Securing File Access Permissions on Linux

Ramses van Zon
SciNet HPC, University of Toronto
27 October 2022

Digital Research Alliance of Canada | Alliance de recherche numérique du Canada

Motivation

- The Linux operating system has built-in tools to control file access.
- This controls which specific users and groups can access which files and directories.
- Very powerful, but needs to be use properly.
- We will teach you
  - what these Linux permissions are;
  - how to use the available tools to control access and sharing;
  - and how to avoid common security pitfalls.

Thanks to

- Jaime Pinto and Ram Sharma from SciNet
- Mohamed Jabir, Darcy Quesnel and Pier-Luc St-Onge from Calcul Québec.

Understanding Linux Permissions

User organization in Linux

- Users in Linux are organized in groups, so called posix groups.
- A user can be part of several posix groups.
- This is very useful if you want to share files with a specific set of users (e.g. your research group).

Try this: Find out what groups you're part of!

```
$ whoami
rzon
$ groups
scinet ccstaff
$ id -gn
scinet
```

The result of `id -gn` is your primary group.

Ownership in Linux

The purpose of these groups is to control access and sharing.

- All files and directories are always ‘owned’ by a specific user.
- In addition, files have a ‘group ownership’ property, which shows to which group they belong.

Try this: Find out what group your files belong to.

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Oct 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Oct 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Oct 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Oct 11 10:08 slide2.pdf
```

This is a long listing (`-l`) of the current directory (`.`) showing all (`-a`) files and directories (even hidden ones).

Understanding the output of `ls -al`

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Oct 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Oct 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Oct 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Oct 11 10:08 slide2.pdf
```

This is a table with one row per file or directory with the following fields:

1. Cryptic-looking permission strings that will be explained soon;
2. A number showing how many places in the file system link to it;
3. The owner of the file or directory;
4. The group membership of the file or directory;
5. The size of a file;
6. Its date of last modification;
7. Its name.

drwxrwxrwx (i.e. file permissions)

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Oct 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Oct 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Oct 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Oct 11 10:08 slide2.pdf
```

There are 10 characters in the first column that represent the permissions set for this file.

The first character is `d` is it's a directory, `l` if it's a link, otherwise `-`.

The next three are the permission for the user, i.e., the owner.
They can be `rwx`, for read, write, and execute permission.
If a permission is not "set", the character at its position is `-`.

The following three are the permissions for members of the group to which the file belongs.

The final three are the permissions for others.

drwxrwxrwx (i.e. file permissions)

```
$ ls -al .
total 24
drwxrwxr-x 2 rzon scinet  4096 Oct 11 00:17 .
drwxrwxr-x 4 rzon scinet  4096 Oct 11 00:17 ..
-rw-rw-r-- 1 rzon scinet  6251 Oct 11 10:07 slide1.pdf
-rw-rw-r-- 1 rzon ccstaff 8245 Oct 11 10:08 slide2.pdf
```

Understand what different users are allowed to do with these files.

- `.` and `..` are directories (in fact, the current directory and its parent). They can be written to, read from, and executed by the user, or anyone in the `scinet` group. Others can only read and execute.
- `slide1.pdf` is not a directory, and cannot be executed, but can be written to and read from by `rzon` and members of the `scinet` group, and read by others.
- Similar for `slide2.pdf`, but writing is restricted to members of the `ccstaff` group.

Let's try another example

```
$ ls -l /
drwxr-xr-x   4 root root     0 Oct 11 00:51 cvmfs/
drwxr-xr-x  18 root root  3200 Oct  6 20:00 dev/
drwxr-xr-x  93 root root  4340 Oct  4 18:20 etc/
drwxrwxrwx   1 root root    14 Oct  4 18:11 home/
lrwxrwxrwx   1 root root     7 Oct  4 18:09 lib -> usr/lib/
drwxr-xr-x   2 root root    40 Apr 11  2018 media/
drwxr-xr-x   2 root root    40 Apr 11  2018 mnt/
drwxrwxrwx   1 root root    17 Oct  4 18:11 project/
dr-xr-x---   3 root root   260 Oct  7 14:39 root/
drwxr-xr-x  23 root root  1060 Oct  4 18:20 run/
lrwxrwxrwx   1 root root     8 Oct  4 18:09 sbin -> usr/sbin/
drwxrwxrwx   1 root root    17 Oct  4 18:11 scratch/
drwxr-xr-x   2 root root    40 Apr 11  2018 srv/
dr-xr-xr-x  13 root root     0 Oct  4 18:11 sys/
drwxrwxrwt 516 root root 26100 Oct 11 00:55 tmp/
drwxr-xr-x  14 root root   300 Oct  4 18:09 usr/
drwxr-xr-x  21 root root   500 Oct  4 18:11 var/
```

Who’s this “root” person?

Who's this ``root"?

- `root` is the all-powerful administrative account.
- That account owns all of the operating system files.
- Other users can access these files and directories because of the permissions set.
- Access to the account is controlled though the `sudo` command.
- On a shared system, you will never have access to the root account or the `sudo` command.
  (in fact, many of the Alliance staff cannot either).

Controlling Ownership and Permissions

How do files get their ownership?

When a user creates a file:

- Its owner is that user.
- Its permissions are those set by the umask.
- Its group is the user's primary group

  or...

  the group of the directory containing the file,
  if its set group id a.k.a. setGID is set.
  (sometimes called the "sticky bit", but that is technically incorrect.)

Try this

```
$ echo 'echo hello world!' > newfile.sh
$ ls -l newfile.sh
-rw-r--r-- 1 rzon scinet 18 Oct 11 10:26 newfile.sh
$ umask -S
u=rwx,g=rx,o=rx
```

Changing Permissions with chmod

```
$ echo 'echo hello world!' > newfile.sh
$ ls -l newfile.sh
-rw-r--r-- 1 rzon scinet 18 Oct 11 10:26 newfile
$ umask -S
u=rwx,g=rx,o=rx
```

This file cannot be executed despite the umask
(umask is a restriction not a prescription):

```
$ ./newfile.sh
bash: ./newfile.sh: Permission denied
```

With `chmod +x` we can change this:

```
$ chmod +x newfile.sh
$ ls -l newfile.sh
-rwxr-xr-x 1 rzon scinet 18 Oct 11 10:26 newfile
$ ./newfile
hello world!
```

Note that my umask caused everyone to get execution permissions.

More specific `chmod` commands

Add permissions (with umask):

```
$ chmod +r newfile.sh
$ chmod +w newfile.sh
$ chmod +rxw newfile.sh
```

Adding permissions for owner

```
$ chmod u+r newfile.sh
$ chmod u+w newfile.sh
$ chmod u+rxw newfile.sh
```

Adding permissions for group

```
$ chmod g+r newfile.sh
```

Adding permissions for other

```
$ chmod o+r newfile.sh
```

Removing permissions (with umask):

```
$ chmod -r newfile.sh
$ chmod -w newfile.sh
$ chmod -rxw newfile.sh
```

Removing permissions for owner

```
$ chmod u-r newfile.sh
$ chmod u-w newfile.sh
$ chmod u-rxw newfile.sh
```

Removing permissions for group

```
$ chmod g-r newfile.sh
```

Removing permissions for other

```
$ chmod o-r newfile.sh
```

Numeric options for chmod

Each character in the permission string is on or off, so it's a bit.

The three characters for rwx can be seen as a 3-bit number, where

```
0 -> no permissions
1 -> r
2 -> w
3 -> rw
4 -> x
5 -> rx
6 -> wx
7 -> rwx
```

Play around and see the net effect

```
$ chmod 755 FILENAME
$ chmod 655 FILENAME
$ chmod 777 FILENAME
```

The umask can also be given in this numerical format.

```
$ umask 0027
$ echo "Hello" > a.txt
$ umask 0077
$ echo "World" > b.txt
$ ls -l a.txt b.txt
-rw-r-----. 1 rzon scinet 6 Oct 11 08:35 a.txt
-rw-------. 1 rzon scinet 6 Oct 11 08:35 b.txt
```

Changing group ownership with `chgrp`

- Selective sharing:

```
$ chgrp GROUP NAME
```

where `NAME` is the name of a file or directory
and `GROUP` is the name of a group of which you are part.

Afterwards, the group permissions of NAME are applied as pertaining to members of `GROUP`.

- Whole-sale sharing:

```
$ chgrp -R GROUP DIR
```

This applies group membership recursively to all files in `DIR`.

However, this does not apply to files created afterwards in that directory.

Future sharing

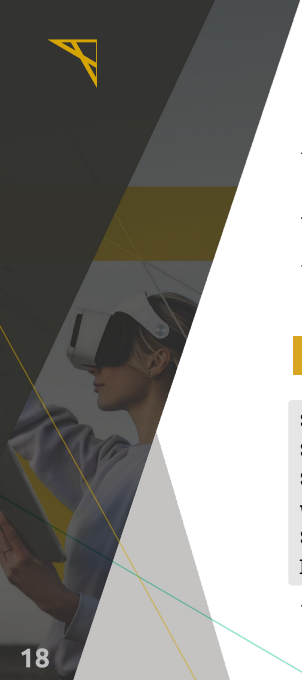
```
$ chgrp GROUP DIR
$ chmod g+s DIR
```

This sets the setGUID bit.

Newly created content in DIR now inherits the group membership.

Try it!

```
$ mkdir shrdir
$ chgrp ccstaff shrdir
$ ls -ld shrdir
drwxr-x--- 2 rzon ccstaff 4096 Oct 11 08:52 shrdir
$ echo "hi" > shrdir/file1
$ chmod g+s shrdir
$ ls -ld shrdir
drwxr-s--- 2 rzon ccstaff 4096 Oct 11 08:52 shrdir
$ echo "there" > shrdir/file2
$ ls -l d
-rw-------. 1 rzon scinet  3 Oct 11 08:58 file1
-rw-------. 1 rzon ccstaff 6 Oct 11 08:58 file2
```



Permissions down a directory tree

It is not enough to give file access to a user or group.

Access must also be given to allow that user or group to descend down into the parent directory of that file, and its parent, and so on.

The “descending into” bit for directories is the “executing” bit for files.

Without execution permission on a directory, there is no access to files or subdirectories of that directory.

Example

```
$ mkdir newdir
$ echo "hello" > newdir/world
$ ls newdir
world
$ cat newdir/world
hello
```

```
$ chmod -x newdir
$ ls newdir
world
```

```
$ cat newdir/world
cat: newdir/world: Permission denied
```

The issue is not the permissions of `world`, but of `newdir`!

Sharing with other users

- What if you need to share with users outside your group, or even with selected users in your group?
- The standard “user, group, other” Linux permissions do not suffice for that.
- You need to use Access Control Lists.

Access Control Lists

- ACL are traditional Windows file permissions that can be overlaid on Linux files.
- They allow more granulated per-user access control.
- The commands differ per file system:

For Lustre and local Linux file systems (Cedar,Graham,Béluga,Narval)

- `getfacl` to see the ACL permissions of FILENAME
- `setfacl` to set PERMISSIONS on FILENAME

For GPFS (Niagara,Mist)

- `mmgetacl` to see the ACL permissions
- `mmputacl` to alter them
- `mmdelacl` to remove any previous added ACL

ACL Examples for Lustre file systems

```
$ setfacl -m u:jdoe:rx my_script.py
```

- Gives user `jdoe` read and execute permissions to `my_script.py`.

```
$ setfacl -d -m u:jdoe:rwX /home/USER/projects/def-PI/shareddata
```

- Sets default access rules to directory `/home/USER/projects/def-PI/shareddata`, so any file or directory created within it inherits the same ACL rule. Required for new data.
- The X attribute above (compared to x) sets the **execute** permission only when the item is already executable.

```
$ setfacl -R -m u:jdoe:rwX /home/USER/projects/def-PI/shareddata
```

- Sets ACL to directory `/home/USER/projects/def-PI/shareddata` and all its current content. So it is applicable only to existing data.

```
$ setfacl -bR /home/USER/projects/def-PI/shareddata
```

- Removes the ACLs in the `shareddata` directory recursively.

ACL Examples for GPFS file systems

`mmputacl` works slightly differently:

- does not have a recursive option.
- requires a permissions file, e.g.

```
user::rwxc
group::----
other::----
mask::rwxc
user:USER:rwxc
user:PI:rwxc              #read and WRITE permissions to group PI
group:OTHERGROUP:r-xc     #read-only for members of OTHERGROUP
```

Because of the lack of a recursive option, you need do down the tree, e.g.

```
$ mmputacl -i permissions.acl /project/g/group/owner
$ mmputacl -i permissions.acl /project/g/group/owner/dir1
$ mmputacl -i permissions.acl /project/g/group/owner/dir1/subdir2
$ mmputacl -d -i permissions.acl /project/g/group/owner/dir1/subdir2
```

In the last command, the `-d` means setting the default permissions for

Common Permission Pitfalls and Security Risks

Overly permissive permissions

Anytime permissions don't work, it is tempting to 'just' run `chmod 777`.

But this opens permissions completely to ***anyone***, which is undesirable, particularly on shared systems.

- Anyone can now read, change, delete or add files and directories to your account.
- You might not even be able to log in anymore.

Be careful and diligent!

Do not give away any more permissions than is necessary.

Confusing chmod with chown or chgrp

`chmod` changes permissions of a file or directory.

`chgrp` changes group membership of a file or directory.

- This changes to which group the permissions set by chmod apply.
- It also changes towards which quota the file or directory counts.

`chown` would change the ownership of a file.

- Only the `root` user can do this, or a user with `sudo` powers.
- Not possible on any on the Alliance clusters.

Trying to set permissions on other's files

- Sometimes, users try to change permissions or apply ACL to files that they do not own.
- That won't work. Only the owner of the file can do this.

So if you need to gain access to files of a collaborator, or of a student, you need to ask them to change the permissions.

Note

Files in project are, by default, in a group associated with an allocation.

On the General Purpose clusters (Cedar, Graham, Béluga, and Narval), by default, files in $HOME and $SCRATCH are in a group with the user as the only member. Thus, other research group member do not have access.

On the Large Parallel cluster Niagara, as well as on Mist, by default, files in $HOME and $SCRATCH are in the group of your sponsor, and are readable by others also in the research group of that sponsor.

Permissions do not work along directory tree

- When you open permissions at a particular sub-level (chmod or ACL), but overlook the levels above, which are oftentimes owned by the PI, the net result is that the permissions still don't work.
- Don't overreact in the worst possible way, and run `chmod -R 777`.

Set the `x` permission of all parent directories, or, if owned by someone else, ask them to set that permission.

Setting permissions on .ssh

- Giving bulk `+r` permission to $HOME will expose contents of $HOME/.ssh
- This is potentially very dangerous as it may contain private ssh keys.
- It can also prevent you from using `ssh`

Resist the urge to given bulk permissions!

Detecting and Fixing Permission Down a Tree

- Users may run the command below to find files/directories with permissions 777 (rwxrwxrwx)

```
$ find -perm -2 -not -type l
```

(search and show, world-writable files under current directory)

- Recursive ways to use chmod, e.g.

```
$ chmod -R o-rwx directory_name
```

- Recursive ways to use chgrp, e.g.

```
$ chgrp -R ccstaff directory_name
```

References

References

- https://docs.alliancecan.ca
- https://docs.alliancecan.ca/wiki/Sharing_data
- https://docs.scinet.utoronto.ca/index.php/Data_Management (Niagara/GPFS specific)