Assignment - 2

Name: Md. Minhazul Mowla

ID: 23201390

Section: 23

Course: CSE321

Submission Date: 7/1/25

Limitations of Standard UNIX File Permissions and The Necessity of Extended Access Control Lists (ACLs)

i. Standard Permission Bottlenecks

The classic UNIX permission model uses three sets of rwx (read, write, execute) bits for:

1. Owner(u) - The file's creator
2. Group(g) - A single group associated with the file.
3. Others(o) - Everyone else on the system

Limitations in Multi-User Environments:

1. Only one group for each file. If a file needs to be accessed by multiple departments like dev, qa, admin ~~etc~~ etc., one must create a new group that includes all required users. It becomes messy.

2. No per-user exceptions. One cannot grant access to one specific user without

either making them the owner ~~of~~ or adding them to the file's group. It may give them unnecessary access to other files.

Example scenario:

Let's say a file named plans.txt. It belongs to a owner named 'Talha'. It also ~~belo~~ belongs to a group named 'devs'.

Now, if one wants a person(A) from another group called 'finance' to read the file. Also, one wants another person(B) from another group named 'qa', to both read and write the file. Group named 'others' has no access.

Based on standard permissions, it's impossible without creating a new group containing just 'A' and 'B'. This is impractical if they need only this one file.

ii. Mechanism of Extended ACLs

POSIX ACL Entry structure:

ACLs extend the inode metadata by adding entries that can specify permissions for multiple users & and groups individually. Each entry follows the format:

tag-type: [tag-qualifier]: permissions

Here,

tag-type → user, group, other, mask....

tag-qualifier - Username or group ID

permissions - r, w, x combinations

Example ACL for plans.txt from before:

```
user::rw-
user:A:r--
user:B:rw-
group::r--
group:qa:rw-
mask::rw-
other:::---
```

This grants specific rights to person 'A' and 'B' ~~without~~ ~~along~~ without changing group membership.

Command Comparison: chmod vs. setfacl/getfacl

Action	Standard chmod	Extended ACL setfacl
i. View permissions	ls -l file	getfacl file
ii. Change owner/ group bits	chmod g+w file	setfacl -m g:qa:rw file
iii. Add specific user	Not possible	setfacl -m u:A:r file
iv. set default ACL (dir)	Not possible	setfacl -d -m u:newuser:rw dir
v. Remove All ACLs	Not Applicable	setfacl -b file

Example:

chmod g+w report.txt // standard way: give group write access

setfacl -m u:A:w report.txt // ACL way: give write access to user 'A'

Permission-Check Priority Logic:

When a process requests file access, the OS checks in the following order:

1. If the user is the file owner, the owner's permissions apply.
2. If a matching named user ACL entry exists (such as user:A:r--), that is used
3. If the user belongs to a matching named group ACL entry, the best-matching group entry is considered
4. If no named entries match, the file's group (group::) permissions are checked.
5. Lastly, the other permissions are used.

The mask entry limits the effective rights for all named users and named groups (but not the

owner on other). For example, if the mask is r--, even if an entry says user:A:rw-, person 'A''s effective permission is only read.

Standard UNIX permissions are simple but inflexible for modern multi-user systems. Extended ACLs solve this by allowing the control per user and per group, without changing group membership. They integrate with the filesystem metadata and follow a clear priority logic, making them essential for detailed access ~~manage~~ management.

References:

1. Arch Linux Wiki - Access control Lists (Usage section)

2. GeeksforGeeks - ACL in Linux (setfacl commands)

3. Red Hat Enterprise Linux Guide - setting Access ACLs