Lab 04- Working with File Permissions

Name: Olivia Flores
Course: IS 3033-ON2
Date: 2/10/2024

INTRODUCTION

In this lab, I will be going over Linux-style access controls on files on an Ubuntu server that I have already previously installed on my Virtual Machine. Some of the main objectives we will cover are how inodes and files are linked, creating Linux user accounts with passwords, the impact of setUID, getting familiar with Linux groups, and customizing extended file permissions. My goal in this lab is to help myself get familiar with an access control system and how in depth it can get with the matter of rules that are set regarding whether subjects are allowed to grant actions like read, write, or execute on the system's objects. Later in this lab, I will introduce to you how I was able to create a group, along with some users and their permissions, as well as some of the fun commands that I used like chmod, chown, and useradd.

Access control models can include attribute-based access control (ABAC), role-based access control (RBAC), rule-based access control (also RBAC), mandatory access control (MAC) and discretionary access control (DAC), or a combination of these. The traditional Unix security model is based on the discretionary access control (DAC) model, which enables users to configure who can access the resources that they "own" [1]. This is exactly what I will be demonstrating as I will be accessing the majority of the users from my Ubuntu server. An important thing about Unix that I want you to note is that it simplifies permissions by only defining rules for the following kinds of subjects:

1. The user that owns the file (u)
2. The file's group (g)
3. Other users (o)

To begin this lab, I will warm up by SSH'ing into my Ubuntu server terminal and use the ls command to display the permissions for the files in the **/bin** directory. You will see that the /bin directory contains essential command-line tools that are commonly used for basic system administration and regular user tasks. Down below is the initial output of the **ls -al /bin** command:

```
total 115732
drwxr-xr-x  2 root root       36864 Feb 26 02:28 .
drwxr-xr-x 14 root root        4096 Aug 10  2023 ..
-rwxr-xr-x  1 root root       51632 Feb  7  2022 [
-rwxr-xr-x  1 root root       35344 Oct 19  2022 aa-enabled
-rwxr-xr-x  1 root root       35344 Oct 19  2022 aa-exec
```

Figure 1: Contents in the /bin directory file.

The "-l" flag instructs **ls** to provide this detailed output. I was also able to recognize that the first part of the output contains the Unix file permissions for the file and what access is authorized for the user, group, and others for example: **drwxr-xr-x**, which I will demonstrate briefly.

When there is a file that has the following permissions listed as : **rwx** we can note that this means that either the user, group or other users can read (r) the contents of the file, write (w) to the contents of the file/make changes to the file, and execute (x) the file as a process, that is if all three of those permissions are listed. On the other hand, directories are almost the same as such files, but for directories, if these permissions are given to the users, groups, or other users, they can see what files are in the directory (r), modify these directories by adding, renaming or deleting names (w), or ‘stat’ (x) the file which essentially is being able to view the file owners and sizes, change directories into that directory and access files within. There can also be a **t (sticky bit)** in place of x within the directory permissions which restricts who can delete files in a directory on Linux Systems. When the sticky bit is set, only the user that owns, the user that owns the directory, or the root user can delete files within the directory [2].

The rest of the output from ls describes how many names/hard links the file has, who owns the file (user and group associated with the file), the file size in bytes, the last access date, and finally, the path and name of the file [1]. Later on in this lab, I will walk you through how file permissions work, how I created some users and groups and test out their access to a file.

Breakpoint 1

In step 1 of this lab, I will demonstrate how two copies of a file can be linked, to where (almost) any modifications apply to both files. In Linux systems, the data structure that does the actual storing of information is called an **Inode**. When accessing an inode, you will be able to see its inode number which defines the location of the file on the disk, along with attributes like the Unix file permissions and access times for the file [1].

As we dive into this next step, I would like you to note that a **hard link** is a file all on its own (a copy of the original file), and the file points to the exact spot on a hard drive where the inode stores the data. A **soft link** is different in that it isn't a separate file, it points to the name of the original file, rather than to a spot on the hard drive. The main difference between the two is that if you delete the original file, it will not affect the hard link from operating, while deleting the soft link would make it inoperable [3].

Let’s start by using the ls -i command to view the inode of the /bin/ls directory using the command:

ls -i /bin/ls

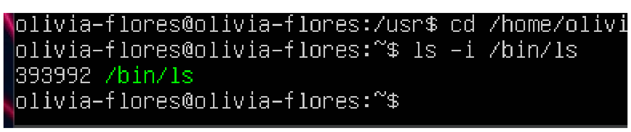


Figure 2: Displaying the inode of the /bin/ls directory.

Note that the inode above does not contain the file’s name; rather, a directory that can contain names that point to inodes. Therefore, we can go ahead and test this out by creating another name (i.e., hard link) that points to the same file. That leads me into our first goal, creating a hard link to the ls program. I can start doing this by inputting the command: **sudo ln /bin/ls /tmp/ls**

I always like to make sure that I can view the changes made by using the ls -l command and here we can use: **ls -l /tmp/ls** to view the new filename known as /tmp/ls.

Now to go back and see the new filename’s inode number using the ls -i command: **ls -i /tmp/ls** as displayed below.

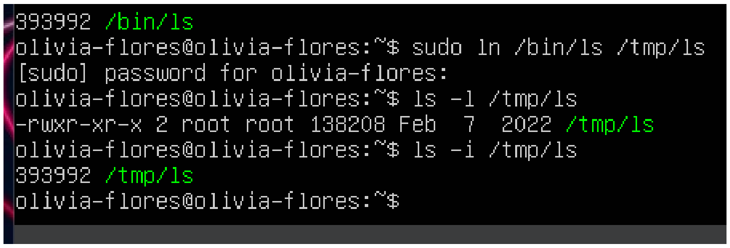


Figure 3: Displaying the same inode number for both the /bin/ls and /tmp/ls directories.

This allows me to see that evidently, both the original and new filename have the same inode number on Figure 3. I want to clarify that there is only one file, but that data can now be accessed using two different names, **/tmp/ls** and **/bin/ls**. If I was to make changes to the /tmp/ls file, those same changes would be made to the other file using the /bin/ls command.

As I introduced sticky bits earlier, the reason why I can see the /tmp directory having one is because as mentioned earlier, the sticky bit is used to prevent unprivileged users from removing or renaming a file in the directory unless they own the file or the directory; this is called the restricted deletion flag for the directory, and is commonly found on world-writable directories like /tmp [4]. In my opinion, the sticky bit set is not necessarily a grand vulnerability, but more of a mechanism for managing file access and preventing accidental deletion of files. The vulnerability can be presented when there is a directory overwritten by an attacker. The reason why this would happen is because the directory for temporary files has to be world writeable to be usable for all users [4].

Next, I am going to demonstrate what happens when I try deleting the /tmp/ls hard link. Deleting one of the names simply decrements the link counter. Only when that reaches 0 is the inode removed [1]. As you can see from the figure below, the 'Operation not permitted' when I try to delete the link as a regular user, but as a root user, deleting the /tmp/ls link is possible and you can see that this link is owned and created by the root user. In addition, you can also see the t in place of the x within the directory permissions, signifying the sticky bit.

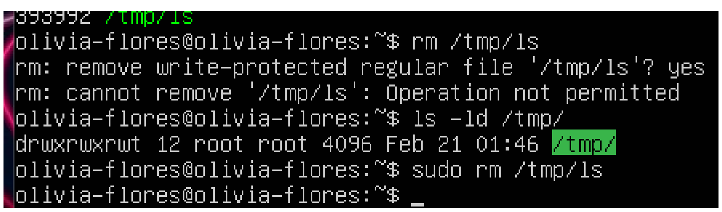


Figure 4: Deletion of hard link as root user.

The stat command can be used to display further information from the inode: **stat /bin/ls**

Looking through the output, I can see that it includes the access rights, along with the last time the file was accessed, modified, and when the inode was last changed (even if the hard linked file is now deleted). Since we now know what inodes are, and the basic output of the command shown in the figure below, an attacker might want to use this information for plenty of things like gathering information on files and directories on a system and finding their potential targets. An attacker can also replace a critical file because as described earlier, these /tmp files are world writable files. This stat command gives the attacker the ability to see when the last times were that the file was accessed or modified even after deletion, helping them get an idea or pattern of the file usage. The stat command can give an attacker or a security analyst so many valuable insights, but it is important to practice using it with reason, we now learned that inodes can be very important in a world of cybersecurity.

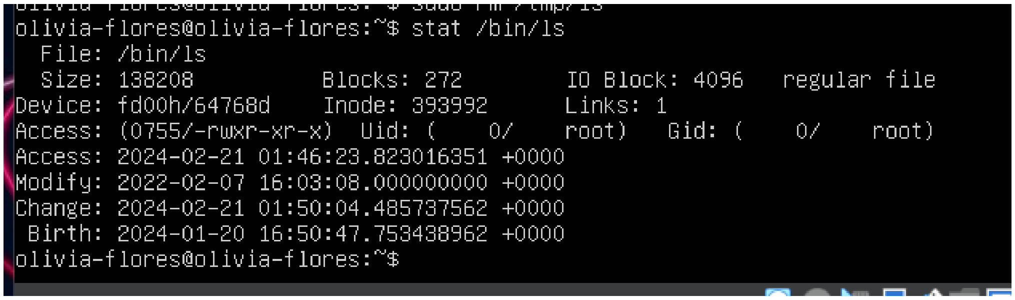


Figure 5: Stats of /bin/ls directory after deletion.

The output from stat includes the format that the information is stored as, along with a more "human readable" output. As we know, user accounts are referred to by UIDs by the system, in this case the UID is 0, as the file is owned by the root user. Similarly, groups are identified by GID, in this case also 0. The actual permissions are stored as four octets (digits 0-7), in this case "0755". This translates to the (now familiar) human-friendly output, **"-rwxr-xr-x"**[1].

Each of the other three octets simply represents the binary for rwx, each represented as a 0 or a 1. The first of the three represents the user, then the group, then the other permission. An easy and quick way to do the conversion is to simply remember that:

- **r = 4**
- **w = 2**
- **x = 1**

And add them together to produce each of the three octets. So for example, rwx = binary 111 = (4 + 2 + 1) = 7. Likewise, r-x = binary 101 = (4 + 1) = 5. Therefore, "-rwxr-xr-x" = 755 [1]. We will cover more on how these permissions can work throughout this lab.

Breakpoint 2

For the rest of this lab, I will be acting as a system administrator for an organizational group. I will demonstrate how I was able to create two new users on my Ubuntu server, and one new user on my Kali client, and set passwords for all three.

Here, I chose ***yellowstone*** as my organization, with ***John*** *and* ***Beth*** as users on the server. ***Jamie*** is another user on my Kali terminal. However, *Jamie* has later obtained my username (olivia-flores) and password and can SSH into the server as me to scope it out. In this step of the lab, I will create my own set of users, passwords, and permissions!

Down below, you can see that I was able to create the two users, John and Beth, as sudo, along with creating their passwords. To do that, I simply used the **sudo useradd -mc** command. The -m will create a default home directory for both of these users, which is important to note for later on in the lab as well. The -c allows for a comment which isn't necessarily important, but I still added one.

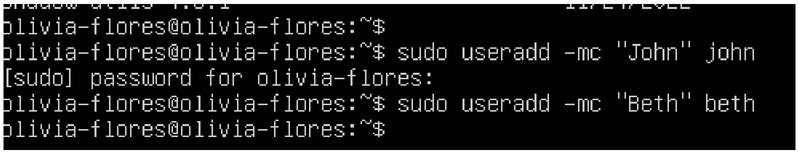


Figure 6: Adding my two users, John and Beth.

Next, I want to SSH into the server as John, then navigate to the parent directory, and list the creation of my two new users. To ssh into the server, I first made a long list of things to begin with and I will briefly explain. I started out by making sure I had my Virtual machine network settings to use the bridged adapter solution, then verified I could ping both my Ubuntu server and Kali client to each other's ip addresses. As a security administrator I want to test out a couple things and make sure that I am following all the right steps, so I want to verify things are working along the way using commands like ls -l or simply asking a generative AI tool for direction when I am not sure which command to use.

So down below I've continued by SSH'ing in the Ubuntu server as John using the **ssh** **john@192.168.1.11** command followed by the password, then was able to navigate to the parent directory shown in figure 7.

The following command is used to provide a detailed listing of files and directories in the current working directory and the whoami command displays the user, both which will be used plenty throughout the lab:

ls -al

whoami

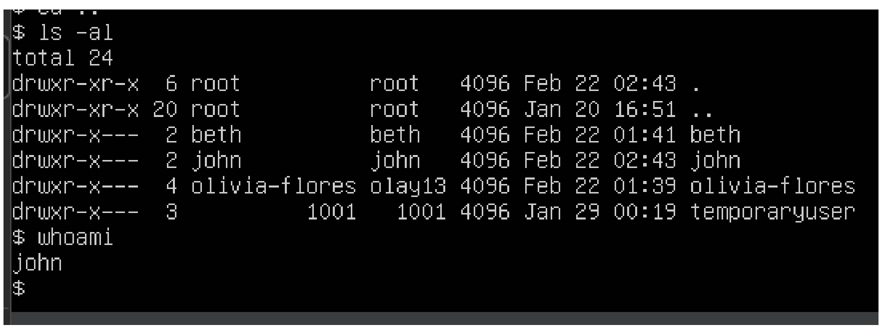


Figure 7: Detailed listing of files and directories in the current working directory.

Next, I logged into my Kali machine with admin privileges and added the third user, Jamie using the same command: **sudo useradd -mc "Jamie" Jamie** as displayed in the figure below:

Figure 8: Adding user to kali machine.

Next, I will demonstrate how I was able to harden the passwords for the users I just created. The passwd command is also used to set and change passwords, and to check the status of a user account, expire a password, set password minimum and maximum lifetimes, disable a user account, and enable a user account.

As demonstrated in the Lab 4 instructions, the following flags can be used to set minimum password lifetime **(-n)**, maximum password lifetime **(-x),** warning before expiration **(-w),** and inactive to disabled **(-i)** in days for each. I did not intentionally put them in the same order, but the order does not matter regardless. It's good to set the **-n** to at least one day because this prevents a user from repetitively changing their passwords, I set mine to two days and the other user to one day [1]. These setting help contribute to the security of the system by reducing the risk associated with long-lived passwords and by promoting users to re-do their passwords periodically, just like any job or organization usually has us keep up with password protection policies.

The figure below shows an example of how I set the password requirements using sudo for both John and Beth. In addition to the password requirements above, I wanted to test out the **-e** flag displayed in the man page for beth's password requirements. This **-e** automatically expires the user's current password if a date is not specified and prompts the user to enter the current and new password.

Commands: **sudo passwd -n 2 -x 100 -w 5 -i 15 john**

sudo passwd -n 1 -x 100 -w 3 -i 15 -e beth

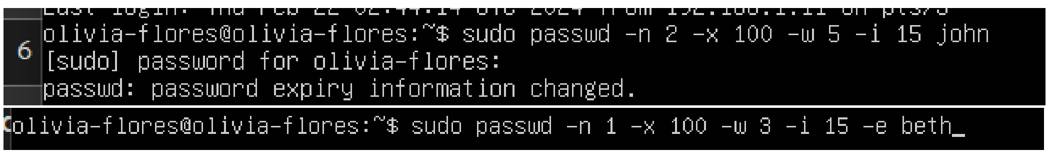


Figure 9: setting password requirements for john and beth.

Next, we verify to see if the password requirements have been changed for the following users. I had no problem signing in as John. I then SSH'ed into Beth's account, and I was able to verify that the -e flag that was set by me, worked along with the other password requirements because I was prompted to enter the current password and create a new password for Beth. Mission accomplished!

Command used to test John's password requirements: **sudo passwd john**

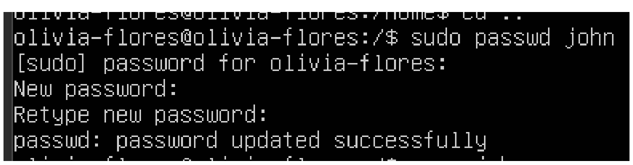


Figure 10: Changing a password for John.

Figure 11: Expired password for Beth, requiring new password change.

I was able to find out more on the password requirements filtering through the **'^password'** in the **/etc/pam/d/common-password** file. The "^" in the regular expression refers to the character after a line break, meaning that "password" should fall at Next, the beginning of a line [1].

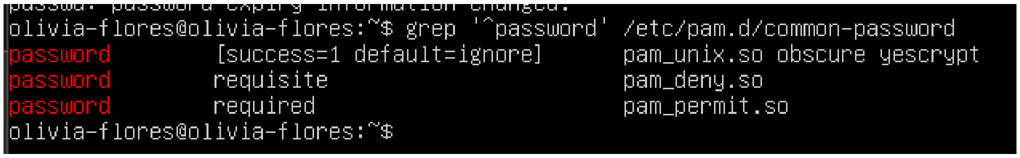


Figure 12: Viewing existing password requirements.

To dive further into changing password parameters, I wanted to see how to set password parameters using the PAM configuration file listed above in the path to common-password, known as **pam.d.** Linux-PAM (short for Pluggable Authentication Modules which evolved from the Unix-PAM architecture) is a powerful suite of shared libraries used to dynamically authenticate a user to applications (or services) in a Linux system.[5] I was not sure how to do make these changes, so I asked ChatGPT to explore a bit more about PAM and how I would go about changing parameters in the file to include a simple password requirement. I kept going in a loop and had many failed attempts at changing password requirements with the knowledge I had initially, but eventually I was able to successfully make changes.

My prompt to ChatGPT was as follows:

"Hi am trying to change the password requirements in my pam.d file, can you help me do that?"

The Generative AI tool recommended that I open the PAM configuration file and something I noted from them was that the PAM configuration files are typically located in the **/etc/pam.d/** directory. The file I am interested in is the **/etc/pam.d/common-password** file as it has already been introduced to us and should be the file I want to make changes to. Next, ChatGPT recommended that I modify the password requirements within this file once I opened it using the command: **sudo nano /etc/pam.d/common-password** and followed the recommendation with the ones below:

"Modify the password requirements: Within the file, you'll find lines that control password policies. The syntax may vary slightly depending on your system, but it's usually a line containing **password requisite** or similar. Look for lines related to password policies.
Edit the line or add new lines: Depending on your specific requirements, you can modify the existing line or add new lines. For example, to set a minimum password length, you might add minlen=12:
This would enforce a minimum password length of 12 characters.
Save the changes and exit the editor.
Test the new configuration: After saving the changes, try changing the password to see if the new requirements are enforced" [6].

As I continued to test this out, I was not sure where exactly to insert the password requirements I was wanting to add, but I tried inserting them where I believe they would go regardless as displayed in the figure below which is probably not safe in any means if there is no back up to this file. Linux systems have a common practice to create backup files or keep old versions of configuration files before making changes. In this case, there was a backup file created with the name **etc/pam.d/common-password.pam-old** signifying the old file before changes were made and I can revert back to, as I wish. This brings me to my next crucial point, PAM has the potential to seriously alter the security of your Linux system. Erroneous configuration can disable access to your system partially, or completely. For instance an accidental deletion of a configuration file(s) under **/etc/pam.d/*** and/or **/etc/pam.conf** can lock you out of your own system, so let's be very careful while making these changes if so [5].

```
# here are the per-package modules (the "Primary" block)
password        [success=1 default=ignore]      pam_unix.so minlen=12 minclass=2 difok=5 d
```

Figure 13: incorrect way to change password requirements in pam.d configuration file.

I kept going back and inputting the command after the file changes were saved using the command: **journalctl | grep pam** to make sure that there were no errors before running the password changes for the users, but I continued to see errors. So, I continued to ask ChatGPT what to do if this was not working for the PAM configuration file, I was trying to make changes to. I eventually understood that I was wanting to make changes to the configuration file but where I was placing my password requirements was obviously an issue because when I would run the **journalctl | grep pam** command, I would see a log error that specified **"minlen** is not allowed for this module type of **pam_unix**." The answer was right in front of me.

Since this was still part of my issue, I then asked ChatGPT another question about what to do if this specification (pam_unix) is not allowed. The Generative AI tool suggested that I **"Modify the password line to include pam_pwquality for password quality checking in the common-password file and to remove the minlen option from the pam_unix line."**[6]
I continued to add a line that displayed the pam_pwquality.so and added minlen=8 right next to it, but there was still an issue when I ran the **journalctl | grep pam** command, because I was trying to add a faulty module named: **pam_pwquality.so**.

Well that made sense, so next I wanted to see how to add that module into the configuration file. I asked ChatGPT again, what do I do in this case? It suggested that I run the **sudo apt-get install libpam-pwquality** to add this to my configuration file. I installed it and when I went and opened the **common-password** file, it already appeared different since the **password requisite pam_pwquality.so retry=3** portion of it was automatically added to the file after the installation, which looked like good news to me!
I went ahead and added the password requirements I had been trying to add as: **minlen=8** as shown below:

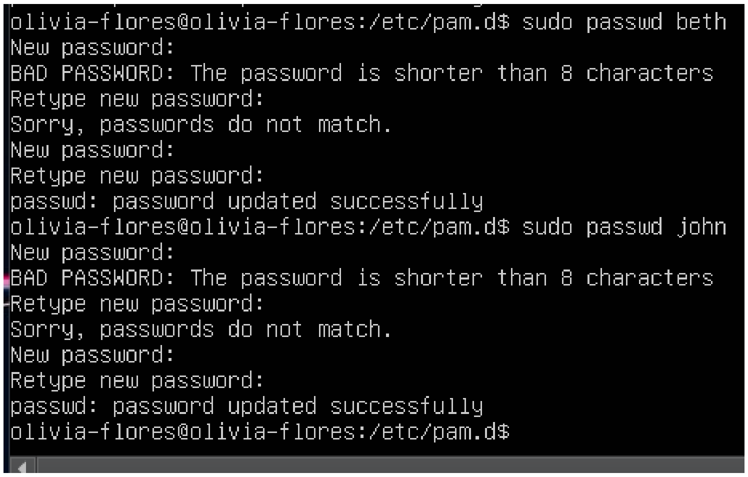
```
# here are the per-package modules (the "Primary" block)
password        requisite                       pam_pwquality.so retry=3 minlen=8
password        [success=1 default=ignore]      pam_unix.so obscure use_authtok try_first_pass yesc>
# here's the fallback if no module succeeds
```

Figure 14: pam.d file after installing pwquality.

As suggested by the Generative AI tool, I went ahead and reloaded the PAM configuration. This is important because depending on the system, you might need to restart services or reload PAM configuration for the changes to take effect. This can typically be done with a command such as:

sudo systemctl restart system-logind

Finally, I wanted to test the password requirements I set by attempting to change a user password that does not meet the requirements. I attempted to do the same for both users, Beth and John. We have now successfully set the requirements for users attempting to log in with a minimum password length of 8.



```
olivia-flores@olivia-flores:/etc/pam.d$ sudo passwd beth
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
Sorry, passwords do not match.
New password:
Retype new password:
passwd: password updated successfully
olivia-flores@olivia-flores:/etc/pam.d$ sudo passwd john
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
Sorry, passwords do not match.
New password:
Retype new password:
passwd: password updated successfully
olivia-flores@olivia-flores:/etc/pam.d$
```

Figure 15: Minimum password length requirement set.

BREAKPOINT 3

Moving on to step 3 of this lab, as system administrator, I am going to create a script that can be owned by me, but can be executed by anyone using commands such as sudo.

In addition to the three standard modes (r, w, x) we briefly went over in step 1 of this lab, we can also identify a set of permissions that are collectively known as **special modes: setUID**, **setGID** and **sticky bit**. In this case, I will be using the setUID for an executable file named lab04.sh, this means the program will run with the privileges of the file's owner (olivia-flores) instead of the user who launched it, in this case, root. However, this default behavior can be changed by modifying the **setuid** flag on the file permissions bits. To start, I will launch the lab04.sh file as root user, then set the Setuid bit through the **chmod** command to further control the file permissions, this will be displayed in the first position of **"4475"** displayed in figure 14 below.

You will also notice in this figure, that in the output of the **ls -al** command, the lab04.sh file shows that it is owned by the root user, it also includes a capital S that indicates that only the set UID flag is on and the executable flag is not. Just as demonstrated in the lab instructions, here we can also see that all other users have read and execute permissions, shown as: **r-x**. This means that when the file lab04.sh is run by a normal user (me), it will be executed with the privileges of the root user, instead of the privileges of the user launching the passwd command.

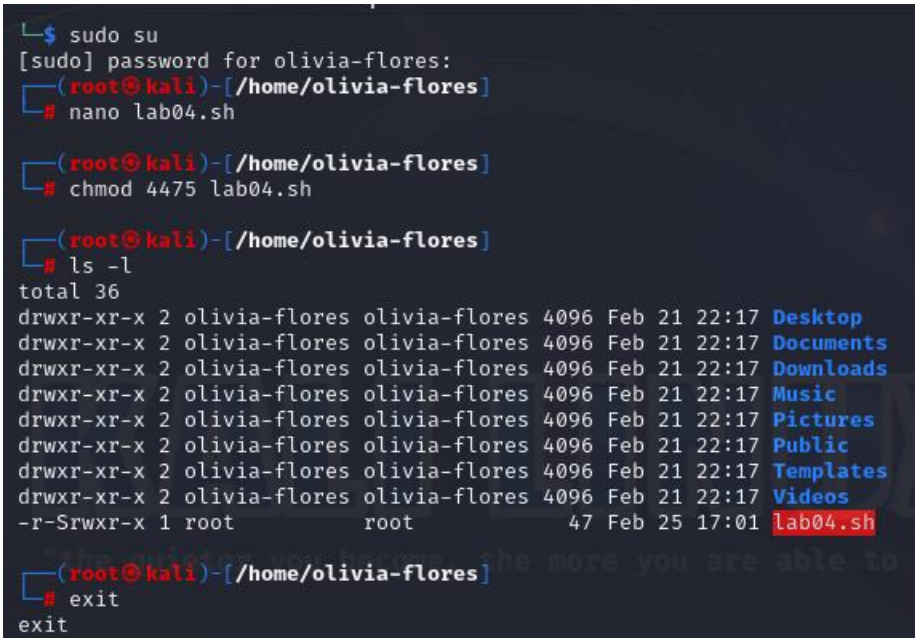


Figure 16: Viewing permissions on a SETUID file.

Next, I entered the command **nano lab04.sh** into the command prompt as other user (olivia-flores) and was able to prove that I could read and execute the file, but not write as displayed below: **[File 'lab04.sh' is unwritable]**

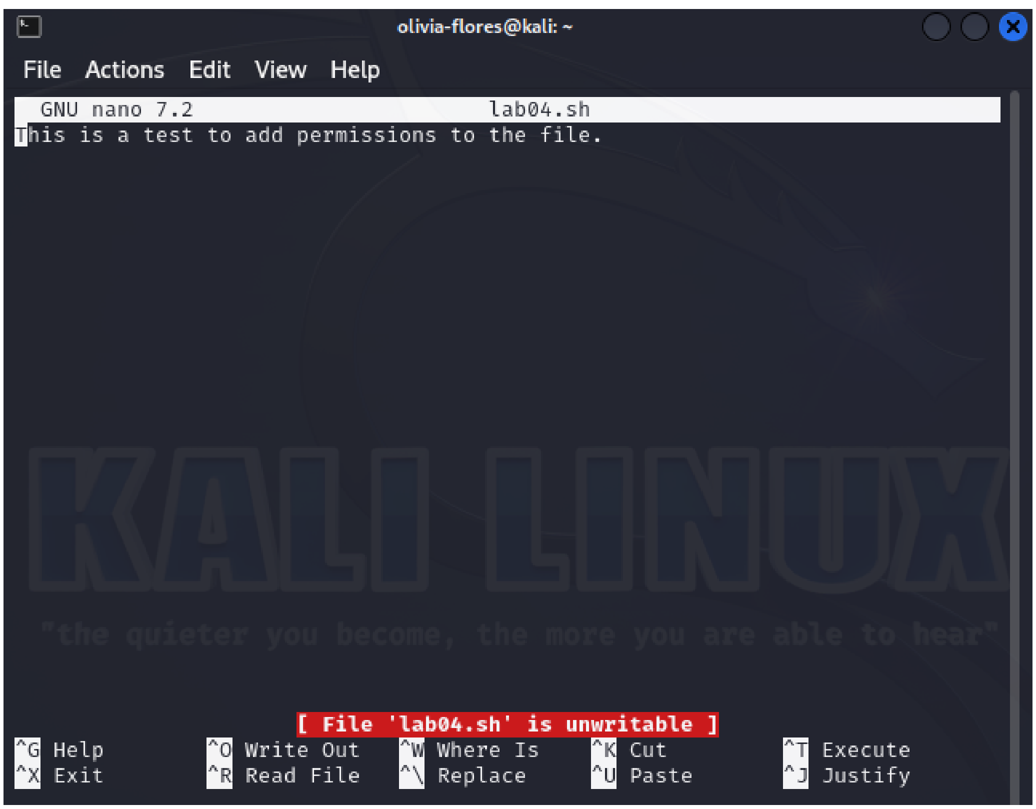


Figure 17: Displaying lab04.sh file with r-x permissions.

While setuid is a powerful feature, it can introduce security vulnerabilities if not used carefully. This meaning it can compromise any security risks that follow such as an attacker exploiting these vulnerabilities to execute arbitrary code with elevated privileges, buffer overruns or through path injection. To prevent such possibilities, some operating systems ignore the setuid bit for executable shell scripts [7]. Some best practices may include only using setuid when necessary, change privileges to only the required privileges absolutely necessary, logging and monitoring to detect unusual activities, and lastly secure programming practices as we covered in Lab 3 of this course with C code.

Breakpoint 4

In this step of the lab, I am going to be setting up groups. First, I want to make sure I have access to my server through the users and passwords we created earlier on in this lab, then I will continue with creating the group for those users in the server. The group, which I named **yellowstone**, will allow these users to share files and protect others from accessing the files.
I will continue by testing my access to the server through each of my server username/passwords, including my own username and then the two new users I created. I used the standard SSH method to complete the following steps.

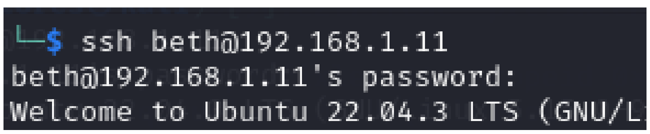


Figure 18: Successfully SSH'ed into server as beth.

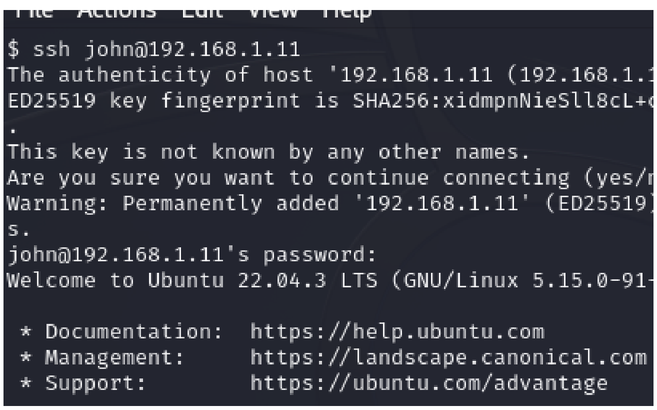


Figure 19: Successfully SSH'ed into server as john.

Figure 20: Successfully SSH'ed into server with my own username.

It is important to remember to continue using the **whoami** command at each turn using the terminal for security purposes. We want to keep track of what user is doing at each step of the way.

Next, I created a test directory, called "**fun-stuff**" under one of my user's (John) /home directory with the **mkdir** command. To verify the directory was created, I used the ls -l command once again to help verify its existence, permissions, and owners.

Now I want to jump right into creating a group with the **groupadd** command, I called my group "yellowstone". To check the existence of the group, I can use the cat command as: **cat /etc/group | grep** *yellowstone*. In this instance, I simply followed along with the lab instructions using the cat command and pipelining (|) into the grep command, but in this case the **find** command is another useful way to use for grep when wanting to find directories in the specified path. The cat command essentially concatenates multiple files and displays them in the output, which is fine here.

Next, I will add the two users I originally created, to the group *yellowstone.* I will do this by using the commands: **usermod-aG yellowstone john** and **usermod-aG yellowstone beth** then check the group's membership again using the same command from above (cat pipelining into grep). I can now see both users displayed next to the group name as shown below:

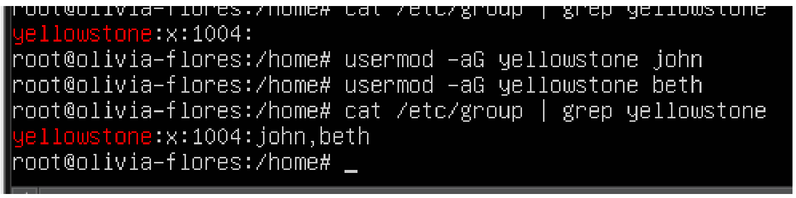


Figure 21: New users added to the group.

Following the group creation and adding the users to the group. I want to now create a new directory in /home with the same name as my group also named ***yellowstone***. I will Change the group ownership of the folder to the same name as the group using the chown command as:
chown :yellowstone yellowstone

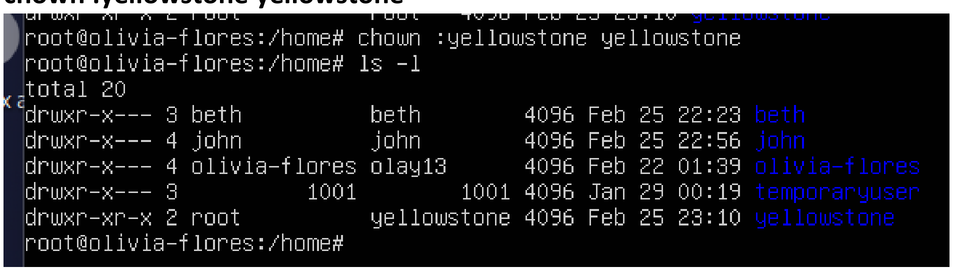


Figure 22: Changing group ownership to yellowstone using the chown command.

In addition to changing the group ownership of the folder to *yellowstone*, I made sure the permissions of folder allow group members to create and delete files by using the command: **chmod g+rwx yellowstone**, which grants the group read, write, and execute permissions to the group. I can verify that the group has these permissions by looking at the 2nd to 4th characters after d. You can also see from the figure below, that the changes to ensure that 'others' have access to the files of the group have been made using the additional command **chmod o+rx** leaving the w out because we only want them to be able to read and execute the files in this directory and I can verify this has been done by looking at the last 3 characters.

Figure 23: Displaying the permissions of the group and others.

Next, I will sign in as the file creator's account who is john in my case and copy the folder they created (/home/john/fun-stuff) in their folder to this group folder (/home/yellowstone). I have not used the scp command previously and seemed like a longer way of just copying a folder over to the group folder. I was curious so I asked ChatGPT, **“why is the scp command used in place of cp?”** and they returned a very good explanation of scp down below:

The **scp** (secure copy) command is used in place of **cp** (copy) when you need to copy files or directories between different hosts over a secure network using the SSH (Secure Shell) protocol. The primary reasons for using **scp** over **cp** include:

1. **Secure File Transfer:**
    - **Encryption: scp** encrypts the data during transmission, providing a secure way to transfer files over a network. This is particularly important when copying files between remote hosts over the internet.
    - **SSH Protocol: scp** utilizes the SSH protocol for authentication and encryption, ensuring a secure and authenticated connection between the source and destination systems.
2. **Remote Copy:**
    - **scp** is designed for copying files between different machines, whereas **cp** is typically used for local file copies on the same machine. **cp** does not support remote file copying.
3. **Usage Across Different Hosts:**
    - **scp** allows you to copy files between local and remote systems or between two remote systems. It's a versatile tool for copying files across different hosts connected via a network.
4. **Syntax Similarity to cp:**
    - The syntax of **scp** is similar to that of **cp**, making it easy for users familiar with the **cp** command to adapt to **scp**. For example, the **-r** option is used for recursive copying in both commands.

While **cp** is excellent for local file copies, **scp** becomes necessary when dealing with remote systems, especially when security and encryption are crucial considerations [8]. They also gave me examples of what commands to use when copying a file either from a local machine to a remote machine or from a remote machine to a local machine. This explained so much and as a security analyst makes a huge difference in day to day operations to complete file transfers as the one shown below:

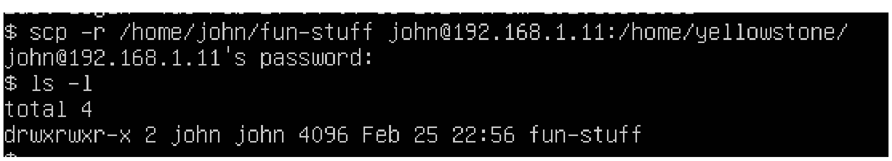


Figure 24: Using 'scp' to copy john's folder to the new group folder.

Now I will exit the root session and switch to my username. Since we completed the copying over of the folder to the group subfolder, we will navigate there and create a script in the folder with my own username olivia-flores. In this ./secret.sh script, I placed a short “echo” statement in the file that states “This file contains secret information!” You can see that I am able to run the file.

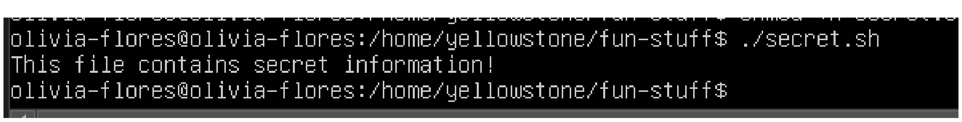


Figure 25: Executing an accessible file.

As instructed, I will also go ahead and change the group ownership of the folder to group members only and verify with the command **cd yellowstone** as shown below:

```
olivia-flores@olivia-flores:/$ cd /home
olivia-flores@olivia-flores:/home$ cd yellowstone
-bash: cd: yellowstone: Permission denied
olivia-flores@olivia-flores:/home$
```

Figure 26: Testing new group permissions.

To explain further on the outputs above and why I want to test these permissions is to verify that they are set in place once a command such as chmod o-rx has been ran. As a security analyst, I find it very important to also use the ls -l command frequently to view the permissions of all three users in this step at once, this gives us exposure to recognizing which ones should or should not have access, or verify those who are part of a group. I have displayed the ending permissions for all three users down below which you can see the permissions to this directory (d), for all other users, is denied and as displayed by the last 3 characters (---) of the 9 file permission characters.

```
olivia-flores@olivia-flores:/home$ ls -l
total 20
drwxr-x--- 3 beth          beth        4096 Feb 25 22:23 beth
drwxr-x--- 5 john          john        4096 Feb 25 23:27 john
drwxr-x--- 5 olivia-flores olay13      4096 Feb 26 00:06 olivia-flores
drwxr-x--- 3          1001        1001 4096 Jan 29 00:19 temporaryuser
drwxr-x--- 3 john          yellowstone 4096 Feb 25 23:55 yellowstone
```

Figure 27: Ending permissions for all three users in step 4.

The examples used in this step of the lab demonstrate the least privilege principle by practicing as a security analyst who these three users' access rights to certain files and directories will be and when dealing with a large organization, this is such a crucial cybersecurity practice. The principle of least privilege (POLP) is a concept in computer security that limits users' access rights to only what is strictly required to do their jobs. POLP can also restrict access rights for applications, systems and processes to only those who are authorized [9].

In this step of the lab, we were also able to get introduced to a secure system design which is super important in cybersecurity as systems are always being exposed as targets. That is why there are secure system designs that help us avoid losing sensitive information or be exposed to an attack where they might gain access to a user's password, files and directories. As mentioned in the NCSC article for Secure design principles, some principles in which an attack can be mitigated include:

1. Establishing the context: Determine *all* the elements which compose your system, so your defensive measures will have no blind spots.
2. Making compromise difficult: an attacker can only target the parts of a system they can reach. Make your system as difficult to penetrate as possible
3. Making disruption difficult: Design a system that is resilient to denial of service attacks and usage spikes
4. Making compromise detection easier: Design your system so you can spot suspicious activity as it happens and take necessary action
5. Reducing the impact of compromise: If an attacker succeeds in gaining a foothold, they will then move to exploit your system. Make this as difficult as possible [10]

Breakpoint 5

In the final step of this lab, I am going to take you along with me to explore more specialized permissions using extended Access Control Lists. With regular permissions, each file and directory can have only one user and group owner at a time. The **extended Access Control Lists** (ACLs) in Linux can resolve this issue. ACLs allow us to apply a more specific set of permissions to a file or directory without (necessarily) changing the base ownership and permissions. They let us "tack on" access for other users or groups [1].

First, I will start off by installing the **acl** package in my Ubuntu server using the command: **sudo apt install acl**

Then I will check the existing ACLs on your group folder with the getfacl command: **getfacl yellowstone**

This displays information about the Yellowstone file, its owner, group that it's a part of, and the user, group and other permissions to this file. In my case, I have john as the owner, and as I demonstrated earlier, we added this file to the group also known as yellowstone, and its final permission look like the following: **rwxr-x---**

Next, I will create a setfacl command that prohibits access to my group folder, yellowstone, to others for all future created files and directories, this can be done using the -d option as shown below as:

setfacl -d –set u::rwx,g::twx,o::---

```
root@olivia-flores:/home# setfacl -d --set u::rwx,g::rwx,o::--- yellowstone
root@olivia-flores:/home# getfacl yellowstone
# file: yellowstone
# owner: john
# group: yellowstone
user::rwx
group::r-x
other::---
default:user::rwx
default:group::rwx
default:other::---
```

Figure 28: Proibiting future access to group folder: yellowstone.

You will now see a **+ sign** by the group folder permissions to indicate that the folder has an ACL setting for future files/directories. However, for existing folders, I will still need to use the **chmod** command to set these permissions. For example, since yellowstone was already an existing folder, I used the chmod command to give the 770 permissions to the user, group and other user, with 7 being rwx permissions, and 0 being none.

```
root@olivia-flores:/home# chmod 770 yellowstone
root@olivia-flores:/home# ls -l
total 20
drwxr-x---  3 beth          beth        4096 Feb 25 22:23 beth
drwxr-x---  5 john          john        4096 Feb 25 23:27 john
drwxr-x---  5 olivia-flores olay13      4096 Feb 26 00:06 olivia-flores
drwxr-x---  3          1001        1001 4096 Jan 29 00:19 temporaryuser
drwxrwx---+ 3 john          yellowstone 4096 Feb 25 23:55 yellowstone
```

Figure 29: Demonstration on how setfacl made changes with +.

I also went ahead and gave my username: olivia-flores access to the parent group folder but not to the subfolder (fun-stuff) that has privileged and secret information. I completed this by inputting the following command as root:

setfacl -m u:olivia-flores:--- /home/yellowstone/fun-stuff

The **'-m'** flag represents the ACL being modified and im specifying permissions for the user (u) as olivia-flores, to then have (---) no read, write or execute permissions to the specified directory shown as /home/yellowstone/fun-stuff.

```
setfacl: Option -m: Invalid argument near character 20
root@olivia-flores:/# setfacl -m u:olivia-flores:--- /home/yellowstone/fun-stuff
root@olivia-flores:/#
```

Figure 7: setfacl command, modifying permissions to olivia-flores user.

So to explain further on the command above and the one below on figures 30 & 31, I want to use my Kali user, for example, Jamie will be able to sign themselves in as me (olivia-flores) and still be able to access the parent folder, but they will not be able to access the subfolder when they enter the command: **cd yellowstone** because they do not have permission to access the subfolder.

```
-bash: cd: /yellowstone: No such file or directory
olivia-flores@olivia-flores:/home$ cd yellowstone
olivia-flores@olivia-flores:/home/yellowstone$ cd fun-stuff
-bash: cd: fun-stuff: Permission denied
olivia-flores@olivia-flores:/home/yellowstone$
```

Figure 31: Proof that olivia-flores does not have access to /home/yellowstone/fun-stuff

However, I can still give them read and execute access to another folder that I have created, called **superfun-stuff**. I will also create a new subfolder under my group folder with another short script in it called **extrasecret.sh** shown below. This script will be made so that specifically me as the user can read (r) and execute the file, as well as the other users. Follow along in the snippets below as it is easier to visualize who is getting what permissions.

Here I added the read permissions to the other users, using the command **chmod +rx extrasecret.sh**, and you will see the user (olivia-flores) has r-x and down below that the other user also has the same r-x permissions.

```
olivia-flores@olivia-flores:/home$ getfacl yellowstone/superfun-stuff/extrase
cret.sh
# file: yellowstone/superfun-stuff/extrasecret.sh
# owner: root
# group: root
user::rwx
user:olivia-flores:r-x
group::rwx
mask::rwx
other::r-x
```

Figure 32: Granting other users permission to read and execute the extrasecret.sh file.

If either, olivia-flores or any other user tries to cd to the superfun-stuff directory and executing the extrasecret.sh file using nano, the permissions set will allow this because both of those have read and execute permissions. The file that is executed will also display the message in figure 33 below.

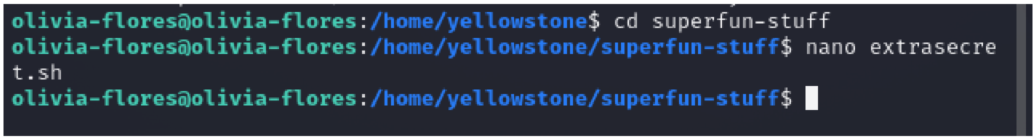


Figure 33: Proof that permissions granted work on other users.

Figure 34: Proof that the file is executable by other users, but not writable.

As I introduced the extrasecret.sh file to you, I want to help us understand how this might be a very useful file in a real-world context to gain information about an intruder scoping out the file. To help me visualize this, I asked ChatGPT the following question in hopes of being able to elaborate on it a bit more.

"So i created a parent folder, and in the parent folder, I added a subfolder with .sh file that can be executed if given the correct permissions. My question is how might I modify the syslog configuration file to log access to this .sh file?"

The Generative AI tool response:

The line **if $programname == 'bash' and $msg contains '/path/to/parent/folder/subfolder/file.sh'** is a condition that filters log messages based on certain criteria. Let's break down what each part of the condition does:

- **$programname == 'bash':** This part of the condition checks if the log message is generated by the "bash" program. It ensures that the log entries are related to the Bash shell.
- **and:** This is a logical operator that combines the two conditions. Both conditions must be true for the overall condition to be true.
- **$msg contains '/path/to/parent/folder/subfolder/file.sh':** This part checks if the log message contains the specified string '/path/to/parent/folder/subfolder/file.sh'. This is likely used to identify log messages related to access or events related to the specific file.

Putting it all together, the entire condition is saying: "If the log message is generated by the 'bash' program and the message contains '/path/to/parent/folder/subfolder/file.sh', then take the specified action." You can see in the image below directly from the syslog configuration file.

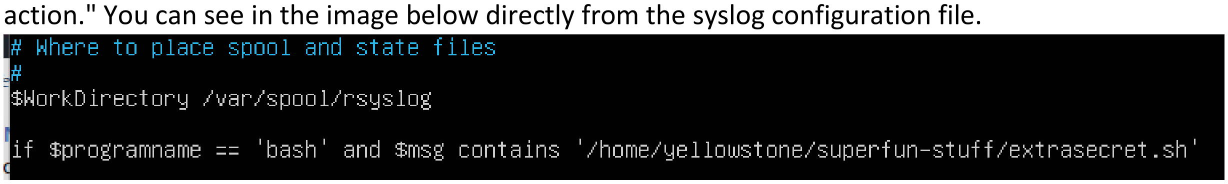


Figure 35: Conditions used within the syslog configuration file.

In the context of syslog configuration, this condition is often used in combination with actions like logging to a specific file or forwarding the log message to another system. In my example, it's directing log messages that match the specified conditions to be written to **/var/log/access.log** (not pictured here).

This kind of conditional filtering allows us to customize how logs are handled based on the content of the log message and the program that generated it. It can be a useful mechanism for organizing and directing logs to different destinations based on specific criteria [11].

Figure 36: Testing the /var/log/access.log

CONCLUSION

In conclusion to this lab, I was able to grasp the Linux-style access controls on files on an Ubuntu server. Some of the main objectives I really had an interest in were the impacts of setUID, getting familiar with specified permissions for groups, and customizing extended file permissions. I was able to accomplish my goal in getting familiar with an access control system and how in depth it got. In addition to that, I also enjoyed reading up more on the PAM configuration related to Linux and lastly the syslog configuration file, as they are both great resources as a system administrator.

REFERENCES

[1] Mitra, Department of Information Systems and Cyber Security. [Online] “Lab 04- Working with File Permissions” UTSA, 2024 [Accessed: February 10, 2024.]

[2] “Linux File Permissions: Understanding Setuid, Setgid, and the Sticky Bit.” *CBT Nuggets*, https://www.cbtnuggets.com/blog/technology/system-admin/linux-file-permissions-understanding-setuid-setgid-and-the-sticky-bit. [Accessed: February 25, 2024.]

[3] “Linux Hard Links versus Soft Links Explained.” *CBT Nuggets*, https://www.cbtnuggets.com/blog/certifications/open-source/linux-hard-links-versus-soft-links-explained. [Accessed: February 25, 2024.]

[4] dalimama. “Can You Describe a Real-Life Scenario of Exploiting Sticky Bits?” *Information Security Stack Exchange*, 21 Nov. 2011, https://security.stackexchange.com/q/9115. [Accessed: February 25, 2024.]

[5] Kili, Aaron. *How to Configure and Use PAM in Linux*. 11 Dec. 2018, https://www.tecmint.com/configure-pam-in-centos-ubuntu-linux/. [Accessed: February 25, 2024.]

[6] “Hi am trying to change the password requirements in my pam.d file, can you help me do that?”*ChatGPT, GPT-3.5 January 2022 version, OpenAI*, https://chat.openai.com/chat. [Accessed: February 25, 2024.]

[7] “SetUID, SetGID, and Sticky Bits in Linux File Permissions.” *GeeksforGeeks*, 5 Aug. 2019, https://www.geeksforgeeks.org/setuid-setgid-and-sticky-bits-in-linux-file-permissions/.[Accessed: February 25, 2024.]

[8] “Why is the scp command used in place of cp?” *ChatGPT, GPT-3.5 January 2022 version, OpenAI*, https://chat.openai.com/chat. [Accessed: February 26, 2024.]

[9] “What Is the Principle of Least Privilege? | Definition from TechTarget.” *Security*, https://www.techtarget.com/searchsecurity/definition/principle-of-least-privilege-POLP. [Accessed: February 26, 2024.]

[10] *Secure Design Principles*. https://www.ncsc.gov.uk/collection/cyber-security-design-principles. [Accessed: February 25, 2024.]

[11] **“So i created a parent folder, and in the parent folder, I added a subfolder with .sh file that can be executed if given the correct permissions. My question is how might I modify the syslog configuration file to log access to this .sh file?”** *ChatGPT, GPT-3.5 January 2022 version, OpenAI*, https://chat.openai.com/chat. [Accessed: February 25, 2024.]

COLLABORATION

In this lab, I made use of the lab instructions provided to us, generative AI tool interactions as I was trying to understand basic commands like scp and get familiar with configuration files as they can be essential to a system administrator, but also pose vulnerabilities to a system. I had a few interactions through our slack workspace, in module 4, as I was having questions about SSH’ing from my client to the server machine, as well as basic setting questions about my Kali linux machine, all which were resolved.