2

Lab 1: Login Security

This lab explores the "hardening" of a program against a given attack. You will begin by writing a simple program. Next, you will try to circumvent protections against the program. Then you will strengthen this program, and so on. In real life, programs often undergo similar enhancements as security vulnerabilities are reported or exploited. This back-and-forth between hardening and exploitation is a form of technological escalation often referred to as an "arms race."

The application we focus on in this lab is a program you have used many times before, but probably never really thought much about: the `login` *program. The* `login` *program ensures that only authorized users are permitted to use a machine. Since* `login` *must read the system's protected* `/etc/shadow` *file, it needs elevated privileges to function. Therefore, bugs in a* `login` *program can cause serious vulnerabilities.*

2.1 *Learning Goals*

In this lab, you will learn:

- how to control an interactive program using a pseudoterminal;
- how to write a "brute force" procedure; and
- an effective set of countermeasures against brute force login attacks.

All of the topics in this lab require skills that you have already developed to some extent. Many students find that this lab clarifies which programming skills are rusty or underdeveloped. Consider this assignment a warm-up for C programming and for using `Makefiles`. If you struggle with parts of this or any other lab, make a note of the problem areas, and see me for help. Computer security often exploits subtle weaknesses in computer systems, and no security practitioner knows *all* of the things they need to know. Instead, they cultivate an awareness of the limits of their knowledge, and develop the habit of rectifying those limits. If you find this lab too easy—great! You'll be ready for the next one.

2.2 *Required Reading*

This lab refers to some other readings. You should read them when you reach those sections.

- Chapter 3 explains how to use a *pseudoterminal* to control a program.

2.3 *Computing Environment*

Remember that this assignment must be completed and submitting using our standard lab environment on your Raspberry Pi computer. See Chapter 1 for instructions.

2.4 *Finding Documentation for C Functions*

Throughout this lab, you will need to find documentation for various C functions. In Linux and in other UNIX-like operating systems, you can find documentation on all system and C standard library calls using the `man` command. `man` is short for "manual," and it is broken into the following nine sections:

Section	Description
1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in `/dev`)
5	File formats and conventions, e.g., `/etc/passwd`
6	Games
7	Miscellaneous (including macro packages and conventions)
8	System administration commands (usually only for `root`)
9	Kernel routines (non-standard)

Your system's `man` page system will refers to entries using a convention like `fgets(3)`. The meaning of this convention is that you can find information about `fgets` in *section 3* of the manual. Type `$ man 3 fgets` to access it.

For example, if I want to obtain documentation for the `fgets` function, which is a part of the C standard library (aka `libc`), I would type the following command into my shell:

```
$ man 3 fgets
```

If you don't know what section a command or function might belong to, you can use the `apropos` command:

```
$ apropos fgets
fgets(3)            - input of characters and strings
fgets_unlocked(3)   - nonlocking stdio functions
fgetspent(3)        - get shadow password file entry
fgetspent_r(3)      - get shadow password file entry
```

The output says that the `fgets` function is in section 3, which is what we used when we called `man` above.

Becoming familiar with `man` is your first step toward becoming a competent systems programmer. If you want to know how to use a function, you should turn to it first, since there are sometimes subtle distinctions between the same function call from one operating system to the next. Only the `man` page installed on your own computer is guaranteed to be correct for your own system (in other words, sometimes Google is wrong!).

2.5 *Starter Code*

This assignment comes with a small set of libraries and a `Makefile` for you to use. You will need to modify the `Makefile` as a part of this assignment, but you need not modify any of the provided libraries.

The starter code contains the following files:

File	Purpose
`password.db`	A password database.
`console.c`	File that contains the `fgets_wrapper` helper method.
`console.h`	API for `console.c`.
`database.c`	Library for reading the `password.db` database.
`database.h`	API for `database.c`.
`ptyhelper.c`	Library for working with pseudoterminals.
`ptyhelper.h`	API for `ptyhelper.c`.
`Makefile`	A `make` specification for building your code.

You will need to add additional files as specified in each part below.

You are strongly advised to use the `fgets_wrapper` function provided in the `console` library to obtain user input.

2.6 *The Password Database*

The password database, `password.db`, uses the following format:

```
username_1:password_1
username_2:password_2
...
username_n:password_n
```

Usernames and passwords may be *up to 8* alphanumeric characters long. Each username and password pair must be terminated with a newline character (i.e., `\n`).

2.7 *Part 1: `login0`, a naïve login program*

In this part, you will write a login program in a file called `login0.c`. You should be able to compile this program by typing `make login0`, which should produce a binary file called `login0`. You will need to modify the `Makefile` to add a `login0` compile target.

Specification:

1. The program should prompt the user to enter a username.
2. The program should attempt to locate the username in the database.
   (a) If the username is in the database, the program should prompt the user to enter a password;
   (b) otherwise, the program should print `USER NOT FOUND` and then go to step 1.
3. If the username is in the password database *and* the entered password matches the stored password in the database, then the program should print `ACCESS GRANTED` and terminate.
4. Otherwise, go to step 1.

The following is a sample `login0` session. Make sure your program behaves *exactly* like this:

```
$ ./login0
Enter a username: barowy
USER NOT FOUND
Enter a username: dbarowy
Enter a password: password
ACCESS GRANTED
```

Observe that I didn't specify one case. What to do is up to you.

2.8 Part 2: Attacking `login0`

Before you write attack code, you will need to write code that lets you supply inputs to `login0`. For those command-line applications that accept input on the "standard input stream" (aka, `stdin`) and provide output on the "standard output stream" (`stdout`), programmatically supplying inputs and capturing outputs is easy. If you've used UNIX long enough, you have probably seen the *pipe character*, `|`. The pipe character "sends" one program's output to another program's input by connecting the first program's `stdout` to the second program's `stdin`. For example, the command `echo "heya" | mail -s "just saying hi" mail@example.com` sends `"heya"` to the `mail` program via `stdin` which then sends email to the given address. Give it a try. [1]

[1] Just be sure to change the email address first.

Unfortunately, *interactive* programs like `login0` are not so straightforward. The problem is that an interactive program is attached to a user's terminal, and it expects user input in a different form than `stdin`. For example, if you use the `fgets_wrapper` function I provide, then when a user types on their keyboard, characters are temporarily stored in an array called a *keyboard buffer*. When the buffer fills up, or if the user presses the Enter key, then it is *flushed*: the characters are removed from the buffer and sent to the program.

Why buffer input? For two reasons. First, for performance reasons. For many programs, there is no need to do work while the user is entering their input. Second, to control the way data is delivered to a program. If a keyboard buffer is 1024 bytes (the default on most Linux systems), then the program can expect to receive data in chunks not exceeding 1024 bytes in length.

`fgets_wrapper` uses buffered input primarily for the second reason, so that if a user types in a 9-character password when we're expecting an 8-character password, our login program can extract the right number of characters from the buffer and discard the rest, preventing a user from accidentally overflowing our password data structure. Unfortunately, this means that if you pipe input to a program using `fgets_wrapper`, it may not work as you expect, because parts of the input will be discarded.

Fortunately, there is a way around this. Instead of blindly trying to feed input to the program through `stdin`, we can instead attach the program directly to a "fake" terminal under *our* control. This fake terminal lets us provide not only outputs, but to change those outputs based on responses we see from the program we attach it to. Because this terminal is not a real physical console, we call it a `pseudoterminal`.

2.8.1 *pty, a pseudoterminal demo*

The `ptyhelper.c` program supplied in your starter code demonstrates how to create a pseudoterminal and attach it to a program you want to control. Chapter 3 explains how to use the helper code to create a program that controls another program.

In this part, you will create a file called `pty.c`. You should be able to compile this program by typing `make pty`, which should produce a binary file called `pty`. Since the supplied `Makefile` does not have a rule to do this, you will need to modify it to add a `pty` target. *Take note* that, when compiling with `gcc`, any program that makes use of the `ptyhelper` library must include the `-lutil` flag. The `-lutil` flag tells `gcc` to find several of the pseudoterminal functions in the `libutil.so` system library. [2]

Specification:

1. Write a program that attaches to `login0`. Call this program `pty`.
2. Call `exec_on_pty` with an appropriately constructed `argv`.
3. Manipulate the file descriptor returned by `exec_on_pty` using `read` and `write` system calls. [3]
4. `pty` should supply a single correct username and password (look in the `password.db` file) to `login0`, print `It worked!` when `login0` returns `ACCESS GRANTED`, and then quit with a non-zero exit code.

[2] `ptyhelper.c` calls the `openpty` C library function, which is not normally in `gcc`'s library search path. Appending `-lutil` tells `gcc` to search for the implementation of this function elsewhere. How did I know to do this? `man openpty` told me to do it.

[3] The `read` and `write` system calls can read and write arbitrary data, including binary data. This means that, if you're reading and writing strings, those calls do not know and they do not help you handle strings. Recall that C strings must always be null-terminated. Does `read` ensure that strings are null-terminated? Read `$ man 2 read` to find out!

2.8.2 *attack0: a "brute force" attack on `login0`*

In this part, you will copy and modify `pty.c` in a new file called `attack0.c`. You should be able to compile this program by typing `make attack0`, which should produce a binary file called `attack0`. You will need to modify your `Makefile` to add an `attack0` target.

Specification:

1. Write a program that "attacks" `login0`. The purpose of this program is simply to harvest usernames, which is often the first step in attacking logins. Call this new program `attack0`.
2. Your attack program should supply a randomly-generated up-to-8-character username at the username prompt. Make sure that you only generate alphanumeric characters. To generate a random integer, use the `rand()` C library call. You must also use `srand()`. See `man 3 rand` for details.
3. If `login0` prompts `attack0` for a password, you have successfully harvested a valid username. Since we don't know the password, if this happens, just provide the password `password` to the prompt. We don't care about the response just yet.

4. Your program should attempt to login up to 10,000 times. If it finds at least one valid username, it should print `SUCCESS` along with that username and quit, otherwise it should keep trying. If it tries 10,000 times without success, it should print `FAILURE` and quit.

5. *Optional.* The above is obviously a naïve method of harvesting usernames. For bonus credit, devise a better method and implement it, being sure to explain your method in a comment. Call the revised program `attack0a.c` and be sure to add a `Makefile` target for it.

2.9 *Part 3: `login1`, an improved login program*

In retrospect, it is obviously a bad idea to tell the user when they have successfully found a username. Instead, we should prompt for a username *and* password before checking the database.

In this part, you will copy `login0.c` into a new file called `login1.c`. You should be able to compile this program by typing `make login1`, which should produce a binary file called `login1`. You will need to modify the `Makefile` to add a `login1` target.

Specification:

1. Modify `login0`. Call this program `login1`.

2. The program should prompt the user to enter a username.

3. The program should prompt the user for a password.

4. If the username is in the password database *and* the entered password matches the stored password in the database *then* the program should print `ACCESS GRANTED` and terminate.

5. Otherwise, it should print `ACCESS DENIED` and go back to step 1.

The following is a sample `login1` session. Make sure your program behaves exactly like this:

```
$ ./login1
Enter a username: barowy
Enter a password: password
ACCESS DENIED
Enter a username:
...
```

2.10 *Part 4: `attack1`, a brute force attack on `login1`*

In this part, you will copy `attack0.c` into a new file called `attack1.c`. You should be able to compile this program by typing `make attack1`, which should produce a binary file called `attack1`. You will need to modify the `Makefile` to add an `attack1` target.

Specification:

1. Modify `attack0` to attack `login1`. Call this program `attack1`.
2. `login1` makes it hard to harvest usernames. Unfortunately, usernames are usually pretty easy to guess even if you can't harvest them. For example, in the CS department, most faculty usernames are the first character of their first name and their last name. Assume that you have already harvested usernames from another source, like a company directory. You may use the usernames (but not the passwords) from the `password.db` file for `attack1`. Create a username database for `attack1` called `usernames.db` and put the usernames in it.
3. Your attack program should randomly select a username from its username database and randomly-generate an up-to-8-character password.
4. Your program should attempt to login up to 10,000 times. If it finds a valid username and password combination, it should print "SUCCESS" along with the username and password and immediately quit, otherwise it should keep trying. If it tries 10,000 times without success, it should print "FAILURE" and quit.
5. *Optional.* Can you think of a better way to generate password guesses? For bonus credit, implement a better guessing procedure. Be sure to document your improvement in a comment. Call the modified program `attack1a.c`.

2.11 *Part 5: `attack2`, an even-better login program*

In this part, you will copy `login1.c` into a new file called `login2.c`. You should be able to compile this program by typing `make login2`, which should produce a binary file called `login2`. You will need to modify the `Makefile` to add a `login2` target.

How might you further improve `login1`? In this last section, you will implement an improvement of your own design. Be sure to document your improvement in a comment at the top of the source file.

2.12 *Reflection Questions*

Answer the following questions in a file called `PROBLEMS.md`, and submit it with the rest of your code.

1. For our username scheme, how many possible usernames are there?
2. For our password scheme, how many possible passwords are there?
3. When you ran `attack0` against `login0`, did it find any working usernames? If not, does it mean that `login0` is "secure"? Why or why not?
4. How might you modify `attack0` to find valid usernames faster? If you did the bonus, explain here how that modification worked.
5. `login1` is obviously better than `login0`. Can you quantify why? Think about your answers to the first two questions.
6. When you ran `attack1` against `login1`, did it find any working password? If not, does it mean that `login1` is "secure"? Why or why not?
7. What was your improvement in `login2`? Is it actually better? How do you know? Be sure to provide a mathematical justification for full credit.

2.13 *Development Tips*

Writing C can be a challenge. One way to deal with this is to log things that happen, and use that information to help debug. Because this assignment puts restrictions on what you consume and print, you *should not* use `printf` to log things. Instead, use a handy function like this one, which prints to a log file instead.

```
void  mylog(char *desc) {
        static int n = 0;
        FILE* file = fopen("DEBUGLOG.txt", "a");
        if(file != NULL) {
                n += 1;
                fprintf(file, "%d: %s", n, desc);
        }
        fclose(file);
}
```

Remember to be patient and systematic. If you don't understand your own code, you should consider setting it aside and starting over.

2.14 *Lab Deliverables*

By the start of lab, you should see a new private repository called `cs331lab01_login-{USERNAME}` in your GitHub account (where `USERNAME` is replaced by your username).

For this lab, please submit the following:

```
cs331lab01_login-{USERNAME}/
             attack0.c
             attack1.c
             console.c
             console.h
             database.c
             database.h
             login0.c
             login1.c
             login2.c (optionally)
             Makefile
             password.db
             PROBLEMS.md
             pty.c
             ptyhelper.c
             ptyhelper.h
             README.md
             usernames.db
```

where the `login*.c`, `attack*.c`, and `pty.c` files contain your *well-documented* source code.

It is always a good practice to create a small set of tests to facilitate development, and you are encouraged to do so here.

As in all labs, your work will be graded on the basis of *design*, *documentation*, *style*, and *correctness*. Be sure to document your program with appropriate comments, including a general description at the top of the file, and a description of each function with pre- and post-conditions when appropriate. Also, use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Whenever you see yourself duplicating functionality, consider moving that code to a helper function. There are several opportunities in this lab to simplify your code by using helper functions.

2.15 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab01_logins-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

2.16 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?

2. Anything else that you want to tell me?

3. Your name, for the bonus point (if you want them).

2.17 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass the auxiliary sensor" you should have
    written "bypass the primary sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too many to count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.