

# 实验一

Linux 系统文件和目录权限设置与辨识 setuid 程序 uid 差异

姓名：齐帅

班级：1303202

学号：1130320201

# 一、设计并实现不同用户对不同类文件的 r、w、x 权限

## 1.查看系统文件的权限设置

a)查看/etc/passwd 文件和/usr/bin/passwd 文件的权限设置，并分析其权限为什么这么设置

(1)使用 ls -l 命令列出/etc/passwd 文件的权限信息

```
-rw-r--r-- 1 root root 1858 Apr 15 09:33 passwd
```

我们可以发现，文件所有者的权限为可读、可写，文件所有者同组的权限为可读，其他用户的权限为可读。

因为该文件是用来管理用户信息的，所以文件拥有者 root 需要在一个用户注册之后修改这个文件，因此 root 需要 w 权限。而对于同组用户与其它用户，我们在登陆的时候需要读取这个文件属于自己的一行，以此来登录并获取用户信息，因此 r 权限是必须的。但是对于同组用户与其它用户，如果其拥有 w 权限，那么就拥有了恶意篡改其他用户信息的机会，就会影响系统的安全性。

(2)使用 ls -l 命令列出/usr/bin/passwd 文件的权限信息

```
qs@ubuntu:/usr/bin$ ls -l passwd
-rwsr-xr-x 1 root root 47032 Jan 26 16:50 passwd
```

我们可以发现，文件所有者的权限为可读、可写、并且设置了 s 位让其他用户以 root 身份来运行，文件所有者同组的权限为可读、可运行，其他用户的权限为可读、可运行。

/usr/bin/passwd 文件是一个可以为用户添加、更改密码的命令。为了让每个用户都可以运行这个命令，所以设置了 s 位来提供 root 权限。但是如果需要这个命令真正生效，那么对于同组用户与其它用户，都必须拥有 x 权限才能真正使 s 位生效。

b)找到 2 个设置了 setuid 位的可执行程序，该程序的功能，该程序如果不设置 setuid 位是否能够达到相应的功能

(1) Sudo

```
-rwsr-xr-x 1 root root 155008 Aug 27 2015 sudo
```

Sudo 程序是用来登录 root 用户的程序，如果不设置 s 位的话，那么普通用户就不能运行这个程序，也就不能登录 root 用户了。

```
NAME
  sudo, sudoedit - execute a command as another user

SYNOPSIS
  sudo -h | -K | -k | -V
  sudo -v [-AknS] [-g group] [-h host] [-p prompt] [-u user]
  sudo -l [-AknS] [-g group] [-h host] [-p prompt] [-U user] [-u user] [command]
  sudo [-ABEHnPS] [-C num] [-g group] [-h host] [-p prompt] [-r role] [-t type] [-u user] [VAR=value] [-i | -s]
    [command]
  sudoedit [-AknS] [-C num] [-g group] [-h host] [-p prompt] [-u user] file ...

DESCRIPTION
  sudo allows a permitted user to execute a command as the superuser or another user, as specified by the security
  policy.
```

## (2) Pppd

```
-rwsr-xr-- 1 root dip 347296 Apr 21 2015 pppd
```

Pppd 程序是用来实现点对点协议的程序，由于在进行网络编程的时候，普通用户也有建立点对点链接的需求，所以如果不设置 s 位的话，那么普通用户就没有权限建立点对点链接了。

```
NAME
    pppd - Point-to-Point Protocol Daemon

SYNOPSIS
    pppd [ options ]

DESCRIPTION
    PPP is the protocol used for establishing internet links over dial-up modems, DSL connections, and many other types of point-to-point links. The pppd daemon works together with the kernel PPP driver to establish and maintain a PPP link with another system (called the peer) and to negotiate Internet Protocol (IP) addresses for each end of the link. Pppd can also authenticate the peer and/or supply authentication information to the peer. PPP can be used with other network protocols besides IP, but such use is becoming increasingly rare.
```

## 2.设置文件或目录权限

a)用户 A 具有文本文件“流星雨.txt”，该用户允许别人下载

```
qs@ubuntu:~/Public$ chmod 744 流星雨.txt
```

```
-rwxr--r-- 1 qs qs 12 May 13 05:32 流星雨.txt
```

使用 chmod 744 流星雨.txt 来让同组用户与其它用户拥有读权限。  
只要其它用户拥有了读权限，那么其就有能力进行下载操作。

b)用户 A 编译了一个可执行文件“cal.exe”，该用户想在系统启动时运行

如果想让一个 exe 程序在开机时自启动，如果我们已经写好了运行脚本，那么在权限方面，所以对于任何用户都要有执行权限才能正常运行，因此我们使用脚本：

```
qs@ubuntu:~/Public$ chmod a+x cal.exe
```

这样就可以让任何用户都拥有了执行权限。

结果如下：

```
-rwxrwxr-x 1 qs qs 8551 May 14 03:45 cal.exe
```

运行结果如下：

```
cal.exe BEGIN
RUNNING.....
cal.exe PAUSE
cal.exe END.....
```

程序设计如下：

```
qs@ubuntu:~/Public$ cat /etc/rc.local
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.
/home/qs/Public/cal.exe
exit 0
```

系统启动时结果如下:

```
* Stopping restore sound card state
* Stopping save kernel messages
* Restoring resolver state...
* Stopping Restore Sound Card State
* Starting cups-browsed - Bonjour remote printer browsing daemon
* Starting automatic crash report generation
cal.exe BEGIN
RUNNING.....
cal.exe PAUSE
cal.exe END.....
```

机内部单击或按 Ctrl+G。

11303020201\_齐... Ubuntu 64 位 - ... linux添加开机自...

c)用户 A 有起草了文件"demo.txt", 想让同组的用户帮其修改文件

由于同组用户需要修改文件, 那么就必须有 w 权限, 所以我们使用

```
qs@ubuntu:~/Public$ chmod g+w demo.txt
```

让同组的人拥有 w 权限。

结果如下:

```
-rw-rw-r-- 1 qs qs 6 May 13 05:46 demo.txt
```

d)一个 root 用户拥有的网络服务程序"netmonitor.exe", 需要设置 setuid 位才能完成其功能。

我们使用

```
qs@ubuntu:~/Public$ sudo chmod u+s netmonitor.exe
```

使其他用户可以以 root 身份来执行该程序。

结果如下:

```
-rwsr--r-- 1 root qs 8551 May 14 03:53 netmonitor.exe
```

## 二、一些可执行程序运行时需要系统管理员权限, 在 UNIX 中可以利用 setuid 位实现其功能

1.设想一种场景, 比如提供 http 网络服务, 需要设置 setuid 位, 并为该场景编制相应的代码

我们假设 http\_server.exe 是一个需要 HTTP 网络服务的程序, 所以我们编制以下代码设置其 setuid 位。

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void show_ids(void)
{
    printf ("real uid: %d\n", getuid());
    printf ("effective uid: %d\n", geteuid());
}

int main(int argc, char *argv[])
{
    int uid;
    if (setuid (getuid()) < 0) //假设将用户切换到http_server.exe的拥有者
        perror ("setuid error");
    show_ids();
    system("echo chmod u+s http_server.exe");
    system("chmod u+s http_server.exe"); //执行chmod u+s, 设置其setuid位
    system("ls -l http_server.exe"); //显示一下http_server.exe的权限位
    system("./http_server.exe"); //执行http_server.exe
    return (0);
}

```

程序运行结果如下：

```

real uid: 1000
effective uid: 1000
chmod u+s http_server.exe
-rwsrwxr-x 1 qs qs 8551 May 14 04:27 http_server.exe
Using HTTP-1.0 Server
Creating Socket on PORT 80
BEGIN
RUNNING.....
CLOSE Socket ID=1001
EXIT

```

在结果的第四行，我们可以看到 http\_server.exe 的 setuid 位被设置。

## 2.如果用户 fork 进程后，父进程和子进程中 euid、ruid、suid 的差别

(1)我们使用 getresuid(ruid, euid, suid)获得 ruid, euid, suid, 因此我们可以写如下脚本。

(2)我们根据 fork()函数的返回值的差异来判断当前处于父进程还是子进程。

(3)当 fork()>0 时我们处于父进程，因此我们可以打印出父进程的 ruid, euid, suid

(4)当 fork()==0 时我们处于子进程，因此我们可以打印出子进程的 ruid, euid, suid



```

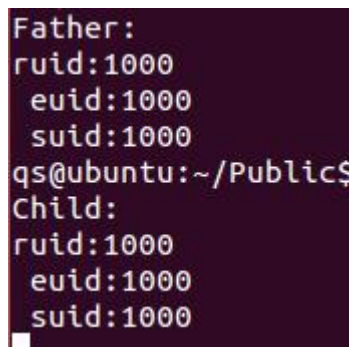
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

void show_ids(int *ruid,int *euid,int *suid)
{
    getresuid(ruid , euid, suid);           //获得ruid euid suid
    printf("ruid:%d\n euid:%d\n suid:%d\n", *ruid , *euid, *suid);
}

int main(int argc, char *argv[])
{
    int ruid,euid,suid;
    if (fork()==0)
    {
        printf("\n") ;
        printf("Child:\n") ;
        show_ids(&ruid , &euid, &suid);
    }
    else
    {
        printf("\n") ;
        printf("Father:\n") ;
        show_ids(&ruid , &euid, &suid);
    }
    return (0);
}

```

程序运行结果为:



```

Father:
ruid:1000
euid:1000
suid:1000
qs@ubuntu:~/Public$
Child:
ruid:1000
euid:1000
suid:1000

```

结果显示: 父进程和子进程的 ruid,euid 和 suid 都是分别相等。

### 3.利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变化

(1)我们首先创建一个 root 为拥有者的程序 execl\_root.exe

```

void show_ids(int *ruid,int *euid,int *suid)
{
    getresuid(ruid , euid, suid);           //获得ruid euid suid
    printf("ruid:%d\neuid:%d\nsuid:%d\n", *ruid , *euid, *suid);
}

int main(int argc, char *argv[])
{
    int ruid,euid,suid;
    printf("After:\n") ;
    show_ids(&ruid , &euid, &suid);
    return (0);
}

```

(2)我们将 execl\_root.exe 设置 s 位

```

-rwsr-xr-x 1 root root 8689 May 14 07:52 execl_root.exe

```

(3)之后我们在普通用户下执行程序，我们这里开了一个子进程，因为不开子进程的话 execl 程序会覆盖我们原本的程序，执行之后我们再次查看 euid、ruid、suid。

程序如下：

```

void show_ids(int *ruid,int *euid,int *suid)
{
    getresuid(ruid , euid, suid);           //获得ruid euid suid
    printf("ruid:%d\neuid:%d\nsuid:%d\n", *ruid , *euid, *suid);
}

int main(int argc, char *argv[])
{
    int ruid,euid,suid;
    int pid;
    if ((pid=fork())==0)
    {
        execl("/home/qs/Public/execl_root.exe", "./execl_root.exe", NULL);
    }
    else if (pid>0)
    {
        printf("Before:\n");
        show_ids(&ruid , &euid, &suid);
    }
    return (0);
}

```

运行结果：

```

qs@ubuntu:~/Public$ ./execl1
Before:
ruid:1000
euid:1000
suid:1000
qs@ubuntu:~/Public$ After:
ruid:1000
euid:0
suid:0

```

从结果我们可以得出运行 `setuid` 后 `euid` 不变，而 `euid` 和 `suid` 变为了 `root`。原因是因为文件的拥有者为 `root`。

#### 4. 程序何时需要临时性放弃 `root` 权限，何时需要永久性放弃 `root` 权限，并在程序中分别实现两种放弃权限方法

如果我们在之后的操作还有可能需要 `ROOT` 权限的时候，我们就暂时放弃 `ROOT` 权限。如果我们之后不需要 `ROOT` 权限了，那么我们就永久放弃 `ROOT` 权限。

我们首先建立一个属于 `root` 的可执行程序，并使用 `chmod 700` 限制除 `root` 用户之外的用户无法执行。

```

-rwx----- 1 root root 8557 May 14 05:39 root_only.exe

```

(1) 随后我们先使用 `root` 身份执行 `root_only.exe`，发现可以成功运行。

(2) 之后我们使用 `seteuid(1000)` 临时放弃 `root` 权限，再次执行 `root_only.exe` 发现执行无权限。

(3) 之后我们使用 `setuid(0)` 重新获取 `root` 权限，再次执行 `root_only.exe` 发现成功运行。

(4) 之后我们使用 `setuid(1000)` 完全放弃 `root` 权限，再次执行 `root_only.exe` 发现执行无权限。

(5) 最后我们使用 `setuid(0)` 重新获取 `root` 权限，发现无法获取失败。

程序设计如下：

```

int main(int argc, char *argv[])
{
    printf("<----->\n");
    printf("以root身份运行root_only.exe\n");
    system("./root_only.exe"); //首先以root身份运行root_only.exe
    printf("<----->\n");
    seteuid(1000); //临时放弃root权限
    printf("临时放弃root权限\n");
    system("./root_only.exe"); //以普通身份运行root_only.exe
    printf("<----->\n");
    setuid(0); //重新获取root权限
    printf("重新获取root权限\n");
    system("./root_only.exe"); //以root身份运行root_only.exe
    printf("<----->\n");
    printf("永久放弃root权限\n");
    setuid(1000); //永久放弃root权限
    system("./root_only.exe");
    printf("<----->\n");
    if (setuid(0)<0) //重新获取root权限失败
        printf("重新获取root权限失败\n");
    return (0);
}

```



运行结果:

```
qs@ubuntu:~/Public$ sudo ./run_root
<----->
ROOT_SERVER BEGIN....
RUNNING.....
EXIT
<----->
临时放弃root权限
sh: 1: ./root_only.exe: Permission denied
<----->
重新获取root权限
ROOT_SERVER BEGIN....
RUNNING.....
EXIT
<----->
永久放弃root权限
sh: 1: ./root_only.exe: Permission denied
<----->
重新获取root权限失败
```

我们找到 sniffer.c 嗅探程序的源代码, 之前我认为 sniffer 程序只能在 root 权限下运行, 但是通过分析源代码, 其实只是有些地方需要 root 权限, 更改过的代码如下:

```
int main(int argc, char *argv[])
{
    int tcp_ck=0,udp_ck=0,icmp_ck=0,prot_ck=0,check=1;
    char src_ip[40],dst_ip[40];
    char src_port[10],dst_port[10];
    int port1,port2;
    int src_i=0,dst_i=0,src_p=0,dst_p=0;
    char ch;
    seteuid(1000); //临时放弃root权限
    printf("临时放弃root权限\n");
    printf("%s\n","[We get those packets:]");
    while ((ch = getopt(argc,argv,"tuia:b:c:d:"))!=-1)
    {
        switch(ch)
        {
            case 't':
                prot_ck=1;
                tcp_ck++;
            case 'u':
                prot_ck=1;
                udp_ck++;
            case 'i':
                prot_ck=1;
                icmp_ck++;
            case 'a':
                prot_ck=1;
                src_i++;
            case 'b':
                prot_ck=1;
                dst_i++;
            case 'c':
                prot_ck=1;
                src_p++;
            case 'd':
                prot_ck=1;
                dst_p++;
        }
    }
}
```

```

}
setuid(0); //重新获得root权限
printf("重新获得root权限\n");
int s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP));
if (s != -1) {
    printf("Create a raw socket with fd: %d\n", s);
} else {
    fprintf(stderr, "Fail to create a raw socket: %s\n",
            strerror(errno));
    exit(EXIT_FAILURE);
}
setuid(1000); //永久放弃ROOT权限
printf("永久放弃ROOT权限\n");
while (1) {
    int size;
    char packet[IP_MAX_SIZE];

```

运行结果如下：

```

qs@ubuntu:~/Public$ sudo ./s.run -t
临时放弃root权限
[We get those packets:]
the TCP packets
重新获得root权限
Create a raw socket with fd: 3
永久放弃ROOT权限
 70 (46)  192 (c0)    0 ( 0)   32 (20)
  0 ( 0)    0 ( 0)   64 (40)    0 ( 0)
  1 ( 1)    2 ( 2)   66 (42)  109 (6d)
192 (c0)  168 (a8)    1 ( 1)    1 ( 1)
224 (e0)    0 ( 0)    0 ( 0)    1 ( 1)
148 (94)    4 ( 4)    0 ( 0)    0 ( 0)
 17 (11)  100 (64)  238 (ee)  155 (9b)
  0 ( 0)    0 ( 0)    0 ( 0)    0 ( 0)

```

其实程序在创建 RAW\_SOCKET 之后就不需要 ROOT 权限了，所以我们在放弃 ROOT 权限之后执行之后的程序，可以增加程序的安全性。

5. **execl** 函数族中有多个函数，比较有环境变量和无环境变量的函数使用的差异。

exec 家族一共有六个函数，分别是：

- (1) `int execl(const char *path, const char *arg, .....);`
- (2) `int execle(const char *path, const char *arg, ..... , char * const envp[]);`
- (3) `int execlv(const char *path, char *const argv[]);`
- (4) `int execlve(const char *filename, char *const argv[], char *const envp[]);`
- (5) `int execlvp(const char *file, char *const argv[]);`
- (6) `int execlp(const char *file, const char *arg, .....);`

其中以 p 结尾的函数，可以向函数传递一个指向环境字符串指针数组的指针。即自个定义各个环境变量，而其它四个则使用进程中的环境变量。

例如，`execlp,execvp`，表示第一个参数 `path` 不用输入完整路径，只有给出命令名即可，它会在环境变量 `PATH` 当中查找命令。

于是我们设计以下程序：

(1)使用 `execl` 函数，不指定路径

程序如下：

```
main()
{
    printf("使用execl函数，不指定路径\n");
    execl("ls","ls","-l",NULL);|
    //printf("使用execlp函数，不指定路径");
    //execlp("ls","ls","-l",NULL);
    //printf("使用execl函数，指定路径\n");
    //execl("/bin/ls","ls","-l",NULL);
}
```

结果如下：

```
qs@ubuntu:~/Public$ sudo ./execlp
使用execl函数，不指定路径
No such file or directory
```

我们发现对于 `ls` 指令，`execl` 函数无法发现路径。

(2)使用 `execlp` 函数，不指定路径

程序如下：

```
main()
{
    //printf("使用execl函数，不指定路径\n");
    //execl("ls","ls","-l",NULL);
    printf("使用execlp函数，不指定路径\n");
    execlp("ls","ls","-l",NULL);
    //printf("使用execl函数，指定路径\n");
    //execl("/bin/ls","ls","-l",NULL);
}
```

结果如下：



```

qs@ubuntu:~/Public$ sudo ./execlp
使用execlp函数，不指定路径
total 168
-rw-rw-r-- 1 qs qs 279 May 14 04:26 cal.c
-rw-rw-r-- 1 qs qs 219 May 14 03:53 cal.c~
-rwxrwxr-x 1 qs qs 8551 May 14 03:45 cal.exe
-rw-rw-r-- 1 qs qs 6 May 13 05:46 demo.txt
-rw-rw-r-- 1 qs qs 0 May 13 05:45 demo.txt~
-rwxrwxr-x 1 qs qs 8744 May 14 05:17 execl1
-rw-rw-r-- 1 qs qs 548 May 14 05:17 execl1.c
-rw-rw-r-- 1 qs qs 537 May 14 05:07 execl1.c~
-rwxrwxr-x 1 qs qs 8604 May 14 06:36 execlp
-rw-rw-r-- 1 qs qs 467 May 14 06:36 execlp.c
-rw-rw-r-- 1 qs qs 465 May 14 06:35 execlp.c~
-rw-rw-r-- 1 qs qs 619 May 14 05:09 fork.c
-rw-rw-r-- 1 qs qs 573 May 14 04:56 fork.c~
-rwxrwxr-x 1 qs qs 8793 May 14 05:09 fork.exe
-rwxrwxr-x 1 qs qs 8846 May 14 04:29 http
-rw-rw-r-- 1 qs qs 719 May 14 04:42 http.c
-rw-rw-r-- 1 qs qs 561 May 14 04:24 http.c~

```

我们可以发现，对于 execlp 函数，可以使用系统环境变量找到 ls 文件。

(3)使用 execl 函数，指定路径

程序如下：

```

main()
{
    //printf("使用execl函数，不指定路径\n");
    //execl("ls","ls","-l",NULL);
    //printf("使用execlp函数，不指定路径\n");
    //execlp("ls","ls","-l",NULL);
    printf("使用execl函数，指定路径\n");
    execl("/bin/ls","ls","-l",NULL);
}

```

结果如下：

```

qs@ubuntu:~/Public$ sudo ./execlp
使用execl函数，指定路径
total 168
-rw-rw-r-- 1 qs qs 279 May 14 04:26 cal.c
-rw-rw-r-- 1 qs qs 219 May 14 03:53 cal.c~
-rwxrwxr-x 1 qs qs 8551 May 14 03:45 cal.exe
-rw-rw-r-- 1 qs qs 6 May 13 05:46 demo.txt
-rw-rw-r-- 1 qs qs 0 May 13 05:45 demo.txt~
-rwxrwxr-x 1 qs qs 8744 May 14 05:17 execl1
-rw-rw-r-- 1 qs qs 548 May 14 05:17 execl1.c
-rw-rw-r-- 1 qs qs 537 May 14 05:07 execl1.c~
-rwxrwxr-x 1 qs qs 8603 May 14 06:37 execlp
-rw-rw-r-- 1 qs qs 467 May 14 06:37 execlp.c
-rw-rw-r-- 1 qs qs 467 May 14 06:36 execlp.c~
-rw-rw-r-- 1 qs qs 619 May 14 05:09 fork.c
-rw-rw-r-- 1 qs qs 573 May 14 04:56 fork.c~
-rwxrwxr-x 1 qs qs 8793 May 14 05:09 fork.exe
-rwxrwxr-x 1 qs qs 8846 May 14 04:29 http
-rw-rw-r-- 1 qs qs 719 May 14 04:42 http.c
-rw-rw-r-- 1 qs qs 561 May 14 04:24 http.c~
-rwsrwxr-x 1 qs qs 8551 May 14 04:27 http_server.e
-rwsrwxr-x 1 qs qs 8551 May 14 03:53 netmonitor.ex

```

通过结果我们发现加入绝对路径的 `execl` 函数与 `execlp` 函数的结果相同。

### 三、心得体会

#### 1.目录与文件权限

Linux 将权限分为了三部分：拥有者、同组、其它。这样做一方面保证文件拥有者正常操作文件，一方面方便了一个文件在组内互动的方便和安全，最重要的时保证了陌生用户正确、安全使用文件。

通过严格限定 `r,w,x` 三个权限，我们可以正确、安全、高效的对每个文件进行使用。

#### 2.Setuid 位的使用

Setuid 位很大程度上提升了 Linux 权限系统的灵活性。一方面，文件拥有者可以讲权限下放，更加方便与其他用户共同协作。另一方面，也给了普通用户成为 ROOT 用户的通道，扩大了 Linux 用户开发、编制底层应用的空间，普通用户可以通过获取 ROOT 权限来运行更多底层驱动。

#### 3.euid、ruid、suid

RUID：用于在系统中标识一个用户是谁，当用户使用用户名和密码成功登录后一个 UNIX 系统后就唯一确定了他的 RUID

EUID：用于系统决定用户对系统资源的访问权限，通常情况下等于 RUID

SUID：用于对外权限的开放。跟 RUID 及 EUID 是用一个用户绑定不同，它是跟文件而不是跟用户绑定。

这三个 ID 共同决定了当前用户的权限，这一方面使系统更加安全，另一方面也为 ROOT 权限的临时获取与失去提供了方便，程序可以选择暂时放弃 ROOT 权限来进行某些操作，以保证在不安全操作下，非法用户不会获得 ROOT 权限。

#### 4.execl 函数族

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。其中只有 `execve` 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。

我们通过灵活的使用函数族中的函数可以安全、高效的完成指定命令。

#### 5.体会

本次实验我在 Linux 环境下进行了目录与文件权限、setuid 位的使用 euid、ruid、suid 观察、execl 函数族使用的实验。通过不断的编程操作，我深刻理解了 Linux 权限的魅力，为我今后在 Linux 系统下进行开发打好了基础。

我也认识到了正确切换 euid、ruid、suid 来进行 ROOT 权限的暂时放弃，在今后编写程序时我会正确使用 ROOT 权限，来保证程序的安全性。

我也学会了使用 exec 函数族，对我今后的 Linux 开发也有着很深的意义。