

# Access Control Lists vs. File Permissions

Fitzgerald & Long, Inc.

Published in *Database Trends "Perspectives"*, September 2000

When we first converted from Prime INFORMATION to uniVerse on UNIX we struggled with our security plan and finally abandoned an attempt to use UNIX permissions for anything more than a "screen door" kind of protection. However we had been part of that community of Prime INFORMATION users who used ACLs, Access Control Lists, to provide another solid layer of security at the O/S level. We wanted this sort of protection from UNIX. When we first converted to UNIX, ACLs were not available on our O/S. Now that they are available we have revisited the "why and how" of using them.

If you have attempted to provide database protection through the use of UNIX file permissions you will have experienced the limitation that each file may have only one owner and one owning group with all remaining users receiving what is called the "other" category of access rights. PRIMOS, Prime Computer's O/S, called "everybody else" \$REST.

UNIX provides three "permissions" with regard to a file. These are permission to read, write and execute. Read and write permissions are obvious but permission to execute applies to UNIX scripts and programs, not PI/open or uniVerse Basic programs. Additionally, permission to execute allows the use of a directory in a pathname. For example, suppose the user wishes to "cd" (change directories) to a path such as /data1/subdir/mydirectory. The user could not use this pathname if he did not have "x" rights to "subdir", the middle directory in the pathname.

The UNIX command "ls -l" is commonly used to display a file's permissions, the owner and the owning group. The permissions are displayed as a string of 10 characters such as "drwxrwxrwx". Inspecting this string we notice that the left most character is a "d" which identifies this file as a directory. The left most occurrence of "rwx" (characters 2, 3, and 4 counting from the left) are the permissions assigned to the directory owner. The next set of three characters (characters 5, 6 and 7) are the permissions assigned to the owning group. The last three characters on the right are permissions for "everybody else", generally called "other". We would characterize the permissions on this file as "wide open" meaning that all users have read, write and execute permissions.

A user generally has a unique login ID, commonly a name that he enters at the login prompt. Each login ID is also associated with a number called a UID (User ID) and a group. Each group has a number called a GID. Most systems have group names associated with the GIDs. When a user creates a file or directory he becomes the owner and his GID becomes the owning group. A default parameter called a "umask" assigns the permissions to the file for owner, owning group and other. When files are loaded via some media from another UNIX system they may bring permissions, owners and groups from the originating system.

The critical shortfall for permissions is that there are only three opportunities to assign permissions. If you are not the owner then you must be a member of the owning group. If you are not the owner and not a member of the owning group then your permissions are determined by the "everybody else" category.

Access Control Lists augment the standard UNIX file permissions by allowing permissions for more than one user and more than one group. With ACLs you can create a list of users and a list of groups in addition to the owner and the owning group (i.e. UID and GID) for each file and directory. Each user and each group is assigned file permissions to allow or deny read, write and execute privileges.

Perhaps an example that demonstrates the major advantages ACLs have to offer is appropriate. Let's assume that the files in the PAYROLL directory are owned by "root" and the owning group is "sys". It is common for UNIX systems to have the same owner and group for all application data files. For our discussion all the permissions in this PAYROLL directory are rwxrwx---. The left most "rwx" indicates that the owner "root" can read and write all files as well as execute UNIX scripts and executable programs such as C programs. All of the members of the group "sys" have the same privileges (rwx) as the owner, "root".

In our scenario there are 10 people who work in the Payroll Department and need varying degrees of access to these files. The last three dashes (---) indicate that those users who are not members of the group (sys) have no privileges. To give these ten users access to these files we must either let them login as the "root" user or make them members of the "sys" group.

Neither of these choices sounds very good from a security point of view because "root" is a superuser and should not be used except by the administrator and his assistants. The group called "sys" is generally used for UNIX system files. This logic would lead us to consider changing the owning group to "pay", for example, to avoid using the "sys" group. Assuming we did that we could then make our 10 users, who work for the payroll office, members of the "pay" group. This strategy would allow all 10 users full access to all files in this directory (still assuming that all files have the same permissions).

There are 2 of these 10 employees who are responsible for producing the payroll. The other 8 people are responsible for areas such as insurance, credit union membership, time cards, pension plans and employee benefits. The application programs that these other 8 use allow them to read data from some of the payroll related files but does not allow them to update these files. These programs do a good job of limiting access. But, can we improve protection at the UNIX level to augment these programmed limitations?

If we could add another two owners OR another group we could improve the protection. We could make our two users "additional owners". Or, we could create/add another group and make our two employees members. The remaining 8 users would be members of a "supplemental" group where they could still read the files but not update them.

Let's consider four different sets of permissions where we can only specify one owner, one group and the "other" which are all the remaining users. Consider the limitations and the security for each scenario.

### **Scenario #1**

All files have:	Owner = root	rwX
	Group = sys	rwX
	other	--- (no permissions)

To read or write files with these permissions (above), all users (other than root) must be members of the "sys" group. All members of the "sys" group can read AND update the files. If a user is not logged in as "root" or with an ID that is a member of the "sys" group, the user can neither read nor write to files with this configuration.

### **Scenario # 2**

All files have:	Owner = root	rwX
	Group = sys	r--
	other	---

To update files with these permissions above, all users must login as the user "root" because the "sys" group has only read access.

### **Scenario # 3**

*All users can read and write all files. There are no limitations --- and no security.*

All files have:	Owner = root	rwX
	Group = sys	rwX
	other	rwX

### **Scenario # 4**

*Here we have changed the group name to avoid using the "sys" group for application purposes. All users can read these files and only members of the "pay" group (and root) can update them.*

All files have:	Owner = root	rwX
	Group = pay	rwX
	other	r--

It doesn't take long to recognize that there aren't enough choices when you can only assign permissions to one owner and one owning group. Now lets consider the use of ACLs because ACLs allow us to have the equivalent of more than one owner and more than one owning group.

### Scenario # 5 (ACLs)

All files have:	Owner = root	rwX
	User = mary	rwX
	User = betty	rwX
	Group = pay	r--
	other	---

Mary and Betty are the two employees who actually “create” the payroll checks and need to be able to update the files. All of the other people need to be able to read certain files but not update them. These additional users are members of the “pay” group. Any other user cannot access the files. Notice that we have three users (root, mary, and betty) with their own set of permissions. Using ACLs allows us to add the users mary and betty. Note that Scenarios 1 – 4 allow only one user ID, the owner.

### Scenario # 6 (ACLs)

All files have:	Owner = root	rwX
	Group = pay	rwX
	Additional Group = ins	r--
	other	---

If we move Mary and Betty into the “pay” group and the other 8 into the “ins” group we can potentially reduce the amount of ACL maintenance we will be faced with in the future. For example, let’s assume there are 800 files in the directory. Each file has an ACL that must be maintained. If Mary should quit we will have to remove her name from all 800 files and later replace her ACL when someone else takes over her duties. Additionally, Alice may be assigned payroll duties while Betty is on sick leave. We would need to add her to the ACL owner’s list. Notice that ACLs allow us to name more than one group. Permissions don’t.

If we use a group instead of the user’s login id (i.e. mary, betty, alice) we can simply add and remove users from the group, one entry, not 800. Another limitation for most UNIX systems is that the number of bytes used to define the ACL has a limit. On SUN systems it is 1,024 bytes. That will accommodate a lot of users and groups but do you really want to do that much data entry? Consider using groups when there is more than one person who needs the same access to a file.

In summary, ACLs offer a lot more flexibility in setting up UNIX level security than the traditional permissions but not all UNIX operating systems offer ACLs. Additionally some that do offer ACLs have commands that are fairly complex; system administrators may find that ACL administration is very time consuming. If you are interested in checking out what your UNIX system has to offer try looking in the “man” pages for “acl”.