COMP2700 Lab 5: Unix Security

In this lab, we will look at some security related concepts and how they are implemented in Linux. These include: the concept of subjects (processes), access control (files and directories permissions), controlled invocation (via SUID) and the *layer below* (circumventing file system access control).

The following exercises assume that we are using the lab VM provided for the course. In particular, we have two groups of users: the `tutors` group and the `students` group. The tutors group consists of users alice, bob and charlie. The students group consists of users dennis, eve, and felix. The administrator username is `admin2700`. To do this lab you need to be familiar with Linux commands discussed in Lab 1 and Lab 2.

Lab setup

Login as alice, download the file lab5.tar.gz and extract it.

```
$ su -l alice
$ wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab5.tar.gz
$ extract-lab lab5.tar.gz
```

This will create a directory lab5 in alice's home directory. The following exercises will be referring to the files and directories contained in lab5.

Users, Ownership and Permissions of Files and Directories

We review here some basic commands for switching users (becoming other users or root) and for changing ownerships and permissions of resources. After the review, do the requested exercises.

Users and groups

The following commands can be used to query the identities of groups and users:

- `whoami`: This command displays the username of the currently logged in user.
- `id`: This command displays the user id associated with the username, and the groups that user belongs to. For example:

  ```
  $ id alice
  ```

  will display alice's uid, primary gid, and all the groups alice belongs to.
- `groups`: This command displays all the groups that the user belongs to.

Escalating privilege

The following commands allows a user to assume the identity of another user -- with proper authentication.

- The `su` command. This can be used to switch to another user in the shell. The syntax is as follows:

```
su [username]
```

where [username] is the target user name.

- The `sudo` command. This command allows a `sudoers` (users in the `sudo` group, e.g., the user `admin2700` in the lab VM) to become the `root` user:

  ```
  sudo [options] [command]
  ```

  where `[command]` is the command to execute under the root privilege. One useful command is the `bash` shell itself:

  ```
  $ sudo /bin/bash
  ```

  which will then run the `bash` shell as root. A shorter version:

  ```
  $ sudo -s
  ```

Changing Ownerships

Syntax:

```
chown [owner:group] [File-or-directory]
```

The chown (change owner) command usually needs to be run under the root privilege (so `sudo` may be needed). As a normal user, you don't often have the privilege to do so. However, you can always change a file that belongs to you to another group which you are in. For example

```
$ chown alice:tutors Documents
```

changes the owner user and the owner group of the directory Documents to `alice` and `tutors`, respectively.

Changing permissions

The chmod command is used to change the permissions of files or directories. Its syntax has the general form:

```
chmod [options] [file_or_directory]
```

The `options` part of the command specifies the permissions to be given to the file. This can be specified using permission bits (via the octal notation) as discussed in the lectures on Unix security, or via a more user friendly way using textual representation as exemplified below.

See the 'man chmod' for more details of various options for changing permissions. Some examples:

- chmod 644 test.txt: This commands gives read and write access to the user owner of the file test.txt, and read access to the group owner, and read access to others.
- chmod 4755 program: This commands gives read, write, execute access to the user owner, read and execute access to the group owner and others, and turns `program' into an SUID binary.
- chmod u+rwx test.txt: This command adds read, write and execute permissions to the owner of the file test.txt
- chmod g-w test.txt: This command removes the write permission from the owner group of test.txt.
- chmod g+x test.txt: This command adds the execute permission to the owner group of test.txt.
- chmod o-r test.txt: This command removes the read permission from the others.
- chmod u+s program: This command sets the SUID bit of the file `program'.

Checking permissions and ownership

Recall from the lectures and previous labs, to display the details of the permissions and ownership of files and directories, you can use the following command:

```
$ ls -l [file_or_directory]
```

For example,

```
$ ls -l ~/lab5/
```

displays details of the permissions and ownership of files and directories in `~/lab5`.

Exercise 1. In the directory `~/lab5/share/`, there are several files. The files currently have all the permission bits set to zero, so no access is allowed to any of these files. Change the permissions and/or group ownership of the files to implement the following access control.

1. File `tutorsonly.txt`: Give alice read and write permissions, and members of the tutors group read permission, and no permissions for the other users.
2. File `courses.txt`: Give alice read and write permissions, members of the tutors group read and write permissions, and read permission for the other users.
3. File `feedbacks.txt`: Give alice read and write permissions, members of the tutors group read permissions, and no permissions for the other users.

4. File `addfeedback`: This is a binary program that appends user's input to the file `feedbacks.txt`. Change the permissions and/or group ownership of this program so that anyone other than the tutors (except for alice) can execute this program to add a feedback to `feedbacks.txt` (you'd need to complete Exercise 1.3 first to change the permissions for `feedbacks.txt` as indicated above in that exercise).

Exercise 2. What are the minimum permissions a user needs to have for a directory in order for the user to read the content of a file in that directory? Assume that the user knows the name of the file and has read and write access to the file. To answer this question, use the VM to test your hypothesis, e.g., create a new directory, say `testdir`, and create a file, say `testfile` in `testdir`. Then experiment with different permission combinations for `testdir` and do a `cat testdir/testfile`.

Processes

Recall from the lectures that processes correspond to "subjects" in the access control mechanisms of Unix/Linux. The `ps` command can be used to list processes running in the system. By default, the command `ps` without any arguments will show the active processes for the current user in the current terminal. To see all processes running in the system (including processes owned by `root`), you can use the following command:

```
$ ps -e -F
```

We can also select the columns we want to display, using the option `-o` (in addition to `-e`). For example, the following command displays information about "real user" (the user who launched this process), the process id, and the command being executed.

```
$ ps -e -o ruser,pid,comm
```

You may notice that some usernames (such as `admin2700`) got truncated and displayed as `admin27+`. You can set the width of a column by specifying the width with a `:N` (where `N` is the desired width), e.g., to set the width of the `ruser` column to 10 characters:

```
$ ps -e -o ruser:10,pid,comm
```

The `ps` command has a rich set of options allowing one to display various information related to processes running in the system. Use `man ps` to see the available options.

Signals

In Linux, the operating system can interact with processes via signals. A common signal is the SIGTERM signal, with numerical value 15, which tells a process to terminate gracefully.

To send a signal to a process, use the kill command:

```
kill -[signal value] [process id]
```

For example,

```
$ kill -15 1658
```

sends the signal 15 (see below for what this signal means) to the process with process id 1658.

The following is an excerpt from Linux man page ('man 7 signal') with relevant details of signals.

```
Signal dispositions                    倾向，性格
   Each signal has a current disposition, which determines how the process
   behaves when it is delivered the signal.

   The  entries  in  the  "Action"  column of the tables below specify the
   default disposition for each signal, as follows:

   Term   Default action is to terminate the process.

   Ign    Default action is to ignore the signal.

   Core   Default action is to terminate the process and  dump  core  (see
          core(5)).
   Stop   Default action is to stop the process.

   Cont   Default  action  is  to  continue the process if it is currently
          stopped.

Standard signals
   Linux  supports the standard signals listed below.  Several signal num-
   bers are architecture-dependent, as indicated in  the  "Value"  column.
   (Where three values are given, the first one is usually valid for alpha
   and sparc, the middle one for x86, arm, and most  other  architectures,
   and  the  last one for mips.  (Values for parisc are not shown; see the
   Linux kernel source for signal numbering on that architecture.)  A dash
   (-) denotes that a signal is absent on the corresponding architecture.

   First the signals described in the original POSIX.1-1990 standard.

   Signal     Value     Action   Comment
   ──────────────────────────────────────────────────────────────────────
   SIGHUP        1       Term    Hangup detected on controlling terminal
                                 or death of controlling process
   SIGINT        2       Term    Interrupt from keyboard
   SIGQUIT       3       Core    Quit from keyboard
   SIGILL        4       Core    Illegal Instruction
   SIGABRT       6       Core    Abort signal from abort(3)
   SIGFPE        8       Core    Floating-point exception
   SIGKILL       9       Term    Kill signal
```

```
    SIGSEGV      11       Core    Invalid memory reference
    SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                  readers; see pipe(7)
    SIGALRM      14       Term    Timer signal from alarm(2)
    SIGTERM      15       Term    Termination signal
    SIGUSR1   30,10,16    Term    User-defined signal 1
    SIGUSR2   31,12,17    Term    User-defined signal 2
    SIGCHLD   20,17,18    Ign     Child stopped or terminated
    SIGCONT   19,18,25    Cont    Continue if stopped
    SIGSTOP   17,19,23    Stop    Stop process
    SIGTSTP   18,20,24    Stop    Stop typed at terminal
    SIGTTIN   21,21,26    Stop    Terminal input for background process
    SIGTTOU   22,22,27    Stop    Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

Exercise 3. In the directory `~/lab5/signals/` there are two programs: whatsmyid and stubborn. (The corresponding C code for these programs are also provided but you don't need to understand them for now).

1. Open a terminal, login as user `alice` and run `whatsmyid`. Then open another terminal, using the same login (`alice`), and then try to figure out what the process id of the `whatsmyid` process in the first terminal is. Then send a SIGTERM to terminate that process. What command(s) do you use?

2. Same as Exercse 3.1. but this time run `stubborn` instead. You may find that this process is harder to terminate as it tries to ignore all signals sent to it. What command do you use to terminate this process? What would be a security implication if a process is allowed to ignore all signals?

Soft links and hard links

Soft links correspond to what is known as shortcuts in other operating systems such as Windows. Both soft links and hard links (discussed next) are created using the ln command. For example, to create a short cut to a file `foo.txt` and call it `bar.txt` simply run:

```
$ ln -s foo.txt bar.txt
```

This will make bar.txt points to foo.txt.

In Linux, to every file the system associates an inode, a data structure that holds various information about the file (permissions, various attributes such as creation/modification time, links to blocks on disk that hold the data, etc). A hard link to a file, say `a.txt`, is a regular file that shares the same inode as `a.txt`. In other words, they have identical contents.

To create a hard link to a file `a.txt`, and call it `b.txt`, simply run the command:

```
$ ln a.txt b.txt
```

The option `-i` in the `ls` command can be used to display the inode of a file. The `find` command also has an option to locate files with the same inode.

Exercise 4. In Linux you can create a symbolic link to a non-existent file. If the existence of a file is a secret information, can this feature be used as a side channel to infer the existence of a file in a target directory? Show an experiment to confirm your answer, e.g., create some files in the /root directory (as root user) and, logging in as a non-root user (e.g., bob), try to observe if there is any difference between a soft link to an existing file vs a soft link to a non-existent file.

Exercise 5. In the directory `~/lab5/links/` there are several files and directories. Find out which files are hard-linked to which. You may want to use the `find` command. Try `man find` for more information on how to use the `find` command for this purpose.

SUID programs

Recall from the lectures that a process can have two user ids associated with it: the real user id (which is the id of the user who launched the process) and the effective user id (the user id that is used to determine the permissions of the process). For a normal program, once it's launched, its process will have the same real user id and effective user id. But for SUID programs, the real user id can be different from the effective user id. SUID programs are used to implement a notion of controlled invocation, typically to secure access to sensitive system resources by normal user.

You can use the `ps` command to display both the real user (`ruser`) and the effective user (`euser`) associated with a process, e.g.,

```
$ ps -e -oruser,euser,pid,comm
```

You can also filter the output of `ps` based on real users (using `-U`) and effective users (using `-u`). For example:

```
$ ps -u root -o ruser,euser,pid,comm
```

shows processes running with the effective user id of the root user.

Exercise 6. The program `/usr/bin/passwd` in Linux is an example of an SUID program that allows a normal user to alter the `/etc/shadow` file to change their password. Run the `passwd` command (but don't change the password) and query the process list to confirm that it is running with the effective user id of the root user.

SUID on shellscripts

In Linux, for security reasons, the SUID bit has no effect on shell scripts. So even if a shell script has the SUID bit set, the effective user id will be the same as the real user id when the script is invoked. In order to make shell scripts executable under a different effective user id, one can call the shell script indirectly via a normal binary program, e.g., using the system function or the execve function in C.

Exercise 7. Consider the following example of controlled invocation that uses a shell script. Alice wants to share a file called `not_for_charlie.txt` with everyone except Charlie. Since she cannot achieve this using permission bits alone, she decided to implement a SUID program to enforce this access policy. But instead of writing a proper SUID binary, she decided to use a C program to wrap the call to a shell script that displays the file after a check on the current username. The code is listed here, but you can also find it in `~/lab5/suid/`. This code has been compiled to an SUID binary called `filter.`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[], char *envp[]) {
  execve("filter.sh",argv,envp);
  return 0;
}
```

The program filter makes a call to the shell script `filter.sh` using the function `execve`. The file `filter.sh` contains the following script

```
#!/bin/bash -p
if [ "$USER" = "charlie" ]; then
   echo "Go away charlie";
   exit 1;
fi
/bin/cat not_for_charlie.txt
```

The script uses the USER environment variable to deny access to Charlie the file `not_for_charlie.txt`. Alice has read and write access to that file, but others have no direct access. Now log in as charlie and try to circumvent Alice's access control mechanism to read the file `not_for_charlie.txt`. How would you (as user charlie) achieve this? What do you think is the issue with the above SUID program?

The 'layer below'

In Linux, block devices such as hard drives are represented in the file system as special files located in the /dev directory. For example, the file `/dev/sda` typically represents the first hard drive in the system. Partitions in the hard drive are also represented individually, e.g., `/dev/sda1` represents the first partition in the hard drive `/dev/sda`.

Not all files in `/dev` represent real devices. For example, we have the following virtual devices:

- `/dev/null`: This is a character device that discards all data written to it. It is useful when used in conjunction with other commands to suppress output or error messages. For example, running

  ```
  $ ls -l > /dev/null
  ```

  will run the command ls but suppress all the output so nothing gets printed.

- `/dev/zero`: This is a character device that represents a stream of zeros. This is useful to wipe a (real) device with zero bits for example.
- `/dev/urandom`: This is a character device that represents a (cryptographically secure) stream of random bytes. This is useful to generate random bit strings, to be used in applications where strong cryptographic randomness is required.

A block device file such as `/dev/sda1` can be read and written to as if it is a normal file, and is subject to the same permission mechanisms governing normal files in a file system. A read access to `/dev/sda`, for example, allows a user to read any block of bits in the hard drive represented by `/dev/sda`. This direct access to the blocks of bits of `/dev/sda` by-passes the access control mechanism of the file system that is stored on `/dev/sda`.

For example, if `/dev/sda` represents the main hard drive of the system, then a user with read access to `/dev/sda` will be able to read any block of bits in the hard drive, including blocks of bits that are linked to sensitive data such as password hashes, etc.

A couple of commands that will be useful to query devices in the system:

- `df`: The `df` command (without arguments) displays all the devices that are currently mounted in the system, including information such as their free space, mount points etc.
- `debugfs`: This command is useful for performing a direct access to a block device. The syntax of the command is

  ```
  debugfs [options] block-device
  ```

  where `block-device` denotes any block device in `/dev` directory. By default, when no options are specified, the block device is opened in a read-only mode. To open it in a write-mode, use option `-w`.

> **WARNING**: DO NOT perform low level disk write to the actual disks on your computers unless you know what you are doing. For this lab, try this only in the provided lab VM, so any damages will be confined to the virtual disk of the VM.

Running `debugfs` will launch a shell where one can issue various commands to access information stored on the device, including, eg., inodes of a file, sectors of a file, etc. Use the `help` information of debugfs to try different commands inside debugfs. One useful command is `cat`, which has the same function as the program `cat` in bash shell, i.e., to display content of a file (or an inode, in the case of debugfs).

Exercise 8a. (VirtualBox Lab VM only)

Note: if you are using Azure Labs VM, skip this and do Exercise 8b instead.

In Unix, access control related to files and directories are enforced at the file system level. One can circumvent this access protection if one has access directly to the device on which the file system resides. In this exercise, we will look at how to by-pass the access control of the file system of in the lab VM.

First, we create a file that is supposed to be readable only by root. Open a new ssh connection to your VM and run (as user `admin2700`):

```
$ echo hello | sudo tee /mnt/test.txt
$ sudo chmod 600 /mnt/test.txt
```

and then switch back to user `alice`.

1. Find out which block device is mounted to the root directory `/`.
2. Which user, other than the root user and the admin2700 user, does have direct access to the block device underlying the directory `/`? Justify your answer.
3. Show how the user mentioned in Exercise 8a.2 above can by-pass the access restriction on the file system to display the content of the file `/mnt/test.txt`.

Exercise 8b. (Azure Labs VM only) This exercise is almost identical to Exercise 8a, so if you have done Exercise 8a, you can skip this. This exercise applies only to Azure Labs VMs, which were set up differently (as part of the default configurations of the cloud-based VMs), so Exercise 8a won't work in this VM.

First, we create a file that is supposed to be readable only by root. Open a new ssh connection to your Azure Labs VM and run (as user `admin2700`):

```
$ echo hello | sudo tee /mnt/test.txt
$ sudo chmod 600 /mnt/test.txt
```

and then switch back to user `alice`.

1. Find out which block device is mounted to the directory `/mnt` directory.
2. Which user, other than the root user and the admin2700 user, does have direct access to the block device underlying the `/mnt` directory? Justify your answer.
3. Show how the user mentioned in Exercise 8b.2 above can by-pass the access restriction on `/mnt/` by showing the content of the file `/mnt/test.txt`.

Extension exercises (optional)

Exercise 9 (VirtualBox VM only). Before attempting the following exercise, make sure you create a backup of the files you want to keep outside the VM.

You can create a `Snapshot` of your VM, so you can restore the state of your VM in case something goes wrong. The VirtualBox website (`https://www.virtualbox.org/manual/ch01.html#snapshots`) provides some instructions on how to do this. The tutor will also guide you on how to create a snapshot.

This is a follow up of Exercise 8a. Log in as the non-root user who has raw disk access (see Exercise 8a). Show how you can change the permission of /etc/shadow file, so that it is readable and writeable by all users, without root privilege. *Hint: you need to modify the inode of the shadow file directly. You may need to restart the VM after you modify the permissions.*

Exercise 10. Programs that are owned by root and have their owner suid bit set are often targetted by vulnerability researchers as such a program, if it contains an exploitable vulnerability, may allow a local attacker to escalate their privilege to root. Use the `find` command to identify all root-owned programs in the lab VM. Hint: you may find the `-perm` and the `-user` options of the `find` command useful. Consult the manual of `find` for details (`man find`).