File Permissions

CIS 2230 Linux System Administration

Lecture 13

Steve Ruegsegger

Review – users

- What is the su command used for?
- What is the uid?
- What is uid 0?
- What command allows you to run as root for 1 command?
- How do you get a root shell prompt using sudo?
- What is the command to change your password?
- What makes a user “a user”?
- What are the 3 things that happens when you make a new user?
- When should you give your pw to a sys admin to help you out?

File permission introduction

- An advantage of Linux = designed for multiple users from the start
- This provided security for the OS and the users.
  - The users are protected from each other
  - The OS is protected from all users
- The OS has built-in file permissions and will only allow actions on files when permitted.
- 3 Actions: read, write, execute
- 3 ids for those actions: user, group, other (a.k.a. everyone or world)
- "3 groups of 3"
- Per id, the actions are combined into "modes"

Permission bits for Files

- Three types of permissions on files, each denoted by a letter
- A permission represents an action that can be done on the file:

Permission (mode)	Letter	Description
Read	r	Permission to read the data stored in the file
Write	w	Permission to append data to the file, to truncate the file, or to overwrite existing data
Execute	x	Permission to attempt to execute the contents of the file as a program

- Occasionally referred to as 'permission bits'
- Note that for *scripts*, you need both execute permission and read permission.

Permission bits for Directories

- The r, w, and x permissions also have a meaning for directories
- The meanings for directories are slightly different:

Permission	Letter	Description
Read	r	Permission to get a listing of the directory
Write	w	Permission to create, delete, or rename files (or subdirectories) within the directory
Execute	x	Permission to change to the directory, or to use the directory as an intermediate part of a path to a file

- The difference between read and execute on directories is specious — having one but not the other is almost never what you want (except when it is).

3 groups of 3

- These bits are in groups of 3
  - A `rwx` set for user, group, world
  - 3 groups x 3 bit = 9 bits (*duh*)
- The ls -l command allows you to look at the permissions:

```
$ ls -la
total 1284                    uid     gid
drwxr-xr-x  2 steve steve    4096 2011-09-15 21:52 .
drwxr-xr-x 31 steve steve    4096 2011-09-15 21:52 ..
-rw-------  1 steve steve   16005 2011-09-04 15:33 who.txt
-rw-------  1 steve steve   19938 2011-09-04 15:33 w.txt
-rw-------  1 steve steve  224268 2011-09-04 15:33 psef.txt
-rw-r--r--  1 steve steve   99982 2011-09-04 16:11 pidkill.ae.gz
-rw-r--r--  1 steve steve   97421 2011-09-04 16:11 pidkill.ad.gz
```

dir

Other rwx permissions

Group rwx permissions

User rwx permissions

What does $ ls -n do?

“modes” from permission bits

- User can do everything, everyone can only read

```
-rwxr--r-- 1 steve steve  1234 2011-09-04 16:11 foo.bar
```

- Everyone can do everything

```
-rwxrwxrwx 1 steve steve  1234 2011-09-04 16:11 foo.bar
```

- Only reading for everyone

```
-r--r--r-- 1 steve steve  1234 2011-09-04 16:11 foo.bar
```

- Only the user can read & write

```
-rw------- 1 steve steve  1234 2011-09-04 16:11 foo.bar
```

Change modes: chmod

- The `chmod` command changes the permissions of a file or directory
- *A file's permissions may be changed only by its owner or by the superuser*
- `chmod` takes an argument describing the new permission mode
- There are two ways to define the new mode
  - *Symbolic*: <id>[+-=]<action>
  - *Octal*: 755

Symbolic chmod

- Symbolic format – set of 3 symbols:
  `[ugoa][+=-][rwxXst]`
- *First* letter indicates who to set permissions for:
  - u for the file's owner
  - g for the group owner
  - o for other users (not in first 2, a.k.a. world/everyone)
  - a for all 3 sets
- *Second* char is an action
  - = sets permissions for files (remove any other bits for that user)
  - + adds permissions to those already set,
  - - removes permissions
- *Third* letters indicate which of the r, w, x permissions to set
  - Usually r, rw, rwx, or rx

Symbolic chmod

- Add execute for me

```
$ chmod u+x backup.sh
```

- Add read and write for everyone

```
$ chmod a+rw lab1.doc
```

- Only I can read and write (everything else is off)

```
$ chmod u=rw,og= lab1.doc
```

- Remove execute from group and world

```
$ chmod og-rx backup.sh
```

- Add read and write to group, but only read to world

```
$ chmod g+rx,o+r lab1.doc
```

Changing the Permissions recursively

- A common requirement is to change the permissions of a directory and all its contents
- The "X" (capital-X) bit means add the x bit only if some other user also has an x bit. (Nice safety precaution)
- chmod accepts a -R option:

```
$ chmod -R g+rwX,o+rX public-directory
```

  - Adds rwx permissions on public-directory for the group owner, and adds rx permissions on it for everyone else (other or world)
  - And any subdirectories, recursively
  - And any contained executable files
  - Contained non-executable files have rw permissions added for the group owner, and r permission for everyone else

3 special permissions bits

- There are *3 more* permission bits, but no more places to put them!
- Sticky bit
  - A 't' or “T” in the world x spot
- Setuid (SUID)
  - A 's' or “S” in the user x spot
- Setgid (SGID)
  - A 's' or “S” in the group x spot

Sticky bit (a t in world x)

- Used to mean “save text mode,” which would tell the kernel to keep a program in memory
- Not needed any longer in modern Unix kernels
  - (They do this automatically with advanced page systems)
- Now only has meaning for directories.
  - You can put it on a file, but it doesn't “do” anything on a file
- For a directory, the sticky bit means:
  - Anyone can write to that directory,
  - But only the user can delete the file from this directory.
    - (or root, of course)

For example, /tmp

- The `/tmp` directory must be world-writable, so that anyone may create/write temporary files within it
- But that would normally mean that anyone may delete any files within it — obviously a security hole, and an inconvenience
- `/tmp` has the 'sticky' bit set
- Expressed with a t (mnemonic: temporary directory) in a listing for world executable:

```
$ ls -ld /tmp
drwxrwxrwt 30 root root 11264 Dec 21 09:35 /tmp
```

- Enable 'sticky' permission with:

```
$ chmod +t /data/tmp
```

Special Directory Permissions: setgid ('s' in group x)

- Also a bit for directories only
- SGID on a directory means that any files created within it acquire the same group ownership (gid) of the directory
  - Directories created within it acquire both the group ownership and setgid permission also
- Useful for a *shared* directory where all users working on its files are in a given group
- Expressed with an s in 'group' x position in a listing:

```
$ ls -l -d /data/projects
drwxrwsr-x 16 root staff 4096 Oct 19 13:14
/data/projects
```

- Enable setgid with:

```
# chmod g+s /data/projects
```

- Now add a file to that dir and new files will have the inherited group

Special File Permissions: setuid (s in user x)

- Finally, a special bit for files (not dirs)
- When set:
  A process run from a setuid file acquires the user ID of the file's owner (called the *effective* user ID)
- For security, Linux *ignores* the SUID for shell scripts (with shebang).
- Expressed with an s in 'user' x position in a listing
- The primary example is the passwd command. Why?

```
$ ll /usr/bin/passwd
-rwsr-xr-x 1 root root 41284 Apr  8  2012 /usr/bin/passwd
```

- Enable setuid with:

```
# chmod u+s /usr/local/bin/program
```

Displaying these special bits -- Review

- **setuid**: (file) *run* this file as the "owner" (not the $USER)
  - s in the user x position
- **setgid**: (dir) all new files in this dir will have the gid of this dir
  - s in the group x position
- **sticky** bit: (dir) only the 'owner' can *delete* a file they are user-owner (even though the dir is world writeable)
  - t in the other x position
- Uppercase vs lowercase
  - Lowercase s or t indicates that the execute bit "underneath" is enabled (i.e., there is an x "behind" the letter)
  - Uppercase S or T indicates that the execute bit "underneath" is disabled

Permissions as Octals

- Sometimes you will find (octal) numbers referring to sets of permissions
- Think binary to decimal:
  - Convert rwx binary to a decimal number
  - if set, consider a 1, else 0
  - r-- = 100b = 4
  - rw- = 110b = 6
  - rwx = 111b = 7
- Combine 3 decimals of user, group & world:
  - rwxrw-r-- = 764
- You may use numerical permissions with chmod:

```
$ chmod 664 *.txt
```

- Is equivalent to:

```
$ chmod ug=rw,o=r *.txt
```

few more examples

- Everyone can read
  - r--r--r-- = 444
- User can read and write
  - rw-r--r-- = 644
- Everyone can read and write
  - rw-rw-rw = 666
- Everyone can exectute
  - rwxrwxrwx = 777

Default file Permissions: umask

- The `umask` command allows you to set the default permissions on newly created files and directories
- `$ umask` returns the current umask value
- `$ umask <octal>` defines a new umask for that shell only
- How it works:
  - It masks the bits to set
  - i.e., it defines the bits to make sure are “off” in the new file permissions

How umask works

	ex 1 file	ex 2 file	ex1 dir	ex1 dir
base perm	666	666	777	777
umask	022	002	022	002
default file	644	664	755	775
permissions	`rw-r--r--`	`rw-rw-r--`	`rwxr-xr-x`	`rwxrwxr-x`
	more strict	less strict		

- The 'base permission' for a file is 666 (or rw-rw-rw-)
- The 'base permission' for a dir is 777 (or rwxrwxrwx)
- Subtract the umask from the default
- The result is the default file (or directory) permissions
- You can convert the triplet decimal number to the individual file permission bits if you want (e.g. `$ umask ug=rwx,o=rx`)
- Notes:
  - You *cannot* use umask to make new files executable.

umask in Ubuntu

- Ubuntu stores the umask as a 4 char octal
  - The “first” zero is the special sticky, setuid an setgid bits.

- The default umask definition
  - Often stored for all users is set in `/etc/profile`
  - For Ubuntu, it's part of PAM and set in `/etc/login.def`
  - Users often overwrite it in `~/.profile`

Changing File Ownership with chown

- The `chown` command changes the ownership of files or directories
- Simple usage:

```
$ sudo chown aaronc logfile.txt
```

  - Makes logfile.txt be owned by the user aaronc
- Specify any number of files or directories
- *Only the superuser can change the ownership of a file*
  - This is a security feature — quotas, set-uid

Changing File Group Ownership with chgrp

- The chgrp command changes the group ownership of files or directories
- Simple usage:

```
$ chgrp staff report.txt
```

  - Makes staff be the group owner of the file logfile.txt
- As for `chown`, specify any number of files or directories
- The superuser may change the group ownership of any file to any group
- The owner of a file may change its group ownership
  - But only to a group of which the owner is a member

Changing the Ownership of a Directory and Its Contents

- A common requirement is to change the ownership of a directory and its contents
- Both chown and chgrp accept a -R option:

```
$ sudo chgrp -R staff shared-directory
```

- Mnemonic: ‘recursive’
- Changes the group ownership of shared-directory to staff
  - And its contents
  - And its subdirectories, recursively
- Changing user ownership (superuser only):

```
$ sudo chown -R root /usr/local/share/misc/
```

2 birds with 1 stone

- The `chown` command can *also* change the group. Just use a colon

```
$ sudo chown steve:steve ~/home/project/*
$ sudo chown fred:students ~/home/fred
$ sudo chown 1003:1000 ~/home/susie
```