Introduction to computer security
Lab : access rights in UNIX systems

Giuseppe Lipari

January 21, 2022

Contents

1 Work environment **1**
1.1 Instructions for writing the report 1

2 Managing Unix access rights **1**
2.1 User ID and Group IDs 1
2.2 File permissions 2
2.2.1 Example 2
2.3 Process identifiers 3
Question 1 3
Question 2 3
2.4 Change privilege 3
2.4.1 Example 4
2.4.2 Questions 5
Question 3 5
Question 4 5
Question 5 5
Question 6 6
2.5 Rules for `access` 6
2.6 Special rules for directories 6
2.6.1 Setgid for directories 6
2.6.2 Sticky bit 7

3 Exercises **7**
3.1 Shared-files server 7
Question 7 8
Question 8 8
Question 9 8
3.2 Client/Server 9
Question 10 9

1 Work environment

This work is intended to be performed on a virtual machine. The University of Lille provides an openstack at `https://cloud.univ-lille.fr/` to create and manage virtual machines using Linux.

Warning: you need to be connected through the vpn to access openstack from outside the campus.

- Go to `https://cloud.univ-lille.fr/` and connect using your login/password of the university ;
- Find your name at the top right, click and select "help/aide" in order to download a pdf which contains the instructions to launch a virtual machine.
  - Security keys are useful to connect to ssh without typing password at each connection.
  - I recommand you to install, at least, a C compiler, make and cmake

    ```
    sudo apt install g++ make cmake
    ```

1.1 Instructions for writing the report

Fork this repository and **add your lab teacher as a developer member**.

Answer questions 1-10 inside `rendu.md`. The code goes to its corresponding `questionX`.

For C programs, remember to give a `Makefile` for compiling and testing your code. We remind you that the repository should only contain sources (`.c` et `.h`), makefiles (`makefile`), bash or python scripts. It shouldn't contain object files (`.o`) or executable files.

Don't forget to `git push` !

2 Managing Unix access rights

2.1 User ID and Group IDs

In UNIX systems, each user corresponds to a unique User ID and one or many Group IDs. You can visualize them using the commande `id` :

```
ubuntu@isi2:~$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),20(dialout),24(cdrom), ...
```

The `root` user can create new users using the command `adduser`, and new groups using `addgroup`. One can modify the parameters of existing users using `usermod`.

I encourage you to read man pages for theses commands (`man adduser`).

If a user belongs to several groups, the first one in the list is considered as its main group, and other ones are considered as *supplementary* groups.

2.2 File permissions

A file has :

- A User ID (id of the owner)
- A Groupe ID (id of the groupe of the file)
- Permission rights for reading, writing and execution for the owner, for the groupe and for the others.

To see theses informations concerning a file, you can use the command `ls -al`.

2.2.1 Example

In the home directory of the user `ubuntu`, we find the following files:

```
ubuntu@isi2:~$ ls -al
total 40
drwxr-xr-x 4 ubuntu ubuntu 4096 Jan  4 11:34 .
drwxr-xr-x 3 root   root   4096 Jan  4 11:05 ..
-rw------- 1 ubuntu ubuntu    8 Jan  4 11:08 .bash_history
-rw-r--r-- 1 ubuntu ubuntu  220 Feb 25  2020 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu 3771 Feb 25  2020 .bashrc
drwx------ 2 ubuntu ubuntu 4096 Jan  4 11:08 .cache
-rw-r--r-- 1 ubuntu ubuntu  807 Feb 25  2020 .profile
drwx------ 2 ubuntu ubuntu 4096 Jan  4 11:05 .ssh
-rw-r--r-- 1 ubuntu ubuntu    0 Jan  4 11:34 .sudo_as_admin_successful
-rw------- 1 ubuntu ubuntu  708 Jan  4 11:33 .viminfo
-rw-rw-r-- 1 ubuntu ubuntu   35 Jan  4 11:33 file.txt
```

The second column contains the id of the owner of the file ; the third column contains the id of the groupe ; the first one contains the access permissions.

The file `.viminfo` has the following permissions :

```
-rw-------
```

The first character is the type of the file (`d` for directory, `-` for a regular file). The three following characters are the reading (`r`), writing (`w`) and execution (`x`) permissions for the owner `ubuntu`. So, the file is readable and writable by the user `ubuntu`. The three next characters represent the same permissions for the users who belongs to the groupe `ubuntu` ; and the last three characters represent the permissions for all the other users. Thus, the file `.viminfo` is only accessible by the user `ubuntu`.

The file `file.txt` is readable by every one ; writable by the user `ubuntu` qnd the users who belong to the group `ubuntu` ; and not executable by anyone.

To change the access rights of a file, you can use the command `chmod`. To change the owner and the groupe of a file you can use the command `chown` (read `man chmod` and `man chown` for the usage).

2.3 Process identifiers

A processus is a running program, whose code is located in an executable file.

Several identifiers are assigned to a process. The *Real UID* (RUID) is the id of the user who has launched the process ; the *Real GID* (RGID) is the id of the main group to which belongs the user who has launched the process. The *Effective UID* (EUID) and the *Effective GID* (EGID) are usually equals to the RUID and RGID, and are used to verify the access rights of the process. EUID and UGID can be modified to change the privilege level of the process.

To obtained the value of this identifiers, the process can use the syscalls `getuid()` and `getgid()` for the real ids, and `geteuid()` and `getegid()` for the effective ids. To obtained the list of supplementary groups, it can use `getgroups()`.

To verify file access rights, the system use the following procedure:

- First, it compares the process EUID with the owner of the file ; if they match, it uses the first triplet of permissions to verify the access rights ;

- If they don't match, it compares the list of group of the process (EGID and supplementary groups) with the group of the file ; if one in the list matches, it use the second triplet.
- If they don't match, it use the third triplet.

Question 1. For this exercice and the following, create a new user `toto` in the system, and add it to the group `ubuntu`.

If a process, that has been launched by the user `toto`, try to open the following `myfile.txt` with writing rights:

```
-r--rw-r-- 1 toto ubuntu    5 Jan  5 09:35 myfile.txt
```

- Tell if the process can write, and why?

□

Question 2. The `x` character indicates that the file is executable.

- What does the `x` character indicate for a directory?
  With the user `ubuntu` create the directory `mydir`, and remove the execution rights to the group `ubuntu`. Now, with the user `toto`, try to enter the directory using `cd mydir`.
- What's happened? Why?
  With the user `ubuntu`, create a file `data.txt` in the directory `mydir`. Now, using the user `toto` try to list the contain of the directory using `ls -al mydir`.
- What's happened? Why?

□

2.4 Change privilege

For an executable file, it can be useful to define its `set-user-id` flag to grants to processes higher privileges.

When this flag is defined, the program is launch with its EUID equals to the id of the owner of the executable file. The flag `set-group-id` also exists and works similarly with the EGID. To change the value of this flag, you can use the command `chmod`.

2.4.1 Example

Regarding the following situation:

```
toto@isi2:~/exec$ ls -al
total 36
drwxrwxr-x 2 ubuntu ubuntu  4096 Jan  7 11:06 .
drwxr-xr-x 7 ubuntu ubuntu  4096 Jan  7 11:03 ..
-rw--w---- 1 ubuntu ubuntu     0 Jan  7 10:42 file_a.txt
-rw-rw-r-- 1 ubuntu ubuntu     0 Jan  7 11:06 file_b.txt
-rwxrwxr-x 1 ubuntu ubuntu 16872 Jan  7 11:05 myopen
-rw-rw-r-- 1 ubuntu ubuntu   361 Jan  7 11:04 myopen.c
```

The code of the program `myopen.c` :

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *f;

  if (argc < 2) {
    printf("Missing argument\n");
    exit(EXIT_FAILURE);
  }
  printf("Hello world\n");
  f = fopen(argv[1], "r");
  if (f == NULL) {
    perror("Cannot open file");
    exit(EXIT_FAILURE);
  }
  printf("File opens correctly\n");
  fclose(f);
  exit(EXIT_SUCCESS);
}
```

The program `myopn` takes as argument the name of a file and tries to open it with reading permission. If the user `toto` launch the command:

```
toto@isi2:~/exec$ ./myopen file_a.txt
Hello world
Cannot open file: Permission denied
```

The program cannot open the file with read access because `file_a` is readable only by the user `ubnutu`.

The user `ubuntu` can define the set-user-id for the file `myopen` that way:

```
ubuntu@isi2:~/exec$ chmod u+s myopen
ubuntu@isi2:~/exec$ ls -al myopen
-rwsrwxr-x 1 ubuntu ubuntu 16872 Jan  7 11:05 myopen
```

You should note the character `s` in the first triplet that indicates the execution permission with set-user-id.

Now, if `toto` launches the same command:

```
toto@isi2:/home/ubuntu/exec$ ./myopen file_a.txt
Hello world
File opens correctly
```

2.4.2 Questions

Question 3. Write a program in C which print the value of its ids (EUID, EGID, RUID, RGID) and the content of the file `my/mydata.txt` (the one created at the question 2). The executable file should belong to the user `ubuntu` and the group `ubuntu`. Launch the program with the user `toto`.

- What are the values of the ids? Can the process open the file `mydir/mydata.txt` with reading permission?

Now, set the `set-user-id` flag of the executable file, and re-launch the program.

- What are the values of the ids? Can the process open the file `mydir/mydata.txt` with reading permission?

□

Question 4. Write a python script that prints the values of the EUID and the EGID. The script should belong to the user `ubuntu`. Set the `set-user-id` flag and launch the script with the user `toto`.

- What are the values of the ids?

□

The *Saved set-user-id* and *saved set-group-id* are used to save the effective ids before changing them, in order to reuse them later.

To obtain the value of all the ids, you can use the syscalls:

```
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
```

and to modify them:

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

A non-privileged process can't set an arbitrary value to the effective ids.

What is the usefulness of the flag `set-user-id`? This flag allows to a user to launch a program with higher privilege than his. This functionality can be used to access files that the user hasn't the right to access.

For example, regarding the file `/etc/passwd` :

```
-rw-r--r-- 1 root root 1802 Jan  4 18:04 /etc/passwd
```

It contains the parameters of all the user account of the system, and it is readable by everyone. However, only the administrator (`root`) has the right to modify it.

How a user can change one of its attributes without asking to the administrator?

Question 5. Visualize the content of the file `/etc/passwd`.

- What's the purpose of the command `chfn`? Give the result of `ls -al /usr/bin/chfn`, and explain the permissions.
- Launch the command `chfn` as the `root` user, answer the questions. Visualize again the content of the file `/etc/passwd` and verify that the information has well been updated.

□

Question 6. You can observe that the file `/etc/passwd` doesn't contain any password.

- Where are stored the users' passwords? Why?

□

2.5 Rules for `access`

The command `access` verify the access rights to file, but it uses the RUID and the RGID instead of the EUID and the EGID.

From `man access`:

> The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.
>
> This allows set-user-ID programs and capability-endowed programs to easily determine the invoking user's authority. In other words, access() does not answer the "can I read/write/execute this file?" question. It answers a slightly different question: "(assuming I'm a setuid binary) can the user who invoked me read/write/execute this file?", which gives set-user-ID programs the possibility to prevent malicious users from causing them to read files which users shouldn't be able to read.

2.6 Special rules for directories

2.6.1 Setgid for directories

If the `setgid` flag is used on a directory, the files that will be created in this directory will have as group ID the one of the directory.

For example, if the directory mydir has the following permissions:

```
drwxrwxr-x 2 ubuntu ubuntu 4096 Jan  4 18:02 mydir
```

If the user `toto` is in the group `ubuntu`.

If toto creates a file in mydir, this file will have as permissions:

```
toto@isi2:/home/ubuntu$ touch mydir/tata.txt
toto@isi2:/home/ubuntu$ ls -al mydir/tata.txt
-rw-rw-r-- 1 toto toto 0 Jan  4 18:06 mydir/tata.txt
```

Now, if we add the setgid to directory `mydir`:

```
ubuntu@isi2:/home/ubuntu$ chmod g+s mydir/
ubuntu@isi2:~$ ls -al mydir/
...
drwxrwsr-x 2 ubuntu ubuntu 4096 Jan  4 18:06 mydir
```

Instead of the character `x` we see a `s` for the triplet corresponding to the group.

If `toto` creates q file in this directory:

```
toto@isi2:/home/ubuntu$ touch mydir/titi.txt
toto@isi2:/home/ubuntu$ ls -al mydir/
total 8
drwxrwsr-x 2 ubuntu ubuntu 4096 Jan  4 18:11 .
drwxr-xr-x 5 ubuntu ubuntu 4096 Jan  4 18:02 ..
-rw-rw-r-- 1 toto   toto      0 Jan  4 18:07 tata.txt
-rw-rw-r-- 1 toto   ubuntu    0 Jan  4 18:11 titi.txt
```

The new file belongs to the group `ubuntu`.

2.6.2 Sticky bit

The *sticky-bit* is a flag that can be assigned to a directory. If the flag is activated for a directory `mydir`, a file in the tree which have `mydir` as root can be renamed or deleted only by the owner of the directory or the owner of the file.

To assign the sticky bit, we use the command `chmod` using the character `t`:

```
ubuntu@isi2:/home/ubuntu$ chmod +t mydir
ubuntu@isi2:~$ ls -l
total 12
drwxrwSr-t 2 ubuntu ubuntu 4096 Jan  5 09:36 mydir
```

3 Exercises

3.1 Shared-files server

We would like to set up a server where users can share files, with restrictions according to the group they belong to.

We have two groups of users, the `groupe_a` and the `groupe_b`. There is a special user named *administrator*: `admin`.

Each group has a directory shared between all the group members, but not accessible to the members of the other group: a directory `dir_a` and a directory `dir_b`. There is a directory `dir_c` which is shared between the users of the two groups.

The members of the `groupe_a`:

- can read all the files and all the sub-directories contained in `dir_a` and `dir_c` ;
- can read, but can't modify files in `dir_c`, rename them, delete them, or create new files ;
- can modify all the files contained in `dir_a` and in sub-directories, and can create new files and directories in `dir_a` ;
- can't delete or rename files in `dir_a` they don't own ;
- can't read, modify or delete files in `dir_b`, and can't create new files in `dir_b`.

Symmetric rules are applied for `groupe_b` members.

The same rules apply for the `admin` user except this one:

- the `admin` can delete, modify, create or rename files in `dir_a`, `dir_b` and `dir_c`.

Other users which don't belong to `groupe_a` or `groupe_b`, can't access to `dir_a`, `dir_b` or `dir_c`.

Question 7. Setup the architecture described above. You need at least to create:

- the users `lambda_a` which belongs to `groupe_a` and `lambda_b` which belongs to `groupe_b` ;
- the user `admin` ;
- the directories `dir_a`, `dir_b` and `dir_c`, and files with the correct permissions.

It is allowed (needed?) to create other groups and users.

Validate accessibility rules with bash scripts, a script for lambda users of each group, and a script for `admin` user. □

Now, we would like to give `groupe_a` (or `groupe_b`) users the possibility to delete file in `dir_a` (or `dir_b`) directory if they belongs to the `groupe_a` (or `groupe_b`, respectively). To do this, we are going to setup an password-based authentication mechanism.

Question 8. Write a program `rmg` which takes as argument a file name to delete. The program should first ask the user for its password. Each user has its own password (different from the one used to login in the system). Theses passwords will be kept in a file `passwd`, which should be readable and writable only by `admin`. The is located in `/home/admin/passwd`.

The program `rmg`:

- should be executable for all system user ;
- verifies, before asking for the password, that the user belongs to the same group as the file (`groupe_a` or `groupe_b`, respectively), otherwise, it print an error message ;
- should have the minimal required privileged to erase a file in `dir_a` or `dir_b`.

Write the program. Validate it using a bash script.

Remarque: It is recommended to separate the implementation in several modules. We recommend you to put the declaration of function that verify the password in a file `check_pass.h`, and their implementation in `check_pass.c`. □

Question 9. We would like now to avoid that `admin` user knows the password of every other users. To do this, we'll allow users to setup their own password using a program `pwg`.

The program will be usable by every users from `groupe_a` or `groupe_b`. If a user has already a password in the file `passwd`, the program will ask for the old password before modifying it. If the check is correct, or if the user hasn't got a password yet, it asks for the password and saves it in the file `passwd` in a ciphered way.

To cipher the password, use the function `crypt()` available in the libc [1]. More information here:
`https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_32.html`

- Write the program `pwg` ;
- Modify the program `rmg` to take into account this modification (note that if the implementation has been split in modules, you just need to modify files `check_pass.h` et `check_pass.c`) ;
- Validate the two programs with bash scripts.

□

3.2 Client/Server

We give the possibility to the `groupe_a` and `groupe_b` users, and `admin` to access remotely to the files.

A program `group_server` listen to the port `4000`, waiting for client connection. The program executes with the `root` privileges.

When a client connects, it sends its username and its password. The server verifies the couple user/password in the file `/home/admin/passwd`, and if correct; it `fork()` to launch a child process that will continue the interaction with the client. The father process wait again for a new connection.

The child process, before interacting with the client, *embody* the user that just connect by modifying its EUID and EGID.

The client can ask for:

- listing the content of a directory among `dir_a`, `dir_b` et `dir_c` with the command:

[1] Even if it's not considered as a safe algorithm, we'll use it for its simplicity.

`list <dirname>`

the server answers with the output of the command `ls <dirname>` (the list, or an error message).

- reading the content of a file contained in one of the directories, using the command:

  `read <filename>`

  where `filename` is the relative pathto the directory `/home/admin` (for example, the command `read dir_a/file_a1.txt` reads the file `/home/admin/dir_a/file_a1.txt`). The server answers with the output of the command `cat <filename>` (the content of the file, or an error message).
- terminate the connection with the command

  `close`

To simplify the exercise, we consider that all files contain only ASCII text.

Question 10.

1. Write the server program `group_server`.
2. Write the client program `group_client <command_file>` which takes as argument the name of a file that contains:
   - the username and the password, separated with a space, on the first line ;
   - on each next line, a command to send to the server.

   The client, after the connection to the server, send the commands contained in `command_file` one by one, wait for the answer, and print the results on the standard output.
3. Test the couple client/server by launching the client different input files. Yo have to test that:
   - The server doesn't crash ;
   - All permissions are correctly setup: `lambda_a` userdoesn't have the right to read a file in `dir_b`, etc...

□