SIO Lab
Linux Security
version 1.0

Authors: João Paulo Barraca, Hélder Gomes, André Zúquete, Pedro Escaleira

Version log:

- 1.0: Initial version

1 Introduction

This guide aims to study some of the security mechanisms of the Linux Operating System. In particular the methods for elevation or restriction of privileges. It should be executed in the Linux OS, preferably in the Virtual Machine used in this course.

2 System Login

To access a system it is required to start the process of login. You can repeat this process by using the key combination `CTRL+ALT+F[1-12]`[1], or executing the command `login` with the super user. The first opens a new virtual console, independent from the session where you logged in (which, for the X Window System, traditionally is at the seventh virtual console). If you open a new virtual console, you can go back to your graphical session by hitting `CTRL+ALT+F7` (the seventh console).

As the result of the login process of a specific user, a session is started using the command interpreter (*shell*) associated with the user account.

2.1 Identity

It is possible to obtain information about the identity of the current user, as well as the groups to which it belongs. One of the possible commands is the following:

```
$ id
```

It is also possible to obtain lists of available users and groups, through the `getent` command. The following command lists the possible entries to be enumerated:

```
$ man getent
```

While the following command obtains all users:

```
$ getent passwd
```

The output of this command is obtained from several sources, depending on the system configuration. In this case, for users and groups, it should consist of `/etc/passwd` and `/etc/group`. However if, for instance, you configured your system to get users from an LDAP server, this command would also present those users.

[1]If you are using Oracle VirtualBox, use `right-CTRL+F[1-12]` instead.

2.2 User `root`

The `root` is the Linux *super user*. With the exception of Mandatory Access Control mechanisms, everything is always allowed. It is omnipotent and has the ID with value of 0.

2.3 Super User Do (`sudo`) command

The syntax `sudo <command>` allows executing the command specified as argument, using the identity of the `root` user. In order for this action to be authorized, the user that executes this command needs to provide his credentials and be on the sudoers file (i.e., the `/etc/sudoers` file).

If you execute the following command, you will validate the user that is used for executing the `id` command:

```
$ sudo id
```

If you execute the command repeatedly, you will not require to introduce the credentials. Moreover, only a restricted set of users may effectively be authorized. The authorization is controlled by the content of the file (`/etc/sudoers`), which lists the users or groups that are allowed to elevate their privileges through this method.

Check the content using:

```
$ sudo cat /etc/sudoers
```

This method is consistent with the general execution of actions, split in at least two parts: **authentication** (validate the credentials) and **authorization** (verification of the rules in the `/etc/sudoers` file).

3 Processes' privileges

Generally speaking, in Linux, processes can be distinguished into two categories, depending on their privileges: privileged and unprivileged. While privileged processes bypass all Kernel permission checks, unprivileged processes depend on the permissions of their effective user or user's group. Usually, if we want to execute a privileged process, we run it as the root user. However, since a machine user is not always an administrator, Linux has features that allow a non-privileged user to launch processes with high privileges without logging in as root.

3.1 Set-UID mechanism

Some commands should always be executed by a specific user, in particular the super-user. Examples of such commands are the `passwd` or `chfn` commands. All manipulate files restricted to the super-user, or invoke actions limited to the super-user. Without a way of elevating the rights for specific actions it would be impossible to execute trivial and essential tasks such as changing the user password.

The `sudo` program is also an example of a program that always requires the elevation of the privileges, so that one user can execute commands are another user, or as the super-user.

Therefore, this is a dilemma: users should be restricted to their privileges, but should also be able to execute action as other users.

The way of dealing with this is the Set-UID mechanism: a bit in the binary file storing a program, specifies that the process should be **always** started using an alternative user, instead of the user that is executing the process.

Analyze the permission of the `passwd` and `sudo`:

```
$ ls -l /usr/bin/passwd
$ ls -l /usr/bin/sudo
```

Then compare the permissions of these files with the ones of the `ls` command (which is not privileged):

```
$ ls -l /bin/ls
```

To see the differences of setting this flag, copy the `/usr/bin/id` file to the home directory of your unprivileged user, execute it, and register the output:

```
$ cd
$ sudo cp /usr/bin/id .
$ ./id
```

Now, set the Set-UID bit of the local file, and execute it again. Compare the results obtained:

```
$ sudo chmod u+s ./id
$ ./id
```

You should notice the addition of a field named `euid`, the short of *Effective User ID*, and you should be able to see that the `UID` is different from the `EUID`. In the case of the Linux OS, the `EUID` is the value that effectively sets the user, and by consequence the permissions of the process.

3.2 Capabilities

Although the Set-UID mechanism is a handy way for a non-privileged user to run privileged processes, it does not comply with the least privilege principle: it gives full privileges to processes when they only need to access specific privileged-related features. Therefore, since Linux 2.2, Linux implements capabilities: specific attributes that a process can access. Consequently, if, for instance, a process only needs access to a specific privileged feature, it does not need to use the Set-UID mechanism, only to have the related capability attributed. In other words, the effective user ID of such a process does not need to be of the root user. However, it accesses a specific privileged feature.

For example, `ping` was a process that previously used the Set-UID mechanism active. This was because it needed to create raw sockets, a privileged action. However, with the introduction of capabilities, `ping` no longer uses the Set-UID mechanism but rather the CAP_NET_RAW capability, which gives it access to creating and using raw sockets. You can check that with the following commands:

```
# Check that ping does not have the Set-UID bit active
$ ls -l /usr/bin/ping

# Get the capabilities of ping
$ getcap /usr/bin/ping
```

4 File Protection Mechanisms

The files and directories in the filesystem have a basic protection which indicates if they can be read, modified, or executed (`rwx`) by their owner, their primary group or other users.

The `ls -l` command allows to inspect the permissions set to any file in the filesystem.

You can also create a directory and file, whose permissions can be inspected:

```
$ mkdir test_dir
$ cd test_dir
$ date > test_file
```

Where `test_dir` should be the name of the directory to create, and `test_file` the name of the file to create. In this case the file is created using the current date as its content.

4.1 Read protection

Check the permissions applied to the file test_file with the following command:

```
$ ls -l test_file
```

Check who is the file owner, the main group and the protections applied to this specific entities, as well as for all other users.

You can show the content of the file using the cat command:

```
$ cat test_file
```

Then, remove the permission for the owner to read the file:

```
$ chmod u-r test_file
```

Try to show the content of the file, which should not be possible. You can use the syntax u+r to enable the owner to read the file.

Correct the permissions and check if you can access the file content (you should be able to do it).

You can manipulate the remaining permissions in order to protect the content of the file, and limit the actions as adequate.

The permissions of the test_dir can also be manipulated. For this purpose, consider the operations of creating a new file, changing the content of an existing file, reading a file and listing the contents.

Remove the write permissions for the directory:

```
$ chmod u-w test_dir
```

And check the impact for the before mentioned actions. Do the same for the remaining permissions (read and execute).

Can you explain the results obtained with these changes? In particular the restriction of the execute permission will also impact the visibility over the directory.

5 Confinement

Besides the permissions of directories and files, as well as groups a user belongs to, it is frequently required (or at least useful) to restrict a program to a subset of the resources and activities available. This is important to implement the basic security principle least privilege: services should only be provided the means required for them to execute their tasks.

If a process is started from the command line, without any other mechanism, this will not be trivial to enforce.

For the purpose of this section, consider the following program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

#define DATADIR "/data"

int main(int argc, char** argv) {
  struct sockaddr_in saddr;
  int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  bzero((char *)& saddr, sizeof(saddr));
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr = htonl(INADDR_ANY);
  saddr.sin_port = htons((unsigned short) 1234);

  bind(sockfd, (struct sockaddr *)&saddr, sizeof(saddr));

  while (1) {
    struct sockaddr_in caddr;
    socklen_t clen = sizeof(caddr);
    int n;
    char buf[1024];
    char fname[2048];
    bzero(fname, 2048);
    bzero(buf, 1024);

    int len = recvfrom(sockfd, buf, 1024, 0, \
                (struct sockaddr *)&caddr, &clen);
    if (len <= 0)
      continue;

    sprintf(fname, "%s/%s", DATADIR, buf);
    FILE* fd = fopen(fname, "r");
    fprintf(stderr, "Serving file %s...", fname);
    if (fd == NULL) {
      fprintf(stderr, "Error\n");
      continue;
    }

    while (len > 0) {
      char data[1400];
      len = fread(data, 1, 1400, fd);
      if (len > 0)
        sendto(sockfd, data, len, 0, \
            (struct sockaddr *)&caddr, clen);
    }
    fclose(fd);
  }
}
```

It is a very simple UDP server that outputs the content of any file requested in the payload of a UDP message. It should provide files that are contained under the `DATADIR` directory, but it has a vulnerability as it allows for directory transversal. Can you find the vulnerability?

Before compiling this code, you will need to install the GCC compiler on your Debian VM:

```
$ sudo apt update
$ sudo apt install build-essential
```

Then, create the `DATADIR` directory on your system, and a file within it:

```
$ sudo mkdir /data
$ echo ola | sudo tee /data/t.txt
```

Now, assuming the name is `server.c`, it can be compiled with the following command:

```
$ gcc -o server server.c
```

The server can be executed issuing:

```
$ ./server
```

Finally **in another terminal**, you can send a command to the server by executing:

```
$ echo -n "t.txt" | nc -u -w 5 127.0.0.1 1234
```

The result should be the content of the `/data/t.txt` file. The `nc` program can be stopped by pressing `CTRL-C`, or by waiting 5 seconds (due to the arguments `-w 5`).

You can request other files to test the program. An attacker would exploit this server by requesting files such as `../etc/shadow` (note the ..), exploiting the directory transversal vulnerability.

Note: The following sub-sections can be executed with any executable service that uses network capabilities. The purpose of using such a simple program is due to the fact that it has a small amount of dependencies from configuration files, data files or other libraries, simplifying the execution of the confinement exercises.

5.1 `chroot`

It is frequent that services do not need to operate over all files of the filesystem, but only over a subset of the files. For example, when you create a container, it only has access to a small subset of the host's filesystem. In the case of our test program, we would like to confine it to the directories required for its operation, plus the `DATADIR`.

The `chroot` command allows to confine the vision that a process has over the filesystem. It works by changing the notion the application has of its root directory. Therefore, a process will see an hierarchy starting in the root directory (as it always would), but this root directory can be manipulated, so that the process is confined to a sub-directory.

Creating the `chroot` implies constructing an environment with all files required for the execution of the program, which also includes the libraries required to load it from a binary file.

In order to create the environment for our server should start by creating a directory for the `chroot` environment:

```
$ sudo mkdir -p /opt/chroot
$ sudo mkdir -p /opt/chroot/bin
$ sudo mkdir -p /opt/chroot/lib
$ sudo mkdir -p /opt/chroot/lib64
$ sudo mkdir -p /opt/chroot/data
```

The `server` binary should be placed in the `/opt/chroot/bin` directory, and all the libraries it requires should be placed in the `/opt/chroot/lib` directory.

We can enumerate the libraries required using the `ldd` command:

```
$ ldd server
    linux-vdso.so.1 (0x00007ffdf5fde000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa0bca48000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fa0bcc40000)
```

Therefore, copy this libraries to the appropriate sub-directory inside the `/opt/chroot` directory:

```
$ sudo cp /lib/x86_64-linux-gnu/libc.so.6 /opt/chroot/lib
$ sudo cp /lib64/ld-linux-x86-64.so.2 /opt/chroot/lib64
```

Afterwards, place some small text files in the `/opt/chroot/data` directory, as these files will be served by our server.

Finally, the server can be started in the `chroot` environment by issuing:

```
$ chroot /opt/chroot /bin/server
```

You can use `nc` to obtain the files in the `DATADIR`, and the problem will still be vulnerable to the previous vulnerability. However, the impact is reduced to the files inside the `chroot` environment.

5.2 Linux Security Modules: Apparmor

The Linux Security Modules allows the configuration of Mandatory Access Control, which are applied to processes, independently of the user that is executing the process. One implementation of these modules is `Apparmor`.

When considering our test server, its access is limited to the user permissions, and if executed with a restricted user, the impact of its vulnerability will be reduced to the files accessible by the user. As an example, an attacker would be able to access personal files of that user, but not of another user, or some system files.

What is appropriate is to restrict the service to network communications, and access to the `DATADIR`. This can be achieved using `Apparmor` with an appropriate profile for our server.

The first step is to generate a profile stating what is allowed or denied. The profile is in `/etc/apparmor.d` and assumes the server is in a specific location. In our case, place the server in `/opt` and create a file name `/etc/apparmor.d/opt.server` with the following content:

```
/opt/server {
  /usr/lib/x86_64-linux-gnu/libc.so.6 mr,

  /data/** r,

  network inet udp,
}
```

This profile allows map and read access to the libraries the binary requests, read access to the contents of the `/data` directory, and network access using UDP sockets. Now, activate the profile by issuing:

```
$ sudo apparmor_parser -r /etc/apparmor.d/opt.server
```

Start the server and request one file that exists in the `DATADIR`. Then, request `../etc/passwd` and analyze what happened. You can have more information in `dmesg`.

The use of `apparmor` is very rich, and you can create audit records for the activities executed by each process. As an example, you can replace the last line of the profile with:

```
audit /data/** r,
```

This will specify that the actions towards those files should be audited, resulting in a log entry. After changing this line and activating the new profile, you can use `dmesg` to analyze the entries that are added when a file is transferred.

6 Bibliography

- Virtual console, https://en.wikipedia.org/wiki/Virtual_console
- User identifier, http://en.wikipedia.org/wiki/User_ID
- Group identifier, http://en.wikipedia.org/wiki/Group_identifier
- Super-user, http://en.wikipedia.org/wiki/Superuser
- Filesystem permissions, http://en.wikipedia.org/wiki/File_system_permissions
- Linux capabilities, https://man7.org/linux/man-pages/man7/capabilities.7.html
- Set-UID mechanism, http://en.wikipedia.org/wiki/Setuid
- `sudo` command , http://en.wikipedia.org/wiki/Sudo
- `chroot` mechanism, http://en.wikipedia.org/wiki/Chroot
- `apparmor` mechanism, https://gitlab.com/apparmor