

Linux Administration Basics

Announcements

- Lab 3 Due Tomorrow @0900
- Quiz on Thursday for Virtualization (10Q)
- No Assignment this week - focus on the lab and studying for midterms
- Lab 4 is released - putting some of the topics we cover this week into practice
- There will still be a quiz on this material, so be sure you spend time and understand it (even if it is easy for you)

Partitions

- Disks can be divided into parts, called partitions
- Partitions allow you to separate data
 - Can protect the overall system and keep users from creating outages
- Single Root Partition + Swap
 - Simplest scheme and decent for most use cases
- Discrete Partitions
 - 1) OS, 2) User home directories
 - 1) OS, 2) Applications, 3) User, 4) Swap
 - As a system administrator, you decide.

It usually makes sense to start by considering a single / partition and then separate out others based on specific use cases like encryption, a shared media partition, file share, etc

Partition Tables

When creating partitions, their metadata is written to a **partition table**

There are two main partition tables - **MBR and GPT**

- **Master Boot Record (MBR)**
 - Can only address 2 TB of disk space
 - Being phased out by GPT
 - Supports up to 4 'primary' partitions
 - Loaded from the BIOS
- **GUID Partition Table**
 - Part of Unified Extensible Firmware Interface (UEFI)
 - UEFI is replacing BIOS
 - Supports up to 128 partitions
 - Supports up to 9.4 ZB disk sizes
 - Not supported on older legacy operating systems

Mount Points

Mount Point - A directory used to access the data on a partition

- / (slash) is always a mount point - it is the **root** of the filesystem
- /home
 - /home/test is on the partition mounted on /home
- /export/foo
 - /export/foo/bar is on the partition mounted on /export/foo

How do things get mounted anyhow?

Mount Points

To mount a partition you use the command **\$ mount <device> <mount point>**

- **\$ mkdir /home/testing**
- **\$ mount /dev/sdc1 /home**
- At this point we have mounted the partition at /dev/sdc1 to the point /home
- What happens to the /home/testing directory?
 - It becomes inaccessible
 - What if we unmount /dev/sdc1 from /home? **\$ umount /home**
 - It becomes accessible again - the data isn't destroyed it just has no way to be accessed

How do we know what partitions are currently mounted?

We can use mount (with no options)

Mount Points

\$ mount

```
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
/dev/sda3 on / type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda8 on /usr type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda1 on /boot type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda6 on /tmp type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda7 on /opt type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda5 on /var type ext4 (rw,relatime,seclabel,data=ordered)
...
/dev/sda2 on / type xfs (rw,relatime,attr2,inode64,noquota)
/dev/sdb3 on /opt type ext4 (rw,relatime,data=ordered)
```

Persisting Mount Points

Filesystems mounted manually with **mount** do not persist after the machine has been restarted - for this we need **fstab** (The filesystem table)

/etc/fstab - Controls what devices get mounted and where on boot

Each entry is made up of 6 fields

- device
- mount point
- file system type
- mount options
- dump
- fsck order

Persisting Mount Points

#	device	mount point	FS	options	dump	fsck
	/dev/sda2	/media	xfs	defaults	0	1
	/dev/sda1	swap	swap	defaults	0	0
	UUID=b4175a08-ef39-4157-87d4-33fee1e2bc8a	/	ext4	defaults	1	1
	UUID=c58ef5c1-c6e1-49d1-b635-feddbe139d58	/boot	ext4	defaults	1	2
	UUID=eb292ef5-9df6-4c74-bb4b-13fecae1900f	/opt	ext4	defaults	1	2
	UUID=a129e6e6-e24b-4bfb-ad22-1d076188182f	/tmp	ext4	defaults	1	2
	UUID=07a5458a-442e-4730-ba02-62b75d77e846	/usr	ext4	defaults	1	2
	UUID=6fc8fb45-8dc4-45e7-92da-4671e9d14c23	/var	ext4	defaults	1	2

So this is all well and good, but where do we get the device id/uuids from?

FSCK

File System Consistency Check (FSCK) is a program that is run automatically on system startup if the system was not shut down cleanly or can be run manually to detect consistency errors with the filesystem.

It examines the file system for errors and attempts to repair them if possible. It uses a combination of built-in tools to check the disk and generates a report of its findings.

You should run `fsck` to check your file system if your system fails to boot, if files on a specific disk become corrupt, or if an attached drive does not act as expected.

It can often be a very slow process (depends on the size of the drive) so you want to avoid it if possible

Viewing Storage Devices

This is where **lsblk** comes in

\$ lsblk

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
sda	8:0	0	149G	0	disk	
├─sda1	8:1	0	1000M	0	part	/boot
├─sda2	8:2	0	31.3G	0	part	[SWAP]
├─sda3	8:3	0	2G	0	part	/
└─sda8	8:8	0	61.1G	0	part	/usr
sdb	8:16	0	149G	0	disk	
└─sdb1	8:17	0	64G	0	part	
sdc	8:32	0	149G	0	disk	
├─sdc1	8:33	0	97.7G	0	part	
└─sdc2	8:34	0	51.4G	0	part	

Viewing Storage Devices

This is where **lsblk** comes in

```
$ lsblk -f
```

NAME	FSTYPE	LABEL	UUID	MOUNTPOINT
sda				
├─sda1	ext4		c58ef5c1-c6e1-49d1-b635-feddbe139d58	/boot
├─sda2	swap		79961e8d-4250-47b0-a062-61ed976432fa	[SWAP]
├─sda3	ext4		b4175a08-ef39-4157-87d4-33fee1e2bc8a	/
└─sda8	ext4		07a5458a-442e-4730-ba02-62b75d77e846	/usr
sdb				
└─sdb1	swap		200b37c6-c551-46b8-9afd-6e317c9d7c36	
sdc				
├─sdc1	swap		3518e426-943a-4b5e-ba25-b8dd19dfecec	
└─sdc2	ext4		8012ba10-0e2c-4592-a39e-c22856279edd	

The df command

The **df** command is used to display free disk space for a given filesystem - and if no filesystem is specified it shows for all the filesystems

```
$ df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda3	1.9G	58M	1.8G	4%	/
devtmpfs	16G	0	16G	0%	/dev
tmpfs	16G	8.0K	16G	1%	/dev/shm
/dev/sda8	61G	7.0G	51G	13%	/usr
/dev/sda1	969M	172M	731M	20%	/boot
/dev/sda6	15G	69M	14G	1%	/tmp
/dev/sda7	9.5G	37M	9.0G	1%	/opt
/dev/sda5	29G	875M	27G	4%	/var

The du command

What if we want to be more granular when determining where our storage is being used?

The **du** command can be used to calculate the size of directories and files within a given filesystem

			\$ du -sh /
			15M /bin
			74M /boot
			5.5M /etc
			31M /home
			594M /lib
			1.1M /run
			16M /sbin
			253M /snap
			956K /tmp
			971M /usr
	\$ du -sh *		
\$ du -sh .	8.0K	README.md	
786M .	32K	assignments	
	4.0K	clone_script.sh	
du -sh labs/ quizzes/	36K	labs	
36K labs/	28K	quizzes	
28K quizzes/	19M	slides	
	614M	tmp	
	136M	virtualConsole	

File Manipulation

There are several ways to create a file on linux

- **\$ touch <file>** - creates an empty file
- **\$ cat > <file>** - redirects input that you type into a new file
- **\$ echo "stuff" > <file>** - creates a file with "stuff" as the contents
- **\$ nano <file>** - opens file for interactive editing in nano
- **\$ vim <file>** - opens file for interactive editing in vim

The list goes on and on and on

File Manipulation

What about reading from a file?

Similar to writing from a file, there are countless ways to read from a file

- **\$ cat <file>** - dump the contents of the file to stdout
- **\$ less <file>** - open file for interactive reading, allowing scrolling and search
- **\$ head <file>** - display the beginning part of a file
- **\$ tail <file>** - display the last part of a file
- **\$ echo "stuff" > <file>** - creates a file with "stuff" as the contents
- **\$ nano <file>** - opens file for interactive editing in nano
- **\$ vim <file>** - opens file for interactive editing in vim

Pipes and Redirection

Now that we have information flowing from stdin to files or from files to stdout, it can be helpful to know how to control the flow of information.

To do this, we use redirection '<', '>', and pipes '|'

Redirection is fairly straightforward - if you have a command and want to redirect its output to a file you can perform the following

- **\$ command > output.txt**

Likewise, if you have a command you want to read input from a file you can simply turn the redirect around

- **\$ command < input.txt**

Pipes and Redirection

When redirecting a commands output, you may still see some output go to the terminal (especially if there were errors/warnings). This is due to the fact that we only redirected stdout to the file. If we want to redirect all output, we can perform the following

- **\$ command > log_file 2> err_file**
 - Puts standard output in one file and standard error into another file
- **\$ command > log_file 2>&1**
 - Puts both standard out and standard error into the same file. This works because it is redirecting stdout to the log_file and it is then redirecting all stderr to stdout.

Pipes and Redirection

What about pipes? How are they different from redirection?

Pipes are used to effectively chain a bunch of commands together via their inputs and outputs

- **\$ command_1 | command_2 | command_3 | | command_N**

This can be used to build complex commands without needing to build entire new utilities. A great example of this is using **grep** to filter out input to what you want to see.

- **\$ cat really_big_file.txt | grep "name"** - will only output the lines that contain "name"
- **\$ cat really_big_file.txt | grep "name" | less** - will allow you to interactively scroll through the results without dumping them all to the terminal

File Manipulation

Copying, Moving, and Deleting Files

\$ **cp** <file> <new location> - copy a file to a new location

\$ **mv** <file> <new location> - move or rename files

\$ **rm** <file> - delete files

\$ **mkdir** <new directory> - make a new empty directory

\$ **mkdir -p** <new directory>/<new sub directory> - create entire paths of new directories at the same time

\$ **rmdir** <directory> - remove an empty directory

\$ **rm -rf** <directory> - remove a directory and everything under it

File Manipulation

What about looking for files or listing the contents of a directory?

```
$ ls
```

```
$ ls -l
```

```
$ ls -la
```

```
$ ls -lh
```

```
$ ls -ltr
```

Names that start with . are known as 'hidden files' and only show up when prompted with the '-a' flag

File Manipulation

Finding files can sometimes be difficult (especially in a large filesystem)

This is where **find** comes in

- Find files by extension
 - `$ find root_path -name '*.ext'`
- Find files by matching multiple patterns
 - `$ find root_path -name '*pattern_1*' -or -name '*pattern_2*'`
- Find files modified in the last 7 days
 - `$ find root_path -mtime -7`
- Run a command for each file (use `{}` for the filename)
 - `$ find root_path -name '*.ext' -exec wc -l {} \;`
 - I.e. ^ this prints out how many lines are in each file

Shell Tips

Tab completion is your friend - don't forget to use it!

Tab completion works for completing file names, commands, and sometimes more

```
$ rm tes<tab>
```

```
$ rm testing
```

```
$ lsb_<tab>
```

```
$ lsb_release
```

```
$ nets<tab>
```

```
$ netstat
```

Shell Tips

Command History - every command you run is kept in the shell history and can be recalled for future use

To access the command history use <up> and <down> arrows to scroll through previous commands

In addition to scrolling through commands you can also use **\$ history** which will print out all the previous session commands along with their corresponding number

You can then run a command with **!<number>** or with **!<relative offset>**
i.e. **\$!-2** will run the second to last command, **\$!12** will run the command with ID 12, etc

Shell Tips

There are also special parts of command history, namely `$!!` and `$!` and **CTRL+R**

- **!!** - Is the same as `!-1` for easy access to running the previous command
 - (this is particularly useful with `$ sudo !!` as we will cover shortly)
- **!\$** - Gives you access to the previous commands arguments
 - I.e. `$ vim /etc/fstab` - the value of `!$` would be `/etc/fstab`
 - This is particularly useful when manipulating files or directories where you may want to look at a file, then edit or delete it
 - `$ less </path/to/file> -> $ rm !$`
- **CTRL+R** - allows you to interactively search through history and run commands directly (super powerful for trying to find older commands)

File Permissions

When talking about file permissions, we need to break down what they look like and what they mean. When running `$ ls -l` you've seen file permissions, but may not have fully recognized them

`$ ls -l`

```
-r----- 1 dave dave 5 Oct  8 00:29 read_only  
-rw----- 1 dave dave 5 Oct  8 00:29 read_write  
-rwx----- 1 dave dave 5 Oct  8 00:29 read_write_execute
```

Permissions are broken up into three sections (Owner, Group, Everyone Else)

```
-----rwx 1 dave dave 5 Oct  8 00:32 everyone_only  
----rwx--- 1 dave dave 5 Oct  8 00:32 group_only  
-rwx----- 1 dave dave 5 Oct  8 00:32 owner_only
```

File Permissions

For each of the three sections (Owner, Group, Everyone Else), permissions can be assigned to allow READ, WRITE, or EXEC for that specific group

- Each of the permissions is assigned a number value
 - 4 - READ
 - 2 - WRITE
 - 1 - EXECUTE
- This means each set of permissions can be expressed as a series of three numbers
 - -rwx----- is 700
 - -rwxr-x--- is 750
 - -r---w--wx is 423

File Permissions

To change file permissions we use the **chmod** command

- **\$ chmod <permission> <file name>**
 - i.e. **chmod 750 ./test**

Permissions can also be changed in symbolic mode where instead of using the numerical permissions you can use strings with **+**, **-**, **or =**

- **\$ chmod <symbolic perms> <file name>**
 - i.e. **\$ chmod +x ./test** - adds the exec permission for all
 - **\$ chmod -r ./test** - removes the read permission for all
 - **\$ chmod g+r ./test** - adds read permission for group
 - **\$ chmod u=rw ./test** - adds read permission for group for user/owner

File Permissions

Now that we can limit files to owners and groups it's important to know how to change the owner and group of a file. To do this we use the **chown** and **chgrp** commands

- **\$ chown <user> <path to file>**
- **\$ chgrp <group> <path to file>**
- **chown** can actually be used to change both the owner and group by specifying both - **\$ chown <user>:<group> <path to file>**
 - As such, **chgrp** is rarely used in practice

File Permissions

But what if we need more granularity than 3 levels for file permissions?

This is where you can leverage file Access Control Lists (ACLs)

```
$ getfacl test
```

```
# file: test
```

```
# owner: dave
```

```
# group: dave
```

```
user::rw-
```

```
group::---
```

```
other::rw-
```

```
$ getfacl test_with_acls
```

```
# file: test_with_acls
```

```
# owner: dave
```

```
# group: dave
```

```
user::rw-
```

```
user:test-user:rw-
```

```
group::---
```

```
mask::rw-
```

```
other::rw-
```

File Permissions

To add additional access controls to a file you use **setfacl**

- **\$ setfacl -m "u:username:permissions" /path/to/file**
 - add permissions for a user
- **\$ setfacl -m "g:groupname:permissions" /path/to/file**
 - add permissions for a group
- **\$ setfacl -dm "entry"**
 - set the default ACL for a directory

After setting ACLs you'll notice the permission string change a bit with **\$ ls -l** to now have a **+** at the end

```
-rw-rw-rw-+ 1 dave dave 5 Oct  8 00:32 test_with_acls
```

PATH and Executing Programs

Paths fall under two categories, Absolute and Relative

- Absolute - starts at the root of the filesystem and names all directories under it
 - i.e. /home/dave/CSE410/testing/labs/lab1.md
- Relative - does not have a preceding / and therefore starts at your current directory
 - i.e. testing/labs/lab1.md
 - There are two special files in every directory . and .. where . means your current directory and .. represents the parent directory of .

To check your current directory you would use **pwd** and to change directories you would use **cd**

PATH and Executing Programs

So why is it that when we create scripts or download binaries we need to call them via absolute paths or preface them with ./?

Well, this comes down to the **PATH** variable which determines when you try to execute a command which directories it will search for the thing you are trying to execute

```
$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/dave/bin
```

It will look through the PATH from left to right and try to locate a file corresponding to the command you executed and eventually gives up

```
my-new-cool-program: command not found
```

PATH and Executing Programs

So it sounds like we just need to add . to our PATH right?

- Well, while that would work, it is a terrible idea. Why?
- Allows for accidental execution of files, or malicious unintentional execution of commands
- What if I have an 'ls' script in my home directory that does something completely different/malicious?

Once we have the files added into our path we also need to ensure they have the correct permissions to execute them

Installing Software

When installing software on linux you have several options.

- Install using a Package Manager (OS Distro specific)
- Download a precompiled binary, make it executable, and place it into the software PATH
- Compile the software from source, make it executable, and place it into the software PATH

Each of these approaches have benefits and tradeoffs

Package Managers

Starting with the option that most are familiar with, Package Managers

- Ubuntu uses the Advanced Package Tool aka APT
 - **\$ apt install htop**
 - **\$ apt update && apt upgrade**
 - Leverages dpkg under the hood
- Centos/RedHat uses Yellowdog Updater, Modified aka YUM
 - **\$ yum install htop**
 - **\$ yum check-update && yum update**
 - Leverages RPM under the hood
- Arch Linux uses pacman
 - **\$ pacman -S htop**
 - **\$ pacman -Syy && pacman -Syu**

Installing Precompiled Binaries

In the event a piece of software is not available via a Package Manager and the maintainer provides precompiled binaries, you can simply download them using **curl** or **wget**, make them executable, and place them in the PATH.

This is what package managers do for you, but with a few extra checks.

When downloading a precompiled binary, it is important to select the correct CPU architecture (x86, arm, etc).

It is also important to validate the checksum (usually **sha256**) if provided to ensure the integrity of the downloaded binary

Installing from Source

Sometimes software is not available from a package manager and the maintainer does not provide precompiled releases. In this case, we need to download the source, configure, compile it, and install it into our PATH.

The process usually follows a similar convention

- Download and unpack source code
- Run **\$./configure** to provide any necessary parameters for compilation
- Run **\$ make** to compile and build the software into a binary
- Run **\$ make install** to install the software into your PATH

User Management

All users must have the following attributes to be considered valid

- Username (or login ID)
- UID (user ID). This is a unique number
- Default group
- Comments
- Shell
- Home directory location

So where is all of this user information stored?

User Management

Well, it is all stored in **/etc/passwd**, which looks like the following

```
root:x:0:0:root:/root:/bin/bash
```

```
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

```
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

```
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
Dave:x:1000:1000:this is dave:/home/dave:/bin/bash
```

/etc/passwd has the following format

```
username:password:UID:GID:comments:home_dir:shell
```


User Management

Username

- Less than 8 characters in length by convention
- Case sensitive (All lowercase by convention)
- Numbers are allowed in usernames
- Do not use special characters

Password

- Encrypted password used to be stored in **/etc/passwd** (why is this a problem?)
- Now, encrypted passwords are stored in **/etc/shadow** which is only readable by root - therefore preventing users from trying to crack passwords
- dave:\$6\$9Et9qq:18176:0:99999:7:::
- Has the format
 - User:password:date when passwd was last changed:days before passwd can be changed again:days after passwd must be changed:days to warn of expiring passwd:days after expiration account is locked:date when account was locked:reserved for future

User Management

User ID (UID)

- UIDs must be unique numbers
- System accounts have UIDs < 1000
- The root account is always UID 0

Group ID (GID)

- The GID listed in the `/etc/passwd` for is the default group for an account
- New files belong to a user's default group
- Users can switch groups by using the **newgrp** command but will be reset to the group in `/etc/passwd` on next login

User Management

Comment Field

- Everyone loves comments - make the useful!
- Typically contains the user's full name
- In the case of a system or application account, it should contain what the account is used for
- May also contain additional information like contact info for the person responsible for the account

Home Directory

- Upon login the user is placed in their home directory
- If that directory doesn't exist (or they have insufficient permissions to access it), they are placed in "/"

User Management

Shell

- The shell will be executed when a user logs in
- A list of available shells are in `/etc/shells`
- The shell doesn't have to be a shell - it can be any executable on the system that the user has access to
- To prevent interactive use of an account, use **`/usr/sbin/nologin`** or **`/bin/false`** as the shell
- Shells can be applications that are accessed remotely by a user which restricts them to that application and gives them ease of setup

User Management

Now that we've spent a bunch of time talking about Users, how do we go about adding them?

For this, there are two main approaches - **useradd** and **adduser**

- **useradd** - minimalist command line utility
 - Takes the form **\$ useradd -c "<comment>" -m (create homedir) -s <shell> -g <primary group> -G <other groups> <username>**
 - **\$ useradd -c "Dave Dobmeier" -m -s /bin/bash -g dave -G group1,group2 dave**
 - Now that the user is created, you need to create a password using **passwd <username>**
- **adduser** - interactive utility for adding users
 - Takes the form **\$ adduser <username>**
 - Handles creating the home directory, comment, shell, password, etc automatically

User Management

What about deleting users if they are no longer required?

Same as with adding users, there are two options (surprise surprise) - **userdel** and **deluser**

```
$ deluser <username>
```

```
$ userdel <username>
```

-r flag - can be specified to either to remove the user's home directory as well

Both pretty much work identically except **deluser** has some extra functionality we will cover in a bit w.r.t groups

Group Management

Speaking of groups, what makes up a group?

- Name
- Password (usually unused)
- GID
- List of accounts which belong to the group

All groups can be found in **/etc/group** which looks like the following
group_name:password:GID:account1,account2,accountN

root:x:0:

daemon:x:1:

bin:x:2:

sudo:x:27:dave

Group Management

Groups are a bit easier to add as they dont have as much information as user accounts

To add a group use the **groupadd** command

- **\$ groupadd <group name>**
- **\$ groupadd testgroup**

Similarly, groups can be deleted with **groupdel**

- **\$ groupdel <group name>**
- **\$ groupdel testgroup**

Group Management

Now that we have groups created, how to we go about adding users to them?

We can use **usermod** to edit the primary and alternate groups for a user

- **\$ usermod -g <primary> -G <alt1>, <alt2>, <altN>**

Or if we just want to add additional groups to a user

- **\$ usermod -aG <newgroup1>,<newgroupN>**

While this seems pretty easy, removing a group from a user is a bit more complicated as it involves using the above command to edit the groups and only specify the groups in which the user is to continue belonging to

Group Management

To make this a bit easier, we can actually use **adduser** and **deluser** to manage user group assignments

- **\$ adduser <username> <group>**
- **\$ deluser <username> <group>**

Lastly, to view the group associations for a user you can use the **id** or **groups** commands

\$ id <user>

uid=1000(dave) gid=1000(dave) groups=1000(dave),27(sudo),108(lxd)

\$ groups <user>

dave : dave sudo lxd

Switching Users and Super User

Firstly, let's start with the **su** command which lets you switch users

- **\$ su <username>** - changes your user id or become super user

superuser, or **root** (on linux based machines) is the conventional name of the user who has all rights or permissions (to all files and programs) in all modes (single- or multi-user)

sudo - Program that lets you execute a command as another user, typically the superuser

- **\$ sudo <command>** - run command as root
- **\$ sudo -u <username> <command>** - run command as the given user
- **\$ sudo su - <username>**
 - Switch to the <username> account with no password (if you use sudo)
- **\$ sudo su -**
 - Changes a user to be superuser from then till they exit the shell

Switching Users and Super User

So how does one go about getting **sudo** access?

Access to **sudo** is defined in the **/etc/sudoers** file which looks like the following

```
# User privilege specification
```

```
root  ALL=(ALL:ALL) ALL
```

```
# Members of the admin group may gain root privileges
```

```
%admin ALL=(ALL) ALL
```

```
# Allow members of group sudo to execute any command
```

```
%sudo  ALL=(ALL:ALL) ALL
```

Switching Users and Super User

Let's break that down a bit starting with user access. The format of a sudoers entry is: user (host)=(user:group) commands

- `root ALL=(ALL:ALL) ALL`
 - A root user can execute any command as any user or any group from any host
- Groups are denoted by starting with a %
- `%sudo ALL=(ALL:ALL) ALL`
 - Any user having a group **sudo** can execute any command as any user or any group from any host
- `%deployer ALL=(ALL) NOPASSWD:/usr/sbin/nginx reload`
 - Any user having a group **deployer** can reload nginx (without entering their password)
 - Used to grant access to specific commands as **sudo** without giving full **sudo** access

Linux Networking

Time to put some of the networking information we talked about in the beginning of the semester into practice

First, lets begin with getting your current ip address - this can be accomplished with **ip addr** (shown below) or **ifconfig**

```
ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state
UP group default qlen 1000
    link/ether 00:50:56:b7:16:62 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.4/24 brd 172.16.0.255 scope global ens160
        valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:feb7:1662/64 scope link
        valid_lft forever preferred_lft forever
```

Linux Networking

Now that we have our IP address and subnet mask, how do we know what the default gateway is for the server?

For this we can use **netstat -nr** (shown below), or **ip route**

Destination	Gateway	Genmask	Flags	MSS	Window	irtt	Iface
0.0.0.0	172.16.0.1	0.0.0.0	UG	0 0	0		ens160
172.16.0.0	0.0.0.0	255.255.255.0	U	0 0	0		ens160

Linux Networking

While finding all of this information is useful for a pre-configured system, what do we need to do if we need to update the networking config or set it up in the first place?

For Ubuntu based systems (until Bionic Beaver 18.x), this used to mean checking **/etc/network/interfaces** and putting all of the relevant config there - similar to the following

```
auto eth0
iface eth0 inet static
    address 172.16.0.4
    netmask 255.255.255.0
    gateway 172.16.0.1
```

```
auto eth0
iface eth0 inet dhcp
```


Linux Networking

While configuring static IP information varies pretty widely from distribution to distribution, the configuration for DNS nameservers is fairly well agreed upon. The vast majority of unix based systems will use **/etc/resolv.conf** for which DNS resolvers to use.

```
$ cat /etc/resolv.conf
```

```
nameserver 128.205.32.8  
nameserver 128.205.106.1  
nameserver 128.205.1.2
```

That said, on Ubuntu 18+ you'll notice in **/etc/resolv.conf** it now tells you not to edit it as it is managed by systemd-resolved and has one hard-coded entry

```
$ cat /etc/resolv.conf
```

```
nameserver 127.0.0.53
```

Linux Networking

This has since been replaced with netplan which handles all of the networking pieces in one file. If you visit **/etc/network/interfaces** on an Ubuntu 18+ system you'll be greeted with the following

**# ifupdown has been replaced by netplan(5) on this system. See
/etc/netplan for current configuration.**

So how does the configuration differ?

network:

ethernets:

ens160:

dhcp4: true

version: 2

network:

ethernets:

ens160:

dhcp4: false

addresses: [172.16.0.4/24]

gateway4: 172.16.0.1

nameservers:

addresses: [1.1.1.1,8.8.8.8]

version: 2

DNS

To find the currently configured hostname we also have several options

- `$ hostname`
- `$ cat /etc/hostname`
- `$ uname -n`
- `(more)`

To set the hostname we can use

- `$ hostname "new-hostname"`
- `$ echo "new-hostname" > /etc/hostname`

DNS

To find the currently configured hostname we also have several options

- `$ hostname`
- `$ cat /etc/hostname`
- `$ uname -n`
- `(more)`

To set the hostname we can use

- `$ hostname "new-hostname"`
- `$ echo "new-hostname" > /etc/hostname`

DNS

So how do we perform DNS lookups for testing/troubleshooting?

There are two main tools for this - **dig** and **nslookup**

```
$ dig google.com
```

```
;; QUESTION SECTION:
```

```
;google.com.          IN  A
```

```
;; ANSWER SECTION:
```

```
google.com.    32  IN  A
```

```
172.217.8.14
```

```
;; Query time: 0 msec
```

```
;; SERVER: 127.0.0.53#53(127.0.0.53)
```

```
;; WHEN: Tue Oct 08 03:34:44 UTC 2019
```

```
;; MSG SIZE rcvd: 55
```

```
$ nslookup google.com
```

```
Server:      127.0.0.53
```

```
Address:     127.0.0.53#53
```

```
Non-authoritative answer:
```

```
Name:  google.com
```

```
Address: 172.217.8.14
```

DNS

So we've determined that when we make a DNS lookup we check **/etc/resolv.conf** it points us at **127.0.0.53** which is a local DNS server running on our machine configured to then resolve against the servers we included in our **/etc/netplan** config.

But are we missing a potential lookup step?

What about **/etc/hosts**? What is that for?

```
$ cat /etc/hosts
```

```
127.0.0.1 localhost
```

```
127.0.1.1 test-srv-02
```

```
10.10.10.1 webserv1.mycompany.com webserv1
```

```
10.0.1.1 dbcluster postgrespool
```

Linux Network Troubleshooting

So what happens when things don't work properly and you get reports of connectivity issues, throughput issues, etc?

The first tool most people use to check connectivity is **ping**

\$ ping <host>

\$ ping google.com

PING google.com (172.217.12.238) 56(84) bytes of data.

64 bytes from iad30s15-in-f14.1e100.net (172.217.12.238): icmp_seq=1 ttl=55 time=13.4 ms

64 bytes from iad30s15-in-f14.1e100.net (172.217.12.238): icmp_seq=2 ttl=55 time=14.1 ms

64 bytes from iad30s15-in-f14.1e100.net (172.217.12.238): icmp_seq=3 ttl=55 time=15.1 ms

64 bytes from iad30s15-in-f14.1e100.net (172.217.12.238): icmp_seq=4 ttl=55 time=13.9 ms

Linux Network Troubleshooting

When you can't ping a host it looks like the following

```
$ ping 1.2.3.4
```

```
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
```

```
--- 1.2.3.4 ping statistics ---
```

```
4 packets transmitted, 0 received, 100% packet loss, time 3068ms
```

What could be the reason we aren't getting any replies?

Does this always mean the host we are pinging is unreachable on the network?

Linux Network Troubleshooting

Let's say **ping** isn't working, what are our next options?

If we have access to the remote system, we should ensure it is currently online and has the correct IP configuration. If that is the case, maybe the web server/service on the machine isn't running, or is running on the wrong port.

- To see what applications are currently listening for network traffic, we can use **netstat**
- The most useful flags for netstat are as follows
 - -n Display numerical addresses and ports
 - -p Display the PID and program used
 - -l Display listening sockets
 - -t Limit the output to TCP
 - -u Limit the output to UDP
 - -r Displays the route table

Linux Network Troubleshooting

Lets see what's listening on this server.

```
$ netstat -tlnp
```

Active Internet connections (only servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program
tcp	0	0	127.0.0.1:6062	0.0.0.0:*	LISTEN	22755/process-agent
tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN	19999/systemd-resol
tcp	0	0	0.0.0.0:22	0.0.0.0:*	LISTEN	16340/sshd
tcp	0	0	127.0.0.1:8126	0.0.0.0:*	LISTEN	22756/trace-agent
tcp	0	0	127.0.0.1:5000	0.0.0.0:*	LISTEN	22754/agent
tcp	0	0	127.0.0.1:5001	0.0.0.0:*	LISTEN	22754/agent
tcp6	0	0	:::80	:::*	LISTEN	15506/apache2
tcp6	0	0	:::22	:::*	LISTEN	16340/sshd

Linux Network Troubleshooting

At this point, we know that the web server is running on the remote system - what else can we use to see why we may not be able to access it?

To see if we can connect to a remote port we can use Netcat (**nc**). Netcat allows us to test connectivity to an arbitrary remote TCP or UDP port.

- **\$ nc -vz <host> <port>**

- -v is for verbose (tell us everything that is happening)
- -z is to just scan for listening daemons without sending any data

\$ nc -vz google.com 443

Connection to google.com 443 port [tcp/https] succeeded!

\$ nc -vz google.com 8080

nc: connectx to google.com port 8080 (tcp) failed: Operation timed out

Linux Network Troubleshooting

While **nc** is super useful if we already have access to the remote system and know it's listening on a specific port, sometimes we need to scan a host and see what ports it may be listening on. For this we can use Network Mapper (**nmap**). This is often used in security to see what hosts may be exposing (more on this next week)

```
$ nmap 172.16.0.2
```

```
Starting Nmap 7.60 ( https://nmap.org ) at 2019-10-10 01:30 UTC
```

```
Nmap scan report for 172.16.0.2
```

```
Host is up (0.00023s latency).
```

```
Not shown: 998 closed ports
```

```
PORT      STATE SERVICE
```

```
22/tcp    open  ssh
```

```
80/tcp    open  http
```

```
Nmap done: 1 IP address (1 host up) scanned in 0.08 seconds
```

Linux Network Troubleshooting

Nmap by default only scans the top 1000 used ports on the IP you specified, however it can be configured to scan for UDP and TCP and even scan entire subnet ranges.

\$ nmap 172.16.0.0/24 -p- (scan the entire /24 and check all 65535 ports)

```
Starting Nmap 7.60 ( https://nmap.org ) at 2019-10-10 01:30 UTC
```

```
Nmap scan report for _gateway (172.16.0.1)
```

```
Host is up (0.00042s latency).
```

```
Not shown: 997 filtered ports
```

```
PORT      STATE SERVICE
```

```
53/tcp    open  domain
```

```
80/tcp    open  http
```

```
443/tcp   open  https
```

```
Nmap scan report for 172.16.0.2
```

```
Host is up (0.00065s latency).
```

```
Not shown: 998 closed ports
```

```
PORT      STATE SERVICE
```

```
22/tcp    open  ssh
```

```
80/tcp    open  http
```

```
Nmap scan report for 172.16.0.3
```

```
Host is up (0.00070s latency).
```

```
Not shown: 998 closed ports
```

```
PORT      STATE SERVICE
```

```
22/tcp    open  ssh
```

```
80/tcp    open  http
```

```
Nmap scan report for test-srv-02 (172.16.0.4)
```

```
Host is up (0.00016s latency).
```

```
Not shown: 999 closed ports
```

```
PORT      STATE SERVICE
```

```
22/tcp    open  ssh
```

```
Nmap done: 256 IP addresses (4 hosts up) scanned in
```

Linux Network Troubleshooting

The last network troubleshooting tool we will cover is **tracroute**. It displays the route (or path) of a network hop and measures transit delays of packets across an IP network and is particularly useful for identifying network latency and routing issues.

\$ **tracroute 1.2.3.4**

traceroute to 1.2.3.4 (1.2.3.4), 64 hops max, 52 byte packets

```
1  10.84.127.253 (10.84.127.253)  10.311 ms  13.177 ms  5.373 ms
2  hayesdfw-opsouth-87-190.cit.buffalo.edu (128.205.87.190)  4.244 ms  3.872 ms  4.560 ms
3  rtr-internetfw9-249.cc.buffalo.edu (128.205.9.249)  3.964 ms  4.066 ms  3.255 ms
4  rtr-norton9-2.cc.buffalo.edu (128.205.9.2)  6.014 ms  11.104 ms  4.203 ms
5  * * *
6  * * *
```

Linux Network Troubleshooting

```
$ traceroute 10.128.1.10
```

```
traceroute to 10.128.1.10 (10.128.1.10), 30 hops max, 60 byte packets
```

```
1  _gateway (172.16.0.1) 0.089 ms 0.049 ms 0.033 ms
```

```
2  10.0.1.1 (10.0.1.1) 0.277 ms 0.254 ms 0.229 ms
```

```
3  10.128.1.10 (10.128.1.10) 0.516 ms 0.601 ms 0.476 ms
```

This can be really useful in identifying which routes your traffic is taking to get to a host, where in the path latency is being added, or potentially where your traffic is getting blocked/stopped.

Note: Even if you can't complete a traceroute it doesn't mean the host is unreachable - the traceroute may be specifically blocked/etc. Try different options for traceroute like **-T** (use TCP), **-I** (use ICMP), **-p <port>** to elicit different results.

Backups

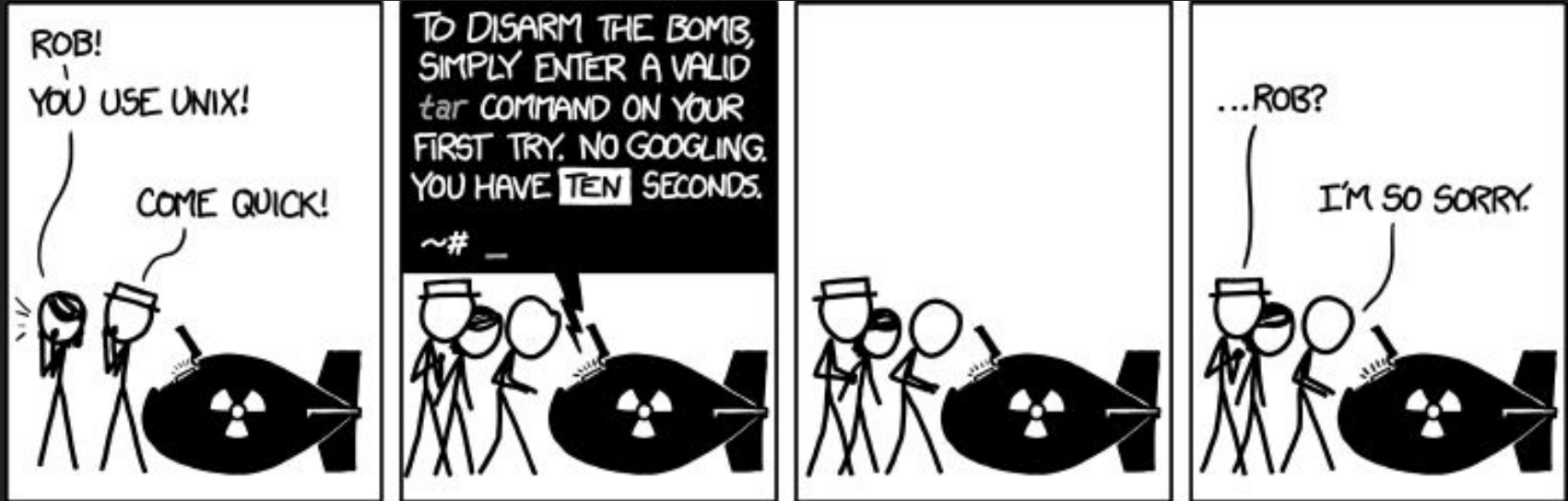
For backups we are going to cover two quick methods that are widely used - **rsync** and **tar**. These two tools likely make up a bunch of basic backup/recovery systems. For business/enterprise use, there are specialized tools just for Backup/DR, but in general case it's good to know **rsync** and **tar**.

First, let's start with Remote Sync aka **rsync**

- **\$ rsync -zavh /home/dave /opt/backup**
 - Archives /home/dave, compresses it, then copies it to /opt/backup
- **\$ rsync -zavh /home/dave dave@172.16.0.4:/opt**
 - Archives /home/dave, compresses it, then copies it to /opt on the 172.16.0.4 remote machine

When running **rsync** it will compare what is different between the source and destination and only copy the differences

TAR Archives



TAR Archives

How to fucking tar.

Make a .tar:

```
tar -czvf archive_name.tgz file_1.ext...
```

Unpack a .tar:

```
tar -xzvf archive_name.tgz
```

^ howthefuckdoitar.com ^

Automating Tasks


Say we want to automate a task on linux (i.e. rsync for daily backups to a remote server), we need some way of scheduling when to run a command or script.

These scheduled commands or tasks are known as “Cron Jobs”. Cron is generally used for running scheduled backups, monitoring disk space, deleting files (for example log files) periodically which are no longer required, running system maintenance tasks and a lot more.

There are two levels of Cron Jobs, system level and user level. Individual users can schedule jobs to run as them at a specified time or system wide jobs can be schedules to run as root.

Automating Tasks

So how do we configure these cron jobs? First we need to take a look at the crontab format.



The diagram shows five horizontal lines representing the fields of a crontab entry. From left to right, these fields are: minute, hour, day of month, month, and day of week. Each field is connected by a vertical line to a downward-pointing arrow. These arrows point to five asterisks (* * * * *) which are then followed by the text 'command to be executed'.

minute (0 - 59)
hour (0 - 23)
day of month (1 - 31)
month (1 - 12)
day of week (0 - 6 => Sunday - Saturday, or
1 - 7 => Monday - Sunday)

* * * * * command to be executed

For examples, check out crontab.guru

Automating Tasks

- The crontab for a user can be shown with **\$ crontab -u <user> -l** and can be edited with **\$ crontab -u <user> -i**.
 - If a user is modifying their own crontab the **-u** flag can be omitted.
- The system crontab is stored in **/etc/crontab** and while it can be edited, it usually does not need to be as there are some handy defaults set up for you.

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 * * * * root    cd / && run-parts --report /etc/cron.hourly
25 6 * * * root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6 * * 7 root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6 1 * * root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
```

Automating Tasks

When we look in `/etc` for cron directories we have the following options

- `cron.hourly` - All scripts in this directory are run once an hour as root
- `cron.daily` - All scripts in this directory are run once a day as root
- `cron.weekly` - All scripts in this directory are run once a week as root
- `cron.monthly` - All scripts in this directory are run once a month as root
- `crontab` - System crontab (special - includes the `<user>` field)
- `cron.d` - Extension of system crontab - used by packages to run scheduled tasks

Processes and Jobs

We often want to see what processes are running on the system, either stuff that we ran, services, or things other users are running. For this we use **ps**

```
$ ps
```

```
28586 pts/15  00:00:00 tcsh
```

```
28635 pts/15  00:00:00 ps
```

ps with no options shows only the processes you are currently running.

If you want to see what a specific user is running specify **-u <user>** and to see more information (like the full command used to run the process) use **-f**

Processes and Jobs

What if we want to see everything running on the system?

- For that we run **\$ ps aux**
 - **u** - display the process' user/owner
 - **a** - show processes for all users
 - **x** - show processes not attached to a terminal

Often this is way too much information to comb through - how can we search through this or at least filter the information?

Use grep!

```
$ ps aux | grep <search> | less
```


Processes and Jobs

With all of this information available to us (without needing to be root/superuser), what is the giant security issue what could be exploited if users aren't careful?

If we can see the command used to start the process and it has a password, guess what, everyone can see it!

Do not supply passwords via flags to commands, it's a terrible idea (more on this next week)

Long Running Processes

If I want to run a process that I know is going to take a long time (i.e. tar-ing up a large directory), sometimes i'd like to do other things while it works. This is where we introduce jobs and job control.

If we want to do other things while a process is running, we can run it in the 'background' by appending a **&** to the end of the command

```
$ <command> &
```

```
$ tar -czvf backup.tar /really/big/directory &
```

But what if we already started running the process?

Long Running Processes

We could use CTRL+C to interrupt and exit the process and start a new one, but that could waste a ton of time (especially if it's been running for a while).

Instead, we can use CTRL+Z to stop/pause the currently running process and make it a job.

```
$ tar -czvf backup.tar /really/big/directory &
```

```
^Z
```

```
[1]+  Stopped                  tar -czvf backup.tar /really/big/directory
```

```
$ jobs
```

```
[1]+  Stopped                  tar -czvf backup.tar /really/big/directory
```

At this point it is stopped, and we have our terminal back. Now what?

Long Running Processes

Well, we have two options for having the job resume. We can either have it resume in the foreground (**fg**) or in the background (**bg**).

If we want to have it start back in the foreground we can do

```
$ fg <job id>
```

Likewise if we want it to keep running, but in the background so we can do other things we can do

```
$ bg <job id>
```

What if we don't want this process running anymore and it is either stopped or it is running in the background and we can't send it CTRL+C?

Killing Processes

When we have a process we want to stop running and we can't interact with it directly, we can do so by sending it a signal using the **kill** command.

```
$ kill <process id>
```

```
$ kill %<job id>
```

This will send it the SIGTERM (15) signal telling the process it needs to clean up and terminate. If the process is misbehaving or ignoring the signal, we can escalate by sending it the SIGKILL (9) signal which causes it to immediately exit and it cannot be ignored.

```
$ kill -9 <process id>
```

There are several other handy signals, but these are the main two you'll regularly use.

Quick note on niceness and limits

With multi-user machines or even on systems where processes may misbehave, it's important to be able to set limits on what a user can do.

For this, we can use **ulimits**. To display your limits on a given system, use **ulimit -a**

```
$ ulimit -a
```

```
file size          (blocks, -f) unlimited
max memory size    (kbytes, -m) unlimited
open files         (-n) 1024
real-time priority (-r) 0
...
cpu time           (seconds, -t) unlimited
max user processes (-u) 3699
virtual memory     (kbytes, -v) unlimited
```

Quick note on niceness and limits

Ulimits can be super useful to prevent a process from creating a ‘fork bomb’ by spinning off processes continually, eating up all the ram, using up all the file descriptors, etc. So, if you are on a system that has plenty of resources and still tells you that it is out of resources, you’re probably encountering a **ulimit**.

Last thing to talk about is **niceness** / **process priority**

- All processes have a priority (0 - 139) and a “nice value” (-20 - 19)
- 0-99 are reserved for “real-time” operating system processes
- 100-139 are used for “user space” - where your programs/tasks are run

When a user starts a process, the default priority value is 120 and niceness value of 0.

Quick note on niceness and limits

The niceness value can then be used to modify the priority of a process (-20 to +19). Therefore $120 + -20 = 100$ and $120 + 19 = 139$ allowing users to adjust their priority within the “user-space” range of 100-139.

So how does this work? Well, there are two commands **nice** and **renice**. **nice** is used when you want to run a command with a different priority level.

```
$ nice -n <niceness-value> <command> <args>
```

```
$ nice -n 5 tar -czf backup.tar.gz ./Documents/*
```

This will run the above tar command with a nice value of 5 - giving it a priority of 125 and allowing other processes to take priority over it when getting scheduled by the OS

Quick note on niceness and limits

So what about **renice**? **renice** is used to change the nice value of an already running process. So if you spun off a long running process and need to change its priority you can after the fact.

```
$ renice -n <nice value> -p <process id>
```

```
$ renice -n 10 -p 1055
```

So if we can change the priority of our processes, why wouldn't all users just make their "nice" value -20 and make their stuff run first?

Well, because they aren't allowed to.

Quick note on niceness and limits

Once a user has increased the nice value of a process, they cannot lower it from its current value. This also means they cannot start processes with negative niceness values.

The only way to lower an existing nice value or start a process with a negative nice value is to be **root**.

Questions?