Limitations of Standard Unix File Permissions and the Necessity of Extended ACLs

Introduction: Let’s assume I have a file on my PC, I am the owner, and I want to give access to that file to one of my friends, specifically him, not any groups or others. In this case, would Unix OS allow me to do that? The answer will be no. As far as we have seen in our labs, we can only give permissions to a file based on three entities: Owner, Group, and Others. Each of them can read, write, or execute upon given permissions using some bits assigned to these. Even if this method is easy to use, understandable, and fast, Unix systems can’t help us with the situation aforementioned, making Unix less effective for modern multi-user environments. Thus, Extended Access Control Lists (ACLs) were introduced. ACLs address these problems by providing fine-grained**,** per-user, and per-group permission control.

1. Standard Permission Bottlenecks of the Unix System

1.1. Architectural Limitations of the 3-Tier(Owner, Group, Others) Permission Model

At the very first, what we can see is that we can’t customize the access for multiple individual users. If any particular user wants to have access, he/she must belong to one group. Here, a file belongs to only one group. Anyone who is not the owner of the file will fall into the group or others. This makes it totally a rigid and fixed permission category, whereas ACLs are context-dependent or user-specific

Secondly, this model shuffles between two fixed controls if it’s not the owner, like binary inclusion. One case would be- the user is in the group or not, another one is- the user is “others” or not. There is no alternative way to give access like- User A can read, User B can read and write, and User C has no access, without reconstructing the groups and ownership.

Thus, in Unix systems, we often create many small groups and dynamically add or remove users to cooperate when necessary, which basically does not lead to any good for the system. We see group management overhead, poor maintainability, and high risk of misconfiguration.

1.2. Scenarios where standard group-based permission schemes fail

Scenario 1: Suppose I am the owner of a file. Now there is another group named developers. I want to give read-only access to my friend Bibek without giving access to that whole group. The standard group-based permission scheme fails here. I can’t do that. To accomplish this, I have to add Bibek to that group, change the ownership, or create a new group.

Scenario 2: Let's say I am doing a thesis in a shared research environment with 3 members, where each team member has been assigned to different jobs. Now I need the access of read/write. Another teammate needs read-only access and the last teammate needs no access. We can't avail this kind of simultaneous permission request in this system. Rather, this system sometimes forces compromises, leading to providing the users more access than necessary.

2. Mechanism of Extended ACLs

2.1 Structure of a POSIX ACL Entry: An extended inode metadata to be added to store the additional permissions entries of a file or directory. Each entry consists of entry type, qualifier(user or group), and permission set(rwx). And a POSIX ACL holds this kind of multiple entries. Below are some common ACL entry types:

Entry Type	**Example**	**Description**
user::	user::rw-	User has read/write
user:username:	user:bob:r--	User Bob has read-only
group::	group::r--	Group permissions are limited by the mask
group:groupname:	group:developers:r--	Group developers have read-only
mask::	mask::r--	Maximum effective permissions
other::	other::---	Other permissions

Extension of Inode Metadata

Extended Access Control Lists (ACLs) do not modify the traditional permission bits rather, it stores extra permission entries as extended attributes linked to the inode. ACL data is directly managed by the file system, whereas it is maintained in separate metadata structures referenced by the inode. The extra facilities ACL provides are per-user and per-group permissions, unlike just the single owner and group in UNIX. When it's time to give permissions, Virtual File System (VFS) checks extra attributes from the inode to detect proper permissions. With this extension, we can implement fine-grained access control without even touching the inode's original layout. The systems that are not likely to support ACLs simply ignore them. This also sustains backward compatibility.

2.2. Command-line implementation differences

Changing Standard Permissions (chmod): Restricted to owner, group and others only. It can not target a specific user and overwrites the previous mode settings. It only deals with 9 permission bits.
Command: **chmod 640 file.txt**
(Here, User->read+write ; Group->read ; Others->no access.)

Managing ACLs (setfacl and getfacl):
Command of just read access to a user(bob): **setfacl -m u:bob:r– file.txt**
Command of viewing ACLs: **getfacl file.txt**
Command for removing ACL: **setfacl -b file.txt**

chmod	**setfacl**
Owner/Group/Others	Per-user & per-group
Coarse	Fine-grained
Simple	Extended metadata
Simple systems	Multi-user environments

2.3. Permission Evaluation Priority Logic

OS evaluates permissions in the following order after a process requests access-

First: Owner (user::) [When Process ID matches file owner]

Second: Named user ACL (user:username) [Exact user match has highest priority]

Third: Group entries i) (group::) ii) (group:groupname)

iii) Any matching group applies

Fourth: Mask (mask::) [limits named users and groups except owner]

Fifth: Others (other::) [When, no user or group match is found]

Conclusion: The traditional Unix is efficient and easy to catch, though it’s not suitable for modern, collaborative, multi-user systems due to its rigid and coarse-grained nature. Thus, Extended ACLs are a must-have extension to deal with complex situations rather than a replacement.