Lecture 32 - More on Command Line

Videos

https://youtu.be/IDBUcCCGu5k - Lect-32-pt1-more-cli.mp4
https://youtu.be/6gmXcDmmxnU - Lect-32-pt2-links-symbolic-links.mp4
https://youtu.be/ZB3ieCtHbxQ - Lect-32-pt3.mp4

From Amazon S3 - for download (same as youtube videos)

http://uw-s20-2015.s3.amazonaws.com/Lect-32-pt1-more-cli.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-32-pt2-links-symbolic-links.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-32-pt3.mp4

Shell Aliases

Beyond just writing scripts the shell provides a “macro” like capability that is called “aliases”. This allows you to type in a command and replace it with a different string.

The most common alias is usually already set up. It is:

```
$ alias ll="ls -lh"
```

This allows you to type

```
$ ll
```

the “ll” is replaced with the “ls -lh” and you get a “long human” listing of a directory.

Aliases are useful for making short-hand visions of commands that you often use.

Setting them up:

```
$ alias alias_name="command_to_alias arg1 arg2"
```

Some common examples:

```
alias gs="git status"
alias gc="git commit"
```

```
alias mk="make"
```

Some that I use to prevent typo problems

```
alias cd..="cd .."
```

If you want to ignore the alias, the prefix your command with a backslash (`\`). Example, you have an alias for 'ls' that you want to ignore:

```
\ls
```

Will ignore the alias and just run the 'ls' command.

You can see the alias definition with:

```
$ alias ls
```

How to setup your aliases and other stuff automatically?

Answer: Dotfiles

Most programs that need configuration (zsh, bash, vim etc) read files in your home directory at startup. These are generally called "." files because they start with a dot.

The Dotfiles system stared by accident.

Let's start with .vimrc - the "RC" part of the mame is for "Run Command" and comes from Minix - the predecessor to Unix. In that system if you had a script that you wanted to run it had to end in "rc".

Let's say you want to turn on line numbers every time you use VIM. You can add a file in your home directory called `.vimrc` with the command in it and when vim starts it will see the file and run the commands.

So put

```
:set nu
```

In the file and you get line numbers!

You also want to set some aliases and your path. For bash the "." files are `.bashrc`, `.profile` and `.bash_profile`. Why? Again it is historical. The "sh" shell - the parent of "ksh" - lead to the development of the "Bourn Again Shell" - bash. "sh" ran `.profile`. Usually your best guess is to try `.bashrc` - and then if that fails to do some poking. Add an "echo" command to each and try again - and see which one is getting run.

So in the correct file, let's say `.bashrc` - you put your "alias" commands.

```
alias gs="git status"
```

I also like to setup variables in the `.bashrc` to directories where I am working:

```
export uw=/Users/pschlump/go/src/github.com/Univ-Wyo-Education
```

This allows me to quickly change to that directory with:

```
$ cd $uw/*2150
```

To get to the path for this class. The `$uw` uses the variable, then pattern match to the exact path for the 2150 class.

SSH uses `~/.ssh` directory and the `~/.ssh/config` file to do its setup.

GIT uses `/.gitconfig`.

These files are usually text fails - but some care should be taken in editing them. I usually make a backup of each file before I edit it in a `~/.bak` directory.

The best way to keep track of your dotfiles is to use git.

I have a git project called "DotFiles" and I keep the originals for all my DotFiles in that. Then I use symbolic links to my home directory so that programs can find the files.

Because my DotFiles contain "secrets" I don't put this out on github.com - I use a fully encrypted private repository for this. See https://github.com/mozilla/sops

Symbolic Links / Hard Links

The Unix/Linux file system provides a powerful facility that is not in Windows at all. This is called Links - In Unix a file is an name with an i-node number. The i-node is the place where the data for

the file is stored. So file names only point to a file - they are not the actual data. You can have more than one “name” for a file.

```
$ vi bob
$ ln bob jane
$ cat jane
$ ls –li bob jane
```

The space for files is re-claimed when the number of references to the file drops to 0. Having a file “open” also counts as a reference. This is one way to create temporary files that only exists during the run of a program and grantee that the space gets cleaned up at the end. You open the file, then remove it. As long as you keep it open the file’s storage will continue to exist.

These links are called “hard” links and can only be applied to “files” not directories.

You can have as many names for a file as you want. Hard links can not span across volumes - so this is per-disk-volume.

This is why removing a file in Unix is called “unlink”.

The second way to have a link is a “symbolic link”. A symbolic link is a file system alias. When you open the file the replacement location is used. That keeps happening until you get to a link that points to nothing - or to a file. (This means that you can have symbolic links to symbolic links to symbolic links. The bad side of this is that you can have circular references! The good side is that you can have links to things that you can not have hard links to. Also you can have directory links that are “up” and cause some programs that follow them to go into infinite loops.

So pitfalls aside... How to crate symbolic links.

```
$ ls –li
$ ln –s bob tim
$ ls –li
$ cat tim
```

The “-s” says create a symbolic link. IT is “from” “to” in order.

You can create symbolic links that don’t point to anything.

```
$ ls –li
$ ln –s abc def
$ ls –li
```

Then create the file

```
$ vi abc
$ ls -li
```

A more robust example

```
$ ls -l ~/.vimrc
```

Output:

```
lrwxr-xr-x  1 pschlump  staff  15 May  1 04:27 /Users/pschlump/.vimrc -> DotFiles/.vimr
```

Getting stuff to work across Machines - Portability of Scripts

Sooner or later you want to set stuff up for different machines. You work on a Mac and Deploy to a Linux Boxes name “peach” and “tomato”. At the same time you want to copy scripts and only have 1 source for them.

```
hn=$(hostname)
echo $hn
if [[ "$hn" == "peach" ]]; then
        {do_something}
fi
if [[ "$hn" == "tomato" ]]; then
        {do_something}
fi
```

or you want to do a different thing between Linux setup an Mac

```
if [[ "$(uname)" == "Linux" ]]; then
        {do_something}
fi
if [[ "$(uname)" == "Darwin" ]]; then
        {do_something}
fi
```

Password-less Login to Remote Machines

We talked about SSH and using it to login to remote machines. The password login system by default in SSH is a strong authentication system based on Secure Remote Passwords - much better

than most login systems. However it still requires that you enter a password for each login. This is far from a seamless login. What SSH will not do is let you put the password in a file and use that to login. So you can't automate a password-based login. It is not considered safe.

SSH has an alternative that is "safe" and reliable. This is based on the use of public/private keys and a digital signature.

The Effect:

```
$ ssh pschlump@192.168.0.27
```

output:

```
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.0-28-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Ubuntu 20.04 LTS is out, raising the bar on performance, security,
   and optimisation for Intel, AMD, Nvidia, ARM64 and Z15 as well as
   AWS, Azure and Google Cloud.

     https://ubuntu.com/blog/ubuntu-20-04-lts-arrives


0 updates can be installed immediately.
0 of these updates are security updates.

Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Thu Apr 30 08:40:40 2020 from 192.168.0.44
```

There was no prompt for a password. It just did it.

What just happened.

1. Ssh on the calling end - talked to the destination. It asked the question what form of attestation do you need?
2. The destination said A) Public/Private Key Pair, B) Password Authentication.
3. The sending end said, I have a "proof" of who I am - based on signing this data with my Private Key.
4. The destination took the signature - and said - yes I can verify that key with your public key. Good to go!

If the P/P key had failed, then a password prompt would have been issued and the sending end would have tried that.

To set this up you first need to generate a key pair. (Well you may first need to install ssh, if you need that then)

```
$ sudo apt-get update
$ sudo apt-get install ssh -y
```

Then Generate the key pair:

```
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

When it asks you for a passphrase just hit enter twice. You should see some output that looks like:

```
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pschlump/.ssh/id_ed25519
Your public key has been saved in /home/pschlump/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:CXJnF7XtOgj0SIHfuENUe1Ga6gfE3Ekun1jmz39jqGo pschlump@ub04
The key's randomart image is:
+--[ED25519 256]--+
|       ...o.+..  |
|      . .+ * B   |
|    . oo=oB @ .  |
|     o *+*.X o   |
|       .S.= + .  |
|        oo o +   |
|         .o + +  |
|   oB    E . o +.|
|        ..... . +|
+----[SHA256]-----+
```

This will generate 2 files in your ~/.ssh directory called id_ed25519 and id_25519.pub

Keep the id_ed25519 private - it is your private key! Don't Loose it - Don't share it!

```
$ ls ~/.ssh
```

Output:

```
authorized_keys  id_ed25519  id_ed25519.pub  id_rsa  id_rsa.pub  known_hosts
```

Now you need to copy the `~/.ssh/id_ed25519.pub` file to the destination system that you want to login to.

```
$ scp ~/.ssh/id_ed25519.pub pschlump@192.168.0.199:/tmp
```

Now login to the destination - you will get prompted for a password this one last time.

```
$ ssh pschlump@192.168.0.199
```

You need to add the contents of the `/tmp/id_ed25519.pub` to the `./.ssh/authorized_keys` file. If you do not have this then create the .ssh directory and create an authorized_keys file. Also you need to set permissions correctly for SSH to use the data.

```
$ mkdir -p .ssh
$ cat /tmp/id_ed25519.pub >>.ssh/authorized_keys
$ chmod 0600 .ssh/authorized_keys
$ chmod 0700 .ssh
$ rm /tmp/id_ed25519.pub
```

The first command `mkdir -p .ssh` will crate the .ssh directory if you do not already have it.

The 2nd command has `>>` in it to append or create the file. This will take the data in `/tmp/id_ed25519.pub` and append it to the `~/.ssh/authorized_keys` file.

Then set permissions on the file - SSH will only use the file if nobody else can read it. The permissions are in 3 groups of 3 - each one with an on/off bit. So you can specify them in octal. The 1st set is the owner - this says that you have read/write for the owner and none for anybody else.

Then set permissions on the .ssh directory. This gives read/write/X=search for the directory to only the owner - you.

Now cleanup and remove the /tmp file.

Logout - and try the login again. This time you should not get prompted for a password.

If you are trying to set this up - there are other options that you may need, and you may want to use the ‘ssh-copy-id’ command.

Another powerful command that uses 'ssh' is 'rsync' - it allows for copying entire trees of files to remote systems and will do it with a minimum amount of network traffic - only copying the files that have chained - been added - or been deleted. Usually you will need to setup the password-less SSH login first to use 'rsync'.

SSH also provides a facility for a thing called "port" forwarding. These are also called SSH tunnels.

Let's say that your company has a fire-wall - and you are working from home. You need to access a server inside the firewall. If you were at work you would just run your program. Suppose the server at work is on the computer "peach" at address 192.168.0.199 (at work) and it is using port 5432. Port 5432 is blocked by the firewall. You cant directly access it.

You can use SSH to create a "tunnel".

A tunnel is a way to forward all the traffic from your computer to a remote system via SSH. In this example I will setup a local port 6543 that forwards to the remote systems 5432 port.

```
$ ssh -L localhost:6543:192.168.0.199:5432 pschlump@uwyo.edu
```

The `localhost:6543` is my local system. The ' `192.168.0.199:5432` port is the port to direct this to on the remote system.

Then I use port 6543 on my local system and all the traffic is sent back and forth to the remote system.

If you use this on a frequent basis you can configure this is the `~/.ssh/config` file.

```
User pschlump
HostName uwyo.edu
Port 22
IdentityFile ~/.ssh/id_ed25519
LocalForward localhost:6543 192.168.0.199:5432
```

Copyright