TIC4303: Lab 7

Introduction

Lab 6 experiments with memory errors which cause stack corruption and control flow hijacking. The `L7-ex1.c` program is a slight modification of the `stack-overflow.c` from the lecture notes. Note that not all exercises will use all the code in the sample program. For this lab, **you should ensure that your submissions work with the lab Linux setup**.

After this lab you will have investigated a simple form of compiler canary/cookie defenses for protecting the stack. In addition, you will have experimented with them as a memory integrity mechanism. You will also see how stack smashing control flow attacks work on a simple program. If you happen to use `gdb`, then you may also learn some `gdb` commands, such as commands to examine memory. Note that this lab only simulates a certain memory defence, and a sanitizer or the compiler would do it better.

Exercise 1: Showing a Stack Memory Error (Graded: 2 Marks)

Using `L7-ex1.c`, show the following:

1. Give an input X which will cause a stack overflow to occur in `buf` causing the process to crash with a "`segmentation fault`". You should compile using the `gcc` C compiler and use the option "`-fno-stack-protector`" which turns off the stack canary protection.
2. Recompile **without** the "`-fno-stack-protector`" option[1] (alternatively, you can use "`-fstack-protector`") and show that the stack memory error with X is detected by the `gcc` stack canary protection.

Explain your answers in the report and give input X as the file "`lab7-ex1-input.txt`".

Exercise 2: Protecting the Return Address (Graded: 6 Marks)

Modify `L7-ex1.c` to `lab7-ex2.c`. The objective of `lab7-ex2.c` is to detect when there is a buffer overflow of array `buf` and to prevent the return address from being corrupted when that happens.

Implement your own canary-like mechanism to detect overflow in `buf` and then call the function `cookie_error()` when a buffer overflow is detected because the canary value is not as expected. As this lab is only to show "in principle", the canary mechanism is only implemented in C. You can just modify the code of `f()` to implement your detector but to be **realistic**, do not change the code between `LINE1` and `LINE2` and hence the buffer data structure is also unchanged. Please ensure that the functionality of `f()` from the original code should be unchanged, you are only making the code more secure against a

[1] In some Linux distributions like Ubuntu 20, Stack Protector is turned on by default for `gcc`, however, in other Linux distributions, it could also be turned off by default. The stack protector is enabled in our lab setup unless you explicitly use the "`-fno-stack-protector`" option. Note that this shows you should not make assumptions about the defaults.

pontential attack by adding code to the rest of `f()`. You may also want to add other utility functions. Note that `main()` should be unchanged.

Your canary mechanism is only used if `my_protect` is `TRUE`, thus, you can turn it on or off on the command line with the `-p` option.[2] The `gcc` stack protector should be turned off, so that it does not interfere with your canary mechanism.

Please test your code with the default optimization level, i.e. `-O0` or no use of `-O`, as well as with `-O2`. Thus, you should test with both compilation commands below:

```
$ gcc -fno-stack-protector lab7-ex2.c
$ gcc -fno-stack-protector -O2 lab7-ex2.c
```

The normal `gcc` stack protector is designed to protect the stack meta-data rather than local variables. It may or may not detect buffer overflow in the local variable. Try to make your mechanism detect buffer overflow in array `buf`.

Describe your defence mechanism in the report and also how you have tested it. Note that input X from Exercise 1 may be program specific, as such may or may not be useful to test Exercise 2. Since the defence can be turned off, longer test input(s) should still cause a crash if `my_protect` is `FALSE`.

Exercise 3: Control Flow Hijacking (Graded: 6 Marks)

Please try Exercise 3 regardless whether you can get the hijack to work. It might be possible that you can only get a segfault rather than the desired hijack.

The experiment in Exercise 3 is to see if you are able to exploit the program in Exercise 1 (`L7-ex1.c`) by making the buffer overflow call `do_not_call()`. This is similar to the example discussed in lectures. Note that `do_not_call()` is not explicitly called anywhere in `L7-ex1.c`, hence, there is no path which connects `main()` to `do_not_call()` in the CFG.

Your submission will be tested on the `L7-ex1` binary which has been compiled with "`-g`". When you unpack the zip file, you will need to make `L7-ex1` executable, e.g.

```
$ chmod u+x L7-ex1
```

While there can be many ways to work out the attack, one way is to use the debugger `gdb`. For this reason, the test binary has been compilede with `-g`. If you are familiar with `gdb`, see the footnote.[3]

[2] In principle, the flag can be handled more elegantly, e.g. with an environment variable, but for simplicity, we just use a command line flag. For students interested, check `getenv()` (there are also other ways).

[3] The `gdb` documentation is at `https://sourceware.org/gdb/current/onlinedocs/gdb`. In particular, some sections more relevant are:
(i) stack frame information (`https://sourceware.org/gdb/onlinedocs/gdb/Frame-Info.html`);
(ii) machine code (`https://sourceware.org/gdb/onlinedocs/gdb/Machine-Code.html`) including the `disassemble` command;
and the (iii) `examine` command (`https://sourceware.org/gdb/onlinedocs/gdb/Memory.html`).
Use of `gdb` is not strictly necessary but it is convenient. gdb is often used to debug the program at the C or assembly language level.

The `printbytes()` function may be useful in your debugging but is also not necessary, it is provided simply as a convenience for students who want to use it. It can also be "`call`" from inside `gdb`.[4]

To allow for deterministic reproduction and for ease of attack, stack protector and PIE have been turned OFF in the `L7-ex1` executable.[5] Note that testing will have ASLR turned on but with PIE and stack protector turned off.

You should submit the input, in a file, which causes the control flow to `do_not_call()`. Your input will be tested in the following way (with filename adustments):

```
$ ./L7-ex1 < lab7-ex3.txt
```

Also write a report which explains how you obtain the input. In particular, explain: (i) the length of the input; (ii) how is the contents of the input obtained, a certain part of the input is arbitrary and another part is critical.

Hint: Addresses in `x86_64` Linux are 64 bit.

Submission

Submit the following for Lab 7: (please note that labs submissions are individual)

- Report for Exercise 1 and data: *Surname*-*StudentNumber*-`lab7-ex1.pdf` and *Surname*-*StudentNumber*-`lab7-ex1-input.txt`.
- Report for Exercise 2 and code: *Surname*-*StudentNumber*-`lab7-ex2.{pdf,c}`. where I can be {1, 2, ...}.
- Report and data for Exercise 3: *Surname*-*StudentNumber*-`lab7-ex3.pdf` and *Surname*-*StudentNumber*-`lab7-ex3.txt`.

Submit all files above in a single zip, e.g. `yap-U123X-lab7.zip`, to the appropriate LumiNUS folder.

[4] `https://sourceware.org/gdb/onlinedocs/gdb/Calling.html`

[5] Reproduction simplifies Exercise 3 and also grading. Note that, in principle, a probabilistic attack can still succeed against stack protector and ASLR.