

# Linux: System Administration

A screenshot of a Linux terminal window. The prompt 'suman@ubuntu: ~\$' is displayed in green and blue text. The command 'sudo' is being entered in white text, followed by a vertical cursor bar. The background is black with a visible pixelated texture.

```
suman@ubuntu: ~$ sudo
```

# LOGISTICS



## **Class Hours:**

- Instructor will set class start and end times.
- There will be regular breaks in class.



## **Telecommunication:**

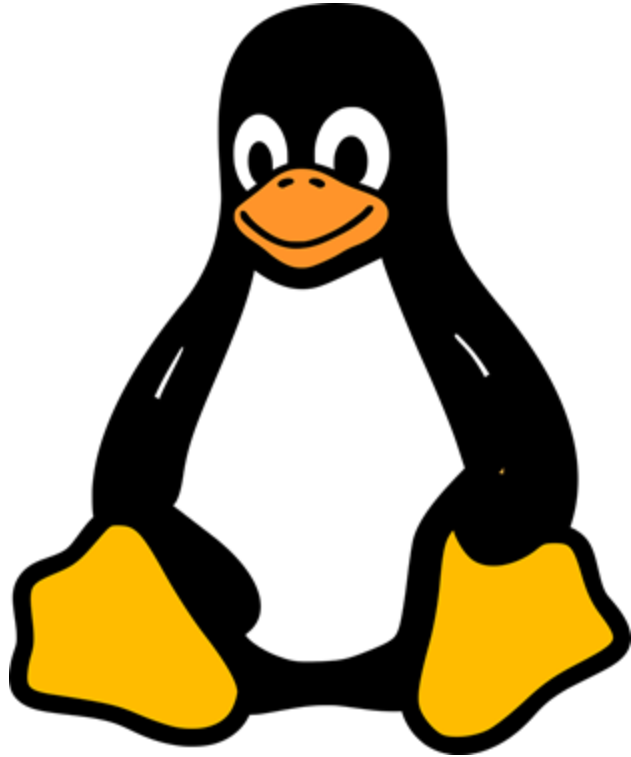
- Turn off or set electronic devices to silent (not vibrate)



## **Learning:**

- Run the commands with the instructor as the slides are presented to you
- Ask questions and participate

# Today's Objectives



1. **Create and execute basic scripts**
2. **Manage and create users, groups and permissions**
3. **Find files based on different properties**

# Linux: Scripting

## Scripting is a Core System Administration Skill

- Scripts let you **automate repetitive tasks**, saving time and reducing errors.
- They make your work **consistent and repeatable**, especially across multiple systems.
- Many administrative tasks — backups, updates, monitoring — can be run **unattended** through scripts.
- Understanding scripting helps you **read, modify, and troubleshoot** automation others have written.

💡 *A good admin doesn't just run commands — they build tools that run them automatically.*



# Linux: Scripting

## The Shebang (#!)

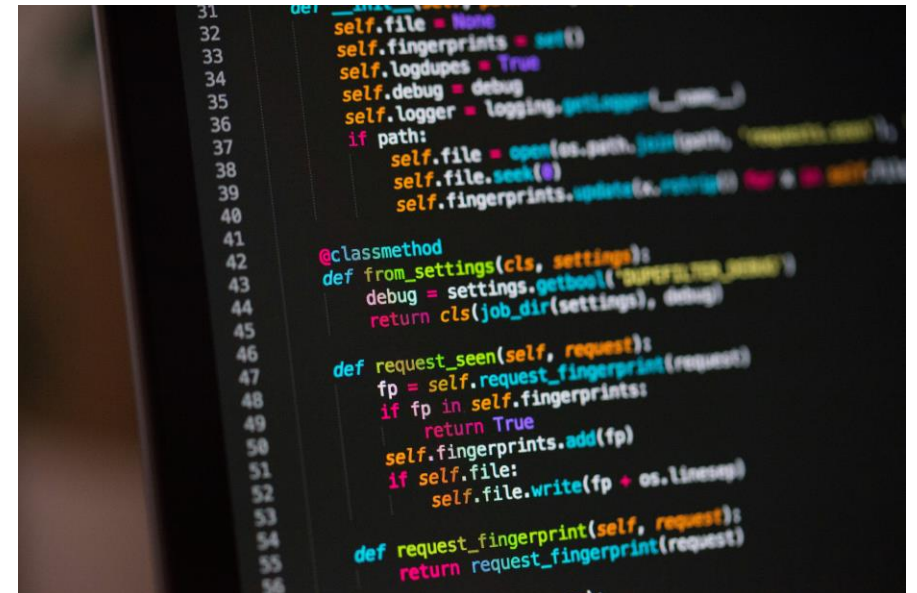
The **first line** of every script is called the **shebang**.  
It tells the system **which interpreter** should execute the script.

### Examples:

```
#!/bin/bash      # Use Bash (most common)
#!/bin/sh        # POSIX shell, for older systems
#!/bin/zsh       # Z shell
#!/usr/bin/python3 # Python script
```

💡 *The shebang defines how your script runs — even if the user's default shell is different.*

In this course, we'll be using:  
`#!/bin/bash`



# Linux: Scripting

`echo` prints text or variable values to the terminal.

```
echo "Hello World"  
echo "Today is a great day!"
```

You can also **insert command output** inside strings using a **subshell**:

```
echo "The date is $(date)"  
echo "Your current directory is `pwd`"
```

Both `$( )` and backticks `` `` run the command inside a **subshell**, replacing it with the output.

💡 *Use subshells to make your output dynamic — commands inside them execute first, and their results are printed.*

# Linux: Scripting

## Creating and Using Variables

Variables store values for later use in scripts or commands.

```
name="Daniel"  
echo "Hello $name"
```

- No spaces around = when assigning.
- Use \$variable to reference its value.

```
count=5  
echo "Count is $count"
```

💡 *Variables make scripts flexible and reusable.*

# Linux: Scripting

## Exporting and Subshell Assignment

export makes a variable available to **child processes** (like other scripts or commands).

```
export PATH=$PATH:/opt/bin
```

You can also assign values using **subshells**:

```
today=$(date)  
echo "Today is $today"
```

💡 *Use subshells when a variable should store the result of a command.*



# Linux: Scripting

## Viewing Environment Variables

You can see which variables are available in your shell environment:

Shows all **exported variables** — these are available to **child processes**.

```
export -p
```

Displays all **environment variables** currently set for your session.

```
printenv
```

Remember — If you create variables in your shell, they will not persist. If you want them to persist, create them in your *.bashrc* or *.bash\_profile* (or other profile files)

💡 *Use these commands to inspect what's in your environment — great for debugging scripts or checking paths.*

# Linux: Scripting

## Quoting and Shell Expansion Tricks

Double quotes (" ") allow variable expansion;  
Single quotes (' ') do not.

```
echo "Hello $USER"    # expands and prints variables value  
echo 'Hello $USER'    # literal text, prints $USER
```

Useful expansions:

```
echo "Home: $HOME"  
echo "Files: $(ls *.sh)"
```

💡 *Double quotes protect text but still expand variables; single quotes print exactly what you type.*

# Linux: Scripting

## Shell Expansion Tricks

You can manipulate variables directly in the shell using **parameter expansion** — no extra commands needed.

**Set a default value (if empty):**

```
echo "User: ${USER:-guest}"
```

Prints \$USER if set, otherwise prints **guest**.

**Get the length of a variable:**

```
name="Daniel"  
echo ${#name}    # 6
```

**Substring (start, length):**

```
word="abcdef"  
echo ${word:2:3}    # cde
```

# Linux: Scripting

## Shell Expansion Tricks

**Substitute text within a variable:**

```
file="report.txt"  
echo ${file/.txt/.log}    # report.log
```

**Remove part of a string:**

```
file="report.txt"  
echo ${file%.txt}        # report
```

💡 *Parameter expansion lets you format, trim, and modify data right inside your scripts — no external tools needed.*

# Linux: Scripting

## Special Shell Variables

Scripts can access built-in variables that provide context about how they're run.

Variable	Description
\$0	Name of the current script
\$1, \$2, ...	Positional arguments passed to the script
\$#	Number of arguments
\$@	All arguments as a single list
\$\$	Process ID (PID) of the current process
\$?	Exit code of the last command

### Example:

```
echo "Running $0 with $# arguments."
```



*These variables make your scripts dynamic — they adapt to input, track processes, and handle results automatically.*

# Linux: Scripting

## Arithmetic in Bash

You can perform basic math directly in the shell:

```
echo $((5 + 3))      # 8  
echo $((10 / 2))     # 5
```

- Works for integers only (no decimals).

For **floating point math**, use `bc`:

```
echo $(bc <<< "5.5 / 2") # 2.75
```

💡 Use `$(( ))` for quick integer math — `bc` for precise decimal calculations.

# Linux: Scripting

## The test Command

The **test** command checks conditions — such as file existence, string comparison, or numeric values — and returns an **exit code** (0 = true, non-zero = false). Read the manual to see all the different test you can perform!

### Examples:

```
test 5 -gt 3           # true
test -f /etc/passwd # true if file exists
test "$USER" = "root"
```

You can also use the **shortcut syntax**:

```
[ 5 -gt 3 ]
[ "$USER" != "root" ]
```

💡 *test is how Bash makes decisions — it powers conditionals like `if` and `while`.*

# Linux: Scripting

## Using `if` / `else` Statements

The **`if` statement** runs commands based on a condition's success or failure. When you use `[ ]`, you're actually using the **`test` command** behind the scenes.

### Example:

```
if [ "$USER" = "root" ]; then
    echo "You are the root user."
else
    echo "You are not root."
fi
```

- `[ condition ]` is just shorthand for `test condition`
- then executes if the exit code is **0 (true)**

💡 *Every `if` checks a command's success — `[ ]` just happens to be one.*



# Linux: Scripting

## Any Command Can Be Used in if

You can use **any command** in an if statement

If the command **succeeds (exit code 0)**, the then block runs.

### Examples:

```
if grep -q "error" /var/log/syslog; then
    echo "Errors found!"
else
    echo "No errors detected."
fi
```

```
if ping -c1 google.com >/dev/null 2>&1; then
    echo "Network is up!"
else
    echo "Network is down!"
fi
```

💡 *if is about command success — not just logic. Any tool that exits cleanly can control the flow.*

# Linux: Scripting

## The Extended Test Command – `[[ ]]`

`[[ ... ]]` is a **modern version** of the test command — safer, more powerful, and easier to read. It supports:

- String comparisons (`=`, `!=`)
- Numeric comparisons (`-eq`, `-gt`, etc.)
- **Pattern matching and basic regex** with `=~`

### Examples:

```
if [[ "$USER" == "Mike" ]]; then
    echo "Welcome, Mike!"
fi
```

```
if [[ "filename.txt" =~ \.txt$ ]]; then
    echo "This is a text file."
fi
```

💡 `[[ ]]` is preferred in Bash — it adds pattern matching, regex, and prevents common quoting errors.

# Linux: Scripting

## Using `elif` and `else`

You can chain multiple conditions using `elif` (else if) and finish with `else` for all remaining cases.

### Example:

```
if [ "$USER" = "root" ]; then
    echo "You are root."
elif [ "$USER" = "admin" ]; then
    echo "Hello Admin!"
else
    echo "Standard user access."
fi
```

- The shell checks each condition **in order** until one succeeds.
- Only the **first true condition** runs — the rest are skipped.

💡 *Use `elif` to handle multiple outcomes cleanly without stacking nested `if` statements.*

# Linux: Scripting

## The case Statement

The **case** statement is used for matching **multiple patterns** — a cleaner alternative to long `if/elif` chains.

### Example:

```
read -p "Enter a number: " num
```

```
case $num in
  1) echo "One";;
  2) echo "Two";;
  3) echo "Three";;
  *) echo "Invalid choice";;
esac
```

- Each pattern ends with `)` and each action ends with `;;`.
- The `*` pattern acts as a **default** (like `else`).

💡 *Use **case** when you have several clear options — it's more readable and faster to maintain than nested **if** statements.*

# Linux: Scripting

## The while Loop

A **while loop** runs as long as its condition is **true**.

```
count=1
while [ $count -le 5 ]; do
    echo "Count: $count"
    ((count++))
done
```

Prints numbers **1 through 5**, one per line.

Use **until** for the opposite behavior:

```
until [ $count -gt 5 ]; do ...
```

Runs **until** the condition becomes true.

💡 *Great for counters, checks, and continuous monitoring loops.*

# Linux: Scripting

## The for Loop

A **for loop** iterates over a list of items — words, numbers, or files.

```
for item in apple banana cherry; do
    echo "Fruit: $item"
done
```

The loop separates items using the **Internal Field Separator (IFS)** — by default, it's **whitespace** (spaces, tabs, newlines).

```
IFS=', ';
for item in one,two,three; do
    echo $item;
Done
```

```
IFS=$' \t\n'    # reset to default
```

💡 *Understanding and resetting IFS ensures consistent behavior when processing text or input.*

# Linux: Scripting

## More for Loop Styles

You can loop over **ranges** and even use **C-style syntax** in Bash.

### Brace Expansion (Numbers & Letters):

```
for i in {1..5}; do
    echo "Number: $i"
done
```

```
for letter in {a..e}; do
    echo "Letter: $letter"
done
```

### C-Style Loop:

```
for ((i=1; i<=5; i++)); do
    echo "Count: $i"
done
```

💡 *Brace expansion is simple and fast; C-style loops give you full control over counters and conditions.*

# Linux: Scripting

## Arrays in Bash

Arrays let you store and manage multiple values in one variable.

```
# Create an array
colors=("red" "green" "blue")

# Access elements
echo ${colors[0]}           # red

# Show all elements
echo ${colors[@]}

# Number of elements
echo ${#colors[@]}

# Add a new item
colors+=("yellow")

# Remove an item by index
unset 'colors[1]'           # removes "green"
```

💡 Arrays make it easy to organize and loop through lists of values dynamically.



# Linux: Scripting

## Looping Over an Array

You can iterate through array elements with a simple **for loop**:

```
colors=("red" "green" "blue")  
  
for color in "${colors[@]}; do  
    echo "Color: $color"  
done
```

💡 Always quote "\${array[@]}" — this preserves spaces and prevents word-splitting issues.


# Linux: Scripting

## Looping Over Script Arguments (\$@)

`$@` expands to **all arguments** passed to a script — perfect for looping through input values dynamically.

```
#!/bin/bash
echo "You passed $# arguments."
```

```
for arg in "$@"; do
    echo "Argument: $arg"
done
```

 `$#` gives the **count** of arguments, and `$1`, `$2`, etc. give individual ones. This is how scripts accept user input or process multiple files in one go.


# Linux: Scripting

## Functions in Bash

A **function** is like creating your own **command** — you call it and pass arguments just like any other command.

```
greet() {  
    echo "Hello, $1"  
    return 0  
}
```

```
greet penguin      # runs the function
```

 Arguments work like \$1, \$2, @\$ — just like in scripts.

# Linux: Scripting

## Return vs Output

- return only sets an **exit status** (0 = success, non-zero = error).
- To capture output, **echo** it and store it in a **subshell**:

```
get_name() {  
    echo "penguin"  
    return 0  
}
```

```
name=$(get_name)  
echo "Name is $name"
```

 *Use return for success/failure — use echo for data.*

# Linux: Scripting

## Error Handling with set Options

You can make your scripts **safer and more predictable** by enabling strict error handling.

```
#!/bin/bash
```

```
# Exit immediately if any command fails  
set -e
```

```
# Treat undefined variables as errors  
set -u
```

```
# Fail if any command in a pipeline fails  
set -o pipefail
```

These options help catch mistakes early and prevent scripts from continuing in an unstable state.

💡 *Use strict modes in production scripts — they turn silent bugs into visible errors.*

# Linux: Scripting

## Running Commands with `bash -c`

You can run a **command or small script** directly from the command line using `bash -c`.

```
bash -c "echo Hello from Bash!"
```

This starts a **new Bash shell**, runs the command inside the quotes, then exits.

You can also chain commands:

```
bash -c "cd /tmp && ls"
```

💡 *Useful for one-liners, automation, or testing script logic without creating a file.*

# Lab 3.1 Scripting Basics / Challenge

Estimated Time: 25



# POP QUIZ:

What is the shell variable \$#?

- A. Number of arguments to the script
- B. An array of arguments to the script
- C. The name of the script
- D. Internal Field Separator





# POP QUIZ:

What is the shell variable \$#?

- A. Number of arguments to the script
- B. An array of arguments to the script
- C. The name of the script
- D. Internal Field Separator

"What variables represent the other options?"



# POP QUIZ:

What is an alias to the test command?

- A. if
- B. [ ]
- C. ttest
- D. There is none



# POP QUIZ:

What is an alias to the test command?

- A. if
- B. ☐
- C. ttest
- D. There is none

"What kind of test can we run?"



# Linux: Users, Groups and Permissions

User and group management is at the heart of Linux security and organization. It defines who can access the system, what they can do, and which files or resources they can modify. Without proper user and group control, even a well-configured system can become vulnerable or chaotic.

**A good administrator doesn't rely on chance — they enforce structure. Understanding users, groups, and permissions means you can protect data, delegate responsibilities safely, and maintain a stable, secure environment for everyone on the system.**





# Linux: Users, Groups and Permissions

## Checking User Information

You can verify your identity and group memberships using these simple commands:

Shows your **current logged-in username**.

`whoami`

Displays your **user ID (UID)**, **group ID (GID)**, and **all groups** you belong to.

`id`

Lists all **groups** associated with your user account.

`groups`

Prints your **username** from the environment variable.

`echo $USER`



*These commands help confirm who you are, what privileges you have, and which groups you belong to.*

# Linux: Users, Groups and Permissions

## Viewing All Users and Groups

View every **user** and **group** configured on the system.

**List all users:**

```
cat /etc/passwd
```

**List all groups:**

```
cat /etc/group
```

💡 *Each line in these files represents a user or group entry used by Linux for authentication and permissions.*

# Linux: Users, Groups and Permissions

## Creating Users and Groups

**Create a new user and group, then verify:**

```
sudo useradd -m penguin # -m is short for --create-home  
sudo groupadd developers  
sudo usermod -aG developers penguin
```

```
id penguin
```

- useradd → creates a new user
- groupadd → creates a new group
- usermod -aG → adds user to group
- id → verifies group membership

💡 *Always verify your changes — users, groups, and permissions form the foundation of system security.*

# Linux: Users, Groups and Permissions

## Switching Users and Managing Passwords

You can switch to another user account (even without their password if you have sudo access):

```
sudo su - penguin
```

Starts a new shell as **penguin**, loading their environment.

```
exit
```

Returns to your original user.

Change a user's password:

```
passwd penguin
```

On most servers, **password logins are disabled** — users authenticate with **SSH keys** instead, stored in: `~/.ssh/authorized_keys`

💡 *SSH key authentication is more secure than passwords and preferred in production systems.*



# Linux: Users, Groups and Permissions

## Removing Users and Groups (Goodbye, Penguin 🐧)

Delete a User

```
sudo userdel penguin
```

Removes the **user account** but keeps the home directory and files.

To delete everything (home + mail + files):

```
sudo userdel -r penguin
```

Delete the groups (most Linux systems create a group when creating a user)

```
sudo groupdel penguin
```

```
sudo groupdel developers
```

Removes the **group** entry.

Make sure no active user or process is still assigned to it.

💡 *Always verify before deleting*

# Linux: Users, Groups and Permissions

## Checking & Disabling SSH Password Authentication

Quickly check the SSH server config for the password auth setting:

```
sudo grep -i "PasswordAuthentication" /etc/ssh/sshd_config
```

If present and set to no, password logins are disabled for SSH:

```
# Explicitly disable PasswordAuthentication.
```

```
PasswordAuthentication no
```

💡 *You can turn this setting **off** (i.e., set to no) to require key-based SSH authentication. After changing this setting you would need to restart sshd. For the purpose of this class, we will not modify this setting. In production environments, password authentication should be turned off.*

# Linux: Users, Groups and Permissions

## Root Login and `sudo su`


You can escalate to root locally with:

```
[ec2-user@ip-172-31-1-87 ~]$ sudo su - root
[root@ip-172-31-1-87 ~]# exit
[ec2-user@ip-172-31-1-87 ~]$
```

Notice that when you are **root**, the prompt ends with **#** instead of **\$**.

### **Security Risk**

Even if SSH password authentication is disabled, users with **sudo privileges** can still become root if their sudo rights allow it. There is no true way to “disable” root access — anyone who can run **su** with **sudo** can effectively log in as any user. Root user can do **anything**.

 **Restrict which commands users may run via sudoers** (e.g., disallow `su`) and disable password logins. Combining both gives you stronger local privilege control. We will cover sudoers later in course.

# Linux: Users, Groups and Permissions

Symbol	Permission	Description	Numeric Value
r	Read	Can view or read file	4
w	Write	Modify or delete file	2
x	Execute	Can execute the file	1
-	No Permission	No access	0

```
[ec2-user@ip-172-31-1-87 ~]$ ls -l
total 0
drwxr-xr-x. 2 ec2-user ec2-user 6 Oct 21 16:48 test-dir
-rw-r--r--. 1 ec2-user ec2-user 0 Oct 21 16:48 test.txt
```

# Linux: Users, Groups and Permissions

## Changing Permissions – Relative (Symbolic)

Modify permissions **relative to current settings**.

```
chmod u+r file.txt      # Add read for user
chmod g-x file.txt      # Remove execute for group
chmod a+rx file.txt      # Add read & execute for everyone
chmod u+x,g-x,o-rw file.sh  # Multiple changes in one command
```

**u** = user   **g** = group   **o** = others   **a** = all  
+ add   - remove   = set exactly

 *Use relative when tweaking existing permissions.*

# Linux: Users, Groups and Permissions

## Changing Permissions – Absolute (Numeric)

Set all permissions **explicitly** using numbers.

```
chmod 744 script.sh
```

Breakdown:

- **7 (rwx)** → user
- **4 (r--)** → group
- **4 (r--)** → others

 *Use absolute when standardizing (e.g., 644 for files, 755 for scripts).*

# Linux: Users, Groups and Permissions

## The Sticky Bit (t)

The **sticky bit** restricts **who can delete files** inside a shared directory. Even if you have write permissions to the directory, you **can't delete or rename files you don't own** when the sticky bit is set.

Example: /tmp Directory

```
ls -ld /tmp
drwxrwxrwt. 10 root root 4096 Oct 26 13:00 /tmp
```

Notice the **t** at the end — that's the sticky bit.  
It means only the **file owner** (or **root**) can delete their own files here.

Set or Remove It Manually

```
chmod +t shared_dir      # Add sticky bit
chmod -t shared_dir      # Remove sticky bit
```

 *Used for shared folders like /tmp to prevent users from deleting each other's files.*

# Linux: Users, Groups and Permissions

## SetUID and SetGID

These special permission bits let a program **run with the privileges of its owner or group**, not the user who runs it.

Example

```
ls -l /usr/bin/passwd  
-rwsr-xr-x 1 root root 54256 Oct 26  passwd
```

- The **s** in the user field → **SetUID** (runs as the file's owner — here, root)
- The **s** in the group field → **SetGID** (runs with the file's group privileges)

### **Security Warning**

If an executable with **SetUID root** is compromised, it can grant attackers **root-level access**.

```
chmod u+s file      # Add SetUID  
chmod g+s file      # Add SetGID  
chmod u-s g-s file  # Remove them
```

 *Only trusted system binaries (like passwd) should ever use SetUID/SetGID.*



# Linux: Users, Groups and Permissions

## Common File Permission Standards

File / Folder Type	Recommended <code>chmod</code>	Purpose / Rationale
Private SSH Key ( <code>~/.ssh/id_rsa</code> )	600 or 400	Owner-only access; SSH refuses keys that are too open
<code>.ssh</code> Directory ( <code>~/.ssh/</code> )	700	Only the owner can read, write, or enter
Executable / Script ( <code>script.sh</code> )	755	Owner can edit; others can run but not modify
Configuration / Text Files ( <code>*.conf</code> , <code>*.txt</code> )	644	Owner read/write; others read-only
Shared Directory ( <code>/shared</code> , <code>/var/www/html</code> )	775	Group collaboration allowed; public read/execute
Temporary Directory ( <code>/tmp</code> )	1777	World-writable but protected by sticky bit ( <code>t</code> )

 **Tip:** Follow *least privilege* — grant only the access users truly need.

# Lab 3.2 Users, Groups and Permissions

Estimated Time: 30 Minutes



# POP QUIZ:

What command is used to add a user to a group?

- A. `user -a -G developers penguin`
- B. `useradd -G developers penguin`
- C. `usermod -a -G developers penguin`
- D. All options are valid



# POP QUIZ:

What command is used to add a user to a group?

"What are -a and -G for?"

- A. user -a -G developers penguin
- B. useradd -G developers penguin
- C. **usermod -a -G developers penguin**
- D. All options are valid



# POP QUIZ:

What command can remove write permissions from group, and add write to file owner?

- A. chmod 777
- B. chmod 200
- C. chmod g-w,o+w
- D. chmod g-w,u+w





# POP QUIZ:

What command can remove write permissions from group, and add write to file owner?

- A. chmod 777
- B. chmod 200
- C. chmod g-w,o+w
- D. **chmod g-w,u+w**

"Why can we use the numbers here?"



# POP QUIZ:

Which command will modify permissions for a key file so that it can be used by ssh?

- A. chmod 600
- B. chmod 777
- C. chmod 644
- D. chmod 640



# POP QUIZ:

Which command will modify permissions for a key file so that it can be used by ssh?

- A. **chmod 600**
- B. chmod 777
- C. chmod 644
- D. chmod 640

"What error will you get with the other permissions?"





# POP QUIZ:

Suppose you are part of the admin group. Will you be able to execute a file in the admin group with 701 permission?

- A. No, permissions are read from left to right
- B. Yes, since others can execute
- C. Only if you are not a other user
- D. Only if you it has sticky bit



# POP QUIZ:

Suppose you are part of the admin group. Will you be able to execute a file in the admin group with 701 permission?

- A. No, permissions are read from left to right
- B. Yes, since others can execute
- C. Only if you are not a "other" user "What if you use sudo?"
- D. Only if you it has sticky bit



# Linux: Finding Files

The find command lets administrators locate files anywhere on the system using powerful filters — by name, size, ownership, modification time, or even permissions. It's an essential tool for quickly pinpointing misplaced configs, outdated logs, or insecure files without guesswork.

**A skilled admin uses find not only to locate files but to audit and secure the system — spotting files with world-writable permissions, tracking down recently changed scripts, or cleaning up old data. It's precision search for keeping systems organized, efficient, and safe.**



# Linux: Finding Files

## Introduction to find


The `find` command searches your filesystem for files and directories that match specific criteria — starting from any path you choose.

Let's start simple inside your **home directory** (~):

```
cd ~  
# Create some files  
touch 1.txt 2.txt 3.sh
```

```
# Find a specific file  
find ~ -name 3.sh
```

```
# Find all .txt files (case-sensitive)  
find ~ -name "*.txt"
```

 `find` automatically searches **recursively**, scanning all subdirectories within your chosen path. Once you master the basics, you can filter by **date**, **size**, or **permissions** to find exactly what you need.

# Linux: Finding Files

## Common find Filters

The `find` command supports powerful filters that let you narrow or expand your search results.

```
# Find only directories
```

```
find ~ -type d
```

```
# Find by name (case-insensitive)
```

```
find ~ -iname "*.sh"
```

```
# Find all .txt files except 2.txt
```


```
find ~ -name "*.txt" ! -name "2.txt"
```

```
# Find .sh OR .txt files with Grouped Expression
```

```
find ~ \( -name "*.sh" -o -name "*.txt" \)
```

### Key Filters:

- `-type` → search by file type (f = file, d = directory)
- `-iname` → case-insensitive name match
- `!` → negate a condition (NOT)
- `-o` → OR operator

 Combine filters to create precise searches — `find` is as flexible as a mini query language for your filesystem.



# Linux: Finding Files

💡 *How can we find all configuration files in /etc?*

```
[ec2-user@ip-172-31-1-87 ~]$ sudo find /etc/ -iname "*.conf" -type f
/etc/dracut.conf.d/ec2.conf
/etc/dnf/plugins/release-notification.conf
/etc/dnf/plugins/copr.conf
/etc/dnf/plugins/debuginfo-install.conf
/etc/dnf/dnf.conf
/etc/dnf/protected.d/dnf.conf
/etc/dnf/protected.d/grub2-tools-minimal.conf
/etc/dnf/protected.d/systemd.conf
/etc/dnf/protected.d/sudo.conf
/etc/dnf/protected.d/grub2-efi-x64-ec2.conf
/etc/pki/ca-trust/ca-legacy.conf
/etc/host.conf
/etc/ld.so.conf
/etc/ld.so.conf.d/dyninst-x86_64.conf
/etc/nsswitch.conf
/etc/xattr.conf
/etc/libaudit.conf
/etc/selinux/semanage.conf
/etc/selinux/targeted/setrans.conf
/etc/request-key.conf
/etc/request-key.d/id_resolver.conf
/etc/dbus-1/session.conf
/etc/dbus-1/system.conf
```

# Linux: Finding Files

## Searching the Entire Filesystem

When searching from /, you may encounter system files and special directories like /proc or /sys, which can generate lots of error messages.

To keep your output clean, you can redirect those errors to /dev/null:

```
sudo find / -name "hosts" 2>/dev/null
```

💡 2> redirects **error output (stderr)** so you only see real results.

This is useful for full-system searches where some paths may be inaccessible or noisy.

# Linux: Finding Files

## Finding Files by Permission

System administrators often search for files with **insecure or specific permissions** to audit the system.

The `-perm` option lets you match files by their permission bits.

```
# Find files with 777 permissions (world-readable, writable, executable)
find ~ -perm 777
```

```
# Find files writable by others (world-writable)
find ~ -perm -o=w
```

```
# Find files that are not world-writable
find ~ ! -perm -o=w
```

### Key Flags:

- `-perm 777` → exact match
- `-perm -o=w` → at least world-writable
- `!` → negate condition (NOT)

 Use this to identify potential **security risks** like files that anyone can modify or execute.



# Linux: Finding Files

## Finding Insecure or Sensitive Files


System administrators can use `find` to detect **potentially dangerous** or **misconfigured** files — such as those with special permissions or exposed private keys.

```
# Find files with SUID (-4000) or SGID (-2000) owned by root
sudo find /usr -type f \( -perm -4000 -o -perm -2000 \) -user root 2>/dev/null
```

 SUID/SGID executables run with elevated privileges and should be reviewed for security risks.

You can also scan for **private key files** with insecure permissions:

```
find ~ -type f -name "*.pem" ! -perm 600
```

 PEM keys should normally be 600 (read/write for owner only).

This helps you identify any that are too open and could be accessed by others.

# Linux: Finding Files

## Finding Files by Size

The `-size` option lets you locate files based on their size — useful for finding **large logs**, **backups**, or **unusually small files**.

```
# Find files larger than 100MB  
find ~ -type f -size +100M
```

```
# Find files smaller than 1KB  
find ~ -type f -size -1k
```

```
# Find files exactly 10MB  
find ~ -type f -size 10M
```

### Suffixes:

- k → kilobytes
- M → megabytes
- G → gigabytes

💡 Use this to track down space hogs, prune old data, or monitor for unexpectedly large files in key directories.

# Linux: Finding Files

## Finding and Removing Empty Files

The `-empty` option finds **empty files or directories**, and when combined with `-delete`, it can quickly clean up unused space.

```
# Find and delete empty files  
find ~ -type f -empty -delete
```

 *Use with caution!* `-delete` permanently removes anything that matches — there's **no undo**.

Cleaning up empty files not only frees disk space but also **releases inodes**, helping keep your filesystem efficient and tidy.

# Linux: Finding Files

## Time-Based Search Options

find can filter files based on **when they were modified, accessed, or changed** — great for log rotation, cleanup, and audits.

Option	Unit	Description
-mtime	days	File <b>content</b> modified (M = modify)
-mmin	minutes	Same as above, in minutes
-atime	days	File <b>accessed</b> (read)
-amin	minutes	Same as above, in minutes
-ctime	days	File <b>metadata changed</b> (permissions, owner, etc.)
-cmin	minutes	Same as above, in minutes

 Use these to find files by their activity — whether for cleanup, backup, or security analysis.

# Linux: Finding Files

## Examples – Time-Based Searches

```
# Files modified in the last 24 hours  
find ~ -type f -mtime -1
```

```
# Files not accessed in 30 days  
find ~ -type f -atime +30
```

```
# Files whose permissions or ownership changed recently  
find /etc -type f -ctime -2
```

```
# Files modified within the last 10 minutes  
find /var/log -type f -mmin -10
```

When servers break, a good strategy for troubleshooting is to look for configuration files that had recent changes

 + means **older than**, - means **newer than**, and no sign means **exactly** that time span.

# Linux: Finding Files

## Using -exec with find

The **-exec** option lets you perform actions on files found by **find** — powerful for **batch operations** like inspection, cleanup, or fixing permissions.

```
find . -name "*.log" -exec ls -lh {} \;
```

Each **{}** represents the **current file**, and **\;** ends the command after each file.

### Variants:

- **-exec command {} \;** → runs the command **once per file**
- **-exec command {} +** → runs the command **once with all files as arguments** (faster and more efficient)

*Note:* The **+** must appear at the **end of the command**, just like **\;**, but without the backslash.

# Linux: Finding Files

## Fixing Insecure File Permissions

You can use `find` with `-exec` to locate and correct **world-writable (777)** files. This is a common **security hardening task** for system administrators.

```
# Find all files with 777 permissions
find ~ -type f -perm 777 -ls
```

```
# Change them to 644 (owner read/write, others read-only)
find ~ -type f -perm 777 -exec chmod 644 {} \;
```

```
# Verify the fix
find ~ -type f -perm 644 -ls
```

 `-perm 777` finds files where **anyone can modify or execute** — changing them to **644** helps protect integrity while keeping readability.

# Linux: Finding Files


## Searching *Inside* Files with grep

While `find` locates files by **attributes**, it doesn't look **inside** them.  
To search for text patterns — like “*Error*” messages or configuration settings — use **grep**.

```
# Search all files in /var/log for the word "error"  
grep -lri "error" /var/log
```

### Options explained:

- **-l** → show only filenames that contain matches
- **-r** → search recursively through directories
- **-i** → ignore case (matches “Error”, “ERROR”, “error”, etc.)

 Use `grep` to quickly pinpoint logs, configs, or scripts containing the information you need.



# Lab 3.3 Finding Files

Estimated Time: 30 Minutes



# POP QUIZ:

Which command can be used to find files with "http-proxy:enabled" in the file?

- A. `grep "http-proxy:enabled"`
- B. `find -search "http-proxy:enabled"`
- C. `locate "http-proxy:enabled"`
- D. `grep -rl "http-proxy:enabled"`



# POP QUIZ:

Which command can be used to find files with "http-proxy:enabled" in the file?

- A. `grep "http-proxy:enabled"`
- B. `find -search "http-proxy:enabled"`
- C. `locate "http-proxy:enabled"`
- D. `grep -rl "http-proxy:enabled"`

"Why are the flags needed?"



# POP QUIZ:

How can we print the contents of every file ending in txt?

- A. `find -type f -name "*.txt" -exec cat {} +`
- B. `find -type f -name "*.txt" -exec cat {} ;`
- C. `cat *.txt`
- D. `grep -rl "*.txt" | cat`





# POP QUIZ:

How can we print the contents of every file ending in txt?

- A. `find -type f -name "*.txt" -exec cat {} +`
- B. `find -type f -name "*.txt" -exec cat {} ;`
- C. `cat *.txt`
- D. `grep -rl "*.txt" | cat`

"Why does `cat *.txt` not work for every file?"



# POP QUIZ:

Suppose a user (penguin) was deleted and we want to find and delete any file belonging to them. Which command will do so?

- A. `find -type f -user penguin -exec rm`
- B. `find -type f -user penguin -rm`
- C. `find -type f -user penguin -delete`
- D. `userdel penguin -files`



# POP QUIZ:

Suppose a user (penguin) was deleted and we want to find and delete any file belonging to them. Which command will do so?

A. `find -type f -user penguin -exec rm`

B. `find -type f -user penguin -rm`

C. `find -type f -user penguin -delete`

D. `userdel penguin -files`

"Why does `cat *.txt` not work for every file?"





# POP QUIZ:

A Web Server is not working as 30 minutes ago. You want to investigate for recent changes. Which command may help you?

- A. `find /etc -time -60m`
- B. `find /etc -mmin 60`
- C. `find /etc -type f -mtime -60`
- D. `find /etc -type f -mmin -60`





# POP QUIZ:

A Web Server is not working as 30 minutes ago. You want to investigate for recent changes. Which command may help you?

- A. `find /etc -time -60m`
- B. `find /etc -mmin 60`
- C. `find /etc -type f -mtime -60`
- D. `find /etc -type f -mmin -60`

"What do each of these options do differently?"



# Lab 3.4 Find Scripting Challenge

Estimated Time: 60 Minutes



# Linux Day 3 Complete

🎉 Great Work Today!

Today you learned how to:

- Write and run **Bash scripts** with variables, loops, arrays, and functions
- Create and manage **users, groups, and permissions** to secure your system
- Use tools like **find** and **grep** to locate files by attributes or content

Give yourself a pat on the back — you've taken another big step toward becoming a **confident and capable Linux Administrator**.

💡 Keep practicing — the more you script, manage, and explore, the more Linux starts working *for you* instead of the other way around.

