Laboratory Assignment 2: Identification and Authentication

For all the assignments in this course, you are expected to work home and only book a lab slot if you are stuck and require a TA's assistance or you are required to demonstrate your solution.

1 Purpose

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

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 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. First, follow the below instructions on how to connect to the Chalmers server running Linux.

On Windows 10, open the ssh client "putty" and connect to `remote11.chalmers.se` using your `cid` and password. You will be using ssh to compile your program. In

order to manage your files on Windows, you should connect to Linux share by typing \\sol.ita.chalmers.se\<cid>.

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.

Make sure your code checks for return values. Failure to do so may result in your solution being rejected.

3.1 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.

3.1.1 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 Canvas. 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.

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.

3.1.2 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 Canvas, 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.

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

3.1.3 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.

3.1.4 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 Canvas (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.

3.1.5 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.

3.1.6 Step 6 — Ensure that the program is 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.

3.2 Lab Demonstration

- Test your program by compiling and running it to ensure it works correctly.
- Schedule a demonstration session:
  - You can either book a slot during the normal week for your demonstration, or
  - Attend the assigned demonstration slot as per the schedule.
- Present your solution during the demonstration session, ensuring that all group members are present and participate.

3.3 Canvas quizz

You are expected to provide your (commented) code from the lab assignment on the quizz.

To pass the lab, you need to perform successful demonstration and then submit the code. Make sure the group member doing the submission keeps an eye for comments from the instructors on your submission to see how you can address any mistakes found. Although comments are provided to one group member, other members are expected to take part in addressing the provided feedback.