

Lab 1

Environment Variable and Set-UID Program

Name: 程昊

Student Number: 57117128

Task1: Manipulating Environment Variables

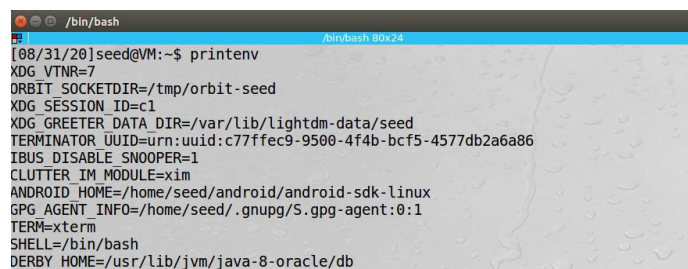
◆ 实验内容:

1. 使用 printenv 或者 env 命令打印环境变量;
2. 使用 export 和 unset 命令设置和取消环境变量.

◆ 实验结果:

1. 利用 printenv 和 env 打印环境变量:

在 SEED Ubuntu 环境下正常打开一个 bash, 输入 printenv 命令, 结果如下图所示:



```
/bin/bash
[08/31/20]seed@VM:~$ printenv
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
TERMINATOR_UUID=urn:uuid:c77fec9-9500-4f4b-bcf5-4577db2a6a86
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
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
```

输入 env 命令:

```
/bin/bash
[08/31/20]seed@VM:~$ env
XDG_VTNR=7
ORBIT_SOCKETDIR=/tmp/orbit-seed
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
TERMINATOR_UUID=urn:uuid:236bfe52-b87e-4109-90b0-361a4c182482
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
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
```

两个命令打印的内容并无差别。另外二者都属于软件命令，而非 shell 内置命令 (/usr/bin/env (printenv))。

2. 使用 export 和 unset 设置、取消环境变量：

首先使用 export 导入一个环境变量 TEST_VAR(:/home)：

```
[08/31/20]seed@VM:~$ export TEST_VAR=/home
```

利用 printenv 查看是否设置成功：

```
[08/31/20]seed@VM:~$ export TEST_VAR=65536
[08/31/20]seed@VM:~$ printenv TEST_VAR
65536
```

可以看到 TEST_VAR 变量已经被成功设置。再使用 unset 取消该环境变量，再次查看 TEST_VAR 的值：

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

此时在终端没有打印任何内容，也即该环境变量不存在(已被取消)。

Task2: Passing Environment Variables from Parent Process to Child Process

◆ 实验内容：

分别打印父进程和子进程的环境变量，并对输出结果进行比较。

◆ 实验结果：

按照实验手册的步骤进行实验，输入如下命令：

```
[08/31/20]seed@VM:~$ cd Desktop
[08/31/20]seed@VM:~/Desktop$ cd test
[08/31/20]seed@VM:~/.../test$ gcc test.c -o test
[08/31/20]seed@VM:~/.../test$ ./test > result_child.txt
[08/31/20]seed@VM:~/.../test$ gcc test.c -o test
[08/31/20]seed@VM:~/.../test$ ./test > result_parent.txt
[08/31/20]seed@VM:~/.../test$ diff result_child.txt result_parent.txt
[08/31/20]seed@VM:~/.../test$ █
```

两次编译的 C 程序分别对应注释父进程或者子进程的 `printenv()` 语句。

利用 `diff` 命令对比父进程和子进程的环境变量结果文本，在控制台并无任何输出，证明父进程和子进程的环境变量完全相同。

◆ 实验结论：

在 Linux OS 下，利用 `fork()` 系统调用生成的子进程继承了父进程的环境变量。

Task3: Environment Variables and `execve()`

◆ 实验内容：

利用 `execve` 方法执行程序 `/usr/bin/env`，并且改变第三个参数 (`envp`)，对比输出结果。

◆ 实验结果：

改变 `envp` 参数分别编译运行程序：

```
[08/31/20]seed@VM:~/.../test$ gedit task3.c
[08/31/20]seed@VM:~/.../test$ gcc task3.c -o task3
[08/31/20]seed@VM:~/.../test$ ./task3 > task3_1.txt
[08/31/20]seed@VM:~/.../test$ gcc task3.c -o task3
[08/31/20]seed@VM:~/.../test$ ./task3 > task3_2.txt
[08/31/20]seed@VM:~/.../test$ diff task3_1.txt taks3_2.txt
diff: taks3_2.txt: No such file or directory
[08/31/20]seed@VM:~/.../test$ diff task3_1.txt task3_2.txt
```

得到的结果如下(部分结果)：

```
0a1,75
> XDG_VTNR=7
> ORBIT_SOCKETDIR=/tmp/orbit-seed
> XDG_SESSION_ID=c1
> XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
> TERMINATOR_UUID=urn:uuid:e4057682-4b7e-4a69-b2e4-865d46e80dcb
> IBUS_DISABLE_SNOOPER=1
> CLUTTER_IM_MODULE=xim
> SESSION=ubuntu
> GIO_LAUNCHED_DESKTOP_FILE_PID=6636
> 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:/home/seed/lib/boost/libboost_system.so.1.64.0
> WINDOWID=58720260
> UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1617
```

事实上当 envp 参数为 NULL 时，程序的输出结果为空。当 envp 参数被指定时，环境变量为 envp 数组指定的参数序列。

◆ 实验结论：

Linux 下生成一个新的进程的方法基本为 fork() 与 execve() 的结合调用，结合 task2 我们可以有如下结论，一个新的进程的环境变量会默认继承自它的父进程，但是 execve() 在覆写调用进程时，会根据传入参数的情况改变进程的环境变量值。

Task4: Environment Variables and system()

◆ 实验内容：

使用 system() 函数执行一个新程序，验证新程序的环境变量是否与调用程序相同。

◆ 实验结果：

编译运行手册中的代码：


```
[08/31/20]seed@VM:~/.../test$ gedit task4.c
[08/31/20]seed@VM:~/.../test$ gcc task4.c -o task4
[08/31/20]seed@VM:~/.../test$ ./task
bash: ./task: No such file or directory
[08/31/20]seed@VM:~/.../test$ ./task4
```

输出结果(部分):

```
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
ORBIT_SOCKETDIR=/tmp/orbit-seed
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
SHLVL=1
J2REDIR=/usr/lib/jvm/java-8-oracle/jre
HOME=/home/seed
QT4_IM_MODULE=xim
OLDPWD=/home/seed/Desktop
DESKTOP_SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE=/usr/share/applications/terminator.desktop
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
GTK_MODULES=gail:atk-bridge:unity-gtk-module
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
```

◆ 实验结论:

system() 方法执行的新程序环境变量确实与调用程序的环境变量相同(通过当前 shell 中输入 env 对比结果即可得知)。

Task5: Environment Variable and Set-UID Programs

◆ 实验内容:

改变编译程序的拥有者，并且设置为 Set-UID 程序，观察其环境变量的变化情况。

◆ 实验结果:

编译手册中提供的代码，然后改变可执行程序的权限和所有者:

```
[08/31/20]seed@VM:~/.../test$ sudo chown root task5
[08/31/20]seed@VM:~/.../test$ sudo chmod 4755 task5
```

再修改环境变量 PATH, LD_LIBRARY_PATH 的值，并添加值 HAO:

```
[08/31/20]seed@VM:~/.../test$ export PATH="$PATH:/home/seed/Desktop/test"
```

```
[08/31/20]seed@VM:~/.../test$ export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/seed/Desktop/test"
```

```
[08/31/20]seed@VM:~/.../test$ export HAO="/home/seed/Desktop/test"
```

运行 task5 程序，得到如下结果:

```
[08/31/20]seed@VM:~/.../test$ ./task5 | 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/sbin:/usr/bin:/usr/games:/usr/local/games:./snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin:/home/seed/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin:/home/seed/Desktop/test
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[08/31/20]seed@VM:~/.../test$ ./task5 | grep LD
OLDPWD=/home/seed/Desktop
[08/31/20]seed@VM:~/.../test$ ./task5 | grep HAO
HAO=/home/seed/Desktop/test
```

LD_LIBRARY_PATH 并未被找到。可以看到程序的输出结果中包含了环境变量修改后的 PATH 和 HAO。

◆ 实验结论:

Set-UID 程序使用的环境变量并非用户完全是用户所定义的环境变量。

Task6: The PATH Environment Variable and Set-UID Programs

◆ 实验内容:

通过改变 path 环境变量，观察所编译程序的运行结果

◆ 实验结果:

编译运行 task6.c 文件，结果如下:

```
[08/31/20]seed@VM:~/.../test$ gedit task6.c
[08/31/20]seed@VM:~/.../test$ gcc task6.c -o task6
[08/31/20]seed@VM:~/.../test$ ./task6
result_child.txt  task3_1.txt  task4      task5      task6.c
result_parent.txt task3_2.txt  task4.c    task5.c    test
task3            task3.c     task4.txt  task6      test.c
```

修改环境变量 PATH，在其最前端插入当前的工作目录：

```
[08/31/20]seed@VM:~/.../test$ export PATH=/home/seed/Desktop/test:$PATH
[08/31/20]seed@VM:~/.../test$ printenv PATH
/home/seed/Desktop/test:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:
```

做一定处理对系统的/bin/sh：

```
[08/31/20]seed@VM:~/.../test$ sudo rm /bin/sh
[08/31/20]seed@VM:~/.../test$ sudo ln -s /bin/zsh /bin/sh
[08/31/20]seed@VM:~/.../test$ ./task6
```

再将/bin/sh 复制到当前目录，命名为 ls：

```
[08/31/20]seed@VM:~/.../test$ cp /bin/sh ls
```

可以看到此时不需要 sudo 命令即可直接执行 chown 和 chmod 命令。说明由 setUID 程序生成的这个 shell 具有 root 权限。

```
[08/31/20]seed@VM:~/.../test$ cp /bin/sh ls
[08/31/20]seed@VM:~/.../test$ ./task6
VM# chown root hello
VM# chmod 4755 hello
VM# exit
```

◆ 实验结论：

Set-UID 程序运行的子程序是具有 root 权限的。PATH 环境变量为可执行程序搜索的路径，可通过修改 PATH 路径调整程序搜索目录的顺序。

Task7: The LD_PRELOAD Environment Variable and Set-UID Programs

◆ 实验内容：

观察 Set-UID 程序如何影响 LD 环境变量。

◆ 实验结果：

首先编写自己的 sleep 函数并且生成动态链接库，同时修改 LD_PRELOAD 环

境变量的值，再编译运行 myproc 程序：

```
[09/01/20]seed@VM:~/.../test$ gcc -fPIC -g -c task7.c
[09/01/20]seed@VM:~/.../test$ gcc -shared -o libmylib.so.1.0.1 task7.o -lc
[09/01/20]seed@VM:~/.../test$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/01/20]seed@VM:~/.../test$ gcc myproc.c -o myproc
myproc.c: In function 'main':
myproc.c:2:2: warning: implicit declaration of function 'sleep' [-Wimplicit-func
tion-declaration]
  sleep(1);
  ^
[09/01/20]seed@VM:~/.../test$ ./myproc
I am not real sleep program
```

可以看到由于 LD_PRELOAD 环境变量的改变，sleep 函数变成了我们自己编写的 sleep 函数。

此时再将 myproc 可执行程序变为 set-UID root 程序，再次运行：

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

停顿一秒之后控制台转入下一行，无打印任何输出。说明此时 myproc 程序对应的 LD_PRELOAD 环境变量并非 seed 用户下所设置的 LD_PRELOAD 值。

此时打开一个新的终端，进入 root 用户，修改 root 用户下的 LD_PRELOAD 值，再次运行 myproc 程序：

```
root@VM: /home/seed/Desktop/test 80x24
[09/01/20]seed@VM:~$ sudo su
root@VM:/home/seed# printenv LD_PRELOAD
/home/seed/lib/boost/libboost_program_options.so.1.64.0:/home/seed/lib/boost/l
boost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_system.so.1.64.0
root@VM:/home/seed# ^C
root@VM:/home/seed# cd Desktop/test
root@VM:/home/seed/Desktop/test# export LD_PRELOAD=libmylib.so.1.0.1
root@VM:/home/seed/Desktop/test# ./myproc
I am not real sleep program
root@VM:/home/seed/Desktop/test#
```

可以看到此时 sleep 函数的运行结果又变为我们自己所编写的 sleep 函数了，但是这是在 root 用户下运行的。

退出 root 用户，回归普通用户再次运行：

```
root@VM:/home/seed/Desktop/test# exit
exit
[09/01/20]seed@VM:~/.../test$ ./myproc
[09/01/20]seed@VM:~/.../test$
```

可以看到此时仍然是真正的 sleep 函数(myproc 仍是 root Set-UID 程序)。

再次回到 root 用户下，可以看到 printenv 的值又变回最开始的值了。

再把程序的所有者改为 hao（新创建的普通用户），再次运行 myproc（此时 seed 下 LD_PRELOAD 环境变量的值仍未 libmylib.so.1.0.1）：

```
[09/01/20]seed@VM:~/.../test$ sudo chown hao myproc
[09/01/20]seed@VM:~/.../test$ ls -l myproc
-rwxr-xr-x 1 hao root 7348 Sep  1 00:05 myproc
[09/01/20]seed@VM:~/.../test$ ./myproc
[09/01/20]seed@VM:~/.../test$
```

此时输出仍为空。

◆ 实验结论：

由以上实验进行对比，可以得出结论，只有 ruid 和 euid 相同时，LD_PRELOAD 变量才会有效。当二者有差异时，该变量不发挥作用。

Task8: Invoking External Programs Using system() versus execve()

◆ 实验内容：

对比 system 和 execve 方法，考察其安全性。

◆ 实验结果：

首先按照实验手册内容编译 task8 文件，并且设置为 set-root-UID 程序。其次，为了做对比试验，在 task8 可执行程序的工作目录下创建一个 dir1 目录，在 dir1 目录中存放一个名为 Mytest 的普通文件，将 dir1 和 Mytest 的权限均设为 700，所有用户为 root。

```
[09/01/20]seed@VM:~/.../test$ mkdir dir1
[09/01/20]seed@VM:~/.../test$ sudo chown root:root dir1
[09/01/20]seed@VM:~/.../test$ sudo chmod 700 dir1
[09/01/20]seed@VM:~/.../test$ cd dir1
bash: cd: dir1: Permission denied
```

这里可以看到 seed 用户没有对 dir1 目录文件进行读写操作的权限。

对于 task8 可执行文件的传参方式我们稍作修改，如下图所示：

```
[09/01/20]seed@VM:~/.../test$ ./task8 "ls.c;ls -l dir1"
#include<stdio.h>
int main() {
    printf("hello, it is task 6.");
    return 0;
}
total 4
-rwx----- 1 root root 18 Sep  1 01:41 Mytest
```

这里相当于传入 system 的参数为/bin/cat ls.c;ls -l dir1。而 system 是调用/bin/sh 开启一个新的 shell 执行该命令行的。我们再尝试删除 Mytest 文件：

```
total 4
-rwx----- 1 root root 18 Sep  1 01:41 Mytest
[09/01/20]seed@VM:~/.../test$ ./task8 "ls.c;rm -r ./dir1/Mytest"
#include<stdio.h>
int main() {
    printf("hello, it is task 6.");
    return 0;
}
[09/01/20]seed@VM:~/.../test$ ./task8 "ls.c;ls -l dir1"
#include<stdio.h>
int main() {
    printf("hello, it is task 6.");
    return 0;
}
total 0
```

可以看到 dir1 目录下的 Mytest 文件已经被成功删除，但 seed 用户本身连访问 dir1 目录的权限都没有，更无法删除 Mytest 文件(删除 Mytest 文件相当于对 dir1 目录文件的写操作)。

将 task8.c 中的 execve 语句注释取消，同时注释掉 sysystem 语句，重新编译，设置 UID，再次按照上述的传参方法传入参数：

```
[09/01/20]seed@VM:~/.../test$ sudo chown root:root task8
[09/01/20]seed@VM:~/.../test$ sudo chmod 4755 task8
[09/01/20]seed@VM:~/.../test$ ./task8 "ls.c;ls -l dir1"
/bin/cat: 'ls.c;ls -l dir1': No such file or directory
[09/01/20]seed@VM:~/.../test$
```

此时已经无法进行越权操作了。

◆ 实验结论：

Execve 与 sysystem 方法的底层逻辑不同，system 方法本质上是开启新的 shell 去执行一段 cmd，而 execve 则是传入可执行程序的参数的方式执行新的程序，execve 方法显然更加安全。System 方法在特权程序中被调用会显得十分不

安全。

Task9: Capability Leaking

◆ 实验内容:

编译运行手册中提供的程序, 模拟攻击者注入了恶意语句, 观察 Capability Leaking 现象。

◆ 实验结果:

首先在/etc 下创建 zzz 文件(我写入了 111):

```
[09/01/20]seed@VM:~/.../test$ sudo gedit /etc/zzz
```

此时 zzz 文件的权限如下:

```
[09/01/20]seed@VM:~/.../test$ ls -ls /etc/zzz
4 -rw-r--r-- 1 root root 4 Sep  1 02:35 /etc/zzz
```

只有 root 用户才对其有写的权限。

再对 task9.c 进行编译、权限修改, Set-UID:

```
[09/01/20]seed@VM:~/.../test$ ./task9
cannot open /etc/zzz
[09/01/20]seed@VM:~/.../test$ sudo chown root:root task9
[09/01/20]seed@VM:~/.../test$ sudo chmod 4755 task9
```

再尝试运行 ./task9 程序, 并且打印 zzz 文件内容

```
[09/01/20]seed@VM:~/.../test$ ./task9
[09/01/20]seed@VM:~/.../test$ cat /etc/zzz
111
Malicious Data
```

可以看到此时 zzz 文件内容已经被修改。:

◆ 实验结论:

之所以 zzz 文件内容可以被修改, 是因为在 task9 可执行程序运行时打开了 zzz 文件, 在执行完相关任务之后却没有在撤销特权前及时关闭 zzz 的文件描述符 fd, 此时 fd 还具有 root 特权, 因此可以执行对 zzz 文件的写操作。