CHAPTER 7

Security and Permissions

7.1 CHMOD COMMAND

With the help of chmod command we can change permissions of a file or a directory of our own.

For any file or directory which is available in UNIX system there exist three types of owners given as :

- Owner (real user)
- Group Member
- Others

For the purpose of administration, users are grouped such that resources can be appropriated. For example, the administrator can appropriate for all second year B.Tech students 1 hour CPU time, 20 hours of Terminal time, 3 pages of hard copy and 10MB of space. Except the disk space others are allocated on weekly basis. These appropriations can be different for final year students. Also, groups make users to share the files.

Similarly, for any file or a directory three types of operations can be carried out namingly :

- Read
- Write
- Execute

If we have reading permissions on a file we can see the content of the file or some other command such as cat which wants to read the file on behalf of us also works. Similarly if we have writing permissions on a file we can modify the content of the file (please note the file can be deleted by only legal owner of the file and super user even if you have writing permissions). Similarly, if we have execution permissions for a file then it can be loaded into RAM and executed if it is executable file. If the file is not executable and having executable permissions will have no effect on the file. We will be knowing in the next chapters how to run a shell script (a simple text file).

Similarly, if we have reading permissions on a directory we can run ls command on it. If we have writing permissions on a directory we can create file or directory in it (try to create a file in /bin). If we have executable permissions then we can enter into it.

For example create a file (say) xyz and run the following command.

ls –l xyz

The result may look like this :

-rw-r—r— _____ xyz

The first string in the above commands output is called as mode string or permissions string which indicates what permissions are available to the file for user, group and other. The first character in the above string is – indicating that xyz is a file. There can be some characters such as d,b,c,p,l at this location which may indicate that xyz is directory, character special file (character device), block special file (block device), pipe file or link file respectively.

The next three characters "rw-" indicates that the user can read, write but not execute. Similarly "r—" for group and others indicates that group members and others can only read the xyz.

The chmod command supports two ways of changing file/directory permissions.

- Octal Approach
- Symbol Approach

Octal approach of changing File Permissions

In octal approach, we specify three digit octal numbers to change permissions such as :

chmod 700 xyz

ls –l xyz

Output of the above command looks like :

-rw——— _____ xyz

In this approach, we have to specify the required permissions without considering what the previous or existing permissions are. Thus this technique is called as absolute approach.

Here, we assume

- Read – 4
- Write – 2
- Execute – 1

If we want all the three permissions then we use 7 (sum of 4+2+1) and vice versa . Like this in the above example 700 we have used as we want all the permissions to be available for the user and none to others and group.

chmod 000 xyz

ls –l xyz

———— _____ xyz

Now if we try to run the following commands, we can not succeed as there is no reading permission for us.

cat xyz

vi xyz

chmod 400 xyz

ls –l xyz

-r———— _____ xyz

Now if we try to run the following commands, we can succeed as there is reading permission for us. However, we can not modify the file content using vi command as we do not have writing permissions.

cat xyz

vi xyz

chmod 200 xyz

ls –l xyz

———— _____ xyz

Now if we try to run the following commands, we can not succeed as there is no reading permission for us.

cat xyz

vi xyz

However, the following command may succeed.

cat>>xyz

Asas
Asas
Asa
Aas
^d

Symbolic way of changing File Permissions

We can change permissions with another approach known symbolic approach. Here, we use the following symbols to refer groups and permissions and + for giving and – (minus)for removing permissions.

- All –a
- User –u
- Group –g
- Others –o
- Read – r
- Write –w
- Execute – x
- = to assign permissions
- + to add permissions
- to remove permissions

For example run the following command gives reading, writing and execution permissions to user or owner.

chmod u=rwx xyz

ls –l xyz

We will see xyz permissions as :

-rwx——— _____ xyz

Now, if we execute the following command

chmod u-x,go+r xyz

ls –l xyz

We will see xyz permissions as :

-rw-r—r— _____ xyz

A Note on Sticky bit, setgid bit, setuid bit

Sticky bit on a file

In the past having the sticky bit set on a file meant that when the file was executed the code for the program would "stick" in RAM. Normally once a program has finished its code was taken out of RAM and that area used for something else.

The sticky bit was used on programs that were executed regularly. If the code for a program is already in RAM the program will start much quicker because the code doesn't have to be loaded from disk.

However today with the advent of shared libraries and cheap RAM most modern UNIX's ignore the sticky bit when it is set on a file.

Sticky bit on a directory

The /tmp directory on UNIX is used by a number of programs to store temporary files regardless of the user. For example when you use elm (a UNIX mail program) to send a mail message, while you are editing the message it will be stored as a file in the /tmp directory. Please note that every user will have his own privacy rules on his files which are stored in such directories.

Modern UNIX operating systems (including Linux) use the sticky bit on a directory to make /tmp directories more secure. Try the command ls -ld /tmp what do you notice about the file permissions of /tmp.

If the sticky bit is set on a directory you can only delete or rename a file in that directory if you are :

- the owner of the directory,
- the owner of the file, or
- the super user

Changing passwords—setuid bit??

When you use the passwd command to change your password the command will actually change the contents of either the /etc/passwd or /etc/shadow files. These are the files where your password is stored. However, we can not directly edit /etc/passwd as we don't have permissions for the same.

Check the file permissions on the /etc/passwd file?.

ls -l /etc/passwd

-rw-r—r— 1 root root 697 Feb 1 21:21 /etc/passwd

Now the file belongs to root and others do not have write permission thus we are unable to modify through vi. Then how do does the passwd command change my password in /etc/passwd file?

The answer is setuid and setgid.

Let's have a look at the permissions for the passwd command (first we find out where it is).

ls -l /usr/bin/passwd

-rws—x—x 1 root bin 7192 Oct 16 06:10 /usr/bin/passwd

Notice the s symbol in the file permissions of the passwd command, this specifies that this command is setuid.

When we execute the passwd command a new process is created. The real UID and GID of this process will match my UID and GID. However the effective UID and GID (the values used to check file permissions) will be set to that of the command. Thus, we are able to modify the file /etc/passwd which belongs to root.

Similarly, setgid bit is useful while enforcing locks on files.

7.2 THE UMASK COMMAND

This command when executed without any argument it displays the current value of the umask. This umask value is used to change the default permissions of any file or directory created. By changing the umask value we can change default permissions of a file or directory created.

For Example let us create a file and see their default permissions.

cat>p1

add

adjda

^d

ls -l p1

-rw-r—r— 1 root root 4 Feb 10 00:32 p1

Now change umask value and create a file to see its permissions.

umask 000

cat>p3

ads

sad

sdsd

^d

ls -l p3

-rw-rw-rw- 1 root root 9 Feb 10 00:35 p3

We can see that permissions of files p1 and p3 are different.

UNIX Kernel uses a mask known as file creation mask (octal 666). While a file is created this mask and umask combinedly plays role in deciding the permissions of a file. Default umask value is 022. Thus, when file p1 is created this is used. Whereas while p3 is created, umask value is taken as 000, which we have specified along the command line.

	P1	P3
File Mask (Binary)	110110110	110110110
Umask	000010010 (022)	000000000 (000)
Exclusive-OR	110100100	110110110
Permissions	rw-r—r—	rw-rw-rw-

The same is applicable to default directory permissions also. UNIX Kernel uses default directory creation mask as 777.

Directory l1 is created after changing the umask where as directory l2 is created before changing. We can see the difference in the permissions.

drwxrwxrwx 2 root root 4096 Feb 10 00:43 l1

drwxr-xr-x 2 root root 4096 Feb 10 00:44 l2

7.3 THE CHOWN COMMAND

With the help of chown command, we can change owner ship of a file or directory. Only real owner (exception for the super user) of the file or directory can change owner ship of a file or directory. Once it is changed, he/she will not have any authority revert it back.

chown username filename

Example

chown rao xyz.c

This command changes owner of the file xyz.c as rao. Of course, real owner of the file can do it; exception for super user.

7.4 THE CHGRP COMMAND

With the help of chgrp command we can change group membership of a file. For example :

chgrp groupname filename

7.5 THE CHATTR COMMAND

It is possible to use ext2 attributes to protect things. These attributes are manipulated with the chattr command. There is an 'append-only' attribute: a file with this attribute may be appended to, but may not be deleted, and the existing contents of the file may not be overwritten. If a directory has this attribute, any files or directories within it may be modified as normal, but no files may be deleted. The 'append-only' attribute is set with.

chattr +a FILE...

There is also an 'immutable' attribute, which can only be set or cleared by root. A file or directory with this attribute may not be modified, deleted, renamed, or (hard) linked. It may be set as follows :

chattr +i FILE...

The ext2fs also provides the 'undeletable' attribute (+u in chattr). The intention is that if a file with that attribute is deleted, instead of actually being reused, it is merely moved to a 'safe location' for deletion at a later date. Unfortunately this feature has not yet been implemented in mainstream kernels; and though in the past there has been some interest in implementing it, it is not (to my knowledge) available for any current kernels.

7.6 THE ID COMMAND

The id command can be used to discover username, UID, group name and GID of any user.

For example, when we have executed id command on our machine we got the following output.

uid = 500(venkat) gid = 100(users) groups = 100(users)

SUID/SGUID files

Suid root refers to a special attribute called set user id. This allows the program to do functions not normally allowed for users to do themselves. Low level networking routines, controlling graphical display functions, changing passwords, and logging in are all examples of programs that rely on executing their functions as a user that is not restricted by standard file permissions. While many programs need this functionality, the program must be bug free in only allowing the user to do the function the program was designed for. Every SUID root program represents a potential security problem.

The first step in controlling SUID root programs is to have a baseline, the list of all SUID program in the system. This can be achieved quite easily by using find :

find / -type f -perm +6000 -exec ls -l {} \; > suid.list

(note: this will find both set user id and set group id programs)

On Solaris POSIX find, the same command becomes :

find / -type f \(-perm -4000 -o -perm -2000 \) -exec ls -l {} \;

These commands will find all the SUID programs on a system and pipes the commands to a file called suid.list. The next step in controlling SUID root programs is to analyze which programs should not be SUID root or can be removed without impeding system functionality. This is the domain of System Adminstrator. Thus, we shall not discuss more.

Finding World Writable, Abandoned and other Abnormal Files

Often system administrators need to detect "abnormal" files (e.g., world writable files, files with no valid owner and/or group, SetUID files, files with unusual permissions, sizes, names, or dates). We already discussed a very important case of SUID/SGUID files. Now let's concentrate of other possibilities. Here are several simplified but potentially useful examples.

- To find all world writable directories :
  find / -perm -0002 -type d -print
- To find all world writable files :
  find / -perm -0002 -type f -print
- Find both files and directories (exclude symbolic links which produce false positives)
  find / -perm -2 ! -type l -ls
- Find files with messed UID or GID :
  find / -nouser -o -nogroup -print
- Find broken symbolic links
  find / -type l -print | perl -nle '-e || print';
  Note : This command starts at the topmost directory (/) and lists all links (-type l -print) that the perl interpreter determines broken links (-nle '-e || print'). You can further pipe the output through xargs and use the rm -f {} if you want to delete such symbolic links.

- List zero-length files
  find . -empty -exec ls {} \;
  After finding empty files, we might choose to delete them by replacing the ls command with the rm command. But it's better to verify the list before jumping the gun...
- Clean out core dumps and temporary files
  find . \(-name a.out -o -name '*.o' -o -name 'core' \) -exec rm {} \;

Those examples are pretty simplistic as in "real life" we need to be able to block traversing of NFS and other non-native file systems and avoid getting to special memory-mapped file systems like proc. As a system administrator one can use find to locate suspicious files (e.g., world writable files, files with no valid owner and/or group, SetUID files, files with unusual permissions, sizes, names, or dates). Here's a final more complex example :

```
find / -noleaf -wholename '/proc' -prune \
     -o -wholename '/sys' -prune \
     -o -wholename '/dev' -prune \
     -o -wholename '/windows-C-Drive' -prune \
     -o -perm -2 ! -type l ! -type s \
     ! \( -type d -perm -1000 \) -print
```

This says to search the whole system, skipping the directories /proc, /sys, /dev, and /windows-C-Drive (presumably a Windows partition on a dual-booted computer). The Gnu -noleaf option tells find not to assume all remaining mounted file systems are UNIX file systems (you might have a mounted CD for instance). The -o is the Boolean OR operator, and ! is the Boolean NOT operator (applies to the following criteria).

Another and potentially simpler and faster approach is to use **-fstype *type*** predicate. It is true if the file system to which the file belongs is of type *type*. For example on Solaris mounted local file systems have type **ufs** (Solaris 10 added **zfs**). For AIX local file system is **jfs** or **jfs2** (journalled file system).

But sometimes the same server uses several types of local file systems (for example ext3 and reisner). In this case you can use predicate OR and create expression that covers each used file system or use generic predicate **local** and in certain circumstances predicate **mount.**

7.7 UNIX/LINUX ACLS AT WORK[1]

The traditional POSIX file system object permission model defines three classes of users called owner, group, and other. Each of these classes is associated with a set of permissions. The permissions defined are read (r), write (w), and execute (x). In this model, the *owner class* permissions define the access privileges of the file owner, the *group class* permissions define the access privileges of the owning group, and the *other class* permissions define the access privileges of all users that are not in one of these two classes.

An ACL consists of a set of entries. The permissions of each file system object have an ACL representation, even in the minimal, POSIX.1-only case. Each of the three classes of users is represented by an ACL entry. Permissions for additional users or groups occupy additional ACL entries. Table 1 shows the defined entry types and their text forms. Each of these entries consists of a type, a qualifier that specifies to which user or group the entry applies, and a set of permissions. The qualifier is undefined for entries that require no qualification.

ACLs equivalent with the file mode permission bits are called ***minimal* ACLs**. They have three ACL entries. ACLs with more than the three entries are called ***extended* ACLs**. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries.

Table 1 Types of ACL Entries

Entry type	**Text form**
Owner	user::*rwx*
Named user	user:*name*:*rwx*
Owning group	group::*rwx*
Named group	group:*name*:*rwx*
Mask	mask::*rwx*
Others	other::*rwx*

These named group and named user entries are assigned to the *group class*, which already contains the owning group entry. Different from the POSIX.1 permission model, the group class may now contain ACL entries with different permission sets, so the group class permissions alone are no longer sufficient to represent all the detailed permissions of all ACL entries it contains. Therefore, the meaning of the group class permissions is redefined: under their new semantics, they represent an upper bound of the permissions that any entry in the group class will grant. This upper bound property ensures that POSIX.1 applications that are unaware of ACLs will not suddenly and unexpectedly start to grant additional permissions once ACLs are supported.

In minimal ACLs, the group class permissions are identical to the owning group permissions. In extended ACLs, the group class may contain entries for additional users or groups. These results in a problem: some of these additional entries may contain permissions that are not contained in the owning group entry, so the owning group entry permissions may differ from the group class permissions.

This problem is solved by the virtue of the mask entry. With minimal ACLs, the group class permissions map to the owning group entry permissions. With extended ACLs, the group class permissions map to the mask entry permissions, whereas the owning group entry still defines the owning group permissions. The mapping of the group class permissions is no longer constant. Figure 2.55 shows these two cases.

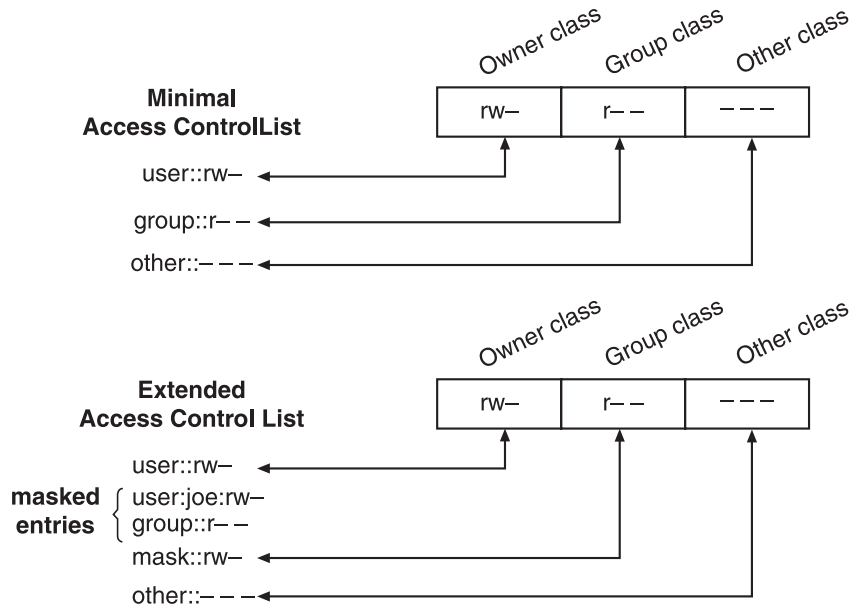


Figure 7.1. Mapping between ACL Entries and File Mode Permission Bits

When an application changes any of the owner, group, or other class permissions (e.g., via the *chmod* command), the corresponding ACL entry changes as well. Likewise, when an application changes the permissions of an ACL entry that maps to one of the user classes, the permissions of the class change.

The group class permissions represent the upper bound of the permissions granted by any entry in the group class. With minimal ACLs this is trivially the case. With extended ACLs, this is implemented by masking permissions (hence the name of the mask entry): permissions in entries that are a member of the group class which are also present in the mask entry are effective. Permissions that are absent in the mask entry are masked and thus do not take effect. See Table 2.

Table 2. Masking of Permissions

Entry type	**Text form**	**Permissions**
Named user	user:joe:r-x	r-x
Mask	mask::rw-	rw-
Effective permissions		r-

The owner and other entries are not in the group class. Their permissions are always effective and never masked.

ACLs that define the current access permissions of file system objects are called ***access* ACL**. A second type called *default* ACL is also defined. They define the permissions a file system object inherits from its parent directory at the time of its creation. Only directories can be associated with default ACLs. Default ACLs for non-directories would be of no use, because no other file system objects can be created inside non-directories. Default ACLs play no direct role in access checks. When a directory is created inside a directory that has a default ACL, the new directory inherits the parent directory's default ACL both as its

access ACL and default ACL. Objects that are not directories inherit the default ACL of the parent directory as their access ACL only.

The permissions of inherited access ACLs are further modified by the *mode* parameter that each system call creating file system objects has. The *mode* parameter contains nine permission bits that stand for the permissions of the owner, group, and other class permissions. The effective permissions of each class are set to the intersection of the permissions defined for this class in the ACL and specified in the *mode* parameter.

If the parent directory has no default ACL, the permissions of the new file are determined as defined in POSIX.1. The effective permissions are set to the permissions defined in the *mode* parameter, minus the permissions set in the current *umask*.

The umask has no effect if a default ACL exists.

Access Check Algorithm

A process requests access to a file system object. Two steps are performed. Step one selects the ACL entry that most closely matches the requesting process. The ACL entries are looked at in the following order: owner, named users, (owning or named) groups, others. Only a single entry determines access. Step two checks if the matching entry contains sufficient permissions. A process can be a member in more than one group, so more than one group entry can match. If any of these matching group entries contain the requested permissions, one that contains the requested permissions is picked (the result is the same no matter which entry is picked). If none of the matching group entries contains the requested permissions, access will be denied no matter which entry is picked.

The access check algorithm can be described in pseudo-code as follows :

If the user ID of the process is the owner, the owner entry determines access

else if the user ID of the process matches the qualifier in one of the named user entries, this entry determines access

else if one of the group IDs of the process matches the owning group and the owning group entry contains the requested permissions, this entry determines access

else if one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access

else if one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching named group entries contains the requested permissions, this determines that access is denied

else

the other entry determines access.

If

the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted

else if the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted

else

access is denied.

ACLs are displayed using the *getfacl* command. We can run getfacl command with file or directory name along the command line such as :

getfacl dir

It displays :

file: dir

owner: rao

group: users

user::rwx

group::r-x

other::—

The first three lines of output contain the file name, owner, and owning group of the file as comments. Each of the following lines contains an ACL entry for one of the three classes of users: owner, group, and other.

If we use omit-header option with getfctl command it displays only ACL's. For example

getfacl —omit-header dir

command outputs

user::rwx

group::r-x

other::—

If we wanted to grant permissions we have to use –m option with setfctl command. For example, we wanted to grant read, write, and execute access to user Ram in addition to the existing permissions we can run the following command.

setfacl -m user:Ram:rwx dir

To see the current permissions we can run the following command.

getfacl —omit-header dir

The output is :

user::rwx

user:Ram:rwx

group::r-x

mask::rwx

other::—

We can observe that two additional entries have been added to the ACL: one is for user Ram and the other is the mask entry. The mask entry is automatically created when needed but not provided. Its permissions are set to the union of the permissions of all entries that are in the group class, so the mask entry does not mask any permission.

The mask entry now maps to the group class permissions. The output of *ls* changes is shown next.

ls -dl dir

drwxrwx—+ ... Rao users ... dir

An additional "+" character is displayed after the permissions of all files that have extended ACLs. This seems like an odd change, but in fact POSIX.1 allocates this character

position to the optional alternate access method flag, which happens to default to a space haracter if no alternate access methods are in use.

The permissions of the group class permissions include write access. Traditionally such file permission bits would indicate write access for the owning group. With ACLs, the effective permissions of the owning group are defined as the intersection of the permissions of the owning group and mask entries. The effective permissions of the owning group in the example are still r-x, the same permissions as before creating additional ACL entries with *setfacl*.

The group class permissions can be modified using the *setfacl* or *chmod* command. If no mask entry exists, *chmod* modifies the permissions of the owning group entry as it does traditionally. The next example removes write access from the group class and checks what happens.

chmod g-w dir

ls -dl dir

drwxr-x—+ ... Rao users ... dir

getfacl —omit-header dir

The output is :

user::rwx

user:Ram:rwx #effective:r-x

group::r-x

mask::r-x

other::—

As shown, if an ACL entry contains permissions that are disabled by the mask entry, *getfacl* adds a comment that shows the effective set of permissions granted by that entry. Had the owning group entry had write access, there would have been a similar comment for that entry. Now see what happens if write access is given to the group class again.

chmod g+w dir

ls -dl dir

drwxrwx—+ ... Rao users ... dir

getfacl —omit-header dir

The output is :

user::rwx

user:Ram:rwx

group::r-x

mask::rwx

other::—

After adding the write permission to the group class, the ACL defines the same permissions as before taking the permission away. The *chmod* command has a nondestructive effect on the access permissions.

In the following example, we add a default ACL to the directory. Then we check what *getfacl* shows.

setfacl -d -m group:toolies:r-x dir

getfacl —omit-header dir

The output is :

```
user::rwx
user:Ram:rwx
group::r-x
mask::rwx
other::—
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::—
```

Following the access ACL, the default ACL is printed with each entry prefixed with "default:". We have only specified an ACL entry for the *toolies* group in the *setfacl* command. The other entries required for a complete ACL have automatically been copied from the access ACL to the default ACL. This is a Linux-specific extension; on other systems all entries may need to be specified explicitly. The default ACL contains no entry for Ram, so Ram will not have access (except possibly through group membership or the other class permissions).

A subdirectory inherits ACLs as shown next. Unless otherwise specified, the *mkdir* command uses a value of 0777 as the *mode* parameter to the mkdir system call, which it uses for creating the new directory. Observe that both the access and the default ACL contain the same entries.

mkdir dir/subdir

getfacl —omit-header dir/subdir

The output is :

```
user::rwx
group::r-x
group:toolies:r-x
mask::r-x
other::—
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::—
```

Files created inside *dir* inherit their permissions as shown next. The *touch* command passes a *mode* value of 0666 to the kernel for creating the file. All permissions not included in the *mode* parameter are removed from the corresponding ACL entries. The same has happened in the previous example, but there was no noticeable effect because the value 0777 used for the *mode* parameter represents a full set of permissions.

touch dir/file

ls -l dir/file

-rw-r——+ ... Rao users ... dir/file

getfacl —omit-header dir/file

The ooutput is :

user::rw-

group::r-x #effective:r—

group:toolies:r-x #effective:r—

mask::r—

other::—

No permissions have been removed from ACL entries in the group class; instead they are merely masked and thus made ineffective. This ensures that traditional tools like compilers will interact well with ACLs. They can create files with restricted permissions and mark the files executable later. The mask mechanism will cause the right users and groups to end up with the expected permissions.

Table displays maxmum number of supported ACL entries in various file systems. Also, Table displays file access time before and after incorporating ACLs in file systems. In addition Table 5 displays the time required for copying files before and after incorporating ACLs in various file systems with file size 137MB and 2.8GB.

Table 3. Maximum Number of Supported ACL entries

File system	Max. entries
XFS	25
Ext2, Ext3	32
ReiserFS, JFS	8191

Table 4. Microseconds for Initially Accessing a File After System Restart, with and without ACLs

	Without ACL	With ACL
Ext2	9	1743
Ext3	10	3804
ReiserFS	9	6165
XFS-256	14	7531
XFS-512	14	14
JFS	13	13

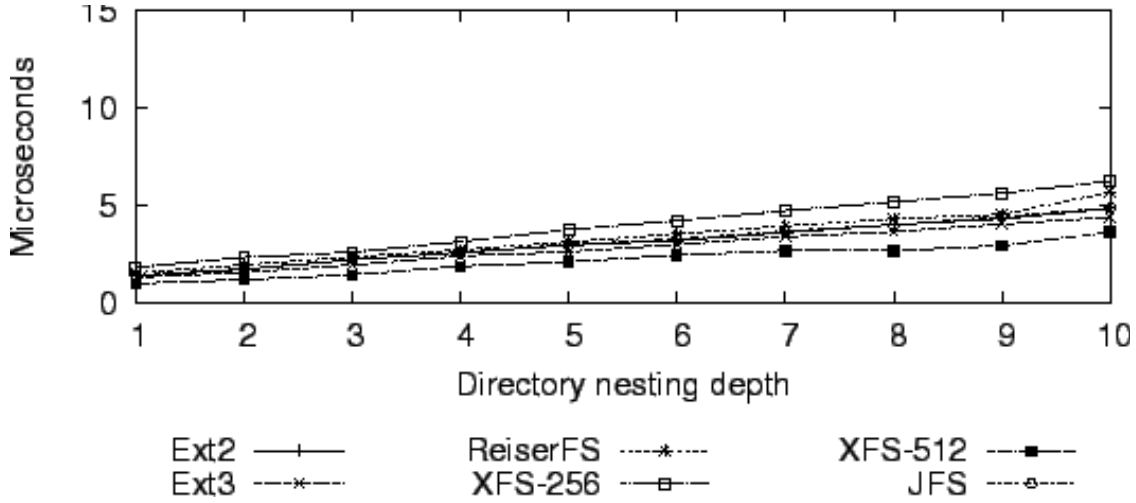


Figure 7.2 Access system call without ACL's

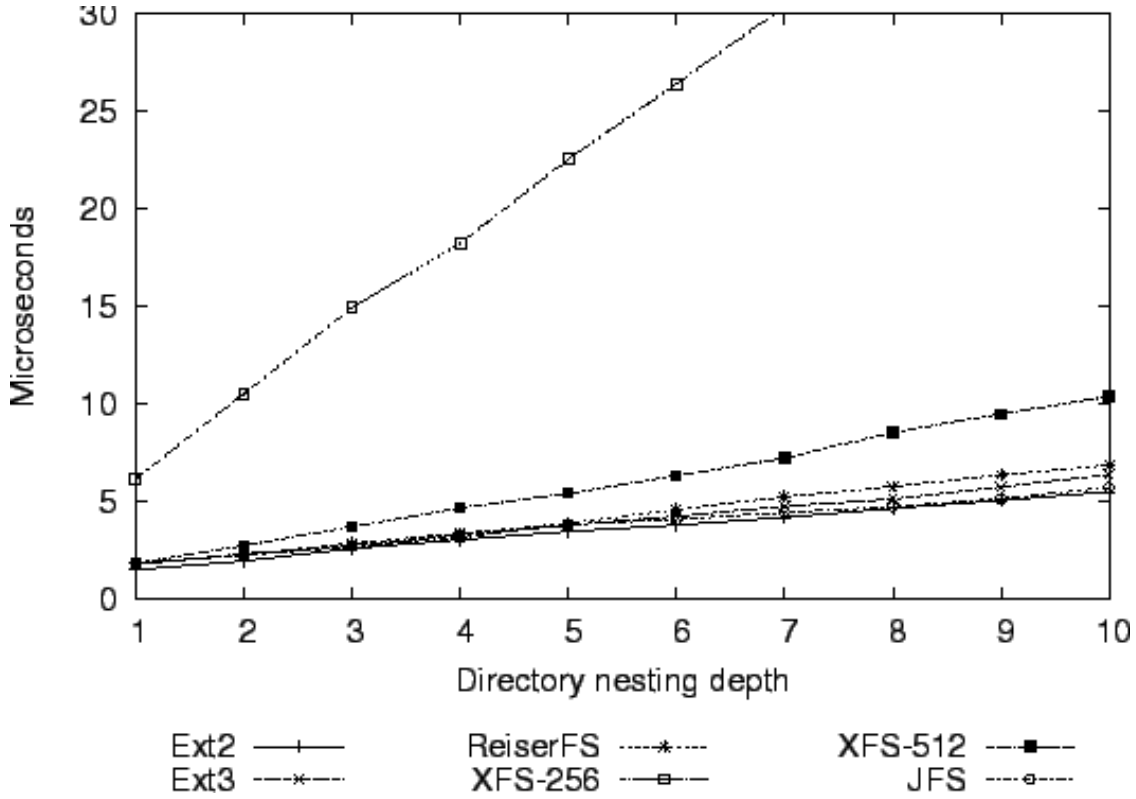


Figure 7.3. Access system call with ACL's

Figure 7.2 shows the time taken for one access system call, depending on the number of directory levels in the pathname argument. Figure 7.3 shows the performance of the same operation with a five-entry access ACL on each directory. XFS's overhead for converting

the ACL from its EA representation to its in-memory representation results in a noticeable difference, particularly with 256 byte i-nodes.

7.8 ENCRYPTION: A SAFETY MEASURE

In order to protect our files, we can use encryption in UNIX/Linux. Command such as crypt, des and pgp can be used.

The Crypt Command

With the help of crypt command, we can encrypt a file. For example :

crypt<aaa >encryptedfile

Here, the file aaa is encrypted and its encrypted file content is stored in the file encryptedfile. While running crypt command it asks for a key like password. The same key one has to use during decrypting.

To decrypt :

crypt <encryptedfile

We can directly use encrypted file with vi command. For example :

vi –x encryptedfile

Of course, vi command asks for the key. We have to give the same key which we have given during encryption.

The des command

This is better secure command than crypt. The des command is a filter that reads from standard input and writes to standard output. It usually accepts the following command-line options :

des -e|-d [-h] [-k key] [-b]

When using the DES, encryption and decryption are not identical operations, but are inverses of each other. The option -e specifies that you are encrypting a file. For example :

des -e <file > encryptedfile

Enter key: ***mykey***

Enter key again: ***mykey***

cat encryptedfile

"UI}mE8NZlOi\Iy|

(Of course, the Enter key: prompt is from the des program; the key is not echoed.)

We have to use the -d option to decrypt the encrypted file: By default decrypted file content appears on the monitor. By redirecting the same, we can store in a file. Do remember that we have to specify the same key which we have given during encryption.

des -d < encryptedfile

Enter key: **mykey**

Enter key again: **mykey**

We can specify the key with –k option also. However, it is not recommended to do so as there is danger of the seeing the same through ps command by others.

The pgp command[2]

PGP (Pretty Good Privacy) is a public key encryption package to protect users files and E-mail messages. It allows us to communicate securely with people we have never met,

with no secure channels needed for prior exchange of keys. By default it compresses the data and encrypts. Thus, encrypted file sizes will be lesser than original files.

Terminology

user id: an ASCII string used to identify a user. User IDs tend to look like "John Q. Public "; please try sticking to that format. When giving a user id to PGP, we may specify any unique (case-insensitive) sub-string. E.g. john, or jqp@xyz.

pass phrase: the secret string used to conventionally encipher our private key. It's important that this be kept secret.

keyring: a file containing a set of public or secret keys. Default names for public and secret rings are "pubring.pgp" and "secring.pgp" respectively.

ASCII armor: the ASCII radix 64 format PGP uses for transmitting messages over channels like E-Mail.

Essentially in this encryption methodology two steps are involved known as :

1. Key generation and management
2. File or Message encryption/authentication

Key Generation and Management

In essence, this method involves use of a pair of keys known as public and private keys. If any one encrypts a message using the public key, it can be decrypted using only respected private key. That is, if one wants to send a message to us, they use our public key and encrypt before sending through Internet. As it can be decrypted through our private key only, even if some one captures the message purposefully or un-expectedly they can not know the message content.

First, create a directory for PGP :

cd

mkdir .pgp

Set the environment variable PGPPATH to point to this directory: If we are using csh, we can do the following :

setenv PGPPATH $HOME/.pgp

We can as well add the above line to our .cshrc file. In the case of bash shell, we can add the following line to .profile or .bashrc.

PGPATH=$HOME/.pgp

export PGPATH.

Now, we can generate our private and public keys by running the command :

pgp -kg

When we are asked for a user ID, type a name followed by email address in angle brackets.

When you are asked for a pass phrase, type in a **good** pass phrase. We can use

pwcheck -h math

to check if our pass phrase is not easy to guess. We will be advised to type in some random text pass phrase.

To check what public keys are available for use, we can run the following command **pgp -kv** (*keys, view list)*

...

Type bits keyID Date User ID

DSS 4096/1024 0xEA5F4D84 2000/05/08 *** DEFAULT SIGNING KEY ***

RITCH CENTER<ritchcenter@yahoo.com>

After we have generated our key, we should do two things immediately :

1. Sign it by oourself. We should always sign our own key right away. Do this as : pgp -ks ritchcenter@yahoo.com
2. Generate a revocation certificate and store it offline somewhere. *Don't send it to anyone!* The idea behind generating the revocation right now is that we still remember the pass phrase and have the secret key available. If something should happen to our stored key, or we forget the pass phrase, the public/private key pair becomes useless. Having the revocation certificate ready in advance allows us to send it out if that should ever happen. We generate the certificate by :

pgp -kx RITCH revoke.pgp

Now, save the *revoke.pgp* file in a safe place, off line.

To extract a printable, ASCII version of our key, use PGP's **-kxaf** (Key extract ASCII filter) command :

pgp -kxaf USerID

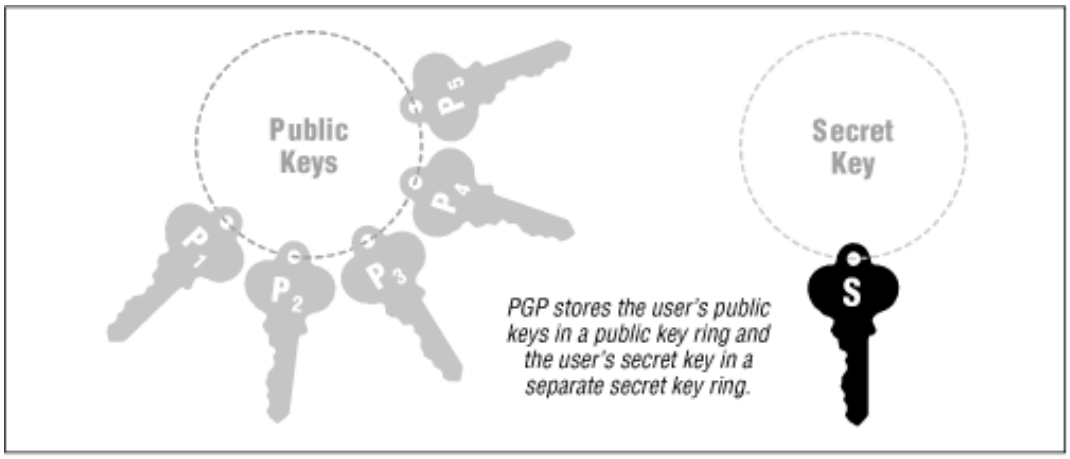
The standard output of the above command can be saved in a text file which becomes our public key (This process is officially called extracting a key). For other people to encrypt mail for you or check your signatures, they must have access to this public key.

If we get somebody else's PGP key, we can add it to our keyring with the PGP -ka (key add) option. Simply save the key in a file, then type :

pgp -ka pgpfilehavingkeyofothers

We can run the **pgp –kv** to check whether it is added to our keyring or not.

The pgp adds all the public keys of a user in the keyring as shown in the Figure.



Encrypting documents

Once we have generated our keys, we can encrypt a file *file* by simply typing :

pgp -e file

When we are prompted with :

A user ID is required to select the recipient's public key. Enter the recipient's user ID :

Type in the user ID we chose during the key generation process. (e.g: *ritchcenter@yahoo.com*) The encrypted file will then be created with the name *file.pgp*. We do not need to use a pass phrase for every document we encrypt. Also, we can now remove the original *file*.

Decrypting documents :

To decrypt *file.pgp*, type

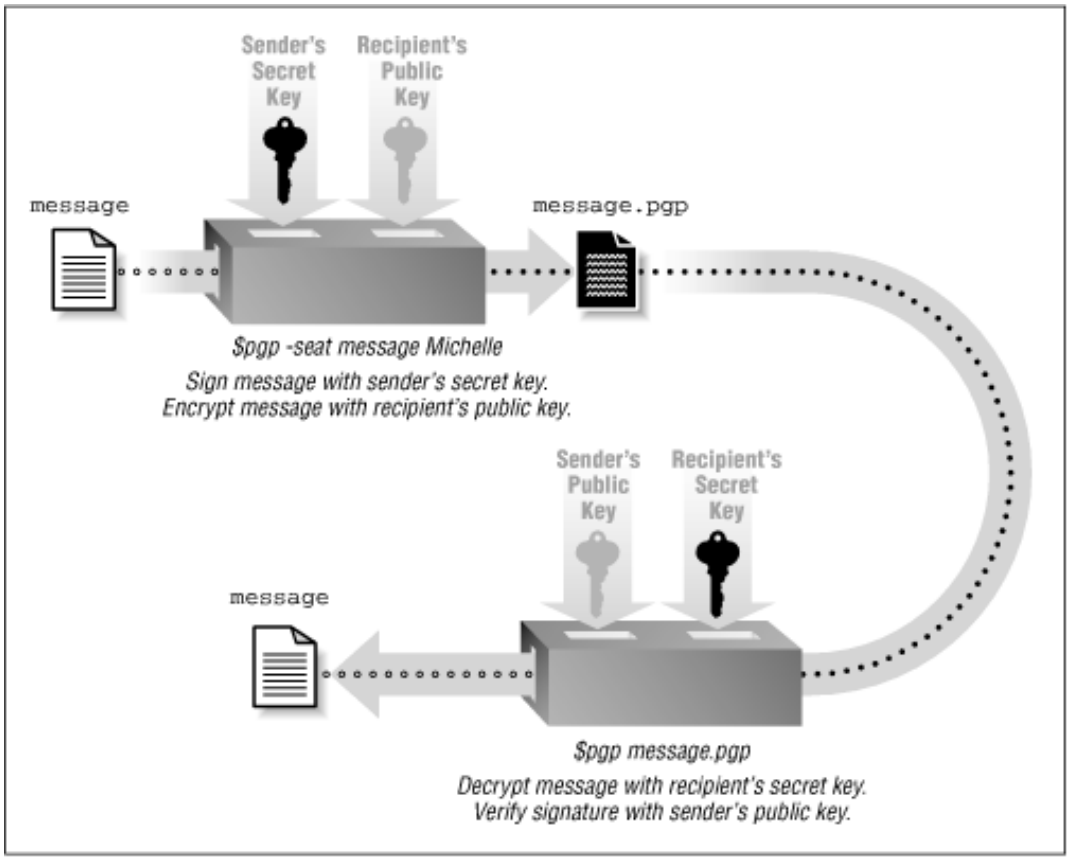
pgp file.pgp

When you are prompted for a pass phrase, type in the personal pass phrase you chose during key generation.

To authenticate a message, we can do the following steps.

1. create file (xyz) to be authenticated.
2. Run the command pgp –sta xyz. It needs our pass phrase.
3. We find xyz.asc as the output which contains our signature attached.
4. We can send this file xyz.asc through email to any one.
5. Assuming that they have saved that message in a file xyz.asc, by simply running pgp xyz.asc they can get the original message. Of course, they would have got our public key in their keyring. If they do not have our key in their keyring then they have to add our public key to it first then only they can decrypt the file xyz.asc.

After we have somebody's public key, we can encrypt a message using the PGP's **-eat** command. This will encrypt the message, save it in ASCII (so we can send it with electronic mail), and properly preserve end-of-line characteristics (assuming that this is a text message). We can sign the message with our own *digital signature* by specifying -seat instead of -eat. This process is shown graphically in Figure.



Some Precautions

- **Multiple Formats:** A document may be stored in different forms. For example, when you use LaTeX, we may have the document in the form of LaTeX source in one set of files and in another form in the resultant DVI file. If we want to protect a document, we have to make sure that it is protected in all forms.
- **Backups:** Text editors (e.g. emacs) and other utilities (e.g. ispell) create backup versions of the files on which they are being used. If you decide to protect a file by encrypting it, you have to make sure that all its backups are removed.

Some of useful PGP Commands

To see a quick command usage summary for PGP, just type :

pgp -h

To encrypt a plaintext file with the recipient's public key :

pgp -e textfile her_userid ...

To sign a plaintext file with our secret key :

pgp -s textfile [-u your_userid]

To sign a plaintext file with your secret key, and then encrypt it with the recipient's public key :

pgp -es textfile her_userid ... [-u your_userid]

To create a signature certificate that is detached from the document :

pgp -sb textfile [-u your_userid]

To encrypt a plaintext file with just conventional cryptography, type :

pgp -c textfile

To decrypt an encrypted file, or to check the signature integrity of a signed file :

pgp ciphertextfile [-o plaintextfile]

To see a quick summary of PGP's key-management commands, just type :

pgp -k

To generate your own unique public/secret key pair :

pgp -kg

To add a public or secret key file's contents to our public or secret key ring :

pgp -ka keyfile [keyring]

To remove a key from our public key ring :

pgp -kr userid [keyring]

To extract (copy) a key from our public or secret keyring :

pgp -kx[a] userid keyfile [keyring]

To view the contents of our public key ring :

pgp -kv[v] [userid] [keyring]

To view the "fingerprint" of a public key, to help verify it over the telephone with its owner :

pgp -kvc [userid] [keyring]

To view the contents and check the certifying signatures of our public key ring :

pgp -kc [userid] [keyring]

To edit the pass phrase for or add a userid to our secret key :

pgp -ke userid [keyring]

To edit the trust parameters for a public key :

pgp -ke userid [keyring]

To remove a key or just a userid from our public keyring :

pgp -kr userid [keyring]

To sign and certify someone else's public key on our public key ring :

pgp -ks her_userid [-u your_userid] [keyring]

To remove selected signatures from a userid on a keyring :

pgp -krs userid [keyring]

Command options that can be used in combination with other command options (sometimes even spelling interesting words) :

To produce a ciphertext file in ASCII radix-64 format, just add the -a option when encrypting or signing a message or extracting a key :

pgp -sea textfile her_userid

pgp -kxa userid keyfile [keyring]

To wipe out the plaintext file after producing the ciphertext file, just add the -w (wipe) option when encrypting or signing a message :

pgp -sew message.txt her_userid

To specify that a plaintext file contains ASCII text, not binary, and should be converted to recipient's local text line conventions, add the -t (text) option to other options :

pgp -seat message.txt her_userid

To view the decrypted plaintext output on our screen (like the UNIX-style "more" command), without writing it to a file, use the -m (more) option while decrypting :

pgp -m ciphertextfile

To specify that the recipient's decrypted plaintext will be shown only on her screen and cannot be saved to disk, add the -m option :

pgp -steam message.txt her_userid

To recover the original plaintext filename while decrypting, add the -p option :

pgp -p ciphertextfile

To use a UNIX-style filter mode, reading from standard input and writing to standard output, add the -f option :

pgp -feast her_userid outputfile

Using PGP for simple encryption is similar to using crypt. To encrypt *file*, type :

pgp -c file

and follow the instructions, including typing in a pass phrase. The encrypted file will be named *file.pgp*. To decrypt *file.pgp*, use :

pgp file.pgp

7.9 CONCLUSIONS

This chapter discusses about UNIX security and encryption. It explains about the chmod command with live examples of changing file permissions. Also, commands such as umask, chattr, chown, chgrp are explained. In addition, a unit on encryption is included so as to make users to aware of their file security. Security through pgp command is explained in detail.