

## Task 1: Manipulating Environment Variables

```
[09/02/20]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:219776c2-a208-4b82-a3af-f7a283919042
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GIO_LAUNCHED_DESKTOP_FILE_PID=5913
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/1000/1373
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
```

图 1: env 打印环境变量 (截取部分输出)

```
[09/02/20]seed@VM:~$ env | grep HOME
ANDROID_HOME=/home/seed/android/android-sdk-linux
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
JAVA_HOME=/usr/lib/jvm/java-8-oracle
HOME=/home/seed
```

图 2: 指定环境变量关键词

```
[09/02/20]seed@VM:~$ export SKWang=seucyber
[09/02/20]seed@VM:~$ env | grep SKWang
SKWang=seucyber
[09/02/20]seed@VM:~$ unset SKWang
[09/02/20]seed@VM:~$ env | grep SKWang
```

图 3: 环境变量的声明与删除

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

```
[09/02/20]seed@VM:~/code$ diff child parent
5c5
< TERMINATOR_UUID=urn:uuid:219776c2-a208-4b82-a3af-f7a283919042
---
> TERMINATOR_UUID=urn:uuid:41b66ee2-fbc8-45c1-acc3-61818d37ce2d
9c9
< GIO_LAUNCHED_DESKTOP_FILE_PID=5913
---
> GIO_LAUNCHED_DESKTOP_FILE_PID=6291
74c74
< _=./t2
---
> _=./t2_2
```

图 4: 对于输出文件的 diff 结果

从结果中可以看出, 子进程和父进程在环境变量上只有很小的差别, 体现在终端的 UUID、进程的 PID 和程序名上, 其他环境变量是完全一致的。查阅 `man fork` 可以发现“The child process is an exact duplicate of the parent process except for the following points: ...”, 而被除外的数据就包括进程 ID 等。因此, 在使用 `fork()` 创建子进程时, 环境变量是继承的。

## Task 3: Environment Variables and `execve()`

```
[09/03/20]seed@VM:~/code$ ./t31
[09/03/20]seed@VM:~/code$
```

图 5: `envp` 设置为 `NULL`

当传递的参数为 `NULL` 时, 新进程不包含任何环境变量。

```
[09/03/20]seed@VM:~/code$ ./t32
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:41b66ee2-fbc8-45c1-acc3-61818d37ce2d
IBUS_DISABLE_SNOOPER=1
CLUTTER_IM_MODULE=xim
```

图 6: `envp` 设置为 `environ`

而当传递的参数为全局变量 `environ` 时, 当前进程的所有环境变量都会传递给新进程并打印出来。

所以, 在使用 `execve()` 创建进程时, 原进程的内存空间会被新程序的数据覆盖, 进程中存储的所有环境变量都将丢失。而如果想让新进程继承环境变量, 则要通过 `execve()` 函数的第三个参数 `envp` 进行指定, 将环境变量数组传递下去。

## Task 4: Environment Variables and system()

```
[09/03/20]seed@VM:~/code$ ./t4
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
```

图 7: 使用 system 调用程序

因为 system() 的实现在调用 execve() 时指定了环境变量参数, 因此环境变量被继承。

## Task 5: Environment Variable and Set-UID Programs

```
[09/03/20]seed@VM:~$ PATH=SKWang:$PATH
[09/03/20]seed@VM:~$ LD_LIBRARY_PATH=SKWang:$LD_LIBRARY_PATH
[09/03/20]seed@VM:~$ export SKWang=SEUcyber
[09/03/20]seed@VM:~$ env | grep SKWang
LD_LIBRARY_PATH=SKWang:/home/seed/source/boost_1_64_0/stage/lib
PATH=SKWang:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/
e/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-ora
x/platform-tools:/home/seed/android/android-ndk/android-ndk-r8c
SKWang=SEUcyber
```

图 8: 在当前 shell 设置环境变量

```
[09/03/20]seed@VM:~/code$ ./t5 | grep SKWang
PATH=SKWang:/home/seed/bin:/usr/local/sbin:/usr/l
e/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/
x/platform-tools:/home/seed/android/android-ndk/a
SKWang=SEUcyber
```

图 9: 运行 Set-UID 程序查看环境变量

一并在 shell 中设置了三个环境变量, 但是 LD\_LIBRARY\_PATH 没有被 Set-UID 子进程继承。这是由于动态链接器的防御策略, 防止 Set-UID 程序的函数调用被 LD\_LIBRARY\_PATH 指定的路径劫持。



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

```
[09/03/20]seed@VM:~/code$ ls
myls.c  t6_victim  t6_victim.c
[09/03/20]seed@VM:~/code$ export PATH=.:$PATH
[09/03/20]seed@VM:~/code$ gcc -o ls myls.c
[09/03/20]seed@VM:~/code$ ./t6_victim
#
```

图 10: 劫持系统命令

程序指定 t6\_victim 应该运行 ls 命令来打印当前目录下的文件，但是实际运行时却启动了一个 root 权限的终端。这是因为程序在运行系统命令时没有指定绝对路径，而是使用的相对路径。相对路径的查找是根据 PATH 环境变量来进行的。通过修改 PATH，迫使其目录下攻击者创建的 ls 程序被实际运行。

攻击者的 ls 运行了 root 权限的终端。这是因为 t6\_victim 本身是 Set-UID 程序，其运行时具有 root 权限，而且相关安全机制被人为关闭了。

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

```
[09/03/20]seed@VM:~/code$ ./t7
I am not sleeping!
```

图 11: 普通用户运行非特权程序

```
[09/03/20]seed@VM:~/code$ ./t7
[09/03/20]seed@VM:~/code$
```

图 12: 普通用户运行 Set-UID root 程序（未设置 LD\_\*）

```
[09/03/20]seed@VM:~/code$ ./t7
[09/03/20]seed@VM:~/code$
```

图 13: 普通用户运行 Set-UID root 程序（设置 LD\_\*）

```
[09/03/20]seed@VM:~/code$ su skwang
Password:
skwang@VM:/home/seed/code$ ls
libmylib.so.1.0.1  mylib.c  mylib.o  myprog.c  t7
skwang@VM:/home/seed/code$ env | grep LD_PRELOAD
skwang@VM:/home/seed/code$ ./t7
skwang@VM:/home/seed/code$ export LD_PRELOAD=./libmylib.so.1.0.1
skwang@VM:/home/seed/code$ env | grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
skwang@VM:/home/seed/code$ ./t7
skwang@VM:/home/seed/code$ ls -l
total 28
-rwxrwxr-x 1 seed seed 7920 Sep  3 16:38 libmylib.so.1.0.1
-rw-rw-r-- 1 seed seed   74 Sep  3 16:37 mylib.c
-rw-rw-r-- 1 seed seed 2580 Sep  3 16:38 mylib.o
-rw-rw-r-- 1 seed seed   56 Sep  3 16:40 myprog.c
-rwsr-xr-x 1 seed seed 7348 Sep  3 16:40 t7
```

图 14: 切换普通用户并尝试运行 t7

仿照 Task5 进行探究。

```
[09/03/20]seed@VM:~/code$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/03/20]seed@VM:~/code$ env | grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
[09/03/20]seed@VM:~/code$ ./check | grep LD_PRELOAD
[09/03/20]seed@VM:~/code$ sudo chown seed check
[09/03/20]seed@VM:~/code$ ./check | grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
[09/03/20]seed@VM:~/code$
```

图 15: 检查环境变量

首先设置环境变量 LD\_PRELOAD，然后运行属于 root 用户的 Set-UID 程序，发现此时进程的环境变量中没有设置的 LD\_PRELOAD。而当改变程序属主，即将程序变为非 Set-UID 程序，在此运行发现 LD\_PRELOAD。这说明当程序是 Set-UID 程序时，LD\_\* 的环境变量是不继承的，所以在 task7 中部分情况的测试用例无法执行我们的 sleep()。这是动态链接器的一种防御策略。

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

```
[09/03/20]seed@VM:~/code$ ls
secret t8 t8.c
[09/03/20]seed@VM:~/code$ ./t8 "secret"
This is a secret!
[09/03/20]seed@VM:~/code$ ./t8 "secret;rm secret"
This is a secret!
[09/03/20]seed@VM:~/code$ ls
t8 t8.c
[09/03/20]seed@VM:~/code$
```

图 16: 利用 system()

如果使用 system() 执行命令，可以通过混淆代码与数据，达到删除（修改）文件的目的。

```
[09/03/20]seed@VM:~/code$ ls
secret t8 t8.c
[09/03/20]seed@VM:~/code$ ./t8 "secret"
This is a secret!
[09/03/20]seed@VM:~/code$ ./t8 "secret;rm secret"
/bin/cat: 'secret;rm secret': No such file or directory
```

图 17: execve() 拒绝删除文件

这是因为 system() 直接将字符串作为命令放入 shell 运行，会造成代码/数据混淆的情况。而 execve() 指定第一个参数为命令，第二个参数为命令的参数，这样就保证了只有指定的命令会运行。

## Task 9: Capability Leaking



```
[09/03/20]seed@VM:~/code$ sudo chown root t9
[09/03/20]seed@VM:~/code$ sudo chmod 4755 t9
[09/03/20]seed@VM:~/code$ ./t9
[09/03/20]seed@VM:~/code$ cat /etc/zxx
Do not touch this!
Malicious Data
[09/03/20]seed@VM:~/code$
```

图 18: 权力泄露

虽然程序使用 `setuid()` 将特权收回，但是在调用 `setuid()` 之前已经使用 `root` 权限打开了 `/etc/zxx` 并保留了文件句柄 `fd`。`fd` 没有被合理的释放，在通过 `fork()` 创建子进程之后，子进程继承了父进程的资源，其中就包括特权的 `fd`。所以子进程通过 `fd` 仍然可以在只读文件中写入数据。