HTTP using the command line interface.

The CLI

Using the Command Line Interface (CLI) is a skill you will need to acquire. Professional Web developers use the CLI on a regular basis: it is easy to use, efficient, and allows you to do many things at once. It is also a status thing: not using a GUI has more status that using one. All the images you see of hackers interacting with keyboard and a screen are using the CLI.

Send me an email that you have finished the activities in this handout no later than Friday September 19th, 2025 at 5 pm.

These lessons should give you a good grasp of what is involved and help you build your skills in Linux/Unix and servers in general.

The CLI is a text only environment and it requires that you develop a clear mental model of what happens, what you can do, and where you are within the server setting. There are not many visual cues to tell you what is going on – so you have to make sure that you know that yourself. An example of a strong mental model is a stop sign. When you see a red octagon what do you do? Does when are away from home? Or do you simply know it wherever you are?

The CLI we are using is based on Linux as the kernel (main software) that uses a shell (application layer) called BASH (stands for Bourne Again SHell). The shell uses a tree metaphor to describe files and folders. The main folder is called "root", and everything branches off from the root. As you move through the folders you move closer and further away from the root folder. The root folder is denoted by a right slash / and then subsequent folders are divided by right slashes from there.

You will need Putty (http://www.putty.org) on a PC or on a Mac you can use the Terminal application.

We are using Humber's server called: apollo.humber.ca – each of you has an account on this server. First of all we need to login to the Linux server. On putty - run the program and in the server details add apollo.humber.ca or in terminal it is

```
ssh username@apollo.humber.ca
```

(If the Humber server is not working or non-responsive – you can pick a Linux emulator to run in your browser here - https://geekflare.com/run-linux-from-a-web-browser/ Either system will work for this exercise.)

HTTP using the command line interface.

Once there you will be asked for your password. The system does not show you any response - so just type it in. As well you may be asked about the connection - just click ok for Putty on a PC and enter on a Mac.

Now that you are in you should see the cursor and it is something like this:

```
[monette@apollo ~]$
```

Where "monette" is my username, "apollo" is the server, and "~" is my home folder: the square brackets delimit the string. If your username is smith then you would see:

```
[smith@apollo ~]$
```

Now that we are in - let's do something.

Let's Build a Playground

Since we are going to do some real file manipulation, let's build a safe place to "play" with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it playground.

Creating Directories

The `mkdir` command is used to create a directory. To create our playground directory, we will first make sure we are in our home directory and then create the new directory:

```
[monette@apollo ~]$ cd
[monette@apollo ~]$ mkdir playground
```

To make playground a little more interesting, let's create a couple of directories inside it called dir1 and dir2. To do this, we will change our current working directory to playground and execute another mkdir:

```
[monette@apollo ~]$ cd playground
[monette@apollo playground]$ mkdir dir1 dir2
```

Notice that the mkdir command will accept multiple arguments, allowing us to create both directories with a single command. The basic format for any command is: command (mkdir) argument (dir1) commands can also take attributes which follow the command and start with a hyphen.

HTTP using the command line interface.

The commands to list all files is ls -a – will show all files including hidden files.

Copying Files

Next, let’s get some data into our playground. We’ll do this by copying a file. Using the cp command, we’ll copy the passwd file from the /etc directory to the current working directory.

```
[monette@apollo playground]$ cp /etc/passwd .
```

cp = copy the following
`/etc/passwd` = the absolute path to the file ‘passwd’ in the directory ‘etc’
`. (dot or period)` = the folder I am currently in (which you can check by issuing the pwd command: “pwd” means “print working directory”).

Manipulating Files and Directories

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an ls, we will see our file:

```
[monette@apollo playground]$ ls -l
total 380
drwx------ 2 monette students   4096 Nov  4 09:19 dir1
drwx------ 2 monette students   4096 Nov  4 09:19 dir2
-rw------- 1 monette students 377626 Nov  4 09:19 passwd
[monette@apollo playground]$
```

Now, just for fun, let’s repeat the copy using the -v option (verbose attribute) to see what it does:

```
[monette@apollo playground]$ cp -v /etc/passwd .

`/etc/passwd' -> `./passwd'
```

The cp command performed the copy again, but this time it displayed a concise message indicating what operation it was performing. Notice that cp overwrote the first copy without any warning. Again, this is a case of cp assuming that you know what you’re doing. To get a warning, we’ll include the -i (interactive) option:

```
[monette@apollo playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a “y” (no quotes) will cause the file to

HTTP using the command line interface.

be overwritten; any other character (for example, n) will cause cp to leave the file alone and end the command.

Moving and Renaming Files

Now, the name `passwd` doesn't seem very playful and this is a playground, so let's change it to something else:

```
[monette@apollo playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
[monette@apollo playground]$ mv fun dir1
```

moves it first to directory dir1. Then

```
[monette@apollo playground]$ mv dir1/fun dir2
```

moves it from dir1 to dir2. Then

`[monette@apollo playground]$ mv dir2/fun .` (this dot refers to the folder you are currently in. A double dot .. means the next folder up)

finally brings it back to the current working directory. Next, let's see the effect of mv on directories. First we will move our data file into dir1 again:

```
[monette@apollo playground]$ mv fun dir1
```

`and then move dir1 into dir2 and confirm it with ls:`

```
[monette@apollo playground]$ mv dir1 dir2
[monette@apollo playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me me 4096 2012-01-11 06:06 dir1
[monette@apollo playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me me 1650 2012-01-10 16:33 fun
```

Note that because dir2 already existed, mv moved dir1 into dir2. If dir2 had not existed, mv would have renamed dir1 to dir2. Lastly, let's put everything back:

```
[monette@apollo playground]$ mv dir2/dir1 .
[monette@apollo playground]$ mv dir1/fun .
```

HTTP using the command line interface.

Creating Hard Links

Now we'll try some links. First the hard links: We'll create some links to our data file like so:

```
[monette@apollo playground]$ ln fun fun-hard
[monette@apollo playground]$ ln fun dir1/fun-hard
[monette@apollo playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file fun. Let's take a look at our playground directory:

```
[monette@apollo playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2012-01-14 16:17 dir1
drwxrwxr-x 2 me me 4096 2012-01-14 16:17 dir2
-rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for fun and fun-hard both contain a 4, which is the number of hard links that now exist for the file. You'll remember that a file will always have at least one link because the file's name is created by a link. So, how do we know that fun and fun-hard are, in fact, the same file? In this case, ls is not very helpful.

While we can see that fun and fun-hard are both the same size (field 5), our listing provides no way to be sure they are the same file. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part, which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an `inode`, which is then associated with the name part. Each hard link therefore refers to a specific `inode` containing the file's contents.

The ls command has a way to reveal this information. It is invoked with the -i option:

```
[monette@apollo playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me me 4096 2012-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me me 4096 2012-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun
```

HTTP using the command line interface.

```
12353538 -rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the `inode` number, and as we can see, both fun and fun-hard share the same `inode` number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: Hard links cannot span physical devices, and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory. These are like links to pages in html – once you know the address of the thing – you can link to it from anywhere. If something changes – then you change the thing but you don't have to change the link (unless you move it).

Creating symbolic links is similar to creating hard links:

```
[monette@apollo playground]$ ln -s fun fun-sym
[monette@apollo playground]$ ln -s ../fun dir1/fun-sym
[monette@apollo playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward: We simply add the -s option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the ls output:

```
[monette@apollo playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 6 2012-01-15 15:17 fun-sym -> ../fun
```

The listing for fun-sym in dir1 shows that it is a symbolic link by the leading l in the first field and the fact that it points to ../fun, which is correct.

Relative to the location of fun-sym, fun is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string ../fun rather than the length of the file to which it is pointing.

When creating symbolic links, you can use either absolute pathnames, like this:

HTTP using the command line interface.

```
[monette@apollo playground]$ ln -s /home/me/playground/fun
dir1/fun-sym
```

or relative pathnames, as we did in our earlier example.

Using relative pathnames is more desirable because it allows a directory containing symbolic links to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[monette@apollo playground]$ ln -s dir1 dir1-sym
[monette@apollo playground]$ ls -l
total 16
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir2
-rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun
-rw-r--r-- 4 me me 1650 2012-01-10 16:33 fun-hard
lrwxrwxrwx 1 me me 3 2012-01-15 15:15 fun-sym -> fun
```

Removing Files and Directories

As we covered earlier, the rm command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let’s delete one of our hard links:

```
[monette@apollo playground]$ rm fun-hard
[monette@apollo playground]$ ls -l
total 12
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir2
-rw-r--r-- 3 me me 1650 2012-01-10 16:33 fun
lrwxrwxrwx 1 me me 3 2012-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file fun-hard is gone and the link count shown for fun is reduced from four to three, as indicated in the second field of the directory listing.

Next, we’ll delete the file fun , and just for enjoyment, we’ll include the -i option to show what that does:

```
[monette@apollo playground]$ rm -i fun
rm: remove regular file `fun'?
```

HTTP using the command line interface.

Enter y at the prompt, and the file is deleted. But let's look at the output of ls now. Notice what happened to fun-sym ? Since it's a symbolic link pointing to a now nonexistent file, the link is broken :

```
[monette@apollo playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir1
lrwxrwxrwx 1 me me 4 2012-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir2
lrwxrwxrwx 1 me me 3 2012-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure ls to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not in and of itself dangerous, but it is rather messy. If we try to use a broken link, we will see this:

```
[monette@apollo playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```
[monette@apollo playground]$ rm fun-sym dir1-sym
[monette@apollo playground]$ ls -l
total 8
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir1
drwxrwxr-x 2 me me 4096 2012-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. However, rm is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use rm with the recursive option (-r) to delete playground and all of its contents, including its subdirectories:

```
[monette@apollo playground]$ cd
[monette@apollo ~]$ rm -r playground
```

We've covered a lot of ground here, and the information may take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links may be a little confusing at first but take the time to learn how they work. They can be a real timesaver.

HTTP using the command line interface.