Access Controls and Unix/Linux File Permissions

License

Contents

General notes about the labs

Often the lab instructions are intentionally open ended, and you will have to figure some things out for yourselves. This module is designed to be challenging, as well as fun!

However, we aim to provide a well planned and fluent experience. If you notice any mistakes in the lab instructions or you feel some important information is missing, please feel free to add a comment to the document by highlighting the text and click the comment icon (), and I (Cliffe) will try to address any issues. Note that your comments are public.

The labs are written to be informative and, in order to aid clarity, instructions that you should actually execute are generally written in this colour. Note that all lab content is assessable for the module, but the colour coding may help you skip to the "next thing to *do*", but make sure you dedicate time to read and understand everything. Coloured instructions in *italics* indicates you need to change the instructions based on your environment: for example, using your own IP address.

You should maintain a **lab logbook / document**, which should include your answers to the questions posed throughout the labs (in this colour).

Preparation

As with all of the labs in this module, start by loading the latest version of the LinuxZ template from the IMS system. If you have access to this lab sheet, you can read ahead while you wait for the image to load.

> To load the image: press F12 during startup (on the boot screen) to access the IMS system, then login to IMS using your university password. Load the template image: LinuxZ (load the latest version).

Once your LinuxZ image has loaded, log in using the username and password allocated to you by your tutor.

The root password -- **which should NOT be used to log in graphically** -- is "tiaspbiqe2r" (**t**his **i**s **a** **s**ecure **p**assword **b**ut **i**s **q**uite **e**asy **2** **r**emember). Again, never log in to the desktop environment using the root account -- that is bad practice, and should always be avoided.

These tasks can be completed on the LinuxZ system. Most of this lab could be completed on any openSUSE or using most other Linux distributions (although some details may change slightly).

Introduction to access control

Access control enforces *authorisation* by determining and enforcing which actions are allowed. Some terminology: a *subject* is an active entity taking actions, such as a user

or program, and an *object* is the (often passive) resource that can be accessed, such as a file or network resource. Access control mediates subjects' access to objects by enforcing a security policy, limiting which actions are and are not allowed. The *policy* expresses what is allowed, either formally or informally as a set of rules.

An access control *mechanism* is the code or thing that enforces a policy. An access control *model* is a way of representing and reasoning about a policy or types of policy.

Unix file permissions and inodes

The traditional Unix security model is based on the *discretionary access control (DAC)* model, which enables users to configure who can access the resources that they "own". Each user can control which other users can access the files that they create. This enables users to grant permissions, without involving a system admin. This is the type of security that has traditionally been built into most consumer OSs such as Windows and Unix.

Unix file permissions uses an abbreviated (simplified) form of access control list (ACL). A (full) ACL involves attaching a list of every subject and what they can do to each file (this is how Windows manages file access). For example, a file may have this ACL: "Joe can read, Frank can write, Alice can read, and Eve can read". Unix simplifies permissions by only defining rules for these three kinds of subjects:

- The user that owns the file (u)
- The file's group (g)
- Other users (o)

Open a terminal console.

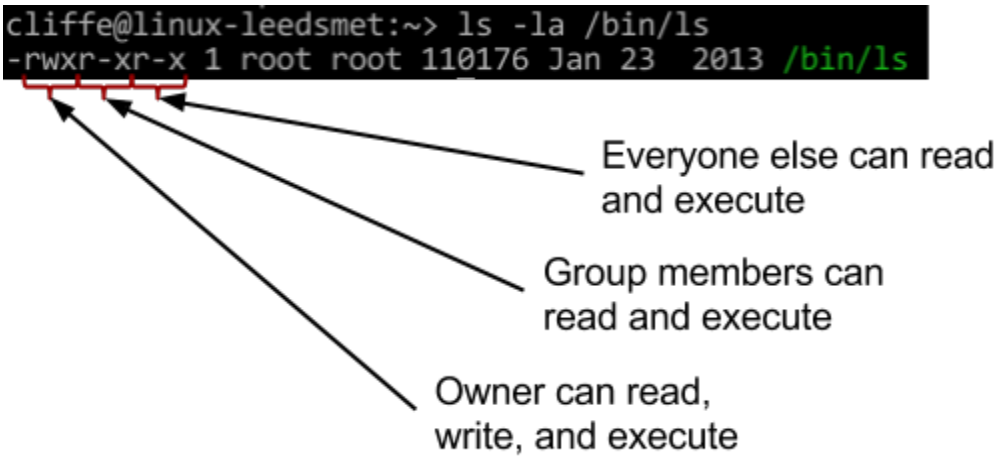
> One way to do this is to start Konsole from KDEMenu → Applications → System → Terminal → Konsole.

Use the `ls` command to display the permissions for a file (the details of the `ls` executable program itself):

```
ls -l /bin/ls
```

The "-l" flag instructs `ls` to provide this detailed output.

As illustrated in the figure below, the first part of the output contains the Unix file permissions for the file: what access the user, group, and other are authorised.



The meaning of these letters is fairly self evident, but does change meaning slightly depending on whether it refers to a normal file or a directory (which is really just a special kind of file).

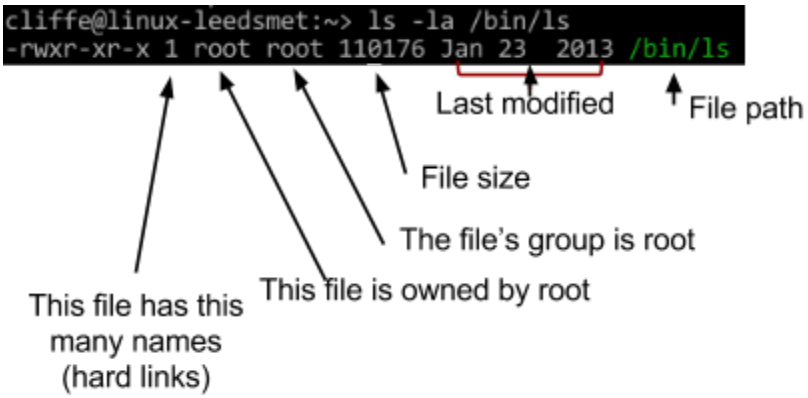
The meaning for a regular file (as is the case for /bin/ls):

- **r**: Read the contents of the file
- **w**: Change the contents of the file
- **x**: Execute the file as a process (The first few bytes describe what type of executable it is, a program or a script)

For a directory:

- **r**: See what files are in the directory
- **w**: Add, rename, or delete names from the directory
- **x**: 'stat' the file (view the file owners and sizes, `cd` into the directory, and access files within)
- **t** (instead of x), AKA the “sticky bit”: write is not enough to delete a file from the directory, in this case you also need to own the file

The rest of the output from `ls` describes how many names/hard links the file has, who owns the file (user and group associated with the file), the file size in bytes, the last access date, and finally the path and name of the file.



The permissions for each file are stored in the file's inode. An inode is a data structure in Unix filesystems that defines a file. An inode includes an inode number, and defines the location of the file on disk, along with attributes including the Unix file permissions, and access times for the file.

View the inode number for this file:

```
ls -i /bin/ls
```

Note that the inode does **not** contain the file's name, rather a directory can contain names that point to inodes. It is therefore possible to create two names (aka hard links) that point to the same file.

Create a hard link to the `ls` program:

```
ln /bin/ls /tmp/ls
```

Now view the details for your new filename, /tmp/ls:

```
ls -l /tmp/ls
```

How many hard links does this report? What are the file ownership and permissions associated with the new name?

View the inode number for this file:

```
ls -i /tmp/ls
```

The inode matches. There is *only one file*, but that data can now be accessed using two different names /tmp/ls and /bin/ls. If the /tmp/ls file was edited, the /bin/ls command would also change.

Deleting one of the names simply decrements the link counter. Only when that reaches 0 is the inode actually removed.

```
rm /tmp/ls
```

Permission denied! Interestingly, in this case as a normal user we can create the link to /bin/ls, but cannot then delete that link since the sticky bit is set for the /tmp/ directory.

```
ls -ld /tmp/
```

Note the “t” in the permissions, and refer to the meaning described above.

What is this directory used for? Why do you think the /tmp/ directory has the sticky bit set -- what does this stop users from doing to each other? How is this related to security?

You can delete the link as root:

```
sudo rm /tmp/ls
```

The stat command can be used to display further information from the inode:

```
stat /bin/ls
```

Look through this information. Note that the output includes the access rights, along with the last time the file was accessed, modified, and when the inode was last changed.

The output from `stat` includes the format that the information is stored as, along with a more "human readable" output. As we know, user accounts are referred to by UIDs by the system, in this case the UID is 0, as the file is owned by the root user. Similarly, groups are identified by GID, in this case also 0. The actual permissions are stored as four octets (digits 0-7), in this case "0755". This translates to the (now familiar) human-friendly output, "-rwxr-xr-x". For now we will ignore the first octet, this is normally 0, we will come back to the special meaning of this later.

Each of the other three octets simply represents the binary for rwx, each represented as a 0 or a 1. The first of the three represents the **u**ser, then the **g**roup, then the **o**ther permission. An easy and quick way to do the conversion is to simply remember:

- r = 4
- w = 2
- x = 1

And add them together to produce each of the three octets.

So for example, rwx = binary 111 = (4 + 2 + 1) = 7.

Likewise, r-x = binary 101 = (4 + 1) = 5.

Therefore, "-rwxr-xr-x" = 755.

Changing file permissions on a Linux system

Login via ssh to the computer at the front of the room (you can do this on your own computer, without ssh, or by starting sshd with "`sudo service sshd start`" and sshing into a classmate's system if doing this outside of class):

```
ssh yourusername@ipaddress
```

Do not login as root, instead use `sudo` as needed.

If you are doing these exercises by yourself, you could use "student" as your second user. You could simply open another console tab and "su - student", and switch between users/tabs as needed.

Create a file named "mysecrets" in your home directory:

```
cat > ~/mysecrets
```

Enter a number of lines of content. Press Ctrl-D when finished entering a "secret" (that others may see).

Your first aim is to ensure your “mysecrets” file is not visible to other users on the same system.

First view the permissions of your newly created file:

```
ls ~/mysecrets
```

Oh no! Its not so secret…

What kind of access do other users on the system have to this file?

The chmod command can be used to set permissions on a file. chmod can set permissions based on absolute octal values, or relative changes.

So for example, you could use chmod to set permissions on a file based on octet:

770 would give the owner and group rwx, and others no permissions

Example: `chmod 770 /home/tmp/somefile`

Or you can make relative changes:

u+x would add the owner (user) the ability to execute the file

Example: `chmod u+x /home/tmp/somefile`

Likewise, o-w would removes *other*'s ability to write to the file

Example: `chmod o-w /home/tmp/somefile`

Use chmod to grant yourself read-write permission to your mysecrets file, and everyone else no permissions to the file:

chmod *XXX* ~/mysecrets

Where *XXX* is three octets that grants the appropriate access.

Try to access anyone/everyone else's “mysecrets” files:

```
less /home/*/mysecrets
```

If they have protected their files correctly you won't be granted access (unless you access the file before they set permissions correctly).

You may wish to let them know that you have access to their secrets.

Test whether you have correctly set permissions. Either by asking a classmate to confirm that they cannot access the file, or by changing to another user:

```
su - student

less /home/yourusername/mysecrets

exit
```

Create a file "~/myshare", and grant everyone read-write access.

Test whether you have correctly set permissions.

Create "mygroupshare", grant only read access to everyone in your group.

Test whether you have correctly set permissions.

Create a new group called "staff", and create a file that you and a fellow classmate (other user) can collaborate on (both edit).

You can work on this problem with a classmate.

Hint, you should both be added to the group. You may require root access for this task. If you do not have root access to the system you are currently sshed to, you may have to complete this on your local system, after exiting the ssh connection. Your classmate may want to ssh to your system.

Test whether you have correctly set permissions. Both users should be able to edit the file, yet other users should not have write access.

Read and write to the shared files created by others, for example:

```
cat /home/dropbear/myshare
```

Challenge:

```
mkdir test

touch test/test1 test/test2 test/test3
```

Use a single chmod command to recursively set the permissions for all files contained in the new "test" directory.

Hint: "man chmod"

umask

Remember that our newly created file started with permissions that meant that everyone could read the file. This can be avoided by setting the ***u**ser file-creation mode **mask** (umask)*. Every process has a umask: an octal that determines the permissions of newly created files. It works by removing permissions from the default "666" for files and "777" for new executables (based on a logical NOT). That is, a umask of "000" would result in new files with permissions "666". A umask of "022" (which is the default value) gives "644", that is "rw- -r- -r-".

The `umask` Bash built in (or system call) can be used to set the umask for the current process.

Check the current umask value:

```
umask
```

Using the `umask` builtin command, set your umask so that new files are only rw accessible by you (but not to your group or others):

```
umask XXX
```

Where XXX is the new umask to use.

Test your new umask value by creating a new file and checking its permissions:

```
touch newfilename
```

```
ls -l newfilename
```

Do the permissions read "rw-------"? If not, change the umask and try again.

Figure out how to (and do) make that setting apply every time you login.

Hint: try Googling bash login or startup config

What would be a safe umask for a shared Linux system? Justify your decision.

Set UID (SUID)

Sometimes a user needs to be able to do things that require permissions that they should not always have. For example, the `passwd` command is used to change your password. It needs read and write access to /etc/shadow. Clearly not every user should have that kind of access! Also, the ping command needs raw network access...

Again not something that every user can do. The Unix solution is *set UID (SUID)*.

Using SUID, processes can be given permission to run as another user. For example, when you run `passwd`, the program actually runs as root (on most Unix systems).

In fact, every process actually has multiple identities, including:

- The real UID (RUID): the user who is running the command
- The effective UID (EUID): the way the process is treated

List all processes real and effective UIDs:

```
ps -eo ruser,euser,comm
```

Look through the list of running processes. Most of the time the RUID and EUID matches. Are there any cases in the output where they do not?

Take a look at how the effective UID is specified:

```
ls -l /usr/bin/passwd
```

```
-rwSr-xr-x 1 root shadow 81792 Oct 29 18:26 /usr/bin/passwd
```

The “s” in the file permissions means that the file UID will be used as the effective UID.

The SUID bit is stored in the first permission octet in the inode:

```
stat /usr/bin/passwd
```

Note the octet representation of the file permissions.

Start another Konsole, and open multiple tabs.

In one tab run:

```
passwd
```

(dont type anything, instead leave the prompt open)

Switch to another tab and run:

```
ps -o ruser,euser,comm -C passwd
```

Make sure you understand the significance of the ruser and euser outputs not matching.

Run this command to find all SUID programs on the system:

```
find / \( -perm -004000 -o -perm -002000 \) -type f -print
```

Identify why each one requires SUID permissions.

Hint: look up each command using man.

Writing a SUID program in C

You are going to create a SUID program, to grant access to the contents of your "mysecret" file to anyone who runs the program, without sharing direct access to the file.

Make sure "~/mysecrets" is only accessible by the owner: ls -la should show "rw-------" for that file.

Create a C program by making a new file "accessmysecret.c":

```
vi accessmysecret.c
```

Remember, vi is modal. Press "i" to enter insert mode, then enter this code:

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

int main()
{
  printf("        UID               GID \n"
      "Real       %d  Real    %d \n"
      "Effective %d  Effective %d \n",
      getuid (),  getgid (),
      geteuid(),  getegid());

  FILE *fp = fopen("mysecrets", "r");
  if (fp == NULL) {
      printf("Error: Could not open file");
      exit(EXIT_FAILURE);
  }
  char c;
  while ((c=getc(fp)) != EOF) {
    putchar(c);
  }
  putchar('\n');
  return EXIT_SUCCESS;
}
```

Save your changes and quit (Esc, ":wq").

Compile the program (which uses the C code to create an executable):

```
gcc accessmysecrets.c -o accessmysecrets
```

Set the permissions for the file (using chmod) to setuid:

```
chmod u+s accessmysecrets
```

Check the permissions include SUID:

```
ls -l accessmysecrets
```

Run the program:

```
./accessmysecrets
```

Note that the program outputs its real and effective identity.

Change to another user, and execute the program:

```
/home/yourusername/accessmysecrets
```

Note that the effective ID is that of the owner of the program. You should also see the contents of the mysecrets file, even though you don't have access to the secrets file directly.

Think about the security of this solution. How secure is it? Would it be safe for root to be the owner of this program? Why not?

Challenge: switch to another user and use the SUID `accessmysecrets` program to get read access to any one of the owner user's files!

Hint: there is a security problem with this code.

Another hint: think about hard links.

Solution:

SPOILER ALERT! SPOILER ALERT! SPOILER ALERT!

There is a security problem caused by not using an absolute filename when opening the file, it opens "mysecrets" rather than "/home/user/mysecrets". Remember, any user can create a hard link to a file (therefore they can make a "copy" of the SUID program wherever they like).

Make a hard link to the SUID program in a directory that the attacker can write to, then also make a hard link to any file the SUID user owns, and name it "mysecrets" in the same directory as the program, then when you execute the program it will write out the contents of the file.

You can exploit this vulnerability as follows:

```
su - student

ln /home/user/accessmysecrets /tmp/access

ln /home/user/someotherfile /tmp/mysecrets

/tmp/access
```

SPOILER ALERT! SPOILER ALERT! SPOILER ALERT!

Challenge: Modify the program to correct the above vulnerability.

Challenge: Modify the program so that only the first line of the mysecrets file is displayed to others.

Challenge: Modify the program so that the script checks the UID and only continues for a specific user (for example, if the user is root).

Hint: "man getuid"

Linux Extended ACLs

We have explored standard Unix permissions. Modern Linux systems (and some other Unix-based systems) now have more complete (and complicated) ACL support.

As previously mentioned, an *access control list (ACL)* is attached to an object (resource) and lists all the subjects (users / active entities) that are allowed access, along with the kind of access that is authorised.

Set a file ACL on your mysecrets file, using the `setfacl` command:

```
setfacl -m u:student:r ~/mysecrets
```

This grants the "student" user read access to the file.

Note that the `stat` program is not usually ACL aware, so won't report anything out of the usual:

```
stat ~/mysecrets
```

The `ls` program can be used to detect File ACLs:

```
ls -la ~/mysecrets
```

```
-rw-r-----+ 1 cliffe users 22 Feb 28 11:47 mysecrets
```

Note that the output includes a "+". This indicates an ACL is in place.

Use `getfacl` to display the permissions:

```
getfacl ~/mysecrets
```

Use Linux File ACLs to grant one or more specific users (other class members) read access to your mysecrets file.

Using ACLs, grant any other group (you choose) read-write access to your mygroupshare file.

Remove the group permission you just added.

Example: `setfacl -x g:staff file`

Conclusion

At this point you have:

- Learned about file permissions, hard links, and inodes
- Learned about octal representations of permissions

- Changed Unix file permissions to grant access to specific users and groups, using chmod
- Used umask to change the permissions applied to new files
- Learned about Set UID (SUID), become more familiar with C, and compiled a SUID C program
    - You may have also done some more programming of your own, (congratulations, if you spotted and/or fixed the security problem)
- Used Linux Extended ACLs to configure more advanced security policies

Well done!