Lecture 4B: File Permissions

Conversion Review

A bit of review which will come in handy for today's topic.

2^2	2^1	2^0
4	2	1

When we see a binary number with three bits, we can convert that into an octal number. An octal digit has a maximum value of 7.

What Do These Permissions Mean?

Now let's revisit the output of our `ls -l` command:

```
-rw-r--r-- 1 eric eric 218 Sep 20 13:57 lecture4b.md
```

-	rwx	rwx	rwx
indicates the type of file	User permissions	Group permissions	Other permissions

The section of our output describes the *permissions* of our file. The first **-** indicates that this is a *regular* file. (a **d** would indicate a directory, for example).

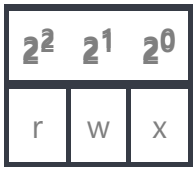
The next three characters (let's call them bits!) indicate the permissions for the *User* (or *owner*), which we can see is `eric`. The owner (me!) has permission to *read* the file (for example, to use `cat` on it) as well as permission to *write* to the file (edit it in `vim`). The dash indicates that the owner does *not* have execute permissions on this file. This makes sense. `.md` indicates a markdown file, which is very similar to a vanilla text file. You don't want to execute text files.

The middle three characters are for members of the `eric` group. These users only have *read* permissions, as indicated by the `r--`. Write and execute permissions are turned off.

Let's also look at the *last* set of permissions, for 'others'. This is for any other user, who isn't a member of the `eric` group. We can see that other users may *read* this file, but not change it.

File Permission Examples

Here's another way to look at permissions:



If an `r` equals **4** and a `w` equals **2**, then we can indicate the permission level of the user as being a **6**. Addtionally the permissions for group members and others are both **4**, so the permission level of this file is **644**.

```
-r-------- 1 eric eric 1679 Jan 29  2018 id_rsa
```

The file you see here is a *private key*. This can be used to log into the Matrix server without a password, and as you might guess is very important to keep secret. The user doesn't even have permission to change it, since modifying the value of the key would basically break it. The permission level of this file is **400**.

```
-rwxr-xr-x 1 root root 131K Jan 18  2018 ls*
```

This is the program `ls` which exists in `/bin`. You *execute* this command constantly. For execute permissions to work properly, you *must also enable read permissions*. You could also `cat` this file, but since it's a binary the results wouldn't make much sense to you. We don't, however, want regular users modifying it since that would pose a serious security risk. The only user capable of *changing* `ls` would be the root user. This might occur if there was an update of `ls`, for example. The permission level of this file is **755**.

What do you think would be the permission level of the file `~uli101/assign1`?

Directory Permissions

```
drwxr-xr-- 82 eric eric 4.0K Sep 22 01:36 eric/
```

The meaning of permissions changes slightly when discussing *directories*. First note the `d` which indicates that this is a directory. This is my `~` sweet `~`.

- `r`: allows reading contents of the directory
- `w`: allows modifying the contents of the directory
- `x`: allows access to files inside (pass-through permission).

In the case of directories, the command we use to *read* the contents of directories is `ls`. Without read permissions on a directory, we are basically blind. (Try using `ls` on the ~uli101 home directory).

Modifying the contents of directories usually means creating and deleting files inside the directory. (Try using `rm` on the assignment).

Pass-through is an important permission. Pass-through is basically like locking the front door to the directory. With pass-through disabled, users won't be able to even use `cd` to navigate into the directory. It doesn't matter how I change the permissions of files inside this directory, or if I enable read and write permissions on that directory.

If pass-through permission for others is turned off, nobody will be able to read, write or execute my files.

Setting File Permissions

First Method

The first way of setting file permissions is absolute: it doesn't matter what the permissions were before you ran this command, it changes them all:

```
chmod 755 test_script
```

We have now allowed *everybody* to read and execute the file `test_script`. Additionally, the user can now also edit the `test_script`. (It should go without saying that you will need to be the owner of this file in order to modify its permissions! Either that or have root access...)

Second Method

The second way of modifying permissions is a little more relative. If we want to preserve the current permission level, but maybe modify it slightly, we can do this:

```
chmod +x test_script
```

This would add execute permission for owners, group members and others for `test_script`.

Who?	Add/Remove	Which Permission?
u (user)	+ (add)	r (read)
g (group)	- (remove)	w (write)
o (other)	= (set)	x (execute)
a or blank(all)		

What will this do?

```
chmod g-rw,o-rw my-diary
```

Note: A lot of inexperienced users will often run `chmod 777` on a file when they run into permission troubles. **This is very bad practice**. Not only is it a huge security risk, but some programs will actually fail if permissions aren't set properly. For example, that private key mentioned above? The `ssh` command will refuse to use keys if they don't have an expected permission level!

---

umask

By default, my `lecture8b.md` file was given **644** permissions. With `umask`, I can set the default permission level for new files, if I'm feeling particularly generous/paranoid.

`umask` uses the *inverse* of the permissions you want to have default. Essentially, every bit that I want to be a dash by default should be counted.

For example, I want my files to have this permission level from now on:

```
-rw-r----- 1 eric eric 218 Sep 20 13:57 lecture4b.md
```

The permission level is set to **640**.

The first octal has one dash: the LSB. This is 1. The second octal has two dashes, one in the 2^1 column and one in the 2^0 column. 2 + 1 = 3. The final octal has three dashes, which combine to equal 7.

I would use:

```
umask 137
```

Now all new files that I create will have this permission level.

Now, notice that:

$7 - 6 = 1$ $7 - 4 = 3$ $7 - 0 = 7$

This is a convenient trick when working with `umask` commands or answering questions on the exam. Take 7, subtract the desired permission level, and you get the correct inverted number.

What would be the result of this command?

```
umask 252
```

> # Summary
>
> **octal** permission *sets* all permissions for a given file. Overwrites whatever you had before. Use `777`
>
> **symbolic** permission *modifies* some permissions, but leaves others intact. Use `ugo+-rwx`
>
> - `chmod` : Use octals to set absolute permissions, ugo+-rwx to manipulate the present permission levels.
> - `umask` : Use this to set a default permission level for all new files.
> - *Files* have a read, write and execute permission option for users, group members and others.
> - *Directories* have a read permission (allow you to `ls` a directory), a write permission (allow you to delete or create files), and pass-through permission (which is required in order to access files inside).

A Nice Summary