**Seminar 2: Memory, Operating System & Device Protection**

*using WSL Ubuntu + Bash CLI + Bash Scripting*

**INTRODUCTION**

Welcome, students!

Today, we’re diving into the world of system security on Linux. Our focus will be **Memory Protection**, **Operating System Protection**, and **Device Protection**, all with the help of WSL Ubuntu and Bash scripting. This seminar will teach you to write scripts and execute commands to explore and strengthen system protection mechanisms while understanding their practical significance.

**Seminar Objectives:**

1. **Understand memory protection** in Linux using process isolation and permission management.
2. Explore **operating system protection mechanisms**, like privilege levels and system calls.
3. Practice **device protection** using access control, monitoring, and permission management.
4. Learn how **file permissions** ensure security in Linux environments.

**Setting the Stage: Understanding the Environment**

Before we dive into the exercises, let’s ensure you have the right setup:

1. **Ubuntu 24.04 running on WSL (Windows Subsystem for Linux)**.
2. **Bash CLI** and **basic scripting knowledge** (don’t worry, we’ll start from the basics!).
3. **Sudo/root access** for some operations (you’ll learn why this is crucial in protection mechanisms).

If everything’s set, let’s begin with **memory protection**.

**Section 1. MEMORY PROTECTION EXERCISES**

**Memory protection** is vital to prevent unauthorized access or modification to a process’s memory space. In Linux, this is achieved through mechanisms like process isolation, memory maps, and protection bits. Let’s explore these concepts step-by-step.

**1.1 PROCESS MEMORY ISOLATION**

**Concept:**  
Each process in Linux has its own memory space, isolated from others. If a program tries to access memory it doesn’t own, it leads to a **segmentation fault**, ensuring the system’s stability.

**Script and Explanation:**

# memory\_test.c - A C program to test memory isolation

cat > memory\_test.c << 'EOF'

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char \*ptr = (char \*)malloc(10); // Allocate 10 bytes

strcpy(ptr, "test"); // Write within allocated space

// Intentionally write beyond allocated memory

ptr[11] = 'x'; // Segmentation fault occurs here

printf("This won't be printed\n");

return 0;

}

EOF

# Compile and run

gcc memory\_test.c -o memory\_test

./memory\_test

**What’s Happening?**

* **malloc(10)** allocates 10 bytes of memory.
* Writing beyond allocated memory (at index 11) triggers a **segmentation fault**. This is Linux enforcing memory protection by terminating the offending process.

***Your Task:***

*Modify the program to see how different memory sizes affect its behavior. Try allocating more memory and accessing it within bounds.*

**1.2 INSPECTING PROCESS MEMORY**

**Concept:**  
Every process has a detailed memory map stored in the /proc filesystem. By analyzing this map, we can observe the memory layout, including stack, heap, and text segments.

**Script and Explanation:**

#!/bin/bash

# inspect\_memory.sh - Script to inspect process memory

# Run a process (sleep command in the background)

sleep 50 &

PID=$!

# Display memory maps of the process

echo "Memory map of process $PID:"

cat /proc/$PID/maps

# Display memory usage statistics

echo -e "\nMemory stats of process $PID:"

cat /proc/$PID/status | grep -E 'VmSize|VmRSS'

# Clean up

kill $PID

**What’s Happening?**

* The maps file shows the process’s memory layout, including:
  + **Text (code)**: Executable instructions.
  + **Heap**: Dynamically allocated memory.
  + **Stack**: Temporary variables and function calls.
* The status file displays memory usage statistics, like:
  + VmSize: Total virtual memory allocated.
  + VmRSS: Resident Set Size (memory actively used).

***Your Task:***

*Change sleep 50 to a program of your choice (e.g., a script or compiled binary) and inspect how its memory layout changes.*

**1.3 MEMORY PROTECTION BITS**

**Concept:**  
Memory pages in Linux have **protection bits** that define whether they’re readable, writable, or executable. Attempting to violate these permissions results in a **segmentation fault**.

**Script and Explanation:**

# readonly\_test.c - A C program to demonstrate memory protection bits

cat > readonly\_test.c << 'EOF'

#include <stdio.h>

#include <string.h>

#include <sys/mman.h>

int main() {

// Create a read-only memory page

char \*addr = mmap(NULL, 4096, PROT\_READ, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

if (addr == MAP\_FAILED) {

perror("mmap");

return 1;

}

// Attempt to write to the read-only memory

addr[0] = 'x'; // This will cause a segmentation fault

return 0;

}

EOF

# Compile and run

gcc readonly\_test.c -o readonly\_test

./readonly\_test

**What’s Happening?**

* **mmap** allocates a memory page with PROT\_READ (read-only) protection.
* Writing to the page triggers a **segmentation fault**, demonstrating Linux’s enforcement of memory protection.

***Your Task:***

*Experiment with different mmap flags, such as PROT\_WRITE and PROT\_EXEC, and observe how they affect program behavior.*

**Let’s Take a Break!**

You’ve just explored memory protection through isolation, inspection, and protection bits.

**Before Moving On:**

* 📝 **Take notes** on what segmentation faults signify and how memory protection mechanisms work.
* 💻 **Experiment** with the provided scripts to solidify your understanding.

**Next Section:**

We’ll explore **operating system protection**, focusing on privilege levels, system calls, and user roles.

**Section 2. OPERATING SYSTEM PROTECTION**

In this section, we’ll explore how the operating system enforces protection mechanisms through **privilege levels**, **system calls** and **user roles**. These ensure that only authorized actions are executed, preventing malicious or accidental system compromise.

**2.1 PRIVILEGE LEVELS AND SYSTEM CALLS**

**Concept:**

The Linux kernel operates using a concept called **protection rings**, where:

* **Ring 0**: The kernel operates with full privileges.
* **Ring 3**: User applications run with limited privileges.

To interact with the kernel, user applications use **system calls** (via predefined interfaces like open, read, and write).

**2.1.1 System Calls in Action**

**Script and Explanation:**

# system\_call\_test.c - A program demonstrating system calls

cat > system\_call\_test.c << 'EOF'

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

// Open a file (system call: open)

int fd = open("test\_file.txt", O\_CREAT | O\_WRONLY, 0644);

if (fd == -1) {

perror("open");

return 1;

}

// Write to the file (system call: write)

const char \*message = "Hello, OS protection!\n";

if (write(fd, message, 23) == -1) {

perror("write");

close(fd);

return 1;

}

// Close the file (system call: close)

close(fd);

return 0;

}

EOF

# Compile and run

gcc system\_call\_test.c -o system\_call\_test

./system\_call\_test

**What’s Happening?**

1. The program uses **system calls** like open, write, and close.
2. These requests are handed off to the kernel, which verifies permissions before execution.
3. If unauthorized, the kernel denies the request, ensuring system integrity.

***Your Task:***

1. *Modify the program to open and write to a file in /etc (e.g., /etc/test\_file.txt).*
2. *Try running it with and without sudo. Observe how the kernel enforces access restrictions.*

**2.1.2 Privilege Elevation with seteuid**

**Script and Explanation:**

# privilege\_test.c - A program requiring elevated privileges

cat > privilege\_test.c << 'EOF'

#include <stdio.h>

#include <unistd.h>

int main() {

// Attempt to change the effective user ID

if (seteuid(0) == -1) {

perror("seteuid");

return 1;

}

printf("Privilege elevated! Effective UID: %d\n", geteuid());

return 0;

}

EOF

# Compile and run

gcc privilege\_test.c -o privilege\_test

./privilege\_test # Should fail without root

sudo ./privilege\_test # Should succeed with root

**What’s Happening?**

* The program uses **seteuid** to elevate privileges.
* Running the program without sudo fails because the current user lacks the necessary permissions.
* When run with sudo, the kernel allows privilege elevation.

**Your Task:**

1. Replace seteuid(0) with another UID (e.g., your current user’s UID) and observe the behavior.
2. Investigate what happens if you remove the sudo command entirely.

**2.2 UNDERSTANDING PROTECTION RINGS**

**Concept:**  
Protection rings enforce privilege levels, restricting access to critical resources and ensuring **user-space processes** cannot directly interfere with **kernel-space operations**.

**2.2.1 Demonstrating Current Privilege Levels**

**Script and Explanation:**

# ring\_level.c - A program to display privilege levels

cat > ring\_level.c << 'EOF'

#include <stdio.h>

#include <unistd.h>

int main() {

printf("Real UID: %d\n", getuid());

printf("Effective UID: %d\n", geteuid());

return 0;

}

EOF

# Compile and run

gcc ring\_level.c -o ring\_level

./ring\_level

**What’s Happening?**

* **Real UID**: Identifies the actual user who launched the program.
* **Effective UID**: Represents the user identity used for access control (e.g., after privilege elevation).

**Your Task:**

1. Run the program with and without sudo and compare the outputs.
2. Investigate scenarios where the real UID and effective UID differ.

**2.2.2 Detecting Privilege Escalation**

**Script and Explanation:**

#!/bin/bash

# monitor\_sudo.sh - Monitor sudo usage

echo "Monitoring sudo commands in real time..."

sudo tail -f /var/log/auth.log | grep -i sudo

**What’s Happening?**

* This script continuously monitors the **auth.log** file for sudo activity.
* You can track who used sudo, when, and for what purpose.

**Your Task:**

1. Execute a few sudo commands while this script is running and observe the logs.
2. Customize the script to filter logs by a specific user or time range.

**Let’s Pause for Reflection!**

You’ve just learned how **Linux enforces privilege levels** and protects critical resources through **system calls** and **protection rings**.

**Key Takeaways:**

* **System calls** act as a gateway between user-space applications and kernel resources.
* Privilege levels (e.g., real and effective UIDs) dictate what a process can access.
* Tools like **sudo** enable temporary privilege elevation, which should be monitored carefully.

**Next Section:**

We’ll dive into **device protection**, focusing on managing access to block devices and USB peripherals.

**Section 3. DEVICE PROTECTION IN LINUX**

In this section, we’ll explore **device protection**, focusing on managing and monitoring access to devices like disks and USBs. Linux enforces device security through strict permissions, access controls, and logging mechanisms. You’ll learn how to manage device permissions, monitor device activity, and understand their security implications.

**3.1 DEVICE ACCESS CONTROL**

**Concept:**

Devices in Linux are represented as files in the /dev directory. Permissions on these device files determine which users or processes can access them. Misconfigured permissions can lead to unauthorized access, so managing them is critical.

**3.1.1 Inspecting Device Files**

**Script and Explanation:**

#!/bin/bash

# inspect\_devices.sh - List devices and their permissions

echo "Listing all block devices and their permissions..."

lsblk -o NAME,TYPE,SIZE,MOUNTPOINT,MODE

echo -e "\nListing character devices in /dev:"

ls -l /dev | grep '^c'

echo -e "\nPermissions for the main storage device (/dev/sda):"

ls -l /dev/sda

**What’s Happening?**

* **lsblk** lists block devices (e.g., disks) along with their type, size, mount point, and mode.
* **/dev** contains all device files. Files starting with c are **character devices** (e.g., keyboards, terminals), while b represents **block devices** (e.g., disks).
* Device permissions are displayed in the format crw-r-----, showing who can read/write to the device.

***Your Task:***

1. *Run the script and note the permissions for /dev/sda.*
2. *Modify permissions using chmod (explained below) and observe the changes.*

**3.1.2 Monitoring Device Access**

**Script and Explanation:**

#!/bin/bash

# monitor\_device\_access.sh - Monitor access to devices

echo "Monitoring access to /dev..."

# Check if inotifywait is installed

if ! command -v inotifywait &> /dev/null; then

echo "Error: inotifywait is not installed. Please install it with:"

echo "sudo apt update && sudo apt install inotify-tools"

exit 1

fi

# Run inotifywait with sudo

#sudo inotifywait -m /dev -e access -e modify -e open

###Each time inotifywait prints a message to the terminal, it generates another modification to /dev/tty,

####which is then detected again. This creates an endless loop of modification events. Thus:

while true; do

sudo inotifywait -m /dev -e access -e modify -e open --exclude '/dev/tty'

sleep 1

done

(install it if necessary)

**What’s Happening?**

* **inotifywait** monitors the /dev directory for access (-e access), modifications (-e modify), and file opens (-e open).
* This provides real-time insights into device usage.
* Open another terminal and type: cat /dev/null and/or echo "test" > /dev/null observe the script reactions.

***Your Task:***

1. *Run this script in one terminal.*
2. *In another terminal, interact with devices (e.g., cat /dev/null or sudo cat /dev/sda) and observe the logs.*

**3.2 MANAGING DEVICE PERMISSIONS**

**Concept:**

Device permissions are managed just like regular files using commands like chmod, chown, and ls. However, improper changes can lead to system instability or security risks.

**3.2.1 Changing Device Permissions**

**Script and Explanation:**

#!/bin/bash

# manage\_device\_permissions.sh - Modify and restore device permissions

DEVICE="/dev/null" # Change this to test other devices

echo "Original permissions for $DEVICE:"

ls -l $DEVICE

# Change permissions

echo "Modifying permissions..."

sudo chmod 600 $DEVICE # Owner: read/write; others: no access

# Display modified permissions

echo "Modified permissions:"

ls -l $DEVICE

# Restore original permissions

echo "Restoring original permissions..."

sudo chmod 666 $DEVICE

ls -l $DEVICE

**What’s Happening?**

1. **chmod 600** restricts the device so only the owner can read/write.
2. **chmod 666** restores full access for everyone.
3. By managing permissions, you can limit unauthorized access to sensitive devices.

***Your Task:***

1. *Test the script on different devices (e.g., /dev/sda, /dev/tty).*
2. *Experiment with permissions like chmod 640 and observe the impact.*

**3.2.2 Creating and Managing Custom Devices**

**Script and Explanation:**

#!/bin/bash

# create\_test\_device.sh - Create and manage a custom device

# Create a test directory for devices

sudo mkdir /dev/test

cd /dev/test

# Create a character device

echo "Creating a test character device..."

sudo mknod testdev c 1 3 # Major=1, Minor=3 corresponds to /dev/null

# Set restricted permissions

echo "Setting permissions for testdev..."

sudo chmod 600 testdev

# Verify device details

echo "Device details:"

ls -l testdev

file testdev

# Clean up

cd ..

sudo rm -r /dev/test

**What’s Happening?**

1. **mknod** creates a character device named testdev with major/minor numbers 1 and 3 (same as /dev/null).
2. Restricted permissions are applied to demonstrate access control.
3. The script cleans up after itself to avoid clutter.

**Your Task:**

1. *Change the mknod parameters to create other types of devices (e.g., b for block devices).*
2. *Test access to the created device with and without sudo.*

**3.3 DEVICE ACCESS MONITORING IN REAL-TIME**

**Concept:**

System administrators often need to monitor devices for unauthorized or suspicious activity. This can be automated using udev rules or monitoring tools.

**~~3.3.1 Real-Time USB Monitoring~~**

**~~Script and Explanation:~~**

~~#!/bin/bash~~

~~# monitor\_usb.sh - Monitor USB device activity~~

~~echo "Monitoring USB device connections and disconnections..."~~

~~udevadm monitor --udev --subsystem-match=usb~~

**~~What’s Happening?~~**

* **~~udevadm monitor~~** ~~observes udev events for USB devices.~~
* ~~The script logs real-time connections and disconnections of USB peripherals.~~

**~~Your Task:~~**

1. ~~Plug and unplug USB devices while the script is running. Observe the logs.~~
2. ~~Customize the script to log events to a file for auditing purposes.~~

**Let’s Pause for Reflection!**

You’ve now learned how to **manage and monitor device permissions** and understand how devices are represented in Linux. 🎉

**Key Takeaways:**

* Devices are represented as files in /dev. Permissions on these files control access.
* Tools like **inotifywait** and **udevadm** help monitor real-time device activity.
* Proper permission management prevents unauthorized access to critical devices.

**Next Section:**

We’ll explore **file permission protection** in Linux, focusing on the role of file ownership, permission bits, and special attributes like SetUID.

**Section 4. FILE PERMISSION PROTECTION IN LINUX**

In this section, we’ll explore how Linux uses **file permissions** to protect the system and ensure security. Permissions are foundational to Linux security, determining who can read, write, or execute a file. You’ll learn how to manipulate permissions, understand their implications, and apply special attributes like SetUID for advanced control.

**4.1 BASICS OF FILE PERMISSIONS**

**Concept:**

Every file and directory in Linux has three permission sets for:

1. **Owner**: The user who owns the file.
2. **Group**: A set of users who can access the file.
3. **Others**: All other users on the system.

Each permission set controls:

* **Read (r)**: View file contents or list a directory.
* **Write (w)**: Modify file contents or create/delete files in a directory.
* **Execute (x)**: Run a file as a program or enter a directory.

**4.1.1 Viewing and Setting File Permissions**

**Script and Explanation:**

#!/bin/bash

# manage\_file\_permissions.sh - Manage file permissions

# Create test files

echo "Creating test files..."

touch file1.txt file2.txt file3.txt

# Set specific permissions

echo "Setting permissions..."

chmod 400 file1.txt # Read-only for owner

chmod 200 file2.txt # Write-only for owner

chmod 100 file3.txt # Execute-only for owner

# Display permissions

echo "Displaying file permissions:"

ls -l file\*.txt

# Test access

echo -e "\nTesting access:"

echo "Reading file1.txt:"

cat file1.txt 2>&1 # Should succeed

echo "Writing to file2.txt:"

echo "test" > file2.txt 2>&1 # Should fail

echo "Executing file3.txt:"

./file3.txt 2>&1 # Should fail (not a script)

# Clean up

rm -f file\*.txt

**What’s Happening?**

* **chmod** modifies file permissions using numeric codes (400 = read-only for owner).
* **ls -l** displays the file’s permissions in the format -rw-------, where:
  + r = read, w = write, x = execute.
* Testing access (cat, echo, ./) shows how permissions affect file usability.

***Your Task:***

1. *Modify the script to test other combinations (e.g., chmod 640).*
2. *Add a directory to the script and test directory permissions.*

**4.1.2 The umask Command**

**Concept:**  
The **umask** command sets default permissions for newly created files. By subtracting the umask value from full permissions (777 for directories, 666 for files), you determine the defaults.

**Example:**

#!/bin/bash

# test\_umask.sh - Test default permissions

# Show current umask

echo "Current umask: $(umask)"

# Create a new file

touch test\_file.txt

# Display default permissions

echo "Permissions of test\_file.txt:"

ls -l test\_file.txt

# Clean up

rm -f test\_file.txt

***Your Task:***

1. *Change the umask value (umask 027) and test how it affects newly created files.*
2. *Explain why a strict umask (e.g., 077) is useful in secure environments.*

**4.2 SPECIAL PERMISSIONS: SetUID, SetGID AND Sticky Bit**

**Concepts:**

1. **SetUID (s for user)**: Allows a program to run with the privileges of its owner.
2. **SetGID (s for group)**: Sets group privileges for a program or enforces group ownership in directories.
3. **Sticky Bit (t)**: Prevents users from deleting files in a shared directory unless they own the file.

**4.2.1 Using the SetUID Bit**

**Script and Explanation:**

# setuid\_test.c - Demonstrate the SetUID bit

cat > setuid\_test.c << 'EOF'

#include <stdio.h>

#include <unistd.h>

int main() {

printf("Real UID: %d\n", getuid());

printf("Effective UID: %d\n", geteuid());

return 0;

}

EOF

# Compile and set SetUID

gcc setuid\_test.c -o setuid\_test

sudo chown root:root setuid\_test

sudo chmod u+s setuid\_test

# Run the program

./setuid\_test

**What’s Happening?**

* **chown root:root** changes ownership to root.
* **chmod u+s** sets the SetUID bit. Now, the program runs with the owner’s privileges (root).
* The program displays both the **real UID** (your user) and the **effective UID** (root).

***Your Task:***

1. *Replace root ownership with another user and observe the changes.*
2. *Explain scenarios where SetUID can be dangerous (e.g., a poorly written SetUID program).*

**4.2.2 Using the Sticky Bit**

**Example and Explanation:**

#!/bin/bash

# test\_sticky\_bit.sh - Demonstrate the sticky bit

# Create a shared directory

sudo mkdir /tmp/shared\_dir

sudo chmod 777 /tmp/shared\_dir

# Set the sticky bit

sudo chmod +t /tmp/shared\_dir

# Display permissions

ls -ld /tmp/shared\_dir

**What’s Happening?**

* **chmod +t** adds the sticky bit (t) to the directory.
* Now, only the file’s owner can delete files inside the directory, even if others have write permissions.
* You can open winexplorer at [\\wsl.localhost\Ubuntu\tmp](file:///\\wsl.localhost\Ubuntu\tmp) to see that shared\_dir ; try to delete it
* Works fine only with $ sudo rmdir /tmp/shared\_dir

***Your Task:***

1. *Create files in the shared directory as different users and test deletion.*
2. *Explain how the sticky bit is useful for shared directories like /tmp.*

**4.3 MONITORING AND AUDITING PERMISSIONS**

**Concept:**

Monitoring permission changes is essential for detecting unauthorized activities. Tools like inotifywait help track changes in real time.

**4.3.1 Monitoring Permission Changes**

**Script and Explanation:**

#!/bin/bash

# monitor\_permissions.sh - Monitor permission changes

# Monitor the current directory for permission changes

echo "Monitoring permission changes in $(pwd)..."

inotifywait -m . -e attrib | while read path action file; do

echo "Permission change detected!"

echo "File: $file"

echo "New permissions: $(stat -c '%A' "$file")"

echo "Owner: $(stat -c '%U' "$file")"

echo "Group: $(stat -c '%G' "$file")"

echo "---"

done

**What’s Happening?**

* **inotifywait** watches for attrib events, triggered by permission changes.
* The script logs changes, including new permissions, owner, and group details.

***Your Task:***

1. *Change permissions of files in the monitored directory and observe the logs.*
2. *Extend the script to log changes to a file for auditing purposes.*

**Let’s Pause for Reflection!**

You’ve now explored **file permission basics**, **special attributes**, and **real-time monitoring**. 🎉

**Key Takeaways:**

* Permissions (rwx) control how files and directories are accessed.
* **Special attributes** like SetUID, SetGID, and the sticky bit provide advanced control but must be used cautiously.
* Monitoring tools like **inotifywait** help detect and log unauthorized permission changes.

**Next Section:**

In the final section, we’ll tackle **practice exercises** that consolidate your learning and explore additional security challenges.

**Section 5. PRACTICE EXERCISES AND CHALLENGES**

Welcome to the final part of this seminar! Here, we’ll consolidate everything you’ve learned with **hands-on exercises** and **challenges**. These tasks are designed to deepen your understanding of memory protection, OS security, device management, and file permissions in Linux.

**5.1 MEMORY PROTECTION CHALLENGES**

**5.1.1 Stack Overflow Protection**

**Exercise: Create a Program to Test Stack Overflow Protection**

# stack\_overflow.c - Demonstrate stack overflow

cat > stack\_overflow.c << 'EOF'

#include <stdio.h>

void cause\_overflow() {

char buffer[10]; // Small buffer

for (int i = 0; i < 20; i++) {

buffer[i] = 'A'; // Write beyond the buffer

}

}

int main() {

printf("Testing stack overflow protection...\n");

cause\_overflow();

return 0;

}

EOF

# Compile and run

gcc -fstack-protector -o stack\_overflow stack\_overflow.c

./stack\_overflow

***Questions to Answer:***

1. *What happens when stack protector (-fstack-protector) is enabled versus disabled?*
2. *Why is stack overflow protection critical for system security?*

**5.1.2 Monitoring Memory Usage**

**Exercise: Write a Script to Monitor and Alert on High Memory Usage**

#!/bin/bash

# monitor\_high\_memory.sh - Monitor memory usage and alert if usage exceeds threshold

THRESHOLD=80 # Set memory usage threshold (percentage)

while true; do

USAGE=$(free | awk '/^Mem/ { printf("%.0f", $3/$2 \* 100) }')

if [ "$USAGE" -gt "$THRESHOLD" ]; then

echo "WARNING: Memory usage is at $USAGE%!"

fi

sleep 5

done

**Questions to Answer:**

1. How can you adapt this script to log alerts into a file?
2. What real-world scenarios might require monitoring high memory usage?

**5.2 OS PROTECTION CHALLENGES**

**5.2.1 Implement Privilege Dropping**

**Exercise: Create a Privileged Program That Safely Drops Privileges**

# drop\_privileges.c - Drop privileges safely

cat > drop\_privileges.c << 'EOF'

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main() {

// Drop privileges

if (setuid(getuid()) == -1) {

perror("setuid");

return 1;

}

printf("Privileges dropped! Effective UID: %d\n", geteuid());

// Simulate privileged action

system("echo 'Performing a low-privilege action...'");

return 0;

}

EOF

# Compile and run

gcc drop\_privileges.c -o drop\_privileges

sudo ./drop\_privileges

***Questions to Answer:***

1. *Why is privilege dropping important for secure programming?*
2. *What risks arise if privileges aren’t dropped after executing privileged operations?*

**5.3 DEVICE PROTECTION CHALLENGES**

**5.3.1 Log Unauthorized Device Access Attempts**

**Exercise: Write a Script to Log Unauthorized Access to /dev/sda**

#!/bin/bash

# log\_device\_access.sh - Log unauthorized attempts to access /dev/sda

LOGFILE="/var/log/device\_access.log"

echo "Monitoring unauthorized access to /dev/sda..."

sudo inotifywait -m /dev/sda -e access -e open | while read path action file; do

echo "$(date): Unauthorized access attempt on /dev/sda" >> $LOGFILE

done

***Questions to Answer:***

1. *What would happen if the user running the script doesn’t have permissions to /dev/sda?*
2. *How can you ensure logs are protected from tampering?*

**5.4 FILE PERMISSION CHALLENGES**

**5.4.1 Create a Secure Directory for Shared Use**

**Exercise: Set Up a Directory with Controlled Access**

#!/bin/bash

# secure\_shared\_dir.sh - Set up a secure shared directory

SHARE\_DIR="/tmp/secure\_shared"

echo "Creating shared directory..."

sudo mkdir -p $SHARE\_DIR

sudo chmod 770 $SHARE\_DIR # Full access for owner and group, none for others

sudo chown $USER:$USER $SHARE\_DIR

echo "Directory $SHARE\_DIR created with permissions:"

ls -ld $SHARE\_DIR

***Questions to Answer:***

1. *What happens if you set the sticky bit on this directory?*
2. *How would you adapt this script to allow multiple groups access?*

**5.4.2 Audit File Permission Changes**

**Exercise: Write a Script to Log File Permission Changes**

#!/bin/bash

# log\_permission\_changes.sh - Log file permission changes in a directory

MONITOR\_DIR=$(pwd)

LOGFILE="permission\_changes.log"

echo "Monitoring permission changes in $MONITOR\_DIR..."

inotifywait -m $MONITOR\_DIR -e attrib | while read path action file; do

echo "$(date): Permissions changed for $file" >> $LOGFILE

done

***Questions to Answer:***

1. *How can this script help in detecting malicious activities?*
2. *What additional events (e.g., create, delete) could you monitor for better auditing?*

**5.5 ADVANCED CHALLENGE: CREATE AN ACCESS CONTROL SCRIPT**

**Exercise: Implement a Simple Access Control System for a Directory**

#!/bin/bash

# access\_control.sh - A simple directory access control system

CONTROL\_DIR="/tmp/restricted\_dir"

# Set up the directory

sudo mkdir -p $CONTROL\_DIR

sudo chmod 700 $CONTROL\_DIR

sudo chown $USER:$USER $CONTROL\_DIR

echo "Access control enabled for $CONTROL\_DIR. Only $USER can access it."

# Monitor unauthorized access attempts

echo "Monitoring unauthorized access attempts..."

sudo inotifywait -m $CONTROL\_DIR -e access -e open | while read path action file; do

echo "$(date): Unauthorized access attempt on $CONTROL\_DIR" >> /var/log/access\_control.log

done

***Questions to Answer:***

1. *How can you notify the system administrator in real time for unauthorized access attempts?*
2. *What are the limitations of using inotifywait for access control?*

**LET’S WRAP UP!**

**Congratulations!**

You’ve now explored:

* **Memory protection mechanisms** like stack overflow protection and memory monitoring.
* **OS security features**, including privilege dropping and real-time monitoring.
* **Device protection techniques** through access control and permission management.
* **File permission best practices** and auditing for a secure Linux environment.

**KEY SKILLS YOU’VE GAINED:**

* Writing Bash scripts to automate security tasks.
* Implementing Linux protection mechanisms for memory, devices, and files.
* Monitoring and auditing real-time events for improved security.

***YOUR NEXT STEPS:***

1. *Apply these concepts to real-world scenarios, such as securing a multi-user system.*
2. *Explore tools like* ***AppArmor****,* ***SELinux****, or* ***auditd*** *for advanced Linux security.*
3. *Experiment with combining these techniques in a project, such as securing a server or developing a monitoring system.*

**Questions or Ready for Feedback?**

If you’d like to dive deeper into any topic or have questions about the exercises, let me know!