



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2020 年秋季学期
计算机学院大三
计算机系网络安全课程

Lab 1 实验报告

姓名	余涛
学号	1180300829
班号	1803202
电子邮件	1063695334@qq.com
手机号码	15586430583

1 实验要求

设想一种场景需要进行普通用户和 root 用户切换，设计程序实现 euid 的安全管理
配合第 3 章 完成进程中 euid 的切换，实现 root 权限临时性和永久性管理，加强程序的安全性

说明：不分组实现

搭建安全的沙盒环境，在沙盒环境中提供必须的常见工具，并提供程序验证沙盒环境的安全性

配合第 3 章 实现系统中的虚拟化限制方法，实现安全的系统加固，测试虚拟化空间的加固程度

说明：2 人一组，分组实现（分工明确，每人讲解自己的部分，并能够相互配合）

2 实验内容

2.1 Linux 系统文件权限设置与辨识 setuid 程序 uid 差异（5 分）

2.1.1 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/etc/bin/passwd 文件的权限设置，并分析其权限为什么这么设置：

分别运行 `ls -l /etc/passwd` 和 `ls -l /usr/bin/passwd` 后显示的结果中，最前面的第 2~10 个字符是用来表示权限。第一个字符一般用来区分文件和目录：

三种权限：

Read 权限：控制读文件内容

Write 权限：控制读文件内容

Execute 权限：控制将文件调入内存并执行

r=4, w=2, x=1

例如，`rw` 属性则可以表示为 $4+2+1=7$ ；`rw-` 属性则可以表示为 $4+2=6$ 。

权限分成三组，每组都是 “rwx” 格式，三组分别代表每个文件拥有者、文件拥有者所在组以及其它组。

```
yt1180300829@ubuntu:~$ sudo su
[sudo] yt1180300829 的密码:
root@ubuntu:/home/yt1180300829# ls -l /etc/passwd
-rw-r--r-- 1 root root 2648 Sep 21 2019 /etc/passwd
root@ubuntu:/home/yt1180300829# ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 63736 Mar 22 2019 /usr/bin/passwd
```

这样设置的原因为：这样写的目的是能够保证文件所有者对该文件有很高的权限，但是其他用户则没有这么多的权限，保护文件内容以达到安全的目的。

所有用户的信息都保存在/etc/passwd 中，该文件由 root 创建，每个用户对/etc/passwd 都有读权限，但是只有 root 对其有写权限，其他用户都没有写权限，保证了普通用户无法修改用户信息，保证了系统安全。/usr/bin/passwd 用于修改用户的密码，任何用户都可以调用，该文件由 root 创建，密码保存在/etc/shadow 文件中，由于/etc/shadow 文件仅仅允许 root 进行读写，所以/bin/passwd 设置密码必须以root身份执行。所以/bin/passwd 设置了 setuid 位，允许普通用户以 root 身份运行该文件。

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，

执行 ls -ls /usr/bin 命令，查询 /usr/bin 文件夹下所有文件的权限。结果如下：

```
root@ubuntu:/home/yt1180300829# ls -ls /usr/bin
总用量 200140
 60 -rwxr-xr-x 1 root root      59576 Jan 14 2019 '['
 32 -rwxr-xr-x 1 root root      30936 Apr 15 2019 aa-enabled
 32 -rwxr-xr-x 1 root root      30936 Apr 15 2019 aa-exec
 24 -rwxr-xr-x 1 root root      22600 Feb 20 2019 aconnect
 16 -rwxr-xr-x 1 root root      14608 Mar  1 2019 acpi_listen
  8 -rwxr-xr-x 1 root root       7258 Apr 11 2019 add-apt-repository
 28 -rwxr-xr-x 1 root root      26704 Feb 22 2019 addpart
  0 lrwxrwxrwx 1 root root         26 Sep 16 2019 addr2line -> x86_64-linux
-gnu-addr2line
 48 -rwxr-xr-x 1 root root      47240 Feb 20 2019 alsabat
 80 -rwxr-xr-x 1 root root      80888 Feb 20 2019 alsaloop
 72 -rwxr-xr-x 1 root root      72120 Feb 20 2019 alsamixer
 16 -rwxr-xr-x 1 root root      14408 Feb 20 2019 alsatplg
 20 -rwxr-xr-x 1 root root      18880 Feb 20 2019 alsaucm
 28 -rwxr-xr-x 1 root root      26704 Feb 20 2019 amidi
 60 -rwxr-xr-x 1 root root      59544 Feb 20 2019 amixer
```

032	-rwxr-xr-x	1	root	root	000120	Sep 3	2019	udevadm
60	-rwxr-xr-x	1	root	root	59464	Mar 4	2019	udisksctl
0	lrwxrwxrwx	1	root	root		9 Nov 27	2018	uic -> qtchooser
0	lrwxrwxrwx	1	root	root		9 Nov 27	2018	uic3 -> qtchooser
16	-rwxr-xr-x	1	root	root	14344	May 3	2018	ul
16	-rwxr-xr-x	1	root	root	14328	Mar 5	2019	unlockmgr_server
180	-rwxr-xr-x	1	root	root	183368	Apr 5	2019	umax_pp
36	-rwsr-xr-x	1	root	root	34888	Feb 22	2019	umount
40	-rwxr-xr-x	1	root	root	39128	Jan 14	2019	uname
88	-rwxr-xr-x	1	root	root	88663	Apr 15	2019	unattended-upgrade
0	lrwxrwxrwx	1	root	root		18 Sep 16	2019	unattended-upgrades -> un
attended-upgrade								
4	-rwxr-xr-x	1	root	root	2345	Jan 5	2019	uncompress
44	-rwxr-xr-x	1	root	root	43224	Jan 14	2019	unexpand
4	-rwxr-xr-x	1	root	root	2762	Nov 12	2018	unicode_start
4	-rwxr-xr-x	1	root	root	530	Nov 12	2018	unicode_stop
52	-rwxr-xr-x	1	root	root	51416	Jan 14	2019	uniq
52	-rwxr-xr-x	1	root	root	51312	Apr 16	2019	pacat
20	-rwxr-xr-x	1	root	root	18504	Apr 16	2019	pacmd
64	-rwxr-xr-x	1	root	root	63576	Apr 16	2019	pactl
4	-rwxr-xr-x	1	root	root	2259	Apr 16	2019	padsp
0	lrwxrwxrwx	1	root	root		23 Sep 16	2019	pager -> /etc/alternative
s/pager								
0	lrwxrwxrwx	1	root	root		5 Sep 16	2019	pamon -> pacat
16	-rwxr-xr-x	1	root	root	14328	Dec 11	2018	paperconf
0	lrwxrwxrwx	1	root	root		5 Sep 16	2019	paplay -> pacat
0	lrwxrwxrwx	1	root	root		5 Sep 16	2019	parec -> pacat
0	lrwxrwxrwx	1	root	root		5 Sep 16	2019	parecord -> pacat
12	-rwxr-xr-x	1	root	root	8837	Oct 29	2018	parsechangelog
108	-rwxr-xr-x	1	root	root	108624	Feb 22	2019	partx
64	-rwsr-xr-x	1	root	root	63736	Mar 22	2019	passwd
40	-rwxr-xr-x	1	root	root	39128	Jan 14	2019	paste
20	-rwxr-xr-x	1	root	root	18512	Apr 16	2019	pasuspender
188	-rwxr-xr-x	1	root	root	190840	Jul 23	2019	patch
40	-rwxr-xr-x	1	root	root	39096	Jan 14	2019	pathchk
16	-rwxr-xr-x	1	root	root	14328	Apr 16	2019	pax11publish
16	-rwxr-xr-x	1	root	root	14472	Aug 3	2018	pcimodules

从上面可以看出，上图中第三个权限位设为了s的文件就是设置了setuid位的可执行文件。我选取了两个设置setuid位的可执行文件：分别为umount、passwd。对于passwd和umount这两个文件来说，其所有者是root，但其他用户也具有对其的执行权限，并且其自身也有SUID权限。所以对于其他用户执行passwd和umount这两个可执行文件来说，产生的进程就是通过root的用户ID来说运行的。

如果不设置setuid位的话，程序执行时就不需要获取暂时的root权限，就无法执行这两个功能了。

（2）设置文件或目录权限

首先进行chmod的参数说明：

u：表示文件拥有者，即 user

g：组拥有者，即 group

o：其它用户拥有者，即 other

a: 所有用户，即相当于 ugo

: 省略不写代表 a，即所有用户

a) 用户 A 具有文本文件“流星雨.txt”，该用户允许别人下载；

只需要设置其他用户对该文本文件“流星雨.txt”的读权限便可以实现下载指令，即设置 r 权限。

首先创建一个测试的用户 test，使其与其他的用户不同组，然后设置密码：

```
root@ubuntu:/home/yt1180300829# sudo useradd test
root@ubuntu:/home/yt1180300829# sudo passwd test
新的 密码:
重新输入新的 密码:
passwd: 已成功更新密码
```

然后创建“流星雨.txt”的文本文件，权限位设置为 644，即设置 r 权限：

```
yt1180300829@ubuntu:~/Lab1_system_security$ vim 流星雨.txt
yt1180300829@ubuntu:~/Lab1_system_security$ chmod 644 流星雨.txt
yt1180300829@ubuntu:~/Lab1_system_security$ ls -l 流星雨.txt
-rw-r--r-- 1 yt1180300829 yt1180300829 40 Dec  5 01:33 流星雨.txt
```

可以看出，权限位的最后三位为 r--，说明其他的用户拥有对“流星雨.txt”文件的读权限，允许别人进行下载。

创建的“流星雨.txt”文件如下：



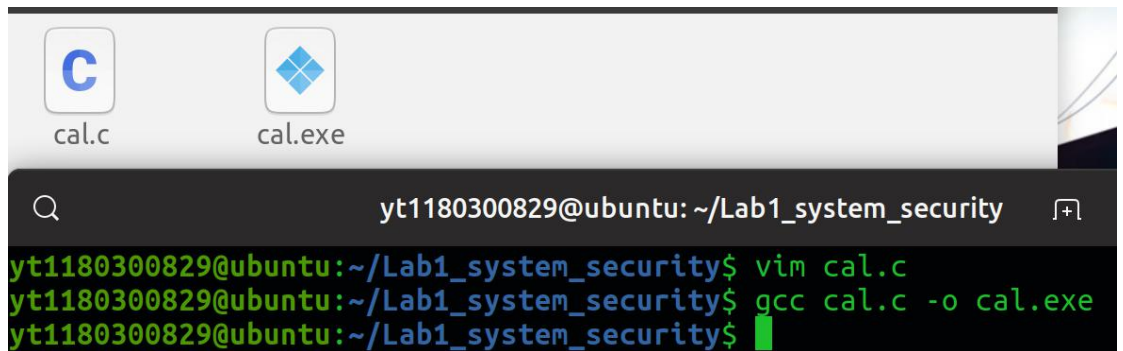
然后使用刚才设置的测试用户进行测试，切换到 test 用户读取“流星雨.txt”：

```
yt1180300829@ubuntu:~/Lab1_system_security$ su test
密码:
$ cat 流星雨.txt
用于测试允许别人下载的文件
```

完成测试。

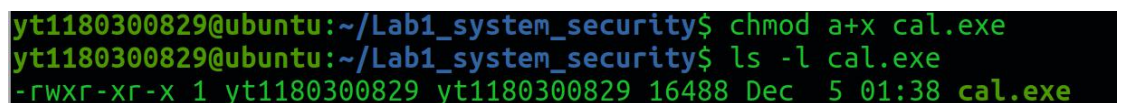
b) 用户 A 编译了一个可执行文件“cal.exe”，该用户想在系统启动时运行；

首先用户创建一个 c 文件并进行编译为 cal.exe：



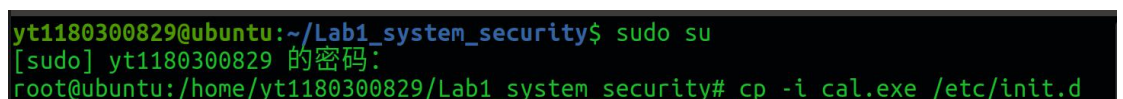
```
yt1180300829@ubuntu: ~/Lab1_system_security
yt1180300829@ubuntu:~/Lab1_system_security$ vim cal.c
yt1180300829@ubuntu:~/Lab1_system_security$ gcc cal.c -o cal.exe
yt1180300829@ubuntu:~/Lab1_system_security$
```

由于在系统刚启动时无法确定是哪个用户，所以需要给所有的用户这个文件的执行权限。

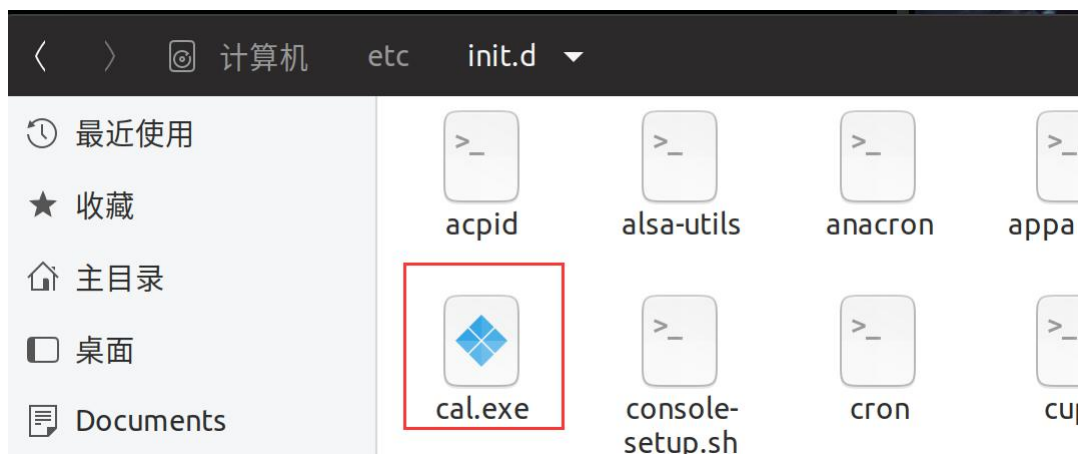


```
yt1180300829@ubuntu:~/Lab1_system_security$ chmod a+x cal.exe
yt1180300829@ubuntu:~/Lab1_system_security$ ls -l cal.exe
-rwxr-xr-x 1 yt1180300829 yt1180300829 16488 Dec  5 01:38 cal.exe
```

为了添加为系统启动时执行，此时需要 root 权限，将可执行文件“cal.exe”放到/etc/init.d 中，然后在/etc/rc3.d 中建立软链接：

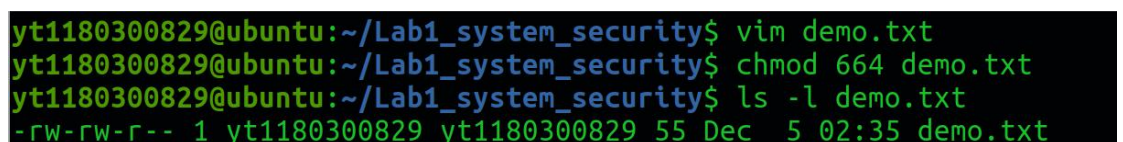


```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo su
[sudo] yt1180300829 的密码:
root@ubuntu:/home/yt1180300829/Lab1_system_security# cp -i cal.exe /etc/init.d
```



```
yt1180300829@ubuntu:/etc/init.d$ sudo su
[sudo] yt1180300829 的密码:
root@ubuntu:/etc/init.d# ln -s cal.exe /etc/rc3.d
```

- 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；
创建“demo.txt”的文本文件，文件的权限位设置为 664，即设置同组的用户拥有读写权限：



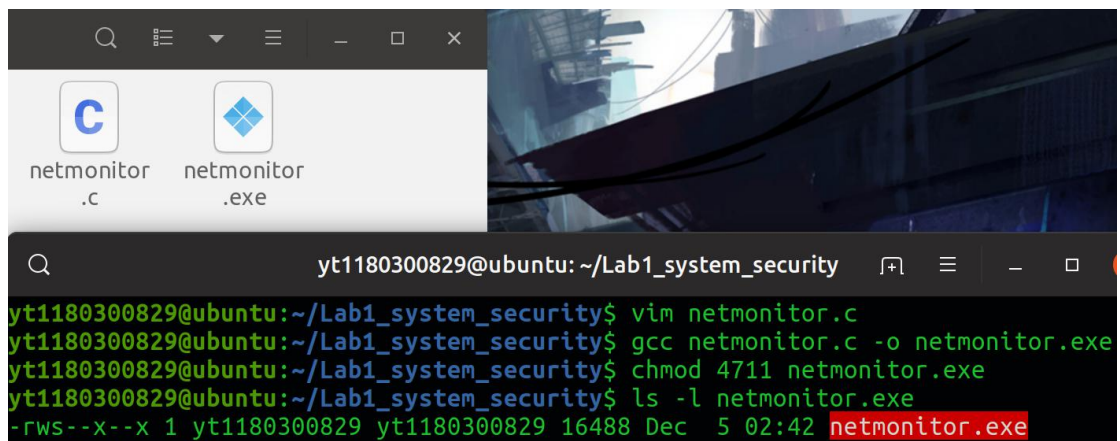
```
yt1180300829@ubuntu:~/Lab1_system_security$ vim demo.txt
yt1180300829@ubuntu:~/Lab1_system_security$ chmod 664 demo.txt
yt1180300829@ubuntu:~/Lab1_system_security$ ls -l demo.txt
-rw-rw-r-- 1 yt1180300829 yt1180300829 55 Dec  5 02:35 demo.txt
```

创建的“demo.txt”文件如下：



- 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。

创建“netmonitor.c”并编译为“netmonitor.exe”后，设置其文件的权限位设置为 4711 即可：



2.1.2 各种场景

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，但 setuid 了的程序运行过程中拥有了 root 权限，因此在完成管理操作后需要切换到普通用户的身份执行后续操作。

(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：

思路如下：

为了实现提供 http 网络服务的功能，我们需要使用 socket 编程中的 bind 函数进行测试，首先需要获取在建立 socket 的套接字之前的 ruid、euid、suid，即实际用户 ID、有效用户 ID 和保存的用户 ID。然后建立 socket 套接字并且使用 bind()函数进行绑定，紧接着判断绑定是否成功，然后检查绑定后的 ruid、euid、suid。

具体代码如下：

分别获取执行前的 ruid、euid、suid 和执行 http 服务后的 ruid、euid、suid

即可：

```
1. // 以下的三个 id 分别对应了实际用户 ID，有效用户 ID，保存的用户 ID
2. uid_t ruid, euid, suid;
3. getresuid(&ruid, &euid, &suid);
4. printf("初始 uid 为: ruid = %d, euid = %d, suid = %d\n",
5.       ruid, euid, suid);
6. // 1. 提供 http 网络服务，需要设置 setuid 位，否则会失败
7. printf("问题一：提供 http 网络服务\n");
8. int server_socket = socket(AF_INET, SOCK_STREAM, 0);
9. if (server_socket < 0)
10. {
11.     printf("erro \n");
12. }
13. // bind 绑定
14. struct sockaddr_in server_sockaddr;
15. memset(&server_sockaddr, 0, sizeof(server_sockaddr));
16. server_sockaddr.sin_family = AF_INET;
17. server_sockaddr.sin_port = htons(80);
18. server_sockaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
19. int is_bind = bind(server_socket, (struct sockaddr*)&server_sockaddr,
20.                   sizeof(server_sockaddr));
21. if (is_bind < 0)
22. {
23.     printf("无权限，绑定失败\n");
24. }
25. else
26. {
27.     printf("有权限，绑定成功\n");
28. }
29. getresuid(&ruid, &euid, &suid);
30. printf("绑定后的 uid 为: ruid = %d, euid = %d, suid = %d\n",
31.       ruid, euid, suid);
```

执行函数后的结果为：

普通用户执行调用 http 服务的结果：

```
yt1180300829@ubuntu:~/Lab1_system_security$ ./httpserver
初始 uid 为: ruid = 1000, euid = 1000, suid = 1000
问题一：提供 http 网络服务
无权限，绑定失败
绑定后的 uid 为: ruid = 1000, euid = 1000, suid = 1000
```

root 用户执行调用 http 服务的结果：


```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo ./httpserver
[sudo] yt1180300829 的密码:
初始 uid 为: ruid = 0, euid = 0, suid = 0
问题一: 提供 http 网络服务
有权限, 绑定成功
绑定后的 uid 为: ruid = 0, euid = 0, suid = 0
```

结论:

可以看出, 只有 root 用户具有执行 http 网络服务的权限, 普通用户不能执行 http 网络服务。

(2)如果用户 fork 进程后, 父进程和子进程中 euid、ruid、suid 的差别;

思路如下:

在父进程中直接执行 fork()来创建子进程, 然后分别查看 fork()前后的 ruid、euid、suid。

具体代码如下:

```
1. // 2. 用户 fork 进程后, 父进程和子进程中 euid、ruid、suid 的差别
2.     printf("问题二: 用户 fork 进程后, 父进程和子进程中 euid、ruid、suid 的差别\n");
3.     if (fork() == 0) //子进程
4.     {
5.         getresuid(&ruid, &euid, &suid);
6.         printf("子进程 uid 为: ruid = %d, euid = %d, suid = %d\n",
7.             ruid, euid, suid);
8.
9.         // 3. 利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变化
10.        printf("问题三: 利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变化\n");
11.        execl("./a", "./a", (char*)0);
12.    }
13.    else //父进程
14.    {
15.        getresuid(&ruid, &euid, &suid);
16.        printf("父进程 uid 为:
17.            ruid = %d, euid = %d, suid = %d\n",ruid, euid, suid);
```

执行函数后的结果为:

普通用户的结果:

父进程:

```
问题二: 用户fork进程后, 父进程和子进程中euid、ruid、suid的差别
父进程 uid为: ruid = 1000, euid = 1000, suid = 1000
```

子进程:

```
子进程 uid 为: ruid = 1000, euid = 1000, suid = 1000
```

root 用户的结果:

父进程:

```
问题二: 用户fork进程后, 父进程和子进程中euid、ruid、suid的差别  
父进程 uid为: ruid = 0, euid = 0, suid = 0
```

子进程:

```
子进程 uid 为: ruid = 0, euid = 0, suid = 0
```

结论:

可以看出不管对于普通用户还是 root 用户, 父进程与子进程中的 ruid、euid、suid 都不变。

(3)利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变化;

思路如下:

创建一个可执行文件, 且将其权限设置为 root 用户的, 设置可执行文件的 setuid 位即可, 然后打印 euid、ruid、suid。

具体代码如下:

```
1. // 3. 利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变化  
2.     printf("问题三: 利用 execl 执行 setuid 程序后, euid、ruid、suid 是否有变  
   化\n");  
3.     execl("./exc", "./exc", (char*)0);
```

可执行文件执行的程序代码为:

```
1. #include<stdio.h>  
2. #include<unistd.h>  
3. int main()  
4. {  
5.     printf("执行了一个可执行文件用来测试\n");  
6.     uid_t ruid, euid,suid;  
7.     getresuid(&ruid, &euid, &suid);  
8.     printf("利用 exec 执行 setuid 程序后 uid 为:  
   ruid = %d, euid = %d, suid = %d\n",ruid, euid, suid);  
9. };
```

执行函数后的结果为:

首先设置可执行文件的 setuid 位, 使之成为 root 用户的文件:

```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo su
[sudo] yt1180300829 的密码:
root@ubuntu:/home/yt1180300829/Lab1_system_security# gcc exc.c -o exc
```

```
root@ubuntu:/home/yt1180300829/Lab1_system_security# chmod 4711 exc
root@ubuntu:/home/yt1180300829/Lab1_system_security# ls -l exc
-rws--x--x 1 root root 16624 Dec  5 05:36 exc
```

然后执行函数：

普通用户执行结果为：

```
问题三：利用 execl 执行 setuid 程序后，euid、ruid、suid是否有变化
执行了一个可执行文件用来测试
利用 exec 执行 setuid 程序后 uid为：ruid = 1000, euid = 0, suid = 0
```

root 用户执行结果为：

```
问题三：利用 execl 执行 setuid 程序后，euid、ruid、suid是否有变化
执行了一个可执行文件用来测试
利用 exec 执行 setuid 程序后 uid为：ruid = 0, euid = 0, suid = 0
```

结论：

可以发现，普通用户执行了 setuid 的可执行程序后，euid 和 suid 变成了 0。

而 root 用户的 euid、ruid、suid 没有发生改变。

(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；

思路如下：

在执行完需要 root 权限才能执行的操作后，如果以后还可能需要 root 权限，就可以临时性放弃 root 权限。如果以后再也不需要 root 权限，就永久性放弃 root 权限。当需要临时性放弃 root 权限时，将当前的 euid 保存在 suid 处，并将 euid 设置为当前的 ruid 即可。当需要永久性放弃 root 权限时，将 euid 和 suid 都设置为当前的 ruid 即可。

具体代码如下：

临时性放弃 root 权限：

```
1. // 4.1 程序临时性放弃 root 权限
2. void lose_root_permission_temporary(uid_t uid_tran)
3. {
4.     uid_t ruid, euid, suid;
5.     getresuid(&ruid, &euid, &suid);
6.     if (euid == 0)
7.     {
```

```

8.      // 临时性放弃 root 权限
9.      int is_seteuid = seteuid(uid_tran);
10.     getresuid(&ruid, &euid, &suid);
11.     if (euid > 0)
12.     {
13.         printf("问题四.一: 临时性放弃 root 权限成功\n");
14.     }
15.     else
16.     {
17.         printf("问题四.一: 临时性放弃 root 权限失败\n");
18.     }
19.     printf("ruid = %d, euid = %d, suid = %d\n", ruid, euid, suid);
20. }
21. else
22. {
23.     printf("问题四.一: 无 root 权限, 无法放弃 root 权限\n");
24. }
25. }

```

永久性放弃 root 权限:

```

1.  // 4.2 永久性放弃 root 权限
2.  void lose_root_permission_permanent(uid_t uid_tran)
3.  {
4.      uid_t ruid, euid, suid;
5.      getresuid(&ruid, &euid, &suid);
6.      if (euid != 0 && (ruid == 0 || suid == 0))
7.      {
8.          setuid(0);
9.          getresuid(&ruid, &euid, &suid);
10.     }
11.     if (euid == 0)
12.     {
13.         // 永久性放弃 root 权限
14.         setresuid(uid_tran, uid_tran, uid_tran);
15.         getresuid(&ruid, &euid, &suid);
16.         if (ruid > 0 && euid > 0 && suid > 0)
17.         {
18.             printf("问题四.二: 永久性放弃 root 权限成功\n");
19.         }
20.         else
21.         {
22.             printf("问题四.二: 永久性放弃 root 权限失败\n");
23.         }

```

```

24.         printf("ruid = %d, euid = %d, suid = %d\n", ruid, euid, suid);
25.     }
26.     else
27.     {
28.         printf("问题四.二: 无 root 权限, 无法放弃 root 权限\n");
29.     }
30. }

```

主程序调用代码为:

```

1. // 4.两种放弃 root 权限的方式
2.         lose_root_permission_temporary(1001); // 临时性放弃 root 权限
3.         lose_root_permission_permanent(1001); // 永久性放弃 root 权限

```

执行函数后的结果为:

普通用户的执行结果:

```

问题四.一: 无 root 权限, 无法放弃root权限
问题四.二: 无 root 权限, 无法放弃root权限

```

root 用户执行结果为:

```

问题四.一: 临时性放弃root权限成功
ruid = 0, euid = 1001, suid = 0
问题四.二: 永久性放弃root权限成功
ruid = 1001, euid = 1001, suid = 1001

```

结论:

普通用户没有放弃 root 的权限, 而 root 用户在临时性放弃 root 权限时, 只有 ruid 和 suid 保持为 0, euid 变为其他用户的 uid, 在 root 用户永久性放弃 root 权限时, ruid、euid 和 suid 都变为其他用户的 uid。

(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。

思路如下:

对于有环境变量和无环境变量的函数使用的差异:

无环境变量时使用 execl()函数, execl()用来执行参数 path 字符串所代表的文件路径, 接下来的参数代表执行该文件时传递过去的 argv(0)、argv[1]....., 最后一个参数必须用空指针(NULL)作结束。

有环境变量时使用 execlp()函数, execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名, 找到后便执行该文件, 然后将第二个以后的参

数当做该文件的 argv[0]、argv[1]....., 最后一个参数必须用空指针(NULL)作结束。

具体代码如下:

```
1. // 5. 比较有环境变量和无环境变量的函数使用的差异。
2.     // 5.1 有环境变量的函数使用
3.     if (fork() == 0)
4.     {
5.         printf("问题五.一: 有环境变量的函数使用\n");
6.         execlp("exc", "./exc", (char*)0);
7.     }
8.     wait(NULL);
9.     if (fork() == 0)
10.    {
11.        // 5.2 无环境变量的函数使用
12.        printf("问题五.二: 无环境变量的函数使用\n");
13.        execl("./exc", "./exc", (char*)0);
14.    }
15.    wait(NULL);
```

2.2 chroot 的配置

利用 chroot 工具来虚拟化管理

2.2.1 实现 bash 或 ps 的配置使用;

首先使用 ldd /bin/bash 命令查看需要使用的动态函数库:

```
yt1180300829@ubuntu:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffed92ff000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f39aa24b000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f39aa245000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f39aa05a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f39aa3bb000)
```

然后创建 lib 对应的 lib、bin 和 lib64 文件夹:

```
yt1180300829@ubuntu:~$ sudo mkdir -p /var/chroot/lib
[sudo] yt1180300829 的密码:
yt1180300829@ubuntu:~$ mkdir -p /var/chroot/bin
```

```
yt1180300829@ubuntu:~$ sudo mkdir -p /var/chroot/lib64
```

然后使用 install 命令拷贝 bash 和对应的动态链接库：

```
yt1180300829@ubuntu:~$ sudo install -C /bin/bash /var/chroot/bin
yt1180300829@ubuntu:~$ sudo install -C /lib/x86_64-linux-gnu/libtinfo.so.6 /var/chroot/lib
yt1180300829@ubuntu:~$ sudo install -C /lib/x86_64-linux-gnu/libdl.so.2 /var/chroot/lib
yt1180300829@ubuntu:~$ sudo install -C /lib/x86_64-linux-gnu/libc.so.6 /var/chroot/lib
yt1180300829@ubuntu:~$ sudo install -C /lib64/ld-linux-x86-64.so.2 /var/chroot/lib64/ld-linux-x86-64.so.2
```

测试 bash 成功：

```
yt1180300829@ubuntu:/var/chroot$ sudo chroot ./
[sudo] yt1180300829 的密码:
bash-5.0#
```

2.2.2 利用 chroot 实现 SSH 服务或 FTP 服务的虚拟化隔离；

由于本机上没有 ftpd，所以首先需要下载 vsftpd：

```
yt1180300829@ubuntu:~$ sudo apt-get install vsftpd
[sudo] yt1180300829 的密码:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装：
  vsftpd
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 185 个软件包未被升级。
需要下载 115 kB 的归档。
解压缩后会消耗 338 kB 的额外空间。
获取:1 http://us.archive.ubuntu.com/ubuntu disco/main amd64 vsftpd amd64 3.0.3-12 [115 kB]
已下载 115 kB，耗时 6秒 (20.1 kB/s)
正在预设定软件包 ...
正在选中未选择的软件包 vsftpd。
(正在读取数据库 ... 系统当前共安装有 202262 个文件和目录。)
准备解压 .../vsftpd_3.0.3-12_amd64.deb ...
正在解压 vsftpd (3.0.3-12) ...
正在设置 vsftpd (3.0.3-12) ...
Created symlink /etc/systemd/system/multi-user.target.wants/vsftpd.service → /li
systemd/system/vsftpd.service
```

然后编辑 vsftpd.conf 的配置文件来实现 chroot 的虚拟化隔离：

```
yt1180300829@ubuntu:~$ sudo gedit /etc/vsftpd.conf
```

修改配置文件如下：

```
userlist_deny=NO
userlist_enable=YES
#允许登录的用户
userlist_file=/etc/allowed_users
seccomp_sandbox=NO
#默认的ftp下载目录
local_root=/home/ftp/
```

允许用户访问的文件目录为 ftp:

```
chroot_local_user=YES
#是否开启用户白名单
chroot_list_enable=NO
# (default follows)
#用户白名单，用户只能访问自己的主目录，不能访问其上级目录
chroot_list_file=/etc/vsftpd.chroot_list
```

然后创建用户白名单和允许登录用户的文本文件:

```
yt1180300829@ubuntu:/etc$ sudo vi /etc/vsftpd.chroot_list
```

```
yt1180300829@ubuntu:/etc$ sudo vi /etc/allowed_users
```

白名单中的用户为:



The screenshot shows a text editor window titled 'vsftpd.chroot_list [只读]' with the file path '/etc' visible. The content of the file is 'test'.

允许登录的用户为:



The screenshot shows a text editor window titled 'allowed_users [只读]' with the file path '/etc' visible. The content of the file is 'test'.

启动 vsftpd 服务:

```
yt1180300829@ubuntu:~$ service vsftpd start
```

然后在浏览器用创建的 test 用户，测试 ftp 连接:



可以发现，无论怎么返回多少层目录，都还在允许用户访问的 **ftp** 目录下，实现了虚拟化隔离：



2.2.3 chroot 后如何降低权限，利用实验一中编制的程序检查权限的合理性；

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
```

```

5. #include <sys/stat.h>
6. int main()
7. {
8.     uid_t ruid, euid, suid;
9.     getresuid(&ruid, &euid, &suid);
10.    printf("chroot 之前:\nruid=%d\neuid=%d\nsuid=%d\n", ruid, euid, suid);
11.    chdir("/var/chroot");
12.    if(chroot("/var/chroot") == 0) {
13.        printf("chroot 成功\n");
14.    } else{
15.        printf("Chroot 失败!\n");
16.        return 1;
17.    }
18.    //在 chroot 之后放弃权限
19.    setresuid(ruid, ruid, ruid);
20.    getresuid(&ruid, &euid, &suid);
21.    printf("chroot 放弃权限
后:\nruid=%d\neuid=%d\nsuid=%d\n", ruid, euid, suid);
22.    execlp("ls", "ls", (char*)0);
23.    return 0;
24. }

```

在执行代码后会用 ruid 替换 ruid、euid、suid 就能放弃 root 权限

```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo gcc reduce_right.c -o reduce_right
```

执行前先设置 setuid，此时的 ruid 应该为 1000

```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo chmod 4755 reduce_right
```

然后执行如下，最后执行了 ls 来查看可执行文件的工作目录中的文件，此时在 /var/chroot 中：

```

yt1180300829@ubuntu:~/Lab1_system_security$ ./reduce_right
chroot之前:
ruid=1000
euid=0
suid=0
chroot成功
chroot放弃权限后:
ruid=1000
euid=1000
suid=1000
bin lib lib64

```


2.2.4 在 chroot 之前没有采用 cd xx 目录，会对系统有何影响，编制程序分析其影响。

将代码中改变可执行文件工作目录的 `chdir("/var/chroot")` 去掉，然后执行代码如下：

```
yt1180300829@ubuntu:~/Lab1_system_security$ sudo gcc reduce_right_cd.c -o reduce_right_cd
yt1180300829@ubuntu:~/Lab1_system_security$ sudo chmod 4755 reduce_right_cd
```

可以发现 `ls` 来查看可执行文件的工作目录中的文件时，此时的工作目录为实际的工作目录，由于没有 `chdir` 工作目录到 `/var/chroot` 中，所以存在虽然讲根目录 `chroot` 到了 `/var/chroot` 中，但是用户仍然可以访问 `chroot` 外的目录的情况，这样就没有实现虚拟化隔离。

```
chroot之前:
ruid=1000
euid=0
suid=0
chroot成功
chroot放弃权限后:
ruid=1000
euid=1000
suid=1000
cal.c      httpserver      reduce_right.c
cal.exe    httpserver.c    reduce_right_cd
demo.txt   netmonitor.c    reduce_right_cd.c
exc        netmonitor.exe  '$'\346\265\201\346\230\237\351\233\250'.txt'
exc.c      reduce_right
```

3 心得体会

通过本次实验，让我了解了系统权限管理中三种 `uid` 的使用，学会了如何分配对不同的文件分配不同的权限。并且学会了使用 `chroot` 虚拟化隔离来运行程序以保证安全，收获颇丰。