04 Users and Permissions

CS6 Practical System Skills
Fall 2019
Leonhard Spiegelberg lspiegel@cs.brown.edu
04.01 Permissions

UNIX is a multi-user system.

How do you protect files from other users, the world?
How do you share files with other users?
How do you protect one from oneself?
On a system you'll find different *logical users*:

*root* ⇒ the OS account which has unlimited rights

*admin* ⇒ one or more accounts which may perform certain actions with root privileges

*regular users* ⇒ You, me & everyone other human out there

*technical users* ⇒ users created to run deployed programs with restricted privileges.

More on adding users, privileges, … in week 3 and the deployment lab.
How Unix categorizes users...
04.01 Users & Files

Each file is owned by a user ⇒ typically the creator

In addition, each file belongs to a group ⇒ smallest group: the user

owner
creator of the file

group
multiple users

other
public, world
Each file in Unix has 3 permissions:

- **read**: the file can be read, i.e. its contents displayed
- **write**: the file can be modified or deleted
- **execute**: the file can be run (i.e. executables or scripts)
UNIX allows you to set (for each file) separate read/write/execute permissions for each party.
## 04.01 Permissions for directories

Because directories are also files, they have read, write, or execute permissions too. The meaning differs though:

<table>
<thead>
<tr>
<th>permission</th>
<th>file</th>
<th>directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>Allows file to be read.</td>
<td>Allows file names in the directory to be read.</td>
</tr>
<tr>
<td>write</td>
<td>Allows file to be modified.</td>
<td>Allows entries to be modified within the directory.</td>
</tr>
<tr>
<td>execute</td>
<td>Allows file to be executed.</td>
<td>Allows access to contents and metadata for entries in the directory.</td>
</tr>
</tbody>
</table>
How can we get information about the permissions of a file?
04.02 ls -l ⇒ the longformat

owner and user are usually the same! Terms are used interchangeably here often.

ls -l

total 88

-rw-r--r-- 1 sealion friends 14 9 Sep 8:01 file.txt
-rw-r--r-- 1 sealion friends 40390 9 Sep 9:00 penguin.jpg
04.02 Permissions

permission string (10 characters)

<table>
<thead>
<tr>
<th>filetype</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular file</td>
<td>-</td>
</tr>
<tr>
<td>directory</td>
<td>d</td>
</tr>
<tr>
<td>symbolic link</td>
<td>l</td>
</tr>
<tr>
<td>pipe</td>
<td>p</td>
</tr>
<tr>
<td>socket</td>
<td>s</td>
</tr>
<tr>
<td>block device</td>
<td>b</td>
</tr>
<tr>
<td>char device</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>permission</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>r</td>
</tr>
<tr>
<td>write</td>
<td>w</td>
</tr>
<tr>
<td>execute</td>
<td>x</td>
</tr>
</tbody>
</table>
04.03 Setting permissions - chmod

chmod mode file ...

change mode, i.e. set or update file permissions

⇒ only the owner (or root) can run this command for a file
⇒ mode can be either a number (numeric mode) or a combination of symbols
## 04.04 chmod - symbolic mode

<table>
<thead>
<tr>
<th>party</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>u</td>
</tr>
<tr>
<td>group</td>
<td>g</td>
</tr>
<tr>
<td>other</td>
<td>o</td>
</tr>
<tr>
<td>all</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>action</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>add permission</td>
<td>+</td>
</tr>
<tr>
<td>remove permission</td>
<td>-</td>
</tr>
<tr>
<td>set to</td>
<td>=</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>permission</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>r</td>
</tr>
<tr>
<td>write</td>
<td>w</td>
</tr>
<tr>
<td>execute</td>
<td>x</td>
</tr>
</tbody>
</table>

### Example:

```
chmod u=rw,g=rx,o= file.txt
```

This command sets permissions for file.txt to `-rw-r-x---`. Combining multiple statements with `,` allows for specifying multiple permissions at once.
04.05 chmod - numeric mode

Instead of using symbols, chmod can be used with an even short syntax using the following encoding.

<table>
<thead>
<tr>
<th>Octal</th>
<th>Binary</th>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>---</td>
<td>no permissions</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>--x</td>
<td>execute only</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>-w-</td>
<td>write only</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>-wx</td>
<td>write and execute</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>r--</td>
<td>read only</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>r-x</td>
<td>read and execute</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>rw-</td>
<td>read and write</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>rwx</td>
<td>read, write and execute</td>
</tr>
</tbody>
</table>

chmod u=rw,g=rx,o= file.txt => chmod 650 file.txt
04.05 chmod - numeric mode

⇒ combining permissions is adding numbers
4 = read    2 = write    1 = execute

Example: set user read and write permissions only:

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>G</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>rw-</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Binary</td>
<td>110</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>Decimal</td>
<td>6 = 4 + 2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

⇒ chmod 600 file.txt
Consider the following output from `ls -l`:

```
drwxr-xr-x 10 sealion animals 320 28 Nov 2018 lecture02
```

Who owns the file?

What permissions does lecture02 have?

What type of file is lecture02?
Consider the following output from `ls -l`:

```
drwxr-xr-x 10 sealion animals 320 28 Nov 2018 lecture02
```

<table>
<thead>
<tr>
<th>permission</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>r</td>
</tr>
<tr>
<td>write</td>
<td>w</td>
</tr>
<tr>
<td>execute</td>
<td>x</td>
</tr>
</tbody>
</table>
## 04.06 chmod - quiz

Fill out the table, use ? if a permission bit can't be deducted.

<table>
<thead>
<tr>
<th>file.txt permissions before</th>
<th>symbolic mode chmod</th>
<th>numeric mode chmod</th>
<th>file.txt permissions after</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwx---rwx</td>
<td>chmod u=,g=,o= file.txt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-w--w-rw-</td>
<td></td>
<td>chmod 777 file.txt</td>
<td></td>
</tr>
<tr>
<td>rw------</td>
<td></td>
<td>chmod 654 file.txt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>chmod u=x</td>
<td>--xrwxrwx</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>chmod u+r,u-r,u=rw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 = read  2 = write  1 = execute
## 04.06 chmod - quiz

### solutions:

<table>
<thead>
<tr>
<th>file.txt permissions before</th>
<th>symbolic mode chmod</th>
<th>numeric mode chmod</th>
<th>file.txt permissions after</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwx---rwx</td>
<td>chmod u=,g=,o= file.txt</td>
<td>chmod 000 file.txt</td>
<td>---------------</td>
</tr>
<tr>
<td>-w--w-rw-</td>
<td>chmod u+rx,g=rxw,o+x file.txt</td>
<td>chmod 777 file.txt</td>
<td>rwxrwxrwx</td>
</tr>
<tr>
<td>-rw------</td>
<td>chmod u=rw,g=rxw,o=r file.txt</td>
<td>chmod 654 file.txt</td>
<td>rw-r-xr--</td>
</tr>
<tr>
<td>???rwxrwx</td>
<td>chmod u=x file.txt</td>
<td>chmod 177 file.txt</td>
<td>--xrwxrwx</td>
</tr>
<tr>
<td>-----------</td>
<td>chmod u+r,u-r,u=rw file.txt</td>
<td>chmod 600 file.txt</td>
<td>rw----------</td>
</tr>
</tbody>
</table>

4 = read  
2 = write  
1 = execute
04.06 chown/chgrp - changing ownership

Change who owns the file and the group:

chown owner:group file ...
chown owner file ...
chown :group file ...

chown :group is the same as chgrp

Change the group the file belongs to to group.

chgrp group file ...

Example:

touch share_this_file.txt
chown tux:friends share_this_file.txt
ls -l
-rw------- 1 tux friends 0 Sep 12 08:15 share_this_file.txt
When to use which permissions?
04.07 Hiding your files from everyone

(1) Protect your files from everyone else

⇒ u=rwx,g=,o= (700)

⇒ u=rw,g=,o= (600)

(2) Protect your files from everyone else and make sure you don't overwrite them or allow execution (no side effects)

⇒ u=r,g=,o= (400)

Tip: chmod also works with wildcards!
04.07 Commonly used permissions

(3) Only you can modify files, others may still read them

⇒ u=rw,g=r,o=r (644)

(4) Only you have write access, others can get information about & read your files

⇒ u=rwx,g=rx,o=rx (755)
04.07 Commonly used permissions

(5) Only you have read/write access, others may still lookup information on your files but not read them

⇒ u=rwx,g=x,o=x (711)
## 04.08 Default guide to chmod for files

<table>
<thead>
<tr>
<th>Type of Files</th>
<th>Permissions</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>world executables files</td>
<td>u=rwx, g=rwx, o=rx</td>
<td>775</td>
</tr>
<tr>
<td>executables by group only</td>
<td>u=rwx, g=rx, o=</td>
<td>750</td>
</tr>
<tr>
<td>group modifiable files</td>
<td>u=rw, g=rw, o=</td>
<td>660</td>
</tr>
<tr>
<td>world readable files</td>
<td>u=rw, g=r, o=r</td>
<td>644</td>
</tr>
<tr>
<td>group readable files</td>
<td>u=rw, g=r, o=</td>
<td>640</td>
</tr>
<tr>
<td>private files</td>
<td>u=rw, g=, o=</td>
<td>600</td>
</tr>
<tr>
<td>private executables</td>
<td>u=rwx, g=, o=</td>
<td>700</td>
</tr>
</tbody>
</table>

**DON'T USE 777 or 666. These permissions pose security risks!**
chmod for directories:

**DON'T** delete the execute bit on your folders.

Why? => you can not anymore access them using `cd` or `ls`!

⇒ If it happens and **you** own the file,
   you can fix this by `chmod 700 path/`

**USE** 700 (private), 711(traversable) or 755(readable) on directories.

Note: 770 is o.k. for shared folders
Example:

sealion wants to access /home/tux (700) and run cat /home/tux/tux_profile.txt (644)

sealion@server:~$ ls -l /home/tux
ls: cannot open directory '/home/tux': Permission denied

sealion@server:~$ cat /home/tux/tux_profile.txt
cat: /home/tux/tux_profile.txt: Permission denied

Explanation:

/home/tux has permissions 700

⇒ sealion has no read/execute permission, hence ls -l /home/tux produces Permission denied.

⇒ cat /home/tux/tux_profile.txt gives Permission denied too, because the location of tux_profile.txt can't be looked up because of the 700 permission on /home/tux
04.08 chmod for directories

Example:

sealion wants to access /home/tux (711) and run cat /home/tux/tux_profile.txt (644)

sealion@server:~$ ls -l /home/tux
ls: cannot open directory '/home/tux': Permission denied

sealion@server:~$ cat /home/tux/tux_profile.txt
Tux

Explanation:

/home/tux has permissions 711

⇒ sealion has no read so ls fails. However, sealion can cd into /home/tux!

⇒ cat /home/tux/tux_profile.txt works, because sealion can lookup file location for /home/tux.

⇒ 711 useful to allow content access of files but no traversal of directories!
04.08 chmod for directories

Example:
sealion wants to access /home/tux (755) and run cat /home/tux/tux_profile.txt (644)

sealion@server:/home/tux$ ls -l /home/tux/
total 8
-rwxrwxrwx 1 tux tux 538 Sep 11 19:44 tux_profile.txt
-rwx------ 1 tux tux  96 Sep 11 18:41 tux_secret.txt

sealion@server:~$ cat /home/tux/tux_profile.txt
Tux

Explanation:
/home/tux has permissions 755
⇒ sealion read to both dir and file
⇒ 755 allows access & browsing.
What about 777 for directories?
Just Don't.
One more thing...
Special linux permissions
Besides the permission for user/group/other, Linux has 3 special permissions which can be combined:

<table>
<thead>
<tr>
<th>permission</th>
<th>octal</th>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>setuid</td>
<td>4</td>
<td>s</td>
<td>Allows a process to run as the owner of the file, not the user executing it</td>
</tr>
<tr>
<td>setgid</td>
<td>2</td>
<td>s</td>
<td>Allows a process to run with the group of the file, not the group of the user executing it</td>
</tr>
<tr>
<td>sticky bit</td>
<td>1</td>
<td>t</td>
<td>prevents a user from deleting another user's files even if they would normally have permission to do so</td>
</tr>
</tbody>
</table>
04.09 Special file permissions

Examples:

- **chmod +t file.txt** ⇒ sets sticky bit for file.txt
- **chmod g+s file.txt** ⇒ sets sgid bit for file.txt
- **chmod u+s file.txt** ⇒ sets suid bit for file.txt
04.09 Special file permissions

`ls -l` for special permissions:

<table>
<thead>
<tr>
<th>setuid</th>
<th>setgid</th>
<th>sticky bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>permission has S where execute bit x is normally located for <strong>user</strong>, s if execute bit x for user is also set for a file.</td>
<td>permission has S where execute bit x is normally located for <strong>group</strong>, s if execute bit x for group is also set for a file.</td>
<td>permission has T where execute bit x is normally located for <strong>other</strong>, t if execute bit x is also set for a file.</td>
</tr>
</tbody>
</table>

**Examples:**

- `chmod 1611 file.txt`  ⇒  `-rw---x--t`  (sticky bit)
- `chmod 2644 file.txt`  ⇒  `-rw-r-Sr--`  (setgid)
- `chmod 4400 file.txt`  ⇒  `-r-S-------`  (setuid)
- `chmod 7777 file.txt`  ⇒  `-rwsrwsrwt`  (ALL permissions set)
Why are they needed?

**sticky bit:**
⇒ prevents other users from deleting files/directories in a public folder. E.g., /tmp where all users store temporary files.

```
ls -l /
drwxrwxrwxrwt  9 root root  4096 Sep 12 01:47 tmp
```

sticky bit set for /tmp. /tmp has 777 rights!
Why are they needed?

**setuid:**
⇒ passwd allows to change the password for a user. However, passwords need to be stored somewhere in a file. With setuid the program passwd runs with root privileges, but the user has no access to the password file.

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

-rwsr-xr-x 1 root root 59640 Mar 22 19:05 /usr/bin/passwd
```

setuid set for passwd, i.e. passwd runs under root permissions because the program is owned by root!
Why are they needed?

**setgid:**

⇒ Files created in a shared folder which has the setgid bit set will belong to the group the folder belongs to.

```
ls /

drwxrws--- 2 sealion friends 4096 Sep 12 02:05 recipes
```

/recipes is a shared folder between tux and sealion. Files created there will belong to friends!
links
04.10 Links

Links are special files which point to another file (in the wider sense).

```
ln -s target link_name
```

creates a symbolic link `link_name` pointing to `target` (Note the order!)
Example:

Assuming we are in Tux's home directory (pwd ⇒ /home/tux), we could create a shortcut to work with Sealion's directory:

```
ln -s /home/sealion sl
ls -l
lrwxrwxrwx 1 tux tux 14 Sep 12 02:46 sl -> /home/sealion/
```  

Permissions do not matter for the link. It's a pointer. When used, the target's permissions will be checked.

```
cd sl
-bash: cd: sl: Permission denied
```

A link is a pointer, thus you see link_name -> target here!
Advice on links:
The link command is very powerful. If you have any doubt on how to use it, use per default
\texttt{ln -s}.
\textbf{Always check the order first!}
Other options may break your system if you don't know what you're doing.
05 Streams & Pipes

CS6 Practical System Skills
Fall 2019
Leonhard Spiegelberg lspiegel@cs.brown.edu
Single commands are great...

... but how about combining them?
05.01 Streams

Where do commands get their input?

Where do commands send their output?

⇒ two special files where output is sent to and one special file where input is read from:

stdin  stdout  stderr

standard input  standard output & standard error
05.01 Streams

⇒ A stream is a sequence of characters
⇒ Each of the three streams is identified by a unique file descriptor (number)
⇒ I.e. streams are actually a special type of file!

<table>
<thead>
<tr>
<th>Stream</th>
<th>file descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>0</td>
</tr>
<tr>
<td>stdout</td>
<td>1</td>
</tr>
<tr>
<td>stderr</td>
<td>2</td>
</tr>
</tbody>
</table>
What is happening when we run a command?

⇒ `ls /home/sealion` with sufficient permissions will print its output to stdout which in turn is displayed by the terminal.

⇒ Without the permissions, an error message will be print to stderr (displayed by the terminal too).
05.02 Standard I/O redirection: output

⇒ Unix allows you to redirect streams from one file to another

n> file

redirects output from file descriptor n to a file, overwrites it if file exists.

n>> file

redirects output from file descriptor n to a file. If file doesn't exist, creates it, else content is appended.
05.02 Output redirection example

cwd is recipes

```bash
ls *.txt > all_txt_files.txt
```

```
calamari.txt
penguin.txt
perch.tst
```
05.02 Output redirection example

Running `ls *.txt >> all_txt_files.txt` then, will set the contents of `all_txt_files.txt` to:

calamari.txt
penguin.txt
perch.txt
all_txt_files.txt
calamari.txt
penguin.txt
perch.txt

output from
`ls *.txt > all_txt_files.txt`
## 05.02 Output redirection

### More examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ls ~ &gt; /dev/null</code></td>
<td>redirects stdout to special file <code>/dev/null</code> which discards data</td>
</tr>
<tr>
<td><code>mkdir /data 2&gt; mkdir_err_log.txt</code></td>
<td>redirects stderr to <code>mkdir_err_log.txt</code> (run as regular user without privileges on <code>/</code>)</td>
</tr>
<tr>
<td><code>cat &gt; write_to_me.txt 1 2 3 Ctrl-d</code></td>
<td>redirects stdout to <code>write_to_me.txt</code>. <code>cat</code> without param allows to interactively write input, stop input mode by pressing Ctrl and d</td>
</tr>
<tr>
<td><code>cat /home/tux/tux_secret.txt 2&gt; err.txt &gt; stolen_secret.txt</code></td>
<td>tries to access Tux's secret file <code>tux_secret.txt</code> (protected through file permissions!) ⇒ error gets written to <code>err.txt</code>, no output to <code>stolen_secret.txt</code> (empty file)</td>
</tr>
</tbody>
</table>
05.03 A new command - echo!

```bash
echo [STRING]
```
prints a new line, containing STRING if provided.

Examples:

```bash
sealion@server:~$ echo Tux is a penguin
Tux is a penguin
sealion@server:~$ echo "usually afraid of seals and sealions"
usually afraid of seals and sealions
sealion@server:~$ echo 'but became friends with sealion!'
but became friends with sealion!
sealion@server:~$ echo "isn't that great?"
isn't that great?
```
05.04 Input redirection

less commonly used than output redirection.

```
man cat ⇒ when cat has no argument, it reads its input from stdin

echo 'hello tux!' > output.txt

will print 'hello tux!' to stdout

```
### 05.05 Some basic text processing commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sort [file]</strong></td>
<td>sorts lines of file, or stdin if no input is given</td>
</tr>
<tr>
<td><strong>head [file]</strong></td>
<td>prints per default first 10 lines of file to stdout, or stdin if no input is given. Use <code>-n &lt;count&gt;</code> to print <code>&lt;count&gt;</code> lines, <code>-c &lt;count&gt;</code> to print <code>&lt;count&gt;</code> bytes</td>
</tr>
<tr>
<td><strong>tail [file]</strong></td>
<td>same as head, just takes the last lines (also with <code>-n</code> / <code>-c</code>)</td>
</tr>
</tbody>
</table>
prints like in C / Java / Python a formatted string to stdout. I.e. stdout will have sealion, tux, penguin, crabby each on one line. (\n is the newline character)

printf "sealion\ntux\npenguin\ncrabby" > temp.txt

sort temp.txt > temp2.txt

head -n 3 temp2.txt > result.txt

rm temp.txt

rm temp2.txt

Can we do better?
05.06 Building pipelines

rewriting commands to use stdin and feeding them temp files

printf "sealion\ntux\npenguin\ncrabby" > temp.txt

sort < temp.txt > temp2.txt

head -n 3 < temp2.txt > result.txt

rm temp.txt

rm temp2.txt

Can we do even better?
**05.07 Pipes**

```
cmd1  |  cmd2
```

**pipe operator** | ⇒ **connects stdout of** cmd1 **to stdin of** cmd2

⇒ allows you to get rid of temporary files

**Example:**

```
printf "sealion\ntux\npenguin\ncrabby"  | 
sort  | 
head -n 3 > result.txt
```
End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477