root，赋予 SUID 特殊权限。

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。

执行 demo8，发现原本只有 root 用户才具有读、写、执行的 MY 文件，已经更名为 my

```
-rwx------ 1 root root 0 Mar 26 10:06 MY
root@VM:/home/seed/Desktop# demo8 "MY;mv MY my"
root@VM:/home/seed/Desktop# ls
a.out  demo3.c  demo6    demo8.c  e.out  ls
b.out  demo4.c  demo6.c  demo.c   f.out  my
c.out  demo5.c  demo8    d.out    g.out  mylib.c
root@VM:/home/seed/Desktop#
```

注释掉 system(command)语句，并取消 execve（）语句;该程序将使用 execve（）来调用该命令。编译程序，并使之成为 Set-UID（由 root 拥有）。你在步骤 1 中的攻击仍然有效吗？请描述并解释你的观察。

重新编译 demo8 文件，并设置其所有者为 root，赋予 SUID 特殊权限。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。

执行 demo8，发现上一步的攻击方法已经失效

```
_=./demo8
OLDPWD=/home/seed
[03/26/21]seed@VM:~/Desktop$ ls
a.out  demo3.c  demo6    demo8.c  e.out  ls  mylib.c
b.out  demo4.c  demo6.c  demo.c   f.out  my
c.out  demo5.c  demo8    d.out    g.out  MY
[03/26/21]seed@VM:~/Desktop$
```