Linux Basics 3: Users, Groups, & Permissions

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- The Filesystem (directories and files)
- **Users, Groups, and File Permissions**
  - **users and groups**
  - **usernames vs. UIDs**
  - **group names vs. GIDs**
  - **the root user**
  - **user/group commands**
  - **file access permissions**
  - **directory permissions**
  - **"special" permissions**
  - **permissions commands**
  - **extra: access control lists**
- Processes (running programs and their attributes)

Users and Groups

Linux is a **multiuser OS**, so every user has a **username** and a login **password**.

Actually, each user is ultimately identified by a *non-negative integer* called the **user ID** (**UID**).

In addition to users, there are entities called **groups**, which represent subsets of the users ("group members").

While each group has a *group name*, a group is ultimately identified by a non-negative integer called the **group ID** (**GID**).

Users and Groups (contd.)

Each user must be a member of at least one group, called the user's **primary group**.

A user may be a member of any number of additional groups, called **supplementary groups**.

Groups provide a mechanism for setting access permissions for a subset of users (though they can provide only simple multiuser access control for files).

UIDs, GIDs, and passwords form the basis for much of Linux' *security*, in conjunction with **file access permissions**.

root

The special user **root** is the **administrator** (also referred to as the **superuser**).

root always has UID of 0 (zero).

There is also a **root group** with GID of 0.

Only root has permission to modify most system files (so normal users cannot damage these files).

root has privileges that all other users lack:
access any file, change file ownership, change scheduling/nice priorities, send signals to any process (to terminate it), etc.

Key User/Group Commands

id – print real and effective user and group IDs:

- `id`

groups — print the groups a user is in

- `groups carver`

chown – change file owner and group:

- `chown carver ~/cs306`
- `chown carver:faculty test.c lab1.c`

passwd – change user's password:

- `passwd`

Key User/Group Commands (contd.)

useradd – add a new user:

- `useradd -u 5001 -g faculty carver`

groupadd – add a new group:

- `groupadd -g 1001 gurus`

usermod – modify user information:

- `usermod -aG gurus carver`

File Access Permissions

Every file (regular or special) has an associated *user* (called the file's **owner**) and an associated *group*.

Every file includes **access permissions** for each of *three* **user classes**:

1. **owner**: the file user/owner
2. **group**: members of the file group (excluding owner)
3. **other**: all other users

Only the *most user-specific permissions apply*:
e.g., if a user is the owner, then only the owner permissions apply even if others are less restrictive

File Access Permissions (contd.)

There are *three* **file access permissions** for each of the three user classes: (1) **read**, (2) **write**, (3) **execute**.

For *regular files*, permissions meanings are as expected:

1. **read** means the ability to *view* a file's contents
2. **write** means the ability to *modify* a file's contents (this includes being able to *"truncate"* the file—but note that it does not allow file *deletion*)
3. **execute** means the ability to invoke the file as a "command" and have the kernel run it in a process

Each permission is *Boolean/binary*, so a file's access permissions can be represented using *9 bits*.

Directory Permissions

For *directories*, permissions meanings are slightly different:

1. **read** means the ability to *list out* the files/directories that are contained in the directory
2. **write** means the ability to modify the directory by *creating* new files/directories in it or *deleting* contained files/directories
3. **execute** means the ability to set the directory as the *"current working directory"* or to "pass through" this directory to reach one of its subdirectories

Execute permission is also called **enter** or **search** permission with directories, because of its meaning with directories.

Execute "pass through" permissions requirements for file access are often not fully understood.

Directory Permissions (contd.)

In order to access a file:

- the file must have appropriate permissions to allow the access
- *all directories in the file path* must have execute permission

For example, to modify the file `/home/carver/Documents/test.text`:

- *write permission* will be needed on the *file*
- *execute permission* will be needed on all these *directories*:
  - `/`
  - `/home`
  - `/home/carver`
  - `/home/carver/Documents`

(Appropriate file permissions settings will depend on the file's owner/group and the UID/GID of the accessing program/user.)

"Special" Permissions

There are actually three other "special" permissions bits:

1. **set UID** (**SUID**)
2. **set GID** (**SGID**)
3. **restricted deletion flag** (also known as the **sticky bit**)

The SUID/SGID bits affect the user/group that are associated with an executing program/command:

- SUID permission causes the program's EUID to be set to the file's owner/UID
- SGID permission causes the program's EGID to be set to the file's group/GID

"Special" Permissions (contd.)

The SUID/SGID permissions are used to create programs that run with "**elevated privileges**" (i.e., have access permissions that the user running the program lacks).

For example, the `passwd` command is "**SUID root**."

This allows a normal user to change his password even though he lacks permission to modify the password file.

The **restricted deletion flag** on a *directory* prevents non-owners from deleting files (even if they have write permission for the directory).

This is useful on **world writable** directories like `/tmp`.

Key File Permissions Commands

ls – list file names plus file information:

- `ls -l test.text`

  ⇒ `-rwxrw-r-- 1 carver faculty 297 Mar 16 13:17 test.text`

  (owner carver has r+w+x permissions, faculty group members have r+w, and all other users have only r)

chmod – change file permissions (mode):

- `chmod 760 test.text`
  (760 is *octal*, equals 111 110 000 binary,
  meaning `rwx rw- ---` for permissions)
- `chmod u=rwx,g=rw,o= test.text`
- `chmod u+x,g-w,o-r test.text`

Extra: Access Control Lists

A key advantage of the basic Linux/UNIX file permissions model is that it is easy to understand, so it is widely used.

By contrast, the permissions model in NTFS on Windows is much more flexible, but also so complex that few users fully understand it, and so few use it effectively.

The key limitation of the basic Linux permissions model is that it is not practical for giving different access permissions to different subsets of users for one file.

Modern Linux filesystems support the use of **access control lists** in conjuntion with file permissions.

With **ACL**s, particular access permissions can be granted to any number of individual users and groups for any file.

Access Control Lists (contd.)

To be backwards compatible with the traditional permissions model, the semantics for Linux ACLs is somewhat complicated.

The two main capabilities ACLs add for file permissions are:

- rwx permissions can be specified directly for any users (without using groups).
- A single **mask permission** can be used by a file's owner to set the *upper bound* for all non-owner permissions.

The commands for getting and setting file ACLs are: `getfacl` and `setfacl`.

ACLs make use of filesystem **extended attributes** (EAs/xattr's), which may have to be enabled when mounting the target filesystem.