浙江大学

课程名称：　信息系统安全

实验名称：　Environment Variable and Set-UID

姓　　名：　王　睿 3180103650

付添翼 3180106182

刘振东 3180105566

2021 年　5 月　18 日

Lab 1：Environment Variable and Set-UID

一、 Purpose and Content 实验目的与内容

1.1 目的

(1)理解环境变量如何影响程序以及系统的行为
(2)理解环境变量是如何工作的，以及如何从父进程传递到子进程
(3)理解 execve()以及 system()函数如何唤起外部程序，以及如何受到环境变量的影响
(4)理解 capability leaking

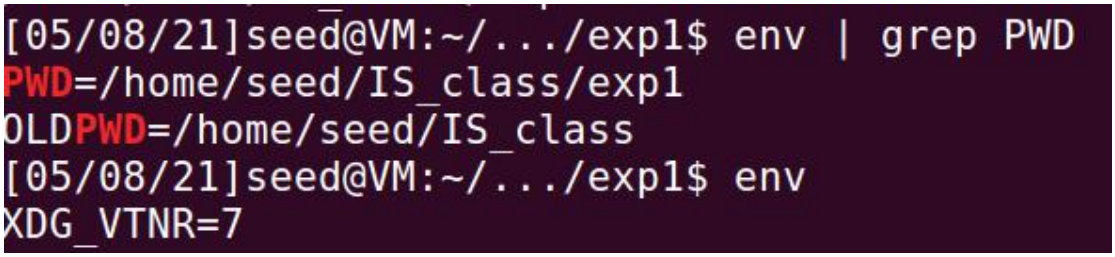
1.2 内容

包含 9 个 task。
(1)操作环境变量 Manipulating Environment Variables
(2)从父进程传递环境变量到子进程 Passing Environment Variables from Parent Process to Child Process
(3)环境变量以及 execve()函数 Environment Variables and execve()
(4)环境变量以及 system()函数 Environment Variables and system()
(5)环境变量以及 Set-UID 程序 Environment Variable and Set-UID Programs
(6)PATH 环境变量以及 Set-UID 程序 The PATH Environment Variable and Set-UID Programs
(7)LD_PRELOAD 环境变量以及 Set-UID 程序 The LD PRELOAD Environment Variable and Set-UID Programs
(8)使用 system()或者 execve()唤起外部程序 Invoking External Programs Using system() versus execve()
(9)Capability 泄漏 Leaking

二、 Detailed Steps 实验过程

2.1 操作环境变量

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

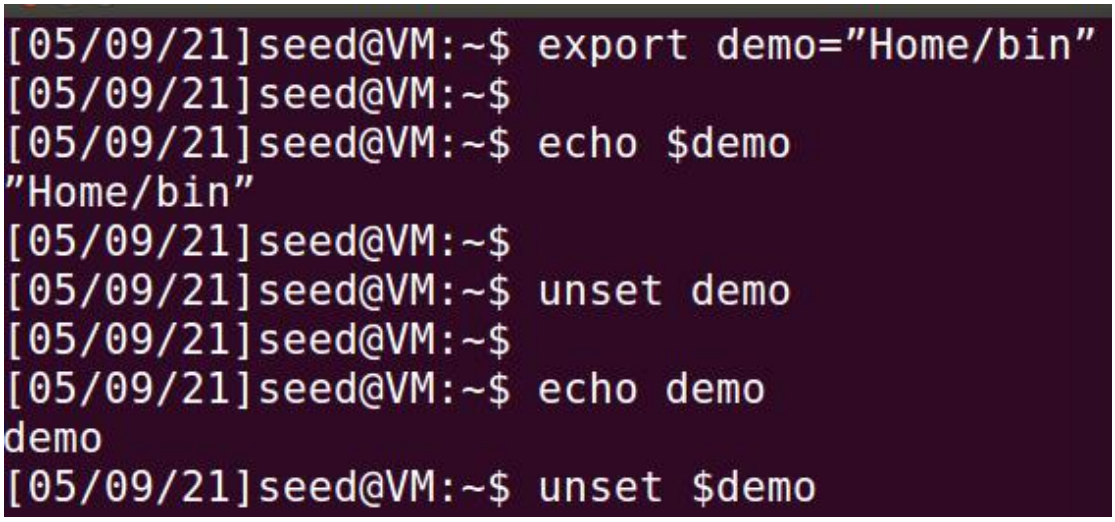


2.1.1 使用 export 和 unset 来 set 或者 unset 环境变量

输入命令：

```
export demo=”Home/bin”
echo $demo
unset demo
echo demo
```

终端输出：



2.2 从父进程传递环境变量到子进程

(1)编辑 c 文件，c 文件内容如下：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

void printenv()
{
int i = 0;
while (environ[i] != NULL) {
printf("%s\n", environ[i]);
i++;
} }
void main()
{
pid_t childPid;
switch(childPid = fork()) {
case 0: /* child process */
printenv(); ①
exit(0);
default: /* parent process */
//printenv(); ②
exit(0);
} }
```

(2)Step1:编译，执行上述 c 程序

```
[05/08/21]seed@VM:~/.../exp1$ gcc main.cpp
[05/08/21]seed@VM:~/.../exp1$ ls
a.out  main.cpp
[05/08/21]seed@VM:~/.../exp1$ a.out >exp1.txt
[05/08/21]seed@VM:~/.../exp1$ cat exp1.txt
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
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/li
b/boost/libboost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_system.so.1.
64.0
WINDOWID=67108874
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1264
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
```

(3)Step2:注释 1，使用 2 部分的 printenv()

(4)Step3:使用 diff 命令查看两个输出文件的不同

```
[05/08/21]seed@VM:~/.../exp1$ diff exp1_1.txt exp1_2.txt
[05/08/21]seed@VM:~/.../exp1$
```

通过比较这两个文件，可以发现，这两个文件输出的环境变量完全相同。

(5)分析：这两个文件输出的环境变量完全相同。说明原环境变量被子进程完全继承。fork 函数通过系统调用创建一个与原来进程几乎完全相同的进程，子进程自父进程继承了进程环境，堆栈与内存根目录等；但是子进程没有继承父进程的某些特性，比如父进程号，文件描述符等。

2.3 环境变量以及 execve()函数

(1)编写以下代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
extern char **environ;
int main()
{
char *argv[2];
argv[0] = "/usr/bin/env";
argv[1] = NULL;
execve("/usr/bin/env", argv, NULL); //①
//execve("/usr/bin/env", argv, environ);
return 0 ;
}
```

(2)Step1:编译，并执行，发现没有输出内容。

```
[05/09/21]seed@VM:~/.../exp1$ vim main3.c
[05/09/21]seed@VM:~/.../exp1$ gcc main3.c
[05/09/21]seed@VM:~/.../exp1$ a.out
[05/09/21]seed@VM:~/.../exp1$
```

(3)Step2:将 execve 语句改为 execve("/usr/bin/env", argv, environ)，再次编译，运行程序，输出如下：

```
[05/09/21]seed@VM:~/.../exp1$ gcc main3.c
[05/09/21]seed@VM:~/.../exp1$ a.out
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
```

(4)分析：函数 execve()的调用格式如下：int execve(const char * filename，char * const argv[], char * const envp[])第一个参数为一个可执行的有效的路径名。第二个参数系利用数组指针来传递给执行文件，argv 是要调用的程序执行的参数序列，也就是我们要调用的程序需要传入的参数。envp 则为传递给执行文件的新环境变量数。所以在 step1，我们赋予新进程的环境变量为 NULL，打印印出环境变量结果为空。在 step2 中，当参数 3 为外部环境变量的数组 environ 时，其传递给新的 program，所以可以打印出环境变量信息。

2.4 环境变量以及 system()函数

(1) 编写 c 程序，代码如下：

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

(2)编译运行，结果输出如下：

```
[05/08/21]seed@VM:~/.../exp1$ gcc main4.cpp -o main4.o
[05/08/21]seed@VM:~/.../exp1$ main4.o
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
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
```

(3)分析：

system 函数定义为 int system（const char * string），该函数调用/bin/sh 来执行参数指定的命令，/bin/sh 一般是一个软连接，指向某个具体的 shell，比如 bash，-c 选项是告诉 shell 从字符串 command 中读取命令。实际上 system（）函数执行了三步操作：1 fork 一个子进程；2 在子进程中调用 exec 函数去执行 command；3 在父进程中调用 wait 去等待子进程结束。

若 fork 失败，system（）函数返回-1。如果 exec 执行成功，也即 command 顺利执行完毕，则返回 command 通过 exit 或 return 返回的值。（注意，command 顺利执行不代表执行成功，例如 command：“rm debuglog.txt”，不管文件存不存在该 command 都顺利执行了）如果 exec 执行失败，也即 command 没有顺利执行，比如信号被中断，或者 command 命令根本不存在，system（）函数返回 127，如果 command 为 NULL，则 system（）函数返回值非 0，一般为 1。在子进程中，执行 execl 作用是调用 execve()，并传递给他环境变量数组，所以可以打印出环境变量信息。

2.5 环境变量以及 Set-UID 程序

(1)Step1:编写代码，在当前进程中打印出所有的环境变量，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
void main()
{
  int i = 0;
  while (environ[i] != NULL) {
  printf("%s\n", environ[i]);
  i++;
}
}
```

(2)Step2:编译，将权限改为 root 权限，使其成为一个 Set-UID 程序,shell 命令如下：

```
#include <stdio.h>
sudo chown root main51.o
sudo chmod 4755 main51.o
```

(3)Step3:在非 root 权限下使用 export，对 PATH、LD_LIBRARY_PATH、ANY_NAME 添加环境变量，shell 命令如下

```
[05/08/21]seed@VM:~/.../exp1$ sudo chown root main51.c
[05/08/21]seed@VM:~/.../exp1$ sudo chmod 4755 main51.c
[05/08/21]seed@VM:~/.../exp1$ export PATH=$PATH:/LZD
[05/08/21]seed@VM:~/.../exp1$ export LZD=&LZD:/555
[1] 7696
bash: LZD:/555: No such file or directory
[1]+  Done                    export LZD=
[05/08/21]seed@VM:~/.../exp1$ export LZD=$LZD:/555
[05/08/21]seed@VM:~/.../exp1$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/LZD
[05/08/21]seed@VM:~/.../exp1$
```

```
export PATH=$PATH:/LZD
export LZD=$LZD:/555
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/LZD
```

(4)shell 输出以及结论：

```
0;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
DESKTOP_SESSION=ubuntu
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:/LZD
LZD=:/555
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
XDG_SESSION_TYPE=x11
PWD=/home/seed/IS_class/exp1
JOB=unity-settings-daemon
```

发现 PATH 多了:/LZD；同时看到多了 LZD:/555；LD_LIBRARY_PATH 变量消失。

(5)分析：LD_LIBRARY_PATH 是 Linux 环境变量，环境变量主要用于指定查找共享库(动态链接库)时除了默认路径之外的其他路径，Loader 要查找共享库的时候，先判断程序是否是 Set-UID 程序，如果是，则忽略掉之前存储的 LD_LIBRARY_PATH，在全局中查找要用的函数的地址；以防止恶意用户修改 LD_LIBRARY_PATH 链接到恶意代码，导致链接到恶意代码的地方。可以看出 Set-UID 程序的进程从 shell 进程继承了 PATH 但是 LD_LIBRARY_PATH 没有被继承，因为为了保证 Set-UID 程序安全，运行时的链接器会忽略环境变量。

2.6PATH 环境变量以及 Set-UID 程序

(1)编写代码如下：

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

(2)如何是的 system()调用的不是/bin/ls,编写 ls.c，如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
extern char** environ;
int main(){
    char*argv[2];
    argv[0]="/usr/bin/whoami";
    argv[1]=NULL;
    execve("/usr/bin/whoami",argv,environ);
    return 0;
}
```

(3)执行以下 shell 命令

```
gcc main6.c -o main6.o
sudo chown root:root main6.o
sudo chmod u+s main6.o

export PATH=.:$PATH
ls -l main6.o
exit
cp /bin/ls ls
main6.o
```

export PATH=.:$PATH，点号加在 PATH 列表的最前面，所以会优先搜索当前目录，这使得找到并运行我们当前目录下的 ls,而不是/bin/ls。

(4)输出结果并分析:

```
[05/08/21]seed@VM:~/.../exp1$ export PATH=.:$PATH
[05/08/21]seed@VM:~/.../exp1$ which ls
./ls
[05/08/21]seed@VM:~/.../exp1$ main6.o
seed
[05/08/21]seed@VM:~/.../exp1$
```

ls.c 是执行一个 whoami 命令，这里 main6.o 调用的不是/bin/ls

2.7LD_PRELOAD 环境变量以及 Set-UID 程序

2.7.1Step1:

(1)编写 mylic.c

```
#include <stdio.h>
void sleep (int s)
{ /* If this is invoked by a privileged program,
   you can do damages here! */
   printf("I am not sleeping!\n");
}
```

(2)使用以下方式变为动态链接库，并设置环境变量 LD_PRLOAD

```
gcc -fPIC -g -c mylib.c
gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
export LD_PRELOAD=./libmylib.so.1.0.1
```

```
[05/08/21]seed@VM:~/.../exp1$ touch mylib.c
[05/09/21]seed@VM:~/.../exp1$ vim mylib.c
[05/09/21]seed@VM:~/.../exp1$
[05/09/21]seed@VM:~/.../exp1$  gcc -fPIC -g -c mylib.c
[05/09/21]seed@VM:~/.../exp1$  gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[05/09/21]seed@VM:~/.../exp1$  export LD_PRELOAD=./libmylib.so.1.0.1
[05/09/21]seed@VM:~/.../exp1$
```

(3)编写 myprog.c,并编译

```
#include<sdio.h>
#include<stdlib.h>
int main()
{
sleep(1);
return 0;
}
```

2.7.2Step2:

(1)run it as a normal user:

```
[05/09/21]seed@VM:~/.../exp1$ ls
a.out       exp1.txt           main3.cpp  main51.0  main6.c   mylib.c  myprog.c
exp1_1.txt  libmylib.so.1.0.1  main4.cpp  main51.c  main6.o   mylib.o  test.txt
exp1_2.txt  ls.c               main4.o    main51.o  main.cpp  myprog
[05/09/21]seed@VM:~/.../exp1$ myprog
I am not sleeping!
[05/09/21]seed@VM:~/.../exp1$
```

有输出。

(2)Make myprog a Set-UID root program, and run it as a normal user.

```
sudo chown root myprog
sudo chmod 4755 myprog
```

```
[05/09/21]seed@VM:~/.../exp1$ sudo chown root myprog
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 4755 myprog
[05/09/21]seed@VM:~/.../exp1$ myprog
[05/09/21]seed@VM:~/.../exp1$
```

没有输出。

(3)Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.

Shell:

```
su
gcc myprog.c -o myprog
chown root myprog
chmod 4755 myprog
export LD_PRELOAD=./libmylib.so.1.0.1
myprog
exit
```

```
[05/09/21]seed@VM:~/.../exp1$ su
Password:
root@VM:/home/seed/IS_class/exp1# gcc myprog.c -o myprog
myprog.c: In function ‘main’:
myprog.c:7:1: warning: implicit declaration of function ‘sleep’ [-Wimplicit-fun
ction-declaration]
 sleep(1);
 ^
root@VM:/home/seed/IS_class/exp1# chown root myprog
root@VM:/home/seed/IS_class/exp1# chmod 4755 myprog
root@VM:/home/seed/IS_class/exp1# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/IS_class/exp1# myprog
I am not sleeping!
root@VM:/home/seed/IS_class/exp1# exit
exit
[05/09/21]seed@VM:~/.../exp1$
```

有输出。

(4)Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account),export the LD PRELOAD environment variable again in a different user’s account (not-root user) andrun it.

先在 seed 中编译 myprog，再在用户二中设置环境变量，在用户二中执行。

Shell：

```
su
useradd LZD
passwd LZD
exit
gcc myprog.c -o myprog
sudo chown root myprog
sudo chmod 4755 myprog
su LZD
export LD_PRELOAD=./libmylib.so.1.0.1
myprog
```

```
root@VM:/home/seed/IS_class/exp1# passwd LZD
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@VM:/home/seed/IS_class/exp1# exit
exit
[05/09/21]seed@VM:~/.../exp1$ gcc myprog.c -o myprog
myprog.c: In function ‘main’:
myprog.c:7:1: warning: implicit declaration of function ‘sleep’ [-Wimplici
ction-declaration]
 sleep(1);
 ^
[05/09/21]seed@VM:~/.../exp1$ chown root myprog
chown: changing ownership of 'myprog': Operation not permitted
[05/09/21]seed@VM:~/.../exp1$ sudo chown root myprog
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 4755 myprog
[05/09/21]seed@VM:~/.../exp1$ su LZD
Password:
LZD@VM:/home/seed/IS_class/exp1$ export LD_PRELOAD=./libmylib.so.1.0.1
LZD@VM:/home/seed/IS_class/exp1$ myprog
LZD@VM:/home/seed/IS_class/exp1$
```

没有输出。

2.7.3 分析

LD_PRELOAD 环境变量是 Unix 动态链接库中的一个环境变量，可以影响程序的运行时的链接，允许定义在程序运行时优先加载的动态链接库。此实验中，mylib.c 通过 sleep 函数，生成了 libmylib.so.1.0.1 链接库，并加到 LD_PRELOAD 环境变量，但是调用动态链接器时，如果真实用户 ID 和有效用户 ID 不一样，进程将忽略 LD_PRELOAAD 和 LD_LIBRARY_PATH 环境变量。第一次实验中真实 uid 和有效 uid 都是 seed，第三次实验中两者都是 root，所以动态链接库不会忽略，从而能实现 sleep 函数，打印出"I am not sleeping"。但是第二次以及第四次的真实 uid 和有效 uid 时不一样的，导致忽略导入的 LD_PRELOAD 环境变量，从而不能输出结果。

2.8 使用 system()或者 execve()唤起外部程序

(1)编写代码如下：

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  char *v[3];
  char *command;
  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
    }
  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;
  command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
  sprintf(command, "%s %s", v[0], v[1]);
  // Use only one of the followings.
  system(command);
  // execve(v[0], v, NULL);
  return 0 ;
}
```

(2)Step1:自己建立文件，设置属性，将程序设置为 Set-UID 程序，shell 如下

```
touch test
sudo chmod 000 test
vim test
gcc main8.c -o main8
sudo chown root main8
sudo chmod 4755 main8
main8 "test;rm test -rf"
```

```
[05/09/21]seed@VM:~/.../exp1$ sudo vim test
[05/09/21]seed@VM:~/.../exp1$ sudo cat test
88888888888888888888
[05/09/21]seed@VM:~/.../exp1$ gcc main8.c -o main8
[05/09/21]seed@VM:~/.../exp1$ sudo chown root main8
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 4755 main8
[05/09/21]seed@VM:~/.../exp1$ main8 "test;rm test -rf"
88888888888888888888
[05/09/21]seed@VM:~/.../exp1$ ls
a.out       libmylib.so.1.0.1  main4.o    main6.c  main.cpp  myprog.c
exp1_1.txt  ls.c               main51.0   main6.o  mylib.c   test.txt
exp1_2.txt  main3.cpp          main51.c   main8    mylib.o   tets
exp1.txt    main4.cpp          main51.o   main8.c  myprog
[05/09/21]seed@VM:~/.../exp1$
```

程序正常输出 test 内容，同时将其删除。

(3)Step2:

注释 system(command)，执行 execve()，重复上述流程:

```
[05/09/21]seed@VM:~/.../exp1$ touch test
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 000 test
[05/09/21]seed@VM:~/.../exp1$ sudo vim test
[05/09/21]seed@VM:~/.../exp1$ sudo cat test
88888888888888888888
[05/09/21]seed@VM:~/.../exp1$ gcc main8.c -o main8
main8.c: In function ‘main’:
main8.c:17:3: warning: implicit declaration of function ‘execve’ [-Wimplicit-fu
ion-declaration]
   execve(v[0], v, NULL);
   ^
[05/09/21]seed@VM:~/.../exp1$ sudo chown root main8
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 4755 main8
[05/09/21]seed@VM:~/.../exp1$ main8 "test;rm test -rf"
/bin/cat: 'test;rm test -rf': No such file or directory
[05/09/21]seed@VM:~/.../exp1$
```

被拒绝，说明 execve(v[0],v,NULL)无法删除指定文件

(4)分析:

system()和 execve()都可以执行新的程序，system()可以正确解析 PATH，获得第一个参数"/bin/cat"绝对路径, 从而进行调用, 将输出作为/bin/cat 的参数, 组成:/bin/cat test;rm test -rf; 相当于执行两条指令，所以文件被删除掉。

而 execve()函数不执行 shell 程序,而是直接请求操作系统来执行该命令，所以它会把“/bin/cat test;rm test -rf”当做它传给系统的一个参数而不是指令，所以系统会认为我们要查看的文件时“/bin/cat test;rm test -rf”，而不是当作指令，所以会提示没有该文件，

2.9Capability 泄漏

一些特权程序会在其执行过程中进行降权（downgrade），变成普通程序，即非特权程序，但是若在降权之前没有释放一些变量，例如文件描述符（file descripter），则存在 Capalibility Leaking 漏洞。

(1) 编写代码如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
void main(){
    int fd;
  /* Assume that /etc/zzz is an important system file,
  * and it is owned by root with permission 0644.
  * Before running this program, you should creat
  * the file /etc/zzz first. */
  fd = open("/etc/zzz", O_RDWR | O_APPEND);
  if (fd == -1) {
    printf("Cannot open /etc/zzz\n");
    exit(0);
  }/* Simulate the tasks conducted by the program */
  sleep(1);
  /* After the task, the root privileges are no longer needed,
  it's time to relinquish the root privileges permanently. */
  setuid(getuid()); /* getuid() returns the real uid */
  if (fork()) { /* In the parent process */
    close (fd);
    exit(0);
  }
  else { /* in the child process */ /* Now, assume that the child process
is compromised, malicious attackers have injected the following
statements into this process */
    write (fd, "Malicious Data\n", 15);
    close (fd);
  }
}
```

(2) 自己新建/etc/zzz 文件, 并自定义输入, 编译上述程序, 并且使它成为 root 拥有的 Set-UID 程序，在 normal 用户下运行，shell 命令如下

```
sudo touch /etc/zzz
sudo vim /etc/zzz
gcc main9.c -o main9
sudo chown root main9
sudo chmod 4755 main9
main9
```

```
[05/09/21]seed@VM:~/.../exp1$ sudo chown root main9
[05/09/21]seed@VM:~/.../exp1$ sudo chmod 4755 main9
[05/09/21]seed@VM:~/.../exp1$ main9
[05/09/21]seed@VM:~/.../exp1$ sudo cat /etc/zzz
99999999999999999
Malicious Data
[05/09/21]seed@VM:~/.../exp1$
```

查看/etc/zzz 发现已经被修改，在文件尾部，添加了 Malicious Data。

(3)分析:

在 fork()调用部分，创建子进程，在子进程中，id 返回为 0，所以执行 else 部分。在 else 部分中，通过文件描述符向文件 zzz 写入 Malicious Data，修改了 zzz 文件，关闭文件；在父进程中，返回的时子进程 id>0，所以执行的 if 部分，关闭文件。不确定关闭文件先后顺序，但是在子进程文件 zzz 关闭前，文件描述符 fd 依然有效，此时仍然可以修改 zzz 文件。

三、 Analysis and Conclusion 实验分析与结论

每一个部分的 task 实验结论与分析见二实验过程。

参考:
https://blog.csdn.net/xiaoyujiale/article/details/112425103
https://www.bilibili.com/video/BV1tZ4y1u7pN/?spm_id_from=333.788.recommend_more_video.-1