

Practical Exercise:  
Execution of actions with different privileges or in a  
confined environment

September 20, 2017

Due date: no date

## Changelog

- v1.0 - Initial Version.

## 1 Introduction

The goal of these laboratory guide is to learn two ways of changing the normal protection mechanisms of a Linux operating system. The first one is the Set-UID and Set-GID mechanisms, designed to overcome the natural protection policies that are based on the UID and GIDs of a process. The second is a way to confine the working environment (namely, the file system view) when executing untrusted and possibly dangerous applications.

These exercises must be executed in Linux systems. It is recommended the usage of the provided Virtual Machine.

## 2 Privilege modification: Set-UID and Set-GID flags

In Linux, the access privileges of a process are defined by its UID and GIDs, which are inherited from the father process. After a login operation, the user's initial process in the session, usually a command interpreter `bash` (or a graphical display manager), gets the user's UID and GIDs, extracted from the `/etc/passwd` and `/etc/group` files. Thereafter, the commands executed from that command interpreter (or launched from the graphical display manager), that usually run in independent processes, also use the same UID and GIDs.

There are, however, some activities that create a contradictory scenario. For instance, when a user named Alice wants to change her login password, she should be able to do so whenever needed. But users' passwords are all stored in a transformed way inside a file (`/etc/shadow/passwd`), and ordinary users (i.e., non-administrators) should not be allowed to freely change the contents of this file, because they could tamper the passwords of other users; they should be allowed to change only their password, but such restriction cannot be expressed by the ordinary file access control rights, which has a file granularity.

Thus, what is required is to provide users some special applications (or executable files) that could correctly perform operations that have security implications, such as correctly changing Alice's own password, but upon request of ordinary users, such as Alice. This is achievable with the Set-UID and Set-GID mechanisms.

The protection of Linux files includes, besides the **rw**x rights for owner, a group and others, three other access control bit flags:

- One to perform a Set-UID operation;
- Other to perform a Set-GID operation; and
- Another historically used as a hint to increase the permanence of the file contents in main memory (named the sticky bit).

Flags			Owner rights			Group rights			Others' rights		
Set-UID	Set-GID	Sticky	R	W	X	R	W	X	R	W	X

When listing a file with any of these flags set, they are observable in the execution right, which is replaced by an **S** or a **T** instead of an **x**.

Execute the following command to locate and observe the protections of all the files in your Linux with the Set-UID bit set:

```
find / -perm -u=s |xargs ls -la
```

Execute the following command to locate and observe the protections of all the files in your Linux with the Set-GID bit set:

```
find / -perm -g=s |xargs ls -la
```

The Set-UID and Set-GID flags, when active in executable files, allow all users to run those executable with the identity (UID or GID) of the executable's owner UID or group GID. Their goal is to allow users to run specific commands (executable files) with temporarily different, possible elevated privileges (given only to certain users/groups) in order to perform specific tasks requiring special privileges.


These identity changing flags are needed for tasks that require different (not necessarily higher) privileges than those which a common user has, such as changing the login password, mounting/unmount a file system, starting a session with another identity, sending a file into the printer spooler, executing a ping over the network, etc.

## 2.1 Real and effective UID / GID

Each process has a real and an effective UID. Usually they are the same. The effective UID is the UID that is *effectively* used by the operating system to establish the process' privileges (e.g., to determine rights when accessing files).

The Set-UID flag changes the effective UID of a process, but not its real UID. The Set-GID flag does a similar action, changes the effective GIDs of the process to an effective GID equal to the GID of the owner group of the executable. Applications can revert their effective UID/GID to their original value before the Set-UID/GID operations, which are the values stored in the process' real UID and GIDs.


## 2.2 Set-UID / Set-GID flags in practice


Create a C program that prints the effective and real UID of the current process, as well as its effective and real GIDs. After its compilation, run the program and check the result. 


Hereafter, some of the commands have to be executed by a super-user. Use a preceding **sudo** command in those cases.



Change the ownership of the executable just created (UID of the owner) with the command **chown**; change also the owner group GID of the executable (with the **chgrp** command). Use the UID of any other user and the GID of any other group. Run the program again and check the results.

Activate the Set-UID flag of the executable with the command **chmod**. Run the program again and check the results. 

Activate the Set-GID flag of the executable with the command **chmod**. Run the program again and check the results. 

Change again the ownership of the executable, or its owner group GID. Run the program again and check the results. 

### 3 Confinement: command chroot

Information stored in file systems is usually protected from unwanted accesses using ownership/identity attributes and corresponding access rights. However, these protections may be circumvented by attackers that are able to get high privileges (e.g. root identification, in Linux) or that are able to impersonate other users/groups that have access to the target files.

A solution to overcome this problem is to reduce the file system view for untrusted applications. This way, it is possible to completely confine all file system accesses performed by those applications to a minimum set of files/directories required for their correct execution.

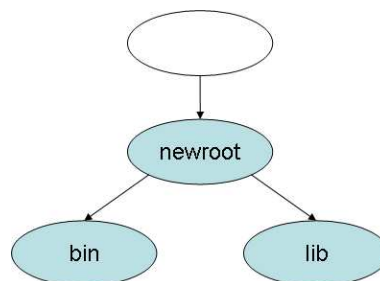
This is the goal of the **chroot** Linux command; check how it works with the command

```
man 8 chroot
```

#### 3.1 Setup of a reduced execution environment

In this experiment we will create a confined environment where only 4 commands exist: **bash**, **ls**, **mkdir**, and **vi**. In this environment it will be possible to list existing files with the **ls** command, to create directories with the **mkdir** command, and to create new files, or to change existing files, with the **vi** command. It will not be possible, for instance, to eliminate files/directories (though it may be possible to eliminate files' contents...) or to change their location.

First, choose a test directory, which will be the new root directory of the confined environment (**newroot** hereafter) and create below it a minimum hierarchy, formed by subdirectories **bin** and **lib**. The first directory will contain the commands above referred; the second directory will contain the dynamic libraries used by those commands.



#### 3.2 Command location and installation

Linux commands do not have a hardwired location, they may exist in any directory. There are, however, well-known locations (directories) for the usual commands, such as the ones we want to transfer to the

confined environment.

Command interpreters, such as **bash**, look for commands using an environment variable (**PATH**). The value of this variable should be a list of directory paths (full path names), separated by colons (:). This value of this variable is changed with the **set** command (internal to **bash**) and its value is obtained with the **\$PATH** expression.

To know where a command is stored, in a file system, one can use the command **which**; for more details execute the command

```
man which
```

This command uses the **PATH** value to locate commands. Use it to locate the commands we intend to include in the confined execution environment and copy them to its **bin** directory.

### 3.3 Dynamic libraries location and installation

The dynamic libraries required by a command can be observed with the command **ldd**; for more details execute the command

```
man ldd
```

.



The output of this command is twofold:

- The name of the dynamic library required by the command;
- The complete path of a dynamic library with a compatible name.

Execute this command with each of the commands copied in the previous section and copy each of the identified dynamic libraries to the **lib** directory of the confined environment. Note: ignore the **linux-gate.so.1** dynamic library.

### 3.4 Command execution within the confined environment

Execute the command

```
chroot newroot command
```

where *newroot* should be the path of the root directory of the confined environment and *command* should be a command existing the environment (e.g. **bash**). Note: **chroot** can only be executed by a super-user; use **sudo** to get a super-user identity.

Check that you can only use the commands installed in the confined environment (besides the **bash** internal commands). Check also the full contents of the confined file system with the command

```
ls -lR /
```

Leave the confined environment (by terminating the command interpreter). Execute again the **chroot** command but now preceding it with **strace**:

```
strace chroot newroot command
```

The command **strace** dumps all the system calls executed by another command (**chroot**, in this case). Check the call to the **chroot** system call before actually looking for **bash** for its posterior execution.

### 3.5 Escaping a chroot jail

It is possible to jump outside an environment confined with **chroot**. This is known as “breaking a **chroot** jail”. But breaking a **chroot** jail requires super-user privileges, to run **chroot** inside the jail. Therefore, **chroot** jails are not secure enough when programs run inside with super-user privileges and have the ability to produce their own code.

The basic steps in order to break a jail are the following:

1. Create some temporary directory
2. Open this directory and save the handle
3. Call the **chroot** operation on the temporary directory, efectively changing the current root to this directory.
4. Call **fchdir** over the temporary directory handle
5. Call **chdir("..")** a few times to get out reach the real root.
6. Change the root of the program to the real root (using **chroot**).

There are numerous examples, written in C, of programs that are able to escape a **chroot** jail. Get one in the Internet, understand how it works and try it.

One popular example is present at <http://www.bpfh.net/simes/computing/chroot-break.html>.