Assignment 1

Areas	Carleton University
Projects	COMP4108B - Computer Systems Security
Date	@January 28, 2026
Priority	Medium Priority
Status	Done
Complete	☑
Created time	@January 6, 2026 12:06 AM
Weight	6%

Assignment 1

> COMP4108: Computer Systems Security
>
> Andrew Wallace - 101210291
>
> andrewwallace3@cmail.carleton.ca
>
> Due: January 28th, 2026

Part A - File System Permissions (35 Marks)

Question 1: Bash Script

Please see `a-1-profile.sh` for implementation details.

Question 2: File/Directory Actions

a. **File Actions**

After running the command for `File1` we see the following output.

```
student@COMP4108-a1:~$ stat /A1/Perms/File1
File: /A1/Perms/File1
Size: 0 Blocks: 0 IO Block: 4096 regular empty file
Device: fc01h/64513d Inode: 393854 Links: 1
Access: (0644/-rw-r--r--) Uid: ( 1000/comp4108) Gid: ( 1000/comp4108)
Access: 2025-06-20 15:00:37.100000000 -0400
Modify: 2025-06-20 15:00:37.100000000 -0400
Change: 2025-06-20 15:01:32.608000000 -0400
Birth: -
```

We see the permissions are `-rw-r--r--`

The privileges for user (u) is `rw-` meaning that the user `comp4108` has read and write permissions but no execute permissions.

The group (g) privileges are `r--` meaning that members of the group `comp4108` only have read access and no write or execute access.

The "others" (o) access is `r--` meaning that others only have read access and no write or execute access.

b. **Directory Actions**

After running the command for `Directory1` we see the following output

```
student@COMP4108-a1:~$ stat /A1/Perms/Directory1
File: /A1/Perms/Directory1
Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: fc01h/64513d
Inode: 393875 Links: 2
Access: (0755/drwxr-xr-x) Uid: ( 5/ games) Gid: ( 1000/comp4108)
Access: 2025-06-20 15:03:15.560000000 -0400 M
odify: 2025-06-20 15:03:15.560000000 -0400
Change: 2025-06-20 15:03:18.712000000 -0400
Birth: -
```

We see the permissions are `drwxr-xr-x`

The first difference we see is that the first component is "d" to signify that this is a directory.

The user (u) privileges are `rwx`, meaning the `games` user can read (list) the files in the directory, write (add or remove) files to the directory, and execute which allows the user to access (step into) the directory.

The group (g) privileges are `r-x`, meaning that members of the group `comp4108` can read (list) the contents of the directory, but cannot write (add or remove) files to the directory, and execute which allows them to access (step into) the directory.

The other (o) privileges are `r-x`, meaning that all others can read (list) the contents of the directory, but cannot write (add or remove) files to the directory, and execute which allows them to access (step into) the directory.

Question 3: File Permissions

a. **Permission Model I**

After running the command `echo "Hello World!" >> /A1/Perms/File2` the following error is shown:

```
-bash: /A1/Perms/File2: Permission denied
```

Inspecting the file permissions, we see:

```
-rw-rw---- 1 comp4108 staff 0 Jun 20  2025 /A1/Perms/File2
```

Since we are not the user `comp4108` and are not a part of the group `staff` we receive the `other` privileges (`---`). This means we have no read, write, or execute access. So, when trying to write to this file, we are denied. To confirm this, we can see that our current user information is:

```
uid=1001(student) gid=1001(student) groups=1001(student),27(sudo),1011(school)
```

This shows that we are not the owner or a part of the group of the file, which confirms our lack of access.

b. **Permission Model II**

The following command was used to write to `File3`:

```
student@COMP4108-a1:~/a1$ echo "Hello World!" >> /A1/Perms/File3
```

`student@COMP4108-a1:~/a1$ cat /A1/Perms/File3Hello World!`

Checking the permissions of `File3` we see:

`-rwxrwxr-- 1 skyle school 13 Jan 17 16:04 /A1/Perms/File3`

We see that the owner is a different user `skyle`, so user (u) privileges do not apply to the `student` user. However, the `student` user belong to the `school` group. This means that the student user has `rwx` access (as specified by the group privileges). This allows us to write to the file.

c. **Secret File**

Since user `bwayne` wishes to grant access to me (`student`) and not necessarily everyone else, running `chmod 666 /A1/Perms/SecretFile` is not advised. This will give read and write access to everyone. From a security perspective, this breaks confidentiality and makes the resource available to adversaries. It is recommended to use fine-grained access control in this scenario. Another option would be to add my user (student) to the `bwayne` group, but this would be less secure if there are other files that the user `bwayne` does not want you to have access to. So, the user `bwayne` should use fine grained access control via:

`setfacl -m u:student:rw /A1/Perms/SecretFile`

This will give my user (`student`) the read and write privileges without granting me privileges to other files.

Question 4: Directory Permissions

a. **Secret Stash**

Prior to running the command, we see the following permissions for the sub-directories:

```
drwx--x--x 2 student student 4096 Jan 17 16:23 A
drwxr--r-- 2 student student 4096 Jan 17 16:23 B
drwxr-xr-x 2 student student 4096 Jan 17 16:23 C
drwxrwxr-x 2 student student 4096 Jan 17 16:23 D
drwx--x--x 2 student student 4096 Jan 17 16:23 E
drwx------ 2 student student 4096 Jan 17 16:23 F
drwxr-xr-x 2 student student 4096 Jan 17 16:23 G
```

```
drwx--x--- 2 student student 4096 Jan 17 16:23 H
drwxr-xr-- 2 student student 4096 Jan 17 16:23 I
drwxr--r-- 2 student student 4096 Jan 17 16:23 J
drwx------ 2 student student 4096 Jan 17 16:23 K
drwxrwxr-x 2 student student 4096 Jan 17 16:23 L
drwxr-xr-- 2 student student 4096 Jan 17 16:23 M
drwx------ 2 student student 4096 Jan 17 16:23 N
drwxr-xr-x 2 student student 4096 Jan 17 16:23 O
drwxr--r-- 2 student student 4096 Jan 17 16:23 P
drwx--x--x 2 student student 4096 Jan 17 16:23 Q
drwx------ 2 student student 4096 Jan 17 16:23 R
drwxr-xr-x 2 student student 4096 Jan 17 16:23 S
drwxr-xr-- 2 student student 4096 Jan 17 16:23 T
drwxr-xr-x 2 student student 4096 Jan 17 16:23 U
drwx------ 2 student student 4096 Jan 17 16:23 V
drwxr-xr-- 2 student student 4096 Jan 17 16:23 W
drwxr--r-- 2 student student 4096 Jan 17 16:23 X
drwxr-xr-x 2 student student 4096 Jan 17 16:23 Y
drwx--x--x 2 student student 4096 Jan 17 16:23 Z
```

After running `chmod -R 775 ./SecretStash` we see:

```
drwxrwxr-x 2 student student 4096 Jan 17 16:23 A
drwxrwxr-x 2 student student 4096 Jan 17 16:23 B
drwxrwxr-x 2 student student 4096 Jan 17 16:23 C
drwxrwxr-x 2 student student 4096 Jan 17 16:23 D
drwxrwxr-x 2 student student 4096 Jan 17 16:23 E
drwxrwxr-x 2 student student 4096 Jan 17 16:23 F
drwxrwxr-x 2 student student 4096 Jan 17 16:23 G
drwxrwxr-x 2 student student 4096 Jan 17 16:23 H
drwxrwxr-x 2 student student 4096 Jan 17 16:23 I
drwxrwxr-x 2 student student 4096 Jan 17 16:23 J
drwxrwxr-x 2 student student 4096 Jan 17 16:23 K
drwxrwxr-x 2 student student 4096 Jan 17 16:23 L
drwxrwxr-x 2 student student 4096 Jan 17 16:23 M
drwxrwxr-x 2 student student 4096 Jan 17 16:23 N
```

```
drwxrwxr-x 2 student student 4096 Jan 17 16:23 O
drwxrwxr-x 2 student student 4096 Jan 17 16:23 P
drwxrwxr-x 2 student student 4096 Jan 17 16:23 Q
drwxrwxr-x 2 student student 4096 Jan 17 16:23 R
drwxrwxr-x 2 student student 4096 Jan 17 16:23 S
drwxrwxr-x 2 student student 4096 Jan 17 16:23 T
drwxrwxr-x 2 student student 4096 Jan 17 16:23 U
drwxrwxr-x 2 student student 4096 Jan 17 16:23 V
drwxrwxr-x 2 student student 4096 Jan 17 16:23 W
drwxrwxr-x 2 student student 4096 Jan 17 16:23 X
drwxrwxr-x 2 student student 4096 Jan 17 16:23 Y
drwxrwxr-x 2 student student 4096 Jan 17 16:23 Z
```

There are two negative side-effects.

Firstly, members in the groups of the sub-directories now have write access to them. This may not be desired. We cannot assume that since the parent directory has write access for it's group that all sub-directories can have the same permissions. For example, there may be a sub-directory that should be locked to members of the group so that they cannot write to this directory. By running this command, we now give them access which is a negative side-effect.

Secondly, you may want to create a group as a blacklist to prevent members of the group from having access. If you run the `chmod -R 775 /A1/Perms/SecretStash`, the blacklisted members now have access.

A better command to run would be to be more specific and only update the "other" (o) users privileges via:

`student@COMP4108-a1:~/a1$ chmod -R o+rx /A1/Perms/SecretStash`

This now only changes the "others" privileges to have read and execute, rather than all of user, group, and other privileges.

b. **Secret Sum**

The owner's command `chmod o-r /A1/Perms/SecretSums` is insufficient to prevent other users from reading the contents of the files for two main reasons.

First, the directory still has execute (x) privileges granted to other users, meaning that they can still access the directory, but can't list (read) the contents of the directory.

Second, the files themselves have read and write privileges granted to other users, meaning that they have the privilege of seeing the contents of the file.

So, if a user knows the name of the file, they can simply run a command to see the contents of the file. For example:

```
cat /A1/Perms/SecretSums/0.txt
```

A more reliable approach would be to also remove read access to the individual files within the directory and/or revoke the execute privileges from "other" users on the `SecretSums` directory.

Question 5: Access Control Lists (ACLs)

a. **Contract Workers**

In this situation, altering the files' ACL entries would be more appropriate rather than modifying group permissions for 3 main reasons.

First, the contract workers only require access to a few shared files and not all the files that belong to the group. If you were to add them to the group, they would have more access than necessary which violates the principle of least privilege.

Second, the contract workers may require different privileges than the group. Altering the ACL entries allow you to maintain the current group privileges while giving the necessary ones to the contract workers.

Thirdly, a file can only belong to one group. If you wanted to achieve this only using groups, another group (if wanting to maintain isolation) would have to be created for these contract workers, and copies of the necessary files would have to be created. This breaks the shared nature of the files and causes more overhead in terms of storage. ACLs keep the shared nature of the files and don't require creation of a new group.

b. **Security Principles**

The two security principles discussed in Chapter 1.7 of the course textbook that ACLs help enforce are:

1. P5: Isolated-Compartments

   ACLs help enforce isolated components by making file permissions more fine grained into individual components rather than large groups. ACLs support this principle by providing strong isolation structures that prevent information leakage. It allows you to manage the permissions of individual users rather than groups of users which increases isolation and security.

2. P7: Modular-Design

   ACLs help enforce modular design because they eliminate the need to embed privileges in large single components (such as the "groups" and "other" access controls). ACLs directly allow for fine grained control of file permissions at the user level, giving you more control and security over your architecture. One interesting point is that the separation of privileges can reduce the likelihood of an insider attack if the attack requires collusion between two pieces of information.

Question 6: setuid

After running the command:

```
student@COMP4108-a1:~/a1$ find /usr/bin -perm -4000
```

We see the following list of binaries:

- /usr/bin/at
- /usr/bin/sudo
- /usr/bin/newgrp
- /usr/bin/chsh
- /usr/bin/pkexec
- /usr/bin/passwd
- /usr/bin/gpasswd
- /usr/bin/netkit-rsh
- /usr/bin/netkit-rlogin

- /usr/bin/netkit-rcp
- /usr/bin/chfn

For these binaries, any user (with necessary privileges) can execute them on behalf of the owner of the file (root). That is, at runtime, the program (file) has the file owners privileges. This does not mean that the user who executed the program has the owner privilege though, it is only the file (running program) itself that has them.

For example, the binary `/usr/bin/chfn` allows a user to change their personal information. For the user to update their personal information, they'd have to open the `/etc/passwd` file and update their specific information. However, other user's information is stored here, so they could easily change another person's information if they had access (of course they do not have access to write to this file). This binary has it's `setuid` set for this very reason. It needs elevated privileges to write to this file. So, when the user runs the binary, the binary is run on behalf of the owning user (root) which has the necessary write privileges and thus the binary can write to the `/etc/passwd` file and update the users personal information.

Part B - Race Conditions (30 Marks)

Question 1: `vuln_slow`

The process to exploit the slow vulnerability is as follows:

1. Run the `vuln_slow` program
2. During it's vulnerable phase run the following commands in a separate terminal:
    a. Rename the debug file `mv /home/student/.debug_log /home/student/temp`
    b. Create the symbolic link for the target file with the name of the debug file via:
    `ln -s /A1/Racing/Slow/root_file /home/student/.debug_log`
    c. Verify your text is written to the root_file via: `cat /A1/Racing/Slow/root_file`

Question 2: `vuln_fast`

High-level Outline of Exploit

a. The file I am trying to write to is the `/root/.rhosts` file. This file contains the host and user name entries of the users that can login as root without a password on this host. We attempt to write to this file so that the `student` user can gain root access to `localhost`.

b. The contents I am trying to write is the entry `localhost student`. This is because `localhost` is the host name in question and we are the `student` user. Having this entry in the `/root/.rhosts` file will allow the `student` user to login as `root` using the `rsh` command.

c. You need an application like `vuln_fast` to do this because the `/root/.rhosts` is only writable by the `root` user. Since `vuln_fast` has it's `setuid` bit set and is owned by the `root` user, any file accessed in this application will be accessed on behalf of the root user. This gives us the opportunity to exploit the application to have it write to a "`root` only" file for us.

d. The timeline for a successful TOCTOU exploit is as follows:

- The `vuln_fast` is running with the command `nice -n 19 ./vuln_fast "localhost student"`
    - The `nice` command is used here with a high "niceness" to so that let's other processes (such as the exploit script) to use the CPU first, giving it a higher probability of achieving the exploit
    - The argument "localhost student" is used because we want that value to be written to the `.rhosts` file to give the `student` user password-less login as the `root` user
- At the time `vuln_fast` checks whether the user (`student`) has access to the `.debug_log` file, the user must have write privileges. That is, the symbolic link must be created after this check.
- After the check passes (i.e., the user has write privileges to the `.debug_log` file), a symbolic link is created with the same name `.debug_log` pointing to the `/root/.rhosts` file before the application writes to it.
    - The command to create the symbolic link is below:

`ln -sf /root/.rhosts /home/students/.debug_log`

The -f argument is used here to delete the `.debug_log` file if it already exists so that the symbolic link will always be created

- At the time the application goes to write to the debug file, it accesses the symbolic link and writes to the `/root/.rhosts` file instead using the provided string “localhost student”.
- This adds the necessary entry to the `.rhosts` file to complete the exploit.

Question 3: Security Principles

The three security principles that were broken and exploited were P1 (Simplicity and Necessity), P4 (Complete Mediation), and P6 (Least Privilege).

P1: Simplicity and Necessity

This principle states that we should minimize functionality and disable unused functionality. The program `vuln_fast` fails to minimize it’s functionality by having its `setuid` bit set. This allows the program to be run on behalf of the owning user (`root`), which gives unnecessary functionality (`root` privileges) during it’s runtime.

P4: Complete Mediation

This principle states that for access to every object, we should verify proper authority. The `vuln_fast` program fails to do this by writing directly to the file without verifying it is the correct file. That is, the program does not check if the calling user has proper authority to the file it is writing to. One solution would be to get the file descriptor for the file in question first, then check if the user has correct write permissions to this file. This prevents the attacker from swapping out the file before writing to it.

P6: Least Privilege

This principle states that you should allocate the fewest privileges needed for the task, and for the shortest duration. The program `vuln_fast` fails to do this by having it’s effective UID set to the `root` for an unnecessary duration of time. That is, there is a vulnerable window where the attacker can exploit these elevated privileges to swap out the original file with a symbolic link to write to a completely different file. One solution would be to set the effective UID to the calling user for most of the

program runtime and only elevate privileges when writing to the file. This will reduce the duration of elevated privileges and prevent the attacker from swapping the file, as the write operation will now be atomic.