UNIVERSIDAD POLITECNICA DE VALENCIA

Local Protection in Linux Systems

Apellidos, nombre	Terrasa Barrena, Andrés (aterrasa@dsic.upv.es), Espinosa Minguet, Agustín (aespinos@dsic.upv.es)
Departamento	Dpto. de Sistemas Informáticos y Computación
Centro	E. T. S. de Ingeniería Informática

UNIVERSIDAD POLITECNICA DE VALENCIA

1. Key Concepts

This document introduces the fundamentals of **local protection** in Unix/Linux systems, which is based on a few basic abstractions and mechanisms:

- The system incorporates two abstractions in order to identify and authorize the people (or *users*) that can log in to use the system: user and group accounts.
- The system stores some protection attributes in any running process and in any system resource (which, in Unix, it always refers to a file).
- The system enforces some protection rules which control the actions each process is allowed to do over each file, depending on the actual protection attributes of the particular process and file involved.

The system administrator is the person in charge of configuring the system in such a way that the right people can use the system in the right way. In order to do so, the administrator needs not only to understand the abstractions and mechanisms above, but also to effectively use the specific system tools available for this configuration. In the case of Unix/Linux, these tools are normally system commands.

2. Objectives

After reading this article, students will be able to:

- Identify the main features of user and group accounts in Unix systems, including the files in which they are stored and the commands to manipulate them.
- List the protection attributes of processes and resources (files) in Unix systems and identify which of them participate in the local protection mechanisms.
- Understand the protection rules in Unix systems and relate them to the corresponding system (shell) commands; identify which of these rules are specific of RedHat-based systems.
- Use the right protection commands to configure a RedHat-based system according to some specific protection requirements.

3. Introduction

Local protection is a topic in System Administration, which is a knowledge area inside Computer Science. The aim of local protection is to configure a computer (or more specifically, the computer's operating system) in order to ensure that the computer is used by the right people and in the right way; or, more specifically, that: (1) only certain people (or users) can access the computer, and (2) when any user is working with the computer, this user cannot compromise, either accidentally or intentionally, the integrity

of any other user's work or the system itself.

In this context, the term "local" is used to denote that, despite the fact that the computer may be connected to other computers, the rules by which protection is enforced are local to that computer (i.e., they do not involve others). This is the most basic case of protection, and it is key to understand other types of protection rules that may be applied to many computers at once (which are outside the scope of this document).

In particular, this document introduces the local protection rules of Linux systems. Linux is an open-source operating system that belongs to a family of systems generally called Unix (please note that Unix also refers to a particular member of this family, but in this case, the registered mark UNIX® should be used instead). In this context, most of the concepts of this document not only apply to Linux but to the general Unix family, while some others are specific to Linux and, particularly, to some Linux distributions based on RedHat Linux (such as RedHat Enterprise, CentOS Linux, Fedora, etc.).

4. The User Table

The valid users of a Unix system are registered in the `/etc/passwd` file. Every line corresponds to a single user and describes the user's attributes. Within this line, attributes are separated by the ':' character. For example, given the following line of the `/etc/passwd` file, Table 1 below describes each of its elements:

```
user1:$1$ZtoHwEykKW/eCLHzSUigg5KAEy:1002:2000:User 1:/home/user1:/bin/bash
```

Element	Description
`user1`	User name (or *login name*)
`$1$ZtoHwEykKW/eCLHzSUigg5KAEy`	Encrypted password
`1002`	User IDentifier (UID)
`2000`	Group IDentifier. This is the identifier of the primary group of the user
`User 1`	Description of the user
`/home/user1`	User's home (private) directory
`/bin/bash`	User's shell (initial program)

Table 1. Elements in the User Table.

There are several special users which are built-in in the system, normally the users whose UIDs are below 500. Among these users, there is one called `root`, whose UID is 0. This user is the system administrator, usually known as the *superuser*. Other special users are used to associate some access level to certain system services, such as the users `mail` and `news`. Finally, another user that should be mentioned is the user `nobody`, which is normally used to represent network connections carried out by unknown or anonymous

users. All these user accounts are created during the installation of the system and should not be modified.

5. The Extension of the User Table

In addition of the user table described in the previous section, most of the current Unix systems use another table, contained in the `/etc/shadow` file, in which the system stores additional information of users.

Each line in this second file corresponds to one in the `/etc/passwd` file, and the information that it contains is now described, following the example above:

```
user1:$1$ZtoHwEykKW/eCLHzSUigg5KAEy:10989:0:99999:7:-1:-1:134538436
```

Element	Description
`user1`	User name (or *login name*)
`$1$ZtoHwEykK/eCLHzSUigg5KAEy`	Encrypted password
`10989:0:99999:7:-1:-1:134538436`	Information about the password age

Table 2. Extension of the user table

When the system uses this new table, passwords are not stored in the `/etc/passwd` file but in the `/etc/shadow` file. In this case, the password element in the `/etc/passwd` file is substituted by the character 'x'. Since `/etc/passwd` must be readable by every user in the system, whereas `/etc/shadow` is only readable by `root`, the use of *shadow passwords* improves system security, preventing users from accessing the encrypted passwords in the system.

The information about the password expiration (or age) in the `/etc/shadow` file is expressed in days, but its interpretation is difficult by reading it directly from the file. For this reason, it is normally recommended to modify and consult this information by using specific administrative tools, such as the **passwd** or **chage** commands, or some graphical applications.

6. The Group Table

In a Unix system, groups are registered in a file named `/etc/group`. Each line in this file corresponds to a single group and describes the group's attributes. In each line, attributes are separated by the '`:`' character. The following example describes each of these attributes:

```
proj1::65534:user1,user2,user3
```

Element	Description
`proj1`	Group name
	Encrypted password (empty in the example). The use of this password allows a user to change his/her primary group. However, this functionality is no longer in use in modern systems
`65534`	Group IDentifier (GID)
`user1,user2,user3`	Comma-separated list of users belonging to that group

Table 3. Elements in the Group Table

As it can be deduced from the example, a given user can be on the list of more than one group, that is, a user can belong to *several* groups. Among all these potential groups, the one which GID appears on the user's line in the `/etc/passwd` file is called the user's *primary group*. The rest of groups that the user may belong to are called the user's *supplementary groups*.

In the same way that it was explained for users above, there are certain groups with a special meaning for the system. These built-in groups have a GID below 500. Analogously to the special users, these groups represent services, anonymous users, etc. Therefore, they should not be modified.

7. Creating Users and Groups

In most Unix systems in the family called *System V* (including systems based on RedHat Linux), the following commands can be used to manage users and groups: **useradd**, **userdel** and **usermod** are used to create, delete and modify the characteristics of user accounts; **groupadd**, **groupdel**, and **groupmod** are used to create, delete and modify the characteristics of group accounts; **passwd** is used to change the password of a user account; and **chage** is used to manage the password age of a user account.

Note

Detailed information about these or any other commands mentioned in this document can be found by using the command **man** on a Linux terminal (e.g., **man useradd**).

Nowadays, the use of graphical tools that facilitate the administrative tasks (such as the User Manager by RedHat) is becoming very common in modern Unix systems. However, the use of commands is still considered by many administrators the native way of managing Unix systems, and it is especially useful when the user or group management is performed inside *shellscripts* (for example, when a large amount of users/groups has to be created automatically).

8. Protection Attributes of Processes

The attributes of a process that participate in the protection mechanism are the following:

1. **Identifiers of the process' owner user**. In fact, every process has two different versions of the user identifier, the so-called *real* version (or rUID) and the *effective* version (or eUID). The real version always corresponds to the user that created the process. The effective version corresponds to the user on behalf of which the process is executing, and it is the one used in the protection mechanism. In any given process, both identifiers are normally equal, except in the case in which the executable file that the process is running has the so-called `SETUID` bit set (as explained later in this document).
2. **Identifiers of the process' owner group**. In this case, the process also has a real version (rGID) and an effective version (eGID) of the group identifier. The real version always corresponds to the primary group of the user that created the process. The effective version corresponds to the group on behalf of which the process is executing, and it is the one used in the protection mechanism. Again, both identifiers are normally equal, except in the case in which the executable file that the process is running has the so-called `SETGID` bit set (as explained later in this document).
3. **List of supplementary groups**. This is the list of all the supplementary groups of the user that created the process.

These attributes are assigned to the process in its creation and they are directly inherited from its parent process. Every user logged in a Unix system has an attention process, which can normally be a graphical desktop or a simple command interpreter or shell. In any of both cases, this process is the parent of all the processes that the user may create. This initial desktop or shell process does not get its attributes by inheritance, but they are assigned by the system when the user logs in by using his/her login name and password. In particular, the rUID and rGID attributes are read from the user table (`/etc/passwd`) while the list of supplementary groups is built by reading the group table (`/etc/group`). The attributes eUID and eGID are initially assigned from their respective real versions. In this context, there is an interesting command named **id**, that displays all these attributes for the current user.

9. Protection Attributes of Files

The attributes of a file that participate in the protection mechanism of the system are the following:

1. `OwnerUID`. Identifier of the user that is the owner of the file.
2. `OwnerGID`. Identifier of the group that is the owner of the file.

3. **Permission bits**. A total of 12 bits that express the operations that are allowed on the file depending on the process that accesses the file. Table 4 below shows the meaning of every one of these bits.

Bit	Meaning
11	SETUID
10	SETGID
9	Sticky
8, 7, 6	Read, write and execute for the owner.
5, 4, 3	Read, write and execute for the group.
2, 1, 0	Read, write and execute for other (i.e., the rest of) users.

Table 4. Meaning of the permission bits in Unix

The meaning of the "read", "write" and "execute" bits is different depending on the type of the file in which they are set. For *regular* files, they have their evident meaning (they allow users to read, modify and run the file, respectively). Obviously, the execution bit only makes sense if the file is an executable binary or contains a *shellscript*.

In a *directory*, the meaning of these three bits is the following:

1. **Read**. Allows listing the contents of the directory.
2. **Write**. Allows the creation, deletion or renaming of any file or directory inside the directory in which this bit is set.
3. **Execute**. Allows using the name of the directory in which the bit is set to form a *path name*. That is, the directory can be used to name a file which path contains the directory.

From all the above, it can be derived that there is not an specific bit controlling the deletion of files/directories in Unix. This permission is actually controlled by means of the "write" bit in the *parent directory* of the file/directory that we want to delete. In some Unix systems (like, for example, RedHat-based Linux distributions), the `sticky` bit is used precisely to modify this rule controlling the deletion of files: if this bit is set in a directory, then a user can only delete a file inside it if the user is the owner of that file. The `SETUID` and `SETGID` bits are explained in detail later in this chapter.

The modification of the protection attributes of files can be performed by means of specific commands: the `ownerUID` can be changed by executing the command **chown**. The `ownerGID` can be changed by executing the command **chgrp**. The permission bits can be changed by running the command **chmod**.

10. The Basic Protection Rules

The basic protection rules are activated when a process notifies the system that it wants

to use a given file. The process also notifies which kind of operation(s) it wants to perform over the file: reading, writing or executing. According to this, the system always applies the following rules:

- If the `eUID` of the process is equal to 0, then the permission is granted (this is the case in which the process belongs to `root`. Otherwise...
- If the `eUID` of the process is equal to the `ownerUID` of the file, then the permission is granted if the operation is allowed in the group of bits 6 to 8 (those corresponding to the file owner). Otherwise...
- If the `eGID` of the process or any of the process' supplementary groups is equal to the `ownerGID` of the file, then the permission is granted if the operation is allowed in the group of bits 3 to 5 (those corresponding to the group). Otherwise...
- In any other case, the permission is granted if the operation is allowed in the group of bits 0 to 2 (those corresponding to the rest of users).

It must be noted that the system applies *one* rule only, precisely the one that first matches the process attributes. In other words, the system determines *first* which group of three bits has to apply and *then* grants (or denies) the operation depending on the operation type and the state of these three bits.

11. Change of the Protection Attributes of Files

Unix establishes some specific protection rules that control the modification of any of the protection attributes of a file, having that these modifications are considered different from writing operations on the file, and therefore cannot be granted/denied by checking the permission bits of the file.

In particular, the rules controlling the modification of such protection attributes are now defined:

1. **Permission bits change**. A process can modify the permission bits of a file only if:
   - The process' `eUID` is equal to `0` (i.e., the process belongs to `root`), or else
   - The process' `eUID` is equal to the `ownerUID` of the file.

   (That is, only root and the owner of a file can modify the file's permission bits.)
2. **Owner change**. Only the superuser can change the owner of a file.
3. **Group change**. A process can modify the owner group of a file only if:
   - The process' `eUID` is equal to `0` (i.e., the process belongs to `root`), or else
   - The process' `eUID` is equal to the `ownerUID` of the file (the user performing the change is the file's owner) *and* the new `ownerGID` corresponds to one of the user's groups.

12. The SETUID and SETGID Bits in Executable Files

These two bits are used to allow a program to run under the privileges of a user which is different from the user that is executing the program. They work like this:

- If the executable file has the `SETUID` bit set, the `eUID` of the process that runs the file is made equal to the file's `ownerUID`.
- If the executable file has the `SETGID` bit set, the `eGID` of the process that runs the file is made equal to the file's `ownerGID`.

Normally programs like these belong to the superuser, and they allow regular users to run privileged tasks under certain given conditions. The ability of a user to change his/her own password is an example of use of this technique.

The existence of these program files in the system must be carefully supervised by the superuser. Many of the security attacks to Unix systems use these files to try to break the system security.

13. The SETGID Bit in Directories

The utilization of the `SETGID` bit in directories is oriented towards facilitating the work of a group of users in a collection of common files and directories. The mechanism works like this: if a directory `D` has the `SETGID` bit set, then:

- if a file is created inside `D`, the `ownerGID` of the file is made equal to the `ownerGID` of `D`.
- if a directory is created inside `D`, the `ownerGID` of the new directory is made equal to the `ownerGID` of `D`, and the new directory has its `SETGID` bit set.

It can be seen that, in essence, it is an inheritance mechanism. When the `SETGID` bit is not set in a directory, any file/directory created inside it gets its `ownerGID` from the `eGID` of the process that creates it. However, when the `SETGID` bit is set, all newly created files and directories will have a common `ownerGID`, even if they are created by different users. By combining this fact with an appropriate set of initial permissions, then these users will be able to read and modify all such files by default.

14. The User Mask

The tools in Unix that create files use by default the protection word `rw-rw-rw-` when creating regular (non-executable) files, and `rwxrwxrwx` when creating directories or executable files. It can be seen that in each case, all the possible permissions (with some meaning) are set by default.

In order to modify this behavior, every user can specify which bits should not be set when creating new files and directories, by means of the *user mask* (or file creation

mask). Any user can set his/her own mask with the command **umask**. Every bit which is set in the user mask is a bit that will be *unset* in new files or directories created by the user (e.g., a mask equal to 022 would unset the "write" bits for the group and others).

The administrator is responsible of assigning a reasonable default user mask to any user. This can be easily achieved by using the file `/etc/profile`, which customizes the initial environment of users when they log in, independently of the shell they use. However, in RedHat-based systems, which use **bash** as the initial shell by default, the initial user mask of users has to be set in the `/etc/bashrc` file.

15. The Private Group Strategy

This strategy is used by default in RedHat-based Linux distributions and it is oriented towards making the assignment of users to groups more flexible and rational. The keys of this strategy are the following:

- Creating a private group for each user (this group is named exactly as the user) and making this group to be the user's primary group.
- Grouping users by using supplementary groups only.
- Combining the use of the `SETGID` bit and a supplementary group in those directories where some given users have to share files/directories.
- Using each user's private group in his/her home directory.
- Assigning `00X` as the default user mask for every regular user in the system; that is, all of the user and group bits set, and whatever the superuser thinks appropriate for the rest of users.

16. Concluding Remarks

This document has presented the fundamentals of local protection in Linux. After reading it, you should be capable of creating and maintaining users and groups, and to effectively configure the permissions of the system in order to create a safe and productive environment on which several users can work, either in common or on their own.

Among the tools which are available to the system administrator to configure the system protection, this document has introduced a series of system commands. The use of system commands is considered the native way of managing Linux systems, and it is very useful when the administrator is performing configurations over great amounts of data (users, groups, files), because commands can be easily embedded in *shellscripts*.

17. Bibliography

[1] Red Hat Enterprise Linux, System Administration Guide (Red Hat, Inc.). Available in: `http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/index.html`