Security Lab

Group Members Name:

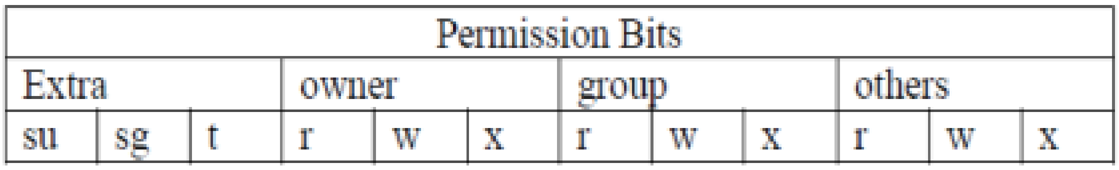
- Neha Goud Baddam - 11519516
- Purandhara Maharshi Chidurala - 11519513
- Kaushik Kalva – 11527375

UNIX: **File Hierarchy**

The Unix/Linux file system is organized as a hierarchy with the root (/) directory at the highest level. Some typical Unix system directories are usr, bin, sbin, home, var, boot, dev, etc, and others. In the figure shown below, “user1” and “user2” are the subdirectories of the directory “home”, “hello.txt” is a plain text file and “link_hello” is a link file that points to “hello.txt”. In order to access the file “/test/temp/hello.txt”, the system begins its search from the root (/) directory, then “test” and “temp” directories consecutively, and then finally it finds the file “hello.txt”.

Ownership and Permissions

Ownership of files in UNIX can be viewed in one of three ways: owner (creator), group or others (i.e., all the other users, apart from the owner and the group members). Using this simple notion of ownership, access to files can be controlled by associating a unique user ID (UID) and a group ID (GID) with twelve permission bits for each file. Typically, these bits are divided into three sets of three bits each, as well as three extra bits as shown in the table below.



Permission Bits											
Extra			owner			group			others		
su	sg	t	r	w	x	r	w	x	r	w	x

The values “r”, “w” and “x” stand for read, write and execute bits for each of the owner, group and others permissions. The values “su”, “sg” and “t” stand for set_user_id, set_group_id and the
sticky bit, respectively. These 4 sets of bits are often represented in their octal digits. For example, “100 111 101 101” is represented as “4755.” When the “su” bit is set, the UID of the process will be that of the owner of the file, no matter who executes the file. Similarly, if the “sg” bit is set, the GID of the process will be that of the group.

Lab Procedures

Use the UbuntuDesktop VM.

Username: sec-lab
Password: untccdc

NOTE: For your lab report, you need to reply questions which are marked in red.

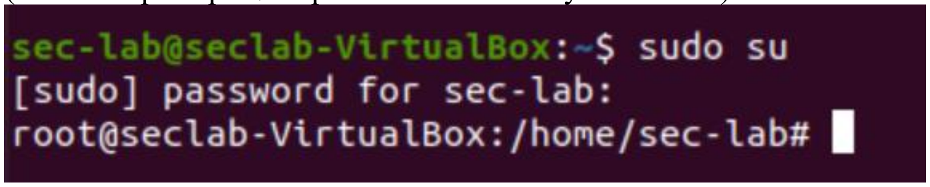
File permissions in Linux

Exercise 2.1: Setting up the file structure and the user space

The objective of this exercise is to setup the file hierarchy structure and the user accounts that are required for the exercises in this section. The **su** command is used to switch users.

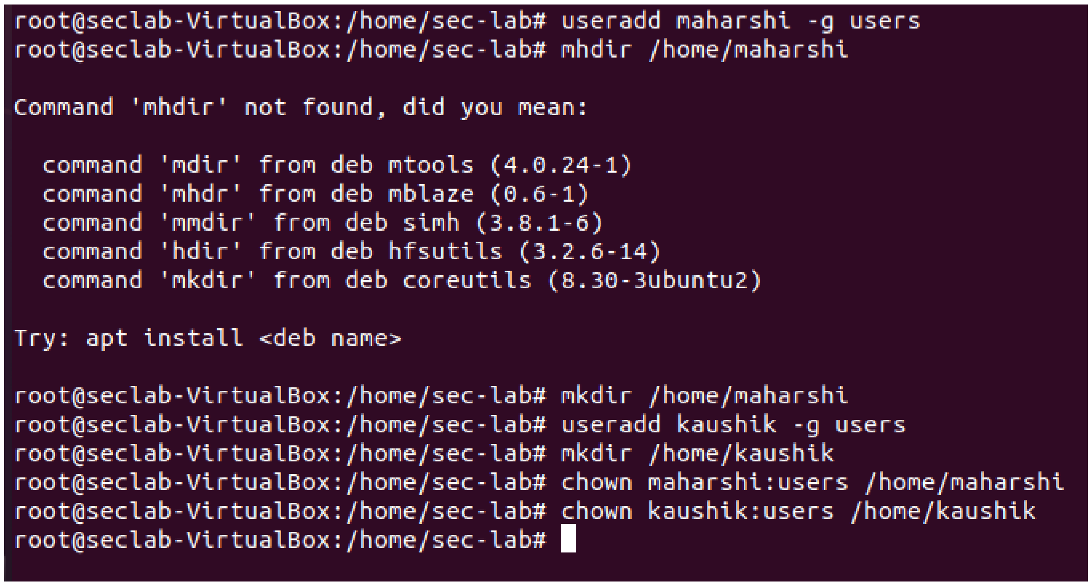
NOTE: Ideally, after all the operations intended to be performed under the privileges of a user are completed, it is better to "exit" from that user. However, for the purposes of the lab, **su** will be done repeatedly in order to train you to switch users.

1. Login as root
   a. sudo su
   b. enter password when prompted.
   c. Now you are root, and the '#' prompt represents the root mode (while the prompt '$' represents the ordinary user mode).



```
sec-lab@seclab-VirtualBox:~$ sudo su
[sudo] password for sec-lab:
root@seclab-VirtualBox:/home/sec-lab#
```

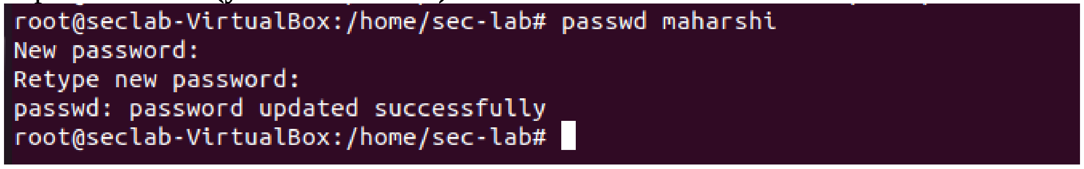
2. Use **useradd** command to create two new users ***user1*** and ***user2 (you can name them by your choice and follow that name in place of user1 and user2 in the document)*** as follows:
   a. useradd user1 -g users
   b. mkdir /home/user1
   c. chown user1:users /home/user1
   d. useradd user2 -g users
   e. mkdir /home/user2
   f. chown user2:users /home/user2



```
root@seclab-VirtualBox:/home/sec-lab# useradd maharshi -g users
root@seclab-VirtualBox:/home/sec-lab# mhdir /home/maharshi

Command 'mhdir' not found, did you mean:

  command 'mdir' from deb mtools (4.0.24-1)
  command 'mhdr' from deb mblaze (0.6-1)
  command 'mmdir' from deb simh (3.8.1-6)
  command 'hdir' from deb hfsutils (3.2.6-14)
  command 'mkdir' from deb coreutils (8.30-3ubuntu2)

Try: apt install <deb name>

root@seclab-VirtualBox:/home/sec-lab# mkdir /home/maharshi
root@seclab-VirtualBox:/home/sec-lab# useradd kaushik -g users
root@seclab-VirtualBox:/home/sec-lab# mkdir /home/kaushik
root@seclab-VirtualBox:/home/sec-lab# chown maharshi:users /home/maharshi
root@seclab-VirtualBox:/home/sec-lab# chown kaushik:users /home/kaushik
root@seclab-VirtualBox:/home/sec-lab#
```

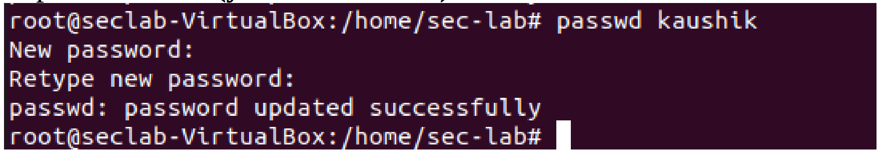
3. Use **passwd** command to set the password for the users you created (required in the case you want to log in). For convenience, set the passwords to be the same as the usernames. You need to retype the passwords and ignore password warnings:

a. passwd user1 **(your user name)**



```
root@seclab-VirtualBox:/home/sec-lab# passwd maharshi
New password:
Retype new password:
passwd: password updated successfully
root@seclab-VirtualBox:/home/sec-lab#
```

b. passwd user2 **(your user name)**

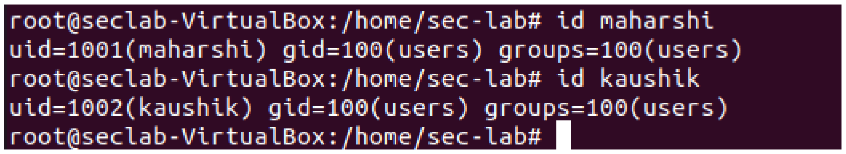


```
root@seclab-VirtualBox:/home/sec-lab# passwd kaushik
New password:
Retype new password:
passwd: password updated successfully
root@seclab-VirtualBox:/home/sec-lab#
```

4. Check user information with the **id** command. Note the uid, gid for each output.

a. id user1

b. id user2

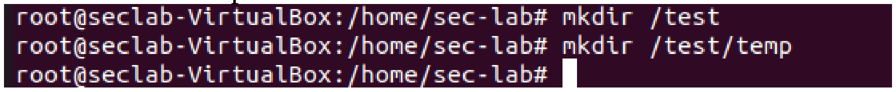


```
root@seclab-VirtualBox:/home/sec-lab# id maharshi
uid=1001(maharshi) gid=100(users) groups=100(users)
root@seclab-VirtualBox:/home/sec-lab# id kaushik
uid=1002(kaushik) gid=100(users) groups=100(users)
root@seclab-VirtualBox:/home/sec-lab#
```

5. Create directory structure

a. mkdir /test

b. mkdir /test/temp



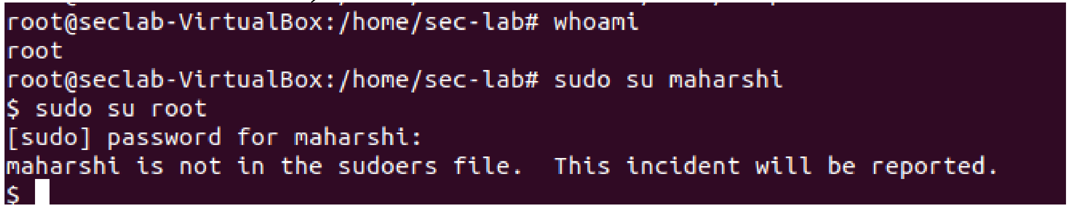
```
root@seclab-VirtualBox:/home/sec-lab# mkdir /test
root@seclab-VirtualBox:/home/sec-lab# mkdir /test/temp
root@seclab-VirtualBox:/home/sec-lab#
```

6. Switch user roles as *user1* and then back to *root* using the **su** command

a. whoami

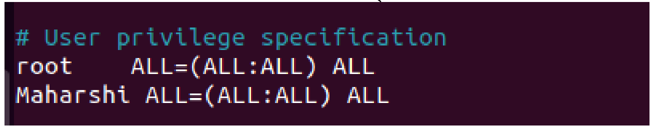
b. sudo su user1 (check '$' indicate user mode)

c. sudo su **OR** sudo su root (unable to switch root? You might want to add user1 to the sudoers file)



```
root@seclab-VirtualBox:/home/sec-lab# whoami
root
root@seclab-VirtualBox:/home/sec-lab# sudo su maharshi
$ sudo su root
[sudo] password for maharshi:
maharshi is not in the sudoers file.  This incident will be reported.
$
```

Added User1 to sudoers file (command: sudo visudo)



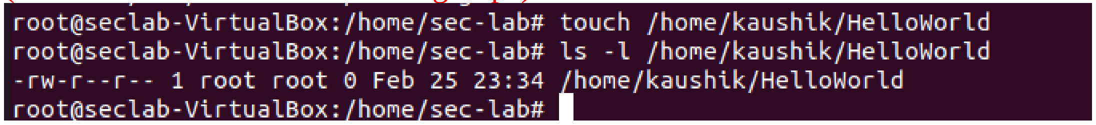
```
# User privilege specification
root    ALL=(ALL:ALL) ALL
Maharshi ALL=(ALL:ALL) ALL
```

7. Create a new file as **root** user

a. touch /home/user2/HelloWorld

b. ls –l /home/user2/HelloWorld

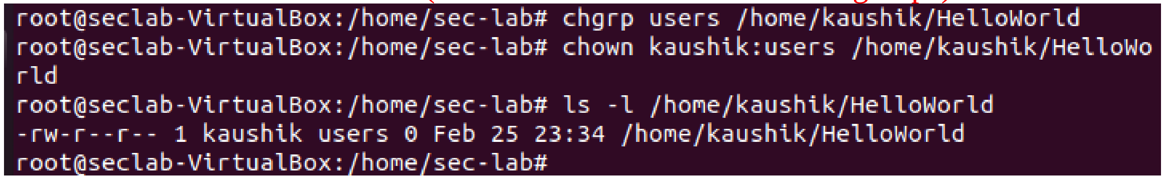
(Who is its owner and what is its group?)



```
root@seclab-VirtualBox:/home/sec-lab# touch /home/kaushik/HelloWorld
root@seclab-VirtualBox:/home/sec-lab# ls -l /home/kaushik/HelloWorld
-rw-r--r-- 1 root root 0 Feb 25 23:34 /home/kaushik/HelloWorld
root@seclab-VirtualBox:/home/sec-lab#
```

Owner is "root" and Group is "root".

8. Change group ownership as well as user ownership of the file
    a. chgrp users /home/user2/HelloWorld
    b. chown user2:users /home/user2/HelloWorld
    c. ls –l /home/user2/HelloWorld (Who is its owner and what is its group?)



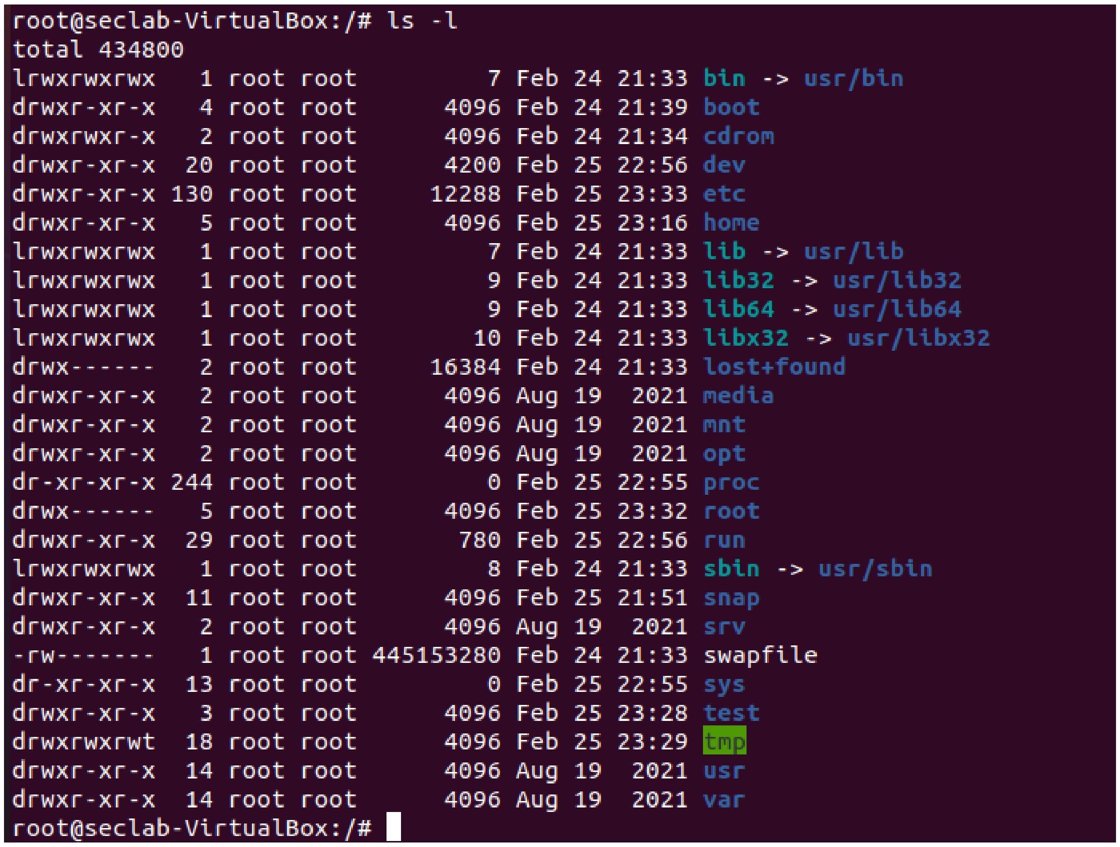
```
root@seclab-VirtualBox:/home/sec-lab# chgrp users /home/kaushik/HelloWorld
root@seclab-VirtualBox:/home/sec-lab# chown kaushik:users /home/kaushik/HelloWo
rld
root@seclab-VirtualBox:/home/sec-lab# ls -l /home/kaushik/HelloWorld
-rw-r--r-- 1 kaushik users 0 Feb 25 23:34 /home/kaushik/HelloWorld
root@seclab-VirtualBox:/home/sec-lab#
```

Owner of file is “Kaushik” and Group is “users”.

Exercise 2.2: Differences in File and Folder Permissions

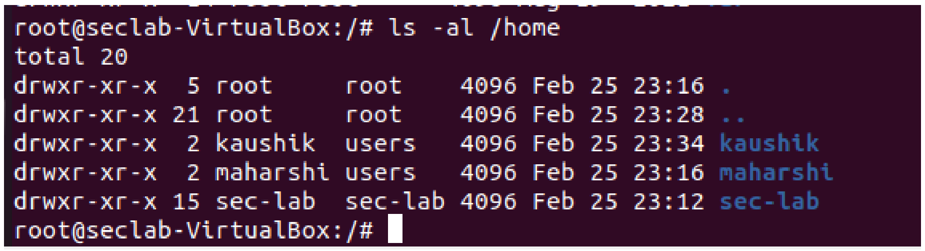
The objective of the following exercises is to see differences between file and folder (directory) permissions. The **chmod** command will be used to change the file and directory permissions to demonstrate these differences.

1. Observe the result of **ls** and **cd** commands
    a. cd /
    b. ls –l



```
root@seclab-VirtualBox:/# ls -l
total 434800
lrwxrwxrwx   1 root root         7 Feb 24 21:33 bin -> usr/bin
drwxr-xr-x   4 root root      4096 Feb 24 21:39 boot
drwxrwxr-x   2 root root      4096 Feb 24 21:34 cdrom
drwxr-xr-x  20 root root      4200 Feb 25 22:56 dev
drwxr-xr-x 130 root root     12288 Feb 25 23:33 etc
drwxr-xr-x   5 root root      4096 Feb 25 23:16 home
lrwxrwxrwx   1 root root         7 Feb 24 21:33 lib -> usr/lib
lrwxrwxrwx   1 root root         9 Feb 24 21:33 lib32 -> usr/lib32
lrwxrwxrwx   1 root root         9 Feb 24 21:33 lib64 -> usr/lib64
lrwxrwxrwx   1 root root        10 Feb 24 21:33 libx32 -> usr/libx32
drwx------   2 root root     16384 Feb 24 21:33 lost+found
drwxr-xr-x   2 root root      4096 Aug 19  2021 media
drwxr-xr-x   2 root root      4096 Aug 19  2021 mnt
drwxr-xr-x   2 root root      4096 Aug 19  2021 opt
dr-xr-xr-x 244 root root         0 Feb 25 22:55 proc
drwx------   5 root root      4096 Feb 25 23:32 root
drwxr-xr-x  29 root root       780 Feb 25 22:56 run
lrwxrwxrwx   1 root root         8 Feb 24 21:33 sbin -> usr/sbin
drwxr-xr-x  11 root root      4096 Feb 25 21:51 snap
drwxr-xr-x   2 root root      4096 Aug 19  2021 srv
-rw-------   1 root root 445153280 Feb 24 21:33 swapfile
dr-xr-xr-x  13 root root         0 Feb 25 22:55 sys
drwxr-xr-x   3 root root      4096 Feb 25 23:28 test
drwxrwxrwt  18 root root      4096 Feb 25 23:29 tmp
drwxr-xr-x  14 root root      4096 Aug 19  2021 usr
drwxr-xr-x  14 root root      4096 Aug 19  2021 var
root@seclab-VirtualBox:/#
```

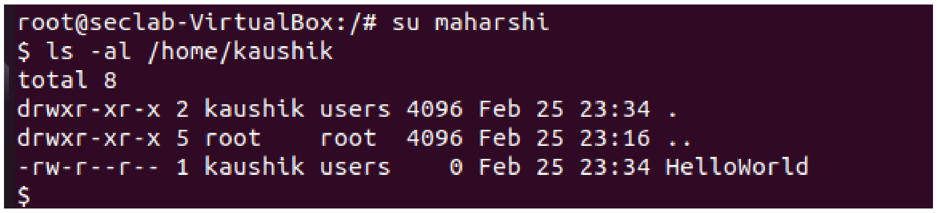
    c. ls -al /home


```
root@seclab-VirtualBox:/# ls -al /home
total 20
drwxr-xr-x  5 root     root    4096 Feb 25 23:16 .
drwxr-xr-x 21 root     root    4096 Feb 25 23:28 ..
drwxr-xr-x  2 kaushik  users   4096 Feb 25 23:34 kaushik
drwxr-xr-x  2 maharshi users   4096 Feb 25 23:16 maharshi
drwxr-xr-x 15 sec-lab  sec-lab 4096 Feb 25 23:12 sec-lab
root@seclab-VirtualBox:/#
```

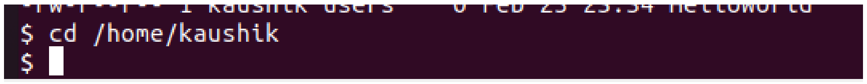
d. What are the directory permissions for ***user1***, ***user2*** and ***test*** directories?
User1, User2 and root permissions: drwxr-xr-x

e. Switch to ***user1*** using **su user1**

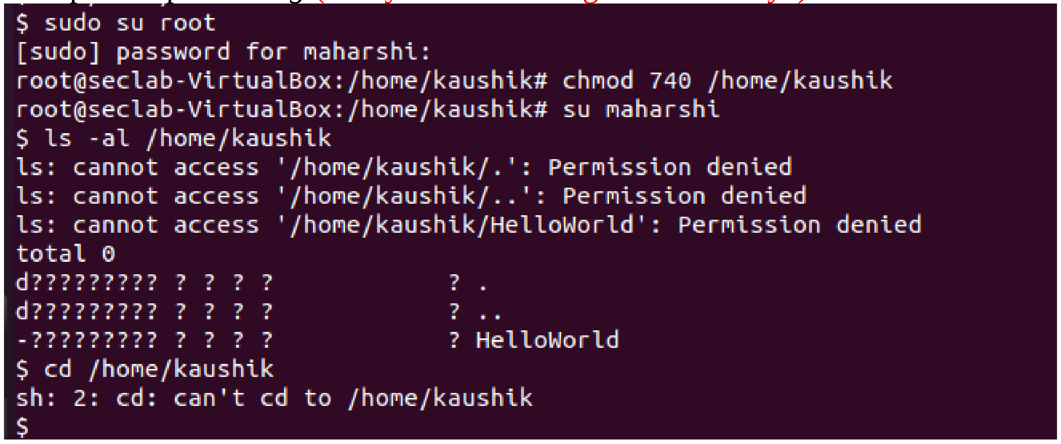
f. ls -al /home/user2 (Can you list the directory?)
Yes


```
root@seclab-VirtualBox:/# su maharshi
$ ls -al /home/kaushik
total 8
drwxr-xr-x 2 kaushik users 4096 Feb 25 23:34 .
drwxr-xr-x 5 root    root  4096 Feb 25 23:16 ..
-rw-r--r-- 1 kaushik users    0 Feb 25 23:34 HelloWorld
$
```

g. cd /home/user2 (Can you change the directory?)
Yes

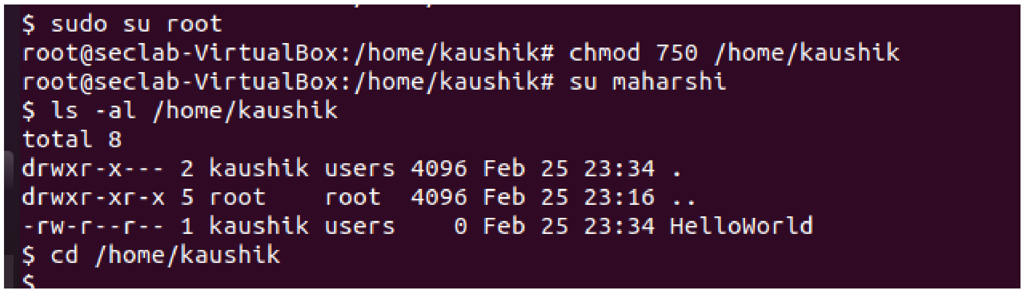

```
$ cd /home/kaushik
$
```

2. Change directory permissions of ***user2*** directory and try again as ***user1***.
a. sudo su root
b. chmod 740 /home/user2
c. Repeat steps *1e* to *1g* (Can you list or change the directory?) No


```
$ sudo su root
[sudo] password for maharshi:
root@seclab-VirtualBox:/home/kaushik# chmod 740 /home/kaushik
root@seclab-VirtualBox:/home/kaushik# su maharshi
$ ls -al /home/kaushik
ls: cannot access '/home/kaushik/.': Permission denied
ls: cannot access '/home/kaushik/..': Permission denied
ls: cannot access '/home/kaushik/HelloWorld': Permission denied
total 0
d????????? ? ? ? ?            ? .
d????????? ? ? ? ?            ? ..
-????????? ? ? ? ?            ? HelloWorld
$ cd /home/kaushik
sh: 2: cd: can't cd to /home/kaushik
$
```

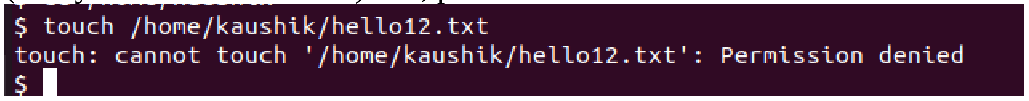
d. sudo su root
e. chmod 750 /home/user2
f. Repeat steps *1e* to *1g* (Can you list or change the directory?)

Yes


```
$ sudo su root
root@seclab-VirtualBox:/home/kaushik# chmod 750 /home/kaushik
root@seclab-VirtualBox:/home/kaushik# su maharshi
$ ls -al /home/kaushik
total 8
drwxr-x--- 2 kaushik users 4096 Feb 25 23:34 .
drwxr-xr-x 5 root    root  4096 Feb 25 23:16 ..
-rw-r--r-- 1 kaushik users    0 Feb 25 23:34 HelloWorld
$ cd /home/kaushik
$
```

g. touch /home/user2/hello12.txt

(Can you create a new file?) No, permission denied.

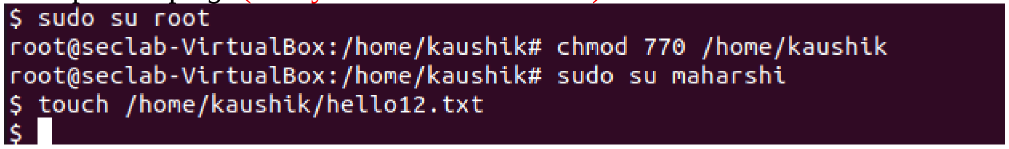

```
$ touch /home/kaushik/hello12.txt
touch: cannot touch '/home/kaushik/hello12.txt': Permission denied
$
```

h. sudo su root

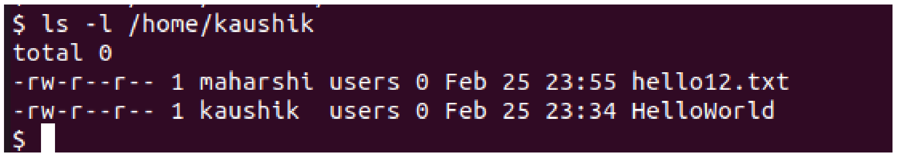
i. chmod 770 /home/user2

j. sudo su user1

k. Repeat step *2g*. (Can you create a new file?) Yes


```
$ sudo su root
root@seclab-VirtualBox:/home/kaushik# chmod 770 /home/kaushik
root@seclab-VirtualBox:/home/kaushik# sudo su maharshi
$ touch /home/kaushik/hello12.txt
$
```

l. ls –l /home/user2


```
$ ls -l /home/kaushik
total 0
-rw-r--r-- 1 maharshi users 0 Feb 25 23:55 hello12.txt
-rw-r--r-- 1 kaushik  users 0 Feb 25 23:34 HelloWorld
$
```

Exercise 2.3: Text files and link files

Unix supports two kinds of link files—a *hard link* and a *symbolic link*. A *hard* link is a file with the actual address space of some ordinary file's data blocks. A *symbolic* (or *soft*) link is just a reference to another file. It contains the pathname to some other file. It is basically a shortcut to a file, which is typically used to access it. (Similarly to shortcuts in Windows.)

1. In the /test/temp/ directory, as the root user, create a new text file ("hello.txt") and fill it with some text.   q

    a. echo something > /test/temp/hello.txt

2. Create a link ***link_hello*** in the ***test*** folder pointing to ***hello.txt*** in the ***temp*** folder (refer to the file structure in the introduction of the file hierarchy).

    a. cd /

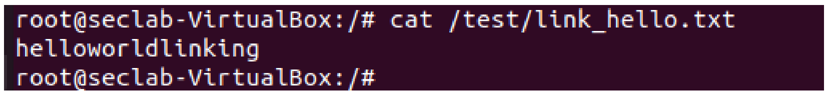
    b. ln -s /test/temp/hello.txt /test/link_hello

    c. Is there any difference in file permissions of *link_hello* and *hello.txt*?

In linux, default permissions given to ordinary symbolic link are 0777. But as the hello.txt is created by root user, other groups can only read the file. However, the permissions given to symbolic link are not used and the access is defined by permissions of the original file

d. cat /test/link_hello.txt —what is the output?

I can see the data stored in that file.



```
root@seclab-VirtualBox:/# cat /test/link_hello.txt
helloworldlinking
root@seclab-VirtualBox:/#
```

Exercise 2.4: Default file permissions and group access control

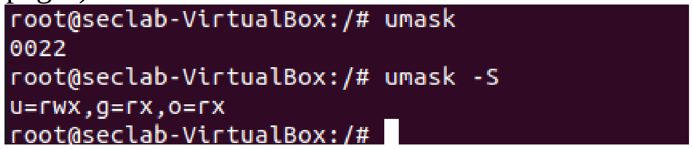
Whenever a new file is created, a default set of permissions is assigned to it. Whatever the permissions are, the UNIX system allows a user to filter out unwanted permissions set by default. This default setting can be set by the user using the **umask** command. The command takes the permissions set during creation of file and performs a bitwise AND to the bitwise negation of the mask value. Some common umask values are 077 (only user has permissions), 022 (only owner can write), 002 (only owner and group members can write), etc.

1. In a terminal window, make sure you are the root user, otherwise switch back to the root user.

2. Use *umask* command to check the current mask permission and assign a new mask.

   a. umask - default value is either 0022 or 0002

   b. What is the current mask? How is it interpreted? (try umask –S or the man pages)



```
root@seclab-VirtualBox:/# umask
0022
root@seclab-VirtualBox:/# umask -S
u=rwx,g=rx,o=rx
root@seclab-VirtualBox:/# 
```

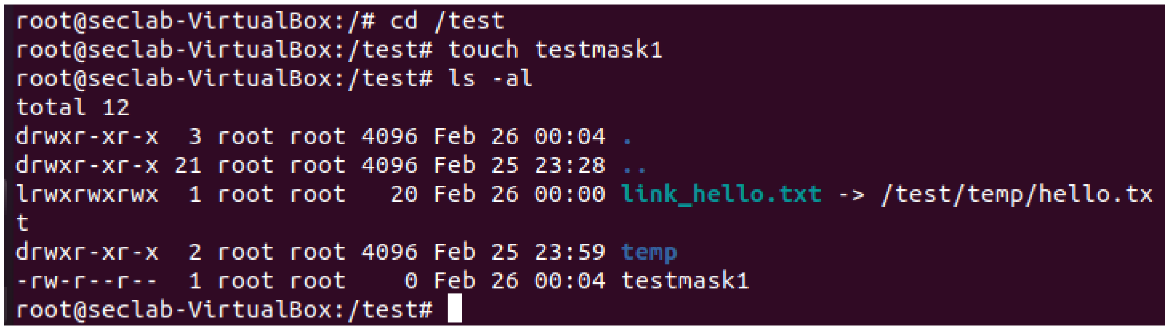
0022, last 3 digits are used to represent u,g and o respectively.
u – Owner(0) - permissions that will be removed from the file or directory Owner
g – Owner Group(2) - permissions that will be removed from the Owning Group.
o – Others(2) - permissions that will be removed from Others on the system.
r = read, w = write and x = execute

c. cd /test
d. touch testmask1

e. ls -al

f. What are the permissions of the file “*testmask1*”?

-rw-r—r--



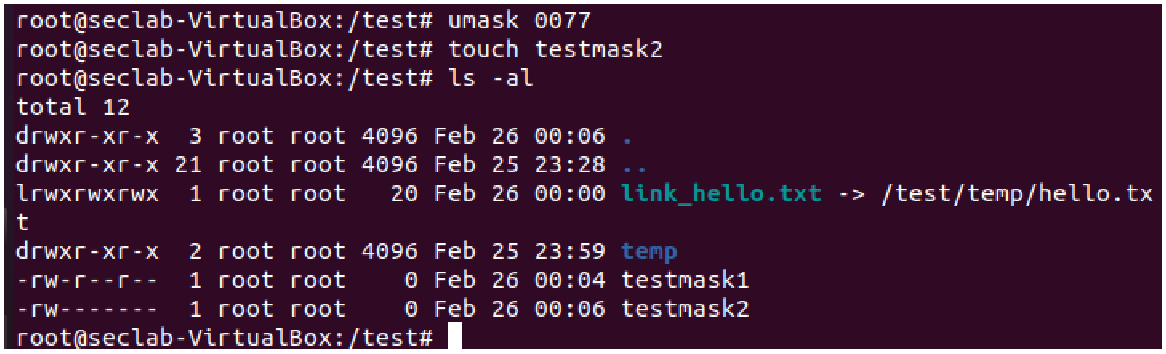
```
root@seclab-VirtualBox:/# cd /test
root@seclab-VirtualBox:/test# touch testmask1
root@seclab-VirtualBox:/test# ls -al
total 12
drwxr-xr-x  3 root root 4096 Feb 26 00:04 .
drwxr-xr-x 21 root root 4096 Feb 25 23:28 ..
lrwxrwxrwx  1 root root   20 Feb 26 00:00 link_hello.txt -> /test/temp/hello.tx
t
drwxr-xr-x  2 root root 4096 Feb 25 23:59 temp
-rw-r--r--  1 root root    0 Feb 26 00:04 testmask1
root@seclab-VirtualBox:/test# 
```

g. umask 0077

h. touch testmask2

i. Now, what are the permissions of the file “*testmask2*”?

-rw-------



```
root@seclab-VirtualBox:/test# umask 0077
root@seclab-VirtualBox:/test# touch testmask2
root@seclab-VirtualBox:/test# ls -al
total 12
drwxr-xr-x  3 root root 4096 Feb 26 00:06 .
drwxr-xr-x 21 root root 4096 Feb 25 23:28 ..
lrwxrwxrwx  1 root root   20 Feb 26 00:00 link_hello.txt -> /test/temp/hello.tx
t
drwxr-xr-x  2 root root 4096 Feb 25 23:59 temp
-rw-r--r--  1 root root    0 Feb 26 00:04 testmask1
-rw-------  1 root root    0 Feb 26 00:06 testmask2
root@seclab-VirtualBox:/test# 
```

3. What does it mean if the umask value is set to 0000?

If umask is 0000, it means that newly created files or directories created will have no privileges revoked initially. It means all types of users can access the files/directories.

4. The risks of setting the extra bits (i.e. assigning their value to “1”) will be covered in Exercise 2.5, which shows that the extra bits should not be set, in general. What should be the umask value to ensure that the extra bits cannot be set?

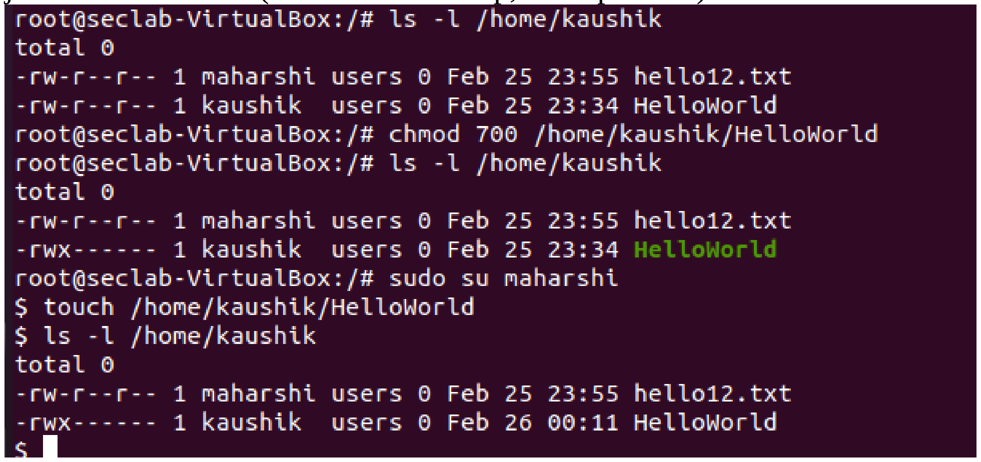
By default, the unmask value should begin with 0. Unmask 0077 file is accessible only by the user

Exercise 2.5: *setuid* bit, *setgid* bit and *sticky* bit

As explained in the *ownership and permission* section, the highest three bits of the permission value of a file represent the *setuid* bit, *setgid* bit and the *sticky* bit. If the *setuid* bit is set then the uid will always be set to the owner of the file during execution. If the *setuid* bit is not set then the uid will be that of the user who executes the process. Similarly, if the *setgid* bit is set then the gid will be set to the group that owns the file during execution. If the *setgid* bit is not set then the gid will be that of the group that executes the process. The sticky bit lets only the owner of the file or directory (and in addition the root user) to delete or rename it.

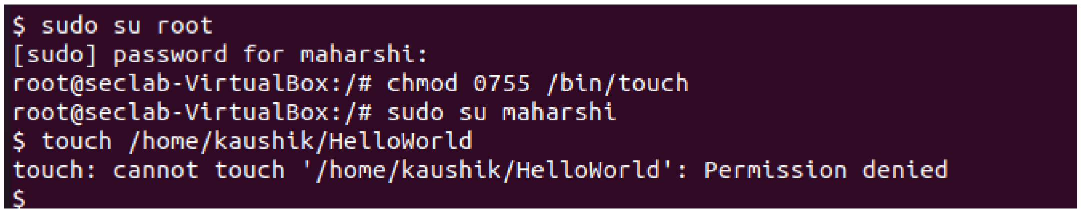
In the following exercise, the objective is to demonstrate how processes are affected when the *setuid* bit is set. The exercise must be begun with root privileges.

a. which touch
b. ls –l /bin/touch
c. chmod 4755 /bin/touch
d. ls –l /bin/touch
e. ls –l /home/user2
f. chmod 700 /home/user2/HelloWorld
g. ls –l /home/user2 (observe timestamp and permissions)
h. sudo su user1
i. touch /home/user2/HelloWorld
j. ls –l /home/user2 (observe timestamp, is it updated?) Yes



```
root@seclab-VirtualBox:/# ls -l /home/kaushik
total 0
-rw-r--r-- 1 maharshi users 0 Feb 25 23:55 hello12.txt
-rw-r--r-- 1 kaushik  users 0 Feb 25 23:34 HelloWorld
root@seclab-VirtualBox:/# chmod 700 /home/kaushik/HelloWorld
root@seclab-VirtualBox:/# ls -l /home/kaushik
total 0
-rw-r--r-- 1 maharshi users 0 Feb 25 23:55 hello12.txt
-rwx------ 1 kaushik  users 0 Feb 25 23:34 HelloWorld
root@seclab-VirtualBox:/# sudo su maharshi
$ touch /home/kaushik/HelloWorld
$ ls -l /home/kaushik
total 0
-rw-r--r-- 1 maharshi users 0 Feb 25 23:55 hello12.txt
-rwx------ 1 kaushik  users 0 Feb 26 00:11 HelloWorld
$
```

k. sudo su root
l. chmod 0755 /bin/touch
m. sudo su user1
n. touch /home/user2/HelloWorld



```
$ sudo su root
[sudo] password for maharshi:
root@seclab-VirtualBox:/# chmod 0755 /bin/touch
root@seclab-VirtualBox:/# sudo su maharshi
$ touch /home/kaushik/HelloWorld
touch: cannot touch '/home/kaushik/HelloWorld': Permission denied
$
```

Why permission denied, while previously allowed?

HelloWorld is owned by User2(Kaushik), that's why permission was denied. In previous case, when touch was run it was run with owner privileges(neha), that is why it worked before. But setuid is reverted to normal now. So, User1(Maharshi) has no permission to access the file.

RESTORE THE SYSTEM

Below are the set of commands that you should issue to restore the system to its original form (commands No. 4 and 5 could take some time).

1. sudo su root
2. umask 0022
3. chmod 0755 /bin/touch
4. userdel user1
5. userdel user2
6. rm –rf /home/user1
7. rm –rf /home/user2
8. rm –rf /test
9. rm –rf /home/test/

Set-UID Program Vulnerability

Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program (executable) is run, it assumes the owner's privileges. For example, if the program's owner is root, then it gains the root's privileges during its execution, no matter who runs it. Set-UID allows ordinary users to gain access to many important functionalities, but unfortunately, it is also the source of many vulnerabilities.

The objectives of this lab are to let the students:

(1) Appreciate its positive side: to understand why Set-UID is needed and how it is implemented.

(2) Become aware of its negative side: to understand potential security problems that it may cause.

Lab Tasks:

This is an exploration lab. Your main task is to "play" with the Set-UID mechanism in Linux. You are required to accomplish the following tasks in Linux:

Note: Please refer to the Lab 1a manual for the basic Linus commands. For example, we assume that the students know that when the lab instructs to copy a file, the **cp** command should be used. For information on working with umask, see, e.g., the following: https://www.linuxnix.com/umask-define-linuxunix/

1. Figure out why "passwd," "chsh," "su," and "sudo" commands need to be Set-UID programs. What will happen if they are not? If you are not familiar with these commands, you should first learn about them by reading their manuals (For example man su, man chsh, etc.).

The commands "passwd," "chsh," "su," and "sudo" need to be Set-UID programs because, these need root permissions to be executed. If they are not Set-UID programs, the user(not root) may need root permissions to perform few activities. So, Set-UID bit is set to avoid running these commands by again logging in as a root. To conclude, set-uid bit is set to make the required programs run as with root.

2. Run Set-UID shell programs in Linux, then describe and explain your observations.

a. Login as root.

1. apt-get update
2. apt-get install zsh
3. Copy /bin/zsh to /tmp (cp /bin/zsh), and make it a set-root-uid program with permission 4755 (chmod 4755 zsh).
4. cd /tmp
5. ls –la //to see the permission change

b. Add a new user.

1. adduser bob (**username of your choice)**
2. Enter your password.
3. Next, enter the password of the new user "bob" (**username)**, after pressing Enter complete the user details.
4. su bob (**username)**

c. Then login as a normal user, and run /tmp/zsh. Will you get root privilege? Yes

1. ./zsh //this is to run the zsh
   Notice: the hostname is now "ubuntu".
2. pwd //see that your current folder; it to should be /tmp
3. cd /root //change directory to root folder
4. pwd //notice that you entered the root folder while you are not root
5. exit //to come out of the zsh shell

d. Perform the same steps as in the above step (c) for bash:

Instead of copying /bin/zsh, copy /bin/bash to /tmp. Make it a set-root-uid program with permission 4755. Run /tmp/bash as a normal user. Try to go to the root directory. Can you get access to the root privilege? No, the privilege access is denied for the bash script execution.

Please describe and explain your observations when performing steps (c) and (d).

When zsh (zsh in tmp has root as its owner and setuid bit is also set) was used, the normal user got the root privileges but for bash(bash in tmp has normal user as its owner and setuid bit is set) the normal user did not get any root privileges, running it as a normal user would raise error to access root folder which everyone doesn't have access to. Bash might be having few pre-default mechanism to prevent this usage. Permission was denied even repeating the process by copying the bash as root to tmp.

3. As you will see from the previous task, /bin/bash has a certain built-in protection that prevents abuse of the Set-UID mechanism. In order to understand the issues which arise

when such the protection is not implemented, we are going to use a different shell program called /bin/zsh.

In some Linux distributions (such as Fedora and Ubuntu), /bin/sh is actually a symbolic link to /bin/bash. To use zsh, we need to link /bin/sh to /bin/zsh. The following instructions describe how to change the default shell to zsh.

```
· sudo su
Password: (enter root password)
· cd /bin
· rm –rf sh //to remove the original sh shell
· ln -s zsh sh //to create a symbolic link of the vulnerable zsh shell to be sh.
```

4. **The PATH environment variable.**

The system (const char *cmd) library function can be used to execute a command within a program. The way system (cmd) works is to invoke the /bin/sh program, and then let the shell program to execute cmd. Since the shell program is invoked, calling system() within a Set-UID program is extremely dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as PATH. These environment variables are under user's control. By changing these variables, attackers (malicious users) can control behavior of Set-UID programs. In bash, you can change the PATH environment variable in the following way (this example adds the directory /home/sec-lab to the beginning of the PATH environment variable):

```
· sudo su
· export PATH=/home/sec-lab:$PATH
```

The Set-UID program below is supposed to execute the **/bin/ls** command; however, suppose that the programmer only uses the relative path for the **ls** command, rather than the absolute path:

A. Create a file: make sure you are still in the bin folder (if not, then **cd /bin**) a) nano setUID.c

b) copy the following code to the file:

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

c) gcc –o setUID setUID.c //this is to compile the c code

d) sudo chmod 4755 setUID

e) ./setUID //to execute the executable file

f) Notice the output.

g) cd /usr/local/

h) ls –la

Notice that the “bin” folder is owned by root (normal users, processes and programs should not have direct access) and your program had access to it as it used the setUID.

A. Can you let this Set-UID program (owned by root) run your code instead of /bin/ls? If you can, is your code running with the root privilege? Describe and explain your observations.

yes, Set-UID program owned by root can run the code instead of /bin/ls and it can have root privilege. setuid bit is true and this leads to running the command as root. The difference between running ls as a root privilege and as a normal user is that, some variables are not assigned (ex: list shown for normal user is colored and with root user it is not because –color variable is set to never with root)

B. Now, change /bin/sh so it points back to /bin/bash, and repeat the above attack. · cd /bin/

· rm –rf sh

· ln -s bash sh

Can you still get the root privilege? Describe and explain your observations.

We can still access the root privilege. As commands executed are with root privilege, it does not matter if we are using zsh or bash.

5. The difference between system() and execve().

NOTE: *Before you work on this task, please make sure that /bin/sh is linked to /bin/zsh, using the following steps:*

a. cd /bin/

b. rm –rf sh

c. ln -s zsh sh

Background: Suppose that Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company’s Unix system. At the same time, to protect the integrity
of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run /bin/cat to display the specified file.

Since the program is running as a root, it can display any file that Bob specifies. However, since the program has no write operations, Vince is sure that Bob cannot use this special program to modify any file.

- Create a file as root in /home/<user-name>/Desktop

//Note: the /<user-name>/ should be replaced with your username (e.g., “sec-admin”). o nano topSecretFile.txt

  - ▪ Enter any text in the file.

- Create a file as a root user in /home/<user-name>/Desktop

//Note: the /<user-name>/ should be replaced with your username (e.g., “sec-admin”). Copy-paste the below code to the file you created.

  - o nano hack.c
    - ▪ Copy the below code to this file.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
 {
 char *v[3];

if (argc < 2)
 {
 printf("Please type a file name.\n");
 return 1;
 }

v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;

/* Set q = 0 for Question a, and q = 1 for Question b */

int q = 0;
if (q == 0)
 {
 char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
 sprintf(command, "%s %s", v[0], v[1]);
 system(command);
 }
 else execve(v[0], v[1], 0);
 return 0 ;
 }
```

I. Once you have copied and pasted the code compile it.

   a. gcc –o code hack.c

   b. ./code topSecretFile.txt

// The text file containing some secret information that you created earlier // is passed as an argument in the command line.

A. Set q = 0 in the program. This way, the program will use system() to invoke the command.

Is this program safe? If you were Bob, can you compromise the integrity of the system? For example, can you delete any file without proper permissions? (Hint: Remember that system() actually invokes /bin/sh, and then runs the command within the shell environment. We tried an attack that exploited the environment variable in the previous task. Here, let us try a different approach. Please pay attention to the special characters used in a normal shell environment).

No, the program is not safe anymore. Yes, we can compromise the integrity of the system, I have tried deleting a file which I created on the desktop and it deleted the file.

B. Set q = 1 in the program. This way, the program will use execve() to invoke the command.

Do your attacks from task (a) still work? Please describe and explain your observations.

No, it's not working in q=1 case. File is not executed and shows no output because execve() is executed in the second case. Execve() only runs the executables and other instructions will not be executed. System creates a new process and exec runs in the same process and here the user is normal user. Hence, the root privileges are not granted.

6**. The LD_PRELOAD environment variable.**

In order to make sure that Set-UID programs are safe from the manipulation of the LD_PRELOAD environment variable, the runtime linker (ld.so) will ignore this variable if the program is a Set-UID root program, except for some conditions. We will figure out what these conditions are in this task.

A. Let us build a dynamic link library. Create the following program, and name it "mylib.c". It basically overrides the sleep() function in libc:

```
#include <stdio.h>
void sleep (int s)
 {
 printf("I am not sleeping!\n");
 }
```

B. We can compile the above program using the following commands (in the arguments below, the third character in "-Wl" and the first character in "-lc" are a lowercase "L", not a digit one):

i. gcc -fPIC -g -c mylib.c
ii. gcc -shared -Wl,-soname,libmylib.so.1 -o libmylib.so.1.0.1 mylib.o –lc

C. Now, set the LD_PRELOAD environment variable:
i. export LD_PRELOAD=./libmylib.so.1.0.1

D. Finally, compile the following program myprog (put this program in the same directory as libmylib.so.1.0.1):

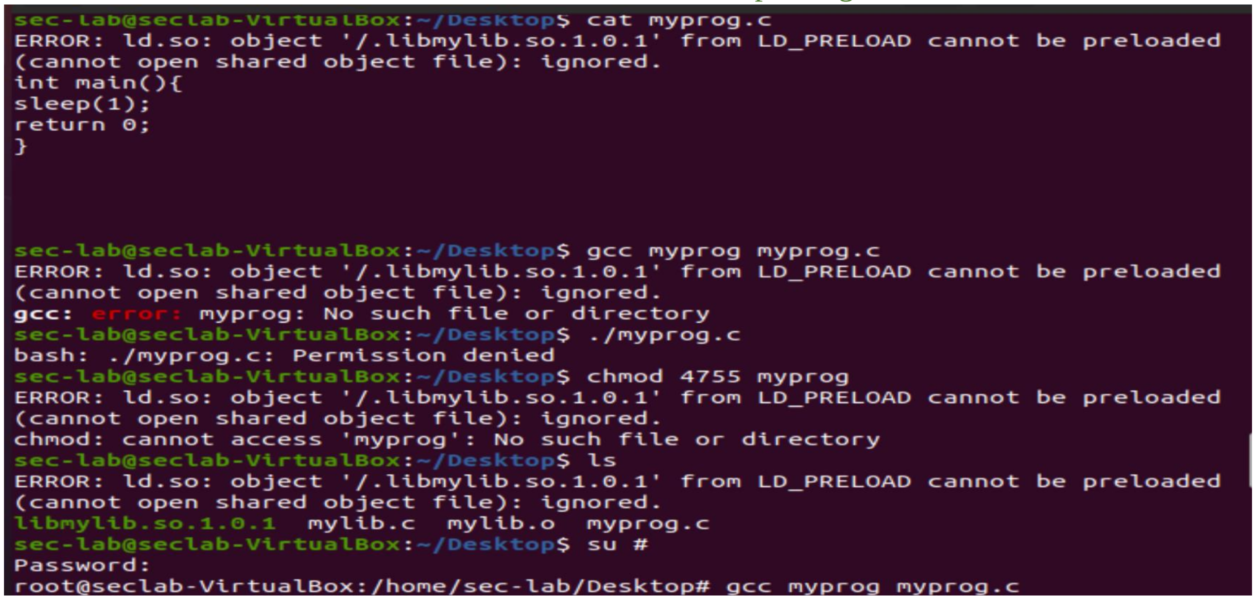
```
/* myprog.c */

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

a. gcc –o myprog myprog.c
// Make sure you are root when compiling the "myprog.c" program.
// Do not execute it immediately, instead, follow the below instructions.

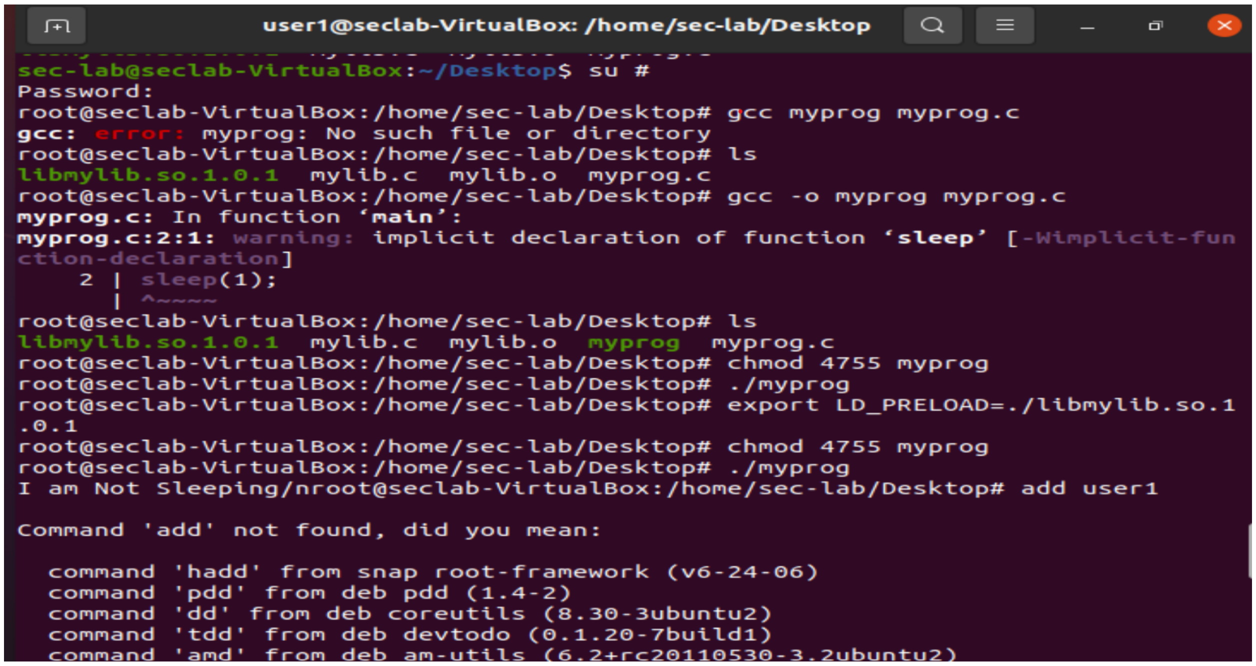
Run "myprog" under the following conditions, and observe what happens. Based on your observations, tell us when the runtime linker will ignore the LD_PRELOAD environment variable, and explain why.

· Run it as a normal user (not as root):
  ○ ./myprog
· Make "myprog" a Set-UID root program, and run it as a normal user: ○ chmod 4755 myprog //to make it set-UID
  ○ ./myprog
· Make "myprog" a Set-UID root program, and run under the root privileges. ○ ./myprog //remember that you are effectively root, when executing it · Make "myprog" a Set-UID user1 program (i.e., its owner is ***user1***, which is another user account), and run it as a normal user (not root).
  ○ adduser user1
    ▪ enter the user password
  ○ chown user1 myprog

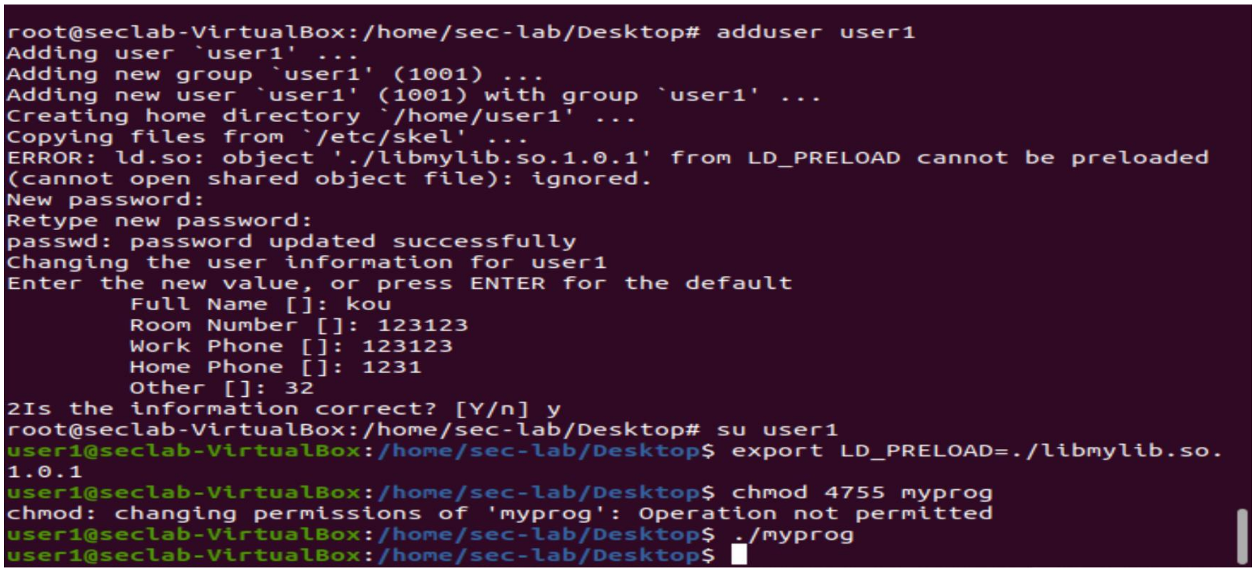
When running from user1 we are unable to see any output, but if we run using root privileges , we can see the output. Here the runtime linker will ignore the LD_PRELOAD environment variable when there is no root privilege. And the runtime linker will not ignore the LD_PRELOAD environment variable when there are root privilege.



```
sec-lab@seclab-VirtualBox:~/Desktop$ cat myprog.c
ERROR: ld.so: object '/.libmylib.so.1.0.1' from LD_PRELOAD cannot be preloaded
(cannot open shared object file): ignored.
int main(){
sleep(1);
return 0;
}



sec-lab@seclab-VirtualBox:~/Desktop$ gcc myprog myprog.c
ERROR: ld.so: object '/.libmylib.so.1.0.1' from LD_PRELOAD cannot be preloaded
(cannot open shared object file): ignored.
gcc: error: myprog: No such file or directory
sec-lab@seclab-VirtualBox:~/Desktop$ ./myprog.c
bash: ./myprog.c: Permission denied
sec-lab@seclab-VirtualBox:~/Desktop$ chmod 4755 myprog
ERROR: ld.so: object '/.libmylib.so.1.0.1' from LD_PRELOAD cannot be preloaded
(cannot open shared object file): ignored.
chmod: cannot access 'myprog': No such file or directory
sec-lab@seclab-VirtualBox:~/Desktop$ ls
ERROR: ld.so: object '/.libmylib.so.1.0.1' from LD_PRELOAD cannot be preloaded
(cannot open shared object file): ignored.
libmylib.so.1.0.1  mylib.c  mylib.o  myprog.c
sec-lab@seclab-VirtualBox:~/Desktop$ su #
Password:
root@seclab-VirtualBox:/home/sec-lab/Desktop# gcc myprog myprog.c
```



```
sec-lab@seclab-VirtualBox:~/Desktop$ su #
Password:
root@seclab-VirtualBox:/home/sec-lab/Desktop# gcc myprog myprog.c
gcc: error: myprog: No such file or directory
root@seclab-VirtualBox:/home/sec-lab/Desktop# ls
libmylib.so.1.0.1  mylib.c  mylib.o  myprog.c
root@seclab-VirtualBox:/home/sec-lab/Desktop# gcc -o myprog myprog.c
myprog.c: In function ‘main’:
myprog.c:2:1: warning: implicit declaration of function ‘sleep’ [-Wimplicit-fun
ction-declaration]
    2 | sleep(1);
      | ^~~~~
root@seclab-VirtualBox:/home/sec-lab/Desktop# ls
libmylib.so.1.0.1  mylib.c  mylib.o  myprog  myprog.c
root@seclab-VirtualBox:/home/sec-lab/Desktop# chmod 4755 myprog
root@seclab-VirtualBox:/home/sec-lab/Desktop# ./myprog
root@seclab-VirtualBox:/home/sec-lab/Desktop# export LD_PRELOAD=./libmylib.so.1
.0.1
root@seclab-VirtualBox:/home/sec-lab/Desktop# chmod 4755 myprog
root@seclab-VirtualBox:/home/sec-lab/Desktop# ./myprog
I am Not Sleeping/nroot@seclab-VirtualBox:/home/sec-lab/Desktop# add user1

Command 'add' not found, did you mean:

  command 'hadd' from snap root-framework (v6-24-06)
  command 'pdd' from deb pdd (1.4-2)
  command 'dd' from deb coreutils (8.30-3ubuntu2)
  command 'tdd' from deb devtodo (0.1.20-7build1)
  command 'amd' from deb am-utils (6.2+rc20110530-3.2ubuntu2)
```



```
root@seclab-VirtualBox:/home/sec-lab/Desktop# adduser user1
Adding user `user1' ...
Adding new group `user1' (1001) ...
Adding new user `user1' (1001) with group `user1' ...
Creating home directory `/home/user1' ...
Copying files from `/etc/skel' ...
ERROR: ld.so: object './libmylib.so.1.0.1' from LD_PRELOAD cannot be preloaded
(cannot open shared object file): ignored.
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for user1
Enter the new value, or press ENTER for the default
        Full Name []: kou
        Room Number []: 123123
        Work Phone []: 123123
        Home Phone []: 1231
        Other []: 32
2Is the information correct? [Y/n] y
root@seclab-VirtualBox:/home/sec-lab/Desktop# su user1
user1@seclab-VirtualBox:/home/sec-lab/Desktop$ export LD_PRELOAD=./libmylib.so.
1.0.1
user1@seclab-VirtualBox:/home/sec-lab/Desktop$ chmod 4755 myprog
chmod: changing permissions of 'myprog': Operation not permitted
user1@seclab-VirtualBox:/home/sec-lab/Desktop$ ./myprog
user1@seclab-VirtualBox:/home/sec-lab/Desktop$ 
```

7. Relinquishing privileges and cleanup.

To be more secure, Set-UID programs usually call setuid() system call to permanently relinquish their root privileges. However, sometimes, this is not enough. When a privileged process transitions to a non-privileged process, one of the common problems that may occur is the so called “capability leaking”. The process may have gained some capabilities when it was still
privileged. When the privileges are downgraded, if the program does not properly clean up these capabilities, they may still be accessible by the non-privileged processes.

For example, after a file is opened, its file descriptor is created, and the latter represents a form of capability, because whoever carries it is able of accessing the corresponding file. The program relinquishes the process’ capability by calling setuid(), but “forgets” to close the file. Now, its descriptor represents a “leaked” capability.

Compile the following program (see the next page), and make the program a set-root-uid program.

· Assuming you call your program hack.c (see its source on the next page): ○
gcc –o hack hack.c
- chmod 4755 hack
- ./hack // when executing the code, run it as a normal user

Describe what you have observed. Will the file /etc/zzz be modified?
Please explain your observation.

The file /etc/zzz is modified and popped “Malicious Data”. Even though file zzz is given only write permission for root, it can be executed by all users because setuid is set by root.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.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 create
         * 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 is 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);
        }
}
```

References

Wenliang Du, “Set-UID Privileged Programs” Lecture Notes of CIS/CSE 643: Computer Security, Syracuse University
http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Set_UID.pdf

The SET-UID man page
http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Vulnerability/Set-UID/files/setuid.pdf

How to write a SETUID Program
http://nob.cs.ucdavis.edu/~bishop/secprog/1987-sproglogin.pdf

Wenliang Du, “Computer & Internet Security – A Hands-on Approach”, 2nd ed., 2019.