Laboratory Assignment 1: Identification and Authentication

1 Purpose

In this laboratory assignment, you will study identification and authentication, and look closer at the requirements of a UNIX application that runs with higher privileges.

The assignment consists of two parts: one to be done at home and another to be done in a laboratory. For the first part, there are a number of questions for you to answer (see Section 3). For the second part, you will be asked to implement a login program for UNIX, and demonstrate your solution in the laboratory (see Section 4).

2 Preparations

Read the material as specified below. Then read the rest of this assignment before you start to work.

2.1 Reading

- Course book, Chapter 3 (User Authentication)
- Lecture slides - UNIX Security
- Offprints - first part

You will be using the C programming language when programming the UNIX application, so if you feel uncertain, it may be a good idea to repeat or study the basics of C before you start programming. This lab will also require you to read and learn from UNIX manual (man) pages.

3 Home assignment and written report

The report should be 2–3 pages (excluding appendix). It should contain at least the following items:

1. Introduction
2. Answers to the questions
3. Conclusions

4. An appendix containing the commented code from the lab assignment.

The introduction should give an overview of the assignment and content of the report. Each question in Section 3.1 should have its own heading. The conclusions section should contain your reflections about the authentication problem and the assignment. The report should be submitted to your instructor using PingPong before the deadline.

3.1 Questions

1. What is password ageing? And what methods exist to implement it?
2. If you want to increase the security of a system, you can use one-time passwords. What are the advantages? What are the disadvantages?
3. Authentication systems are often based on some knowledge shared by the computing system and the user. This knowledge can be of three types: something the user *knows*, *has* or *is*. For each of these types, answer the following questions:

   a. How can the authentication mechanism be implemented?

   b. What advantages and disadvantages does the authentication method have with respect to e.g. user friendliness, cost of introduction, and accuracy.

   c. How can the disadvantages be overcome?
4. What authentication method would you recommend for use in computer systems in the following environments?

   Motivate your answer, and state the assumptions you have made concerning the security needs in each environment.

   a. A university

   b. A military facility

   c. A middle-sized company

   d. A personal home computer
5. In some systems you do not only implement authentication of the user against the system, but also of the system against the user. What is the purpose of doing this?
6. A user is running a program containing the system call `setuid()`. Depending on who is running the program and on the value of the argument to `setuid()`, different things can happen.

   Check the manual page for `setuid(2)` *on the lab system* (`man -s2 setuid`).

   a. Study Figure 1. What are the values of the real user ID (ruid) and the effective user ID (euid) at i) and ii)?

   b. Table 1 illustrates six cases where root or a normal user account starts a normal program which uses `setuid()` with different arguments (user IDs). Fill in the third and fourth column in the table. The third column should state whether the system call will succeed or fail. The fourth column should state the user ID (ruid) of the program after the `setuid()` call.

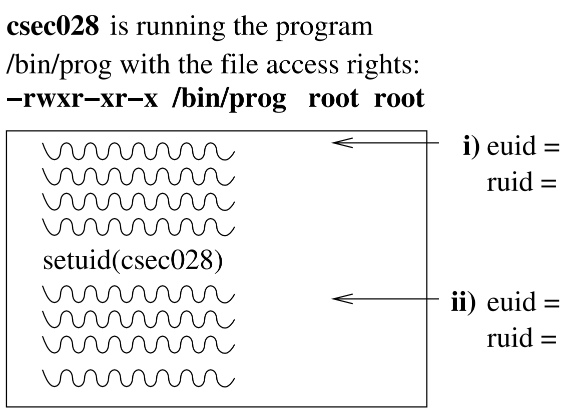
csec028 is running the program
/bin/prog with the file access rights:
-rwxr-xr-x /bin/prog root root
setuid(csec028)
i) euid =
ruid =
ii) euid =
ruid =

Figure 1

Table 1

	current user	setuid(UID)	success/failure	user ID after setuid()
1	root	UID: 0 (root)		
2	root	UID: 20757 (csec069)		
3	root	UID: 20716 (csec028)		
4	csec028	UID: 0 (root)		
5	csec028	UID: 20757 (csec069)		
6	csec028	UID: 20716 (csec028)		

7. On Unix systems, file-access permissions on programs may be set to set-user-ID (these are often called SUID programs) as in the `passwd` program:

```
 csec@legolas~ > ls -l /usr/bin/passwd
-rwsr-xr-x  1 root root 27888 Jul 26 09:22 /usr/bin/passwd
```

   Study Figure 2 while answering the following questions:

   a. What are the values of the real UID (ruid) and the effective UID (euid) at *i*), when the user `csec028` runs the SUID program `/bin/prog`?
   b. What is the purpose of using SUID on programs?

   Hint: *Observe that SUID programs do not necessarily use the* `setuid()` *system call to set the special rights of the program. SUID is a separate operating system mechanism built on special file-access permissions. Make sure you understand the difference.*

8. Sometimes the `setuid()` (or perhaps more likely, `seteuid(2)`) function is used in programs where the set-user-ID bit is activated on the program's file-access permissions. This is done, for example, in the `/usr/bin/passwd` program.

   a. What are the values of `ruid` and `euid` at *i*) and *ii*) in Figure 3?

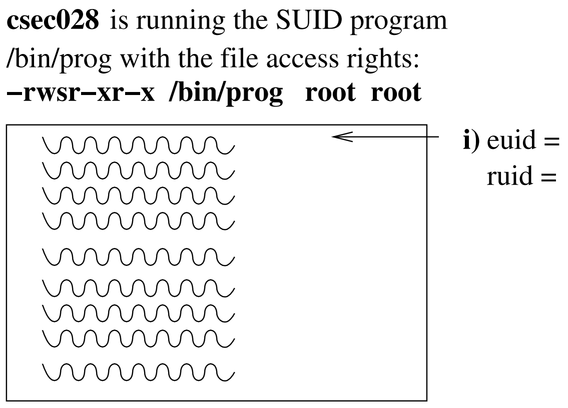
csec028 is running the SUID program
/bin/prog with the file access rights:
−rwsr−xr−x /bin/prog root root
i) euid =
ruid =

Figure 2

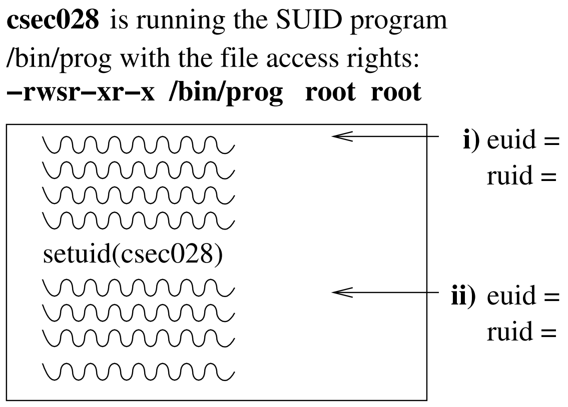
csec028 is running the SUID program
/bin/prog with the file access rights:
−rwsr−xr−x /bin/prog root root
setuid(csec028)
i) euid =
ruid =
ii) euid =
ruid =

Figure 3

b. What is the purpose of doing this?

Hint: *what happens if the program has a buffer-overflow vulnerability before the* `setuid()` *function call? What happens if the vulnerability is after the* `setuid()` *function call?*

4 Laboratory Assignment

This task is described in several steps below. Make sure that each step is working well before you proceed to the next one.

Note that when you submit your work, we are only interested in the complete program, which is the result of solving all the required steps.

The implementation should be done in the C programming language. The reason for this is that all major operating systems used today are written in C. Since Linux is developed in C, you have direct access to many library routines, some of which are discussed below.

Before you begin, read these three important items:

- Library routines do nothing "magical". They are simply C functions that use system calls and possibly other library routines. You can complete your program without them, but then you will be forced to write more code.
- Make sure that you read the manual page for the correct routine! If you, for example, write `man time`, you get the manual page for the command `time(1)`. But in order to get the page that describes the library function `time(2)`, you will need to write `man -s2 time`. **If any `include`-lines exist under Synopsis in the manual page, you must include these to be able to use the routine in question.**
- Even the most experienced programmer may create buggy code, with possibly catastrophic results if the code is critical. Some of these defects may be detected by checking the return values of system calls and other functions. **Always do that, otherwise your lab will not be approved!**

4.1 The Program

Normally, the program `getty` displays the text "login:" when users log in on a text terminal. This program also accepts a username and executes another program called `/bin/login` with this username as an argument. The program `/bin/login` prints "Password:" and checks if the entered password is in fact the correct password for the actual user. If it is correct, a command interpreter is executed. However, in our case, the most practical approach is to let the `login` program handle both the identification (username) and the authentication (password) that the user supplies.

4.1.1 Step 1 — Mimic UNIX login

Write or study a small program that mimics the most common login procedures in UNIX. You have two options here: either you write it yourself according to the specifications below, or you download the file `login_linux.c` from the assignment folder on PingPong. If you choose to write it yourself, the following should be implemented:

- The program begins with displaying "login:" and takes the username as input.
- Then the program writes "Password:" and waits for the password to be entered, which should not be visible on the terminal. Use the function `getpass(3)` that, among other things, will make sure that the text is not "echoed" on the terminal.
- The program queries the system's user database to see if the username exists. If this is the case, it encrypts the entered password (with a known algorithm) and compares the result to the stored encrypted password of that user. Suitable library routines are `getpwnam(3)`, `crypt(3)` and `strncmp(3)`.
- If the username does not exist, or if the password is wrong, the program displays "Login incorrect" and restarts from the beginning. Otherwise it writes something like "Welcome to this System!" and terminates.

Test that your program works by compiling and running it.

Note: *You only need to compile and get your program to run, not make it succeed to log in (which is impossible in the lab system: the encrypted passwords are not readable here). When your program starts, proceed to the next step.*

4.1.2 Step 2 — The user database

In this step, you need to add a database that contains login information and make your program use it.

- Now You will modify the program to use its own user database to look for data (instead of using `getpwnam(3)` like in step 1). The routines for accessing the database are already given. Download the files `pwent.h`, `pwent.c` and `Makefile` from the assignment folder on PingPong, and store them in your own lab directory.

- The given routines follow `getpwnam(3)`. Read its man page for more explanations. Note that the line `‘‘struct passwd *passwddata’’` needs to be changed to `‘‘mypwent *passwddata’’` and NOT `‘‘struct mypwent *passwddata’’`.

- If you copied `Makefile`, you can compile your program with the command `make` now.

- The name of the database is assumed to be `passdb`[1] (a common text-file).

- Each record in passdb should have the following syntax:
  `name:uid:passwd:salt:no_of_failed_attempts:password_age`

In this step, your program only needs to *use* the fields *name*, *uid* and *passwd*. All six fields need to be present, however. In order to make troubleshooting simpler, the passwords may be stored as plaintext in this step.

Test that your program works by compiling and running it.

4.1.3 Step 3 — Resistance to buffer-overrun attacks

A common way to “crack” programs is to enter longer text-strings than the program expects or can handle—the so-called “buffer overruns”. Test what happens when you enter more characters than the username buffer length allows. Now make sure that it is impossible to overwrite memory locations. That is, do not fetch more characters from the keyboard or a file than will fit in the allocated buffer. If `fgets` is used instead of `gets` you will get a "\n" at the end of the string that need to be replaced by "\0".

Test that your program works by compiling and running it.

[1]That is, passdb and not passdb.txt or PASSDB.

4.1.4 Step 4 — Additions to the user database

Now you shall add more information to the user database.

Hint: *In order to continuously display changes to the database, you may issue the command* `'tail -F passdb'` *in a different terminal window (here it is assumed that the name of your database is* `passdb`).

- The number of failed login attempts, `no_of_failed_attempts`, should be recorded in the database. When the login (finally) succeeds, this number should be displayed and thereafter be set to zero. This field will be further used in step 5.
- The "age" of the password, which is the number of successful login attempts, should be recorded in the database. When this number exceeds a certain limit, for example 10, the user should be continuously alerted to change the password. This variable should be set to zero according to the user action, just like the program `passwd` in UNIX.)
- The passwords shall no longer be stored in plaintext. Use `crypt(3)` or a better algorithm for encryption! For password generation, you may download (and compile) `makepass.c` from the assignment folder on PingPong (check source code for usage). The `makepass` program is used manually to generate a password, and the password is manually written to the `passdb` file. You do not need to write a program to do this!

Test that your program works by compiling and running it.

4.1.5 Step 5 — Prevention of repeated online password guesses

Now you must prevent an attacker from breaching the security of the system by guessing likely passwords online (through the login program). That is, you need to protect against a potential *bruteforce attack* on your program. If the attacker can copy the database, the guessing procedure may happen offline, but that is a case we do not consider at this point. The system can tolerate a user entering the wrong password a few times, but repeated erroneous passwords signify an intrusion attempt.

Use the field with the number of failed login attempts in order to implement a barrier against repeated guesses of passwords. Several alternative solutions exist, but it is enough if you implement what you consider reasonable.

Test that your program works by compiling and running it.

4.1.6 Step 6 — Make the program secure and start a command interpreter

A program, like the one you have just written, needs to run with super-user privileges. Only then will UNIX allow us to start a program (e.g. a command interpreter) with another user's rights. However, this setting could also be dangerous if a user, who could make the program do something forbidden, receives access to super-user privileges. When writing such programs, you must therefore be very careful. Up to this point, you have ensured that the program is immune to buffer overflows and malformed input. However, you also need to take care of the following items:

- A basic way to control the execution (such as terminating or suspending) of a process in UNIX terminals is by pressing certain key combinations. Make sure that all signals from the keyboard are ignored, so that it is impossible to cancel the program through the use of, for example, `ctrl-c`. Are there any other key combinations that the program should protect itself against? Use the system call `signal(2)`.

  Hint: *Check out* `signal(7)` *for a listing of signals.*

- Upon a successful login attempt, a command interpreter (`/bin/sh`) should be started with the correct user access rights. You do not need to handle group belongings.

Test that your program works by compiling and running it. Then, go back to the beginning of the assignment and read what is required for the report.