TIC4303: Lab 6

Introduction

In Lab 2, we saw examples of memory errors which can occur with pointers to data (data pointers), e.g. segmentation fault in Unix/Linux. In this lab, we experiment with code where memory errors can arise with function pointers (code pointers) rather than data pointers. This can lead to segfault but also other kinds of exceptions.

Basically, a function pointer is a pointer which points to a function, i.e. code, instead of data. A function pointer can be called just like a function including passing arguments and getting a return result. Some uses of function pointers are for writing generic functions, sometimes in a kind of Object-Oriented style, and also for implementing callbacks. If you are not familiar with function pointers in C, the following link give an introduction:

`https://www.cprogramming.com/tutorial/function-pointers.html`
`http://crasseux.com/books/ctutorial/Function-pointers.html`

The syntax for function pointers is slightly different from normal pointers, e.g. `L6-function-ptr.c` declares `f` to be a function pointer as follows:

```
int (*f)(int);
```

Calling `f` is written as

```
(*f)(argument);
```

Rather than writing as a pointer dereference (`*`), it can also be written in normal function call notation – this is used in `L6-function-ptr.c` as:

```
f(argument);
```

You should also contrast the kinds of memory errors in this lab with Lab 2. The kinds of techniques used in this lab are also used in JIT compilers such as JavaScript compilers in the browser, e.g. Chrome, which turn data into code and then call it, e.g. Exercise 3. While Lab 6 is meant to be relatively straightfoward, real bugs may be more complex with a mix of data and code execution errors. Note that attacks which try to get code execution may use variants of this lab. This lab also illustrates non-deterministic bugs.

After this 2 and 6, students should have an idea of some of the effects of memory errors in Unix or Linux. Namely, arbitrary data corruption and code execution. Of course, there could also be information leakage due to an attack but the focus in Lab 2 and 6 is on integrity of the software. Although our experiments are on Linux, similar problems can happen in Windows.

Exercise 1: Function Pointers in `qsort` (not graded)

A sample program which illustrates the use of function pointers is `L6-qsort.c`. To understand why `qsort()` uses function pointers, it is because `qsort()` is a generic sorting routine `qsort()` for any kind of array data, as such, different data types require a different way of comparing them, which is supplied by a user provided comparison function, `compar()`. In `L6-qsort.c`, we can see to different ways of sorting by changing the comparison function. Exercise 1 is only an example to illustrate the use of function pointers in C. I would recommend trying it out to understand the use of function pointers (if you are not familiar with function pointers). Note: in C++, there are virtual calls which is a more controlled way of using an analog of function pointers with C++ classes.

Exercise 2: Exploring Function Pointers (Graded: 18 Marks)

This exercise is graded. Read, Compile and Run: `L6-function-ptr.c`. (Note that Exercise 2 should be run in our Ubuntu setup, just `gcc` compiler defaults are sufficient).

The program uses `mmap()` which is one way of getting new memory in Unix. You may want to see where in the memory map, e.g. in `/proc`, the chunk of memory comes from. On x86 Linux, `mmap()` works by using virtual memory and the hardware support for page tables. Also read the man page to understand `mmap()` and reasons why `mmap()` may fail.

Some parts of the code are commented out. You will need to experiment with uncommenting and recommenting back some of the code. The program consists of a number of experiments. For each experiment below:

a. explain what happens
b. **elaborate on the reason** for the result.

The questions for Exercise 2 are as follows, note that each question may have sub-questions:

1. Uncomment the block of code with "`f = NULL;`", test and undo the changes.
2. What is the difference between the function called by `f()` at the line with comment LINE1 with `f()` at the line with comment LINE2. Is the result the same? Explain the similarities and differences between `f()` and `addnum()`.
3. What happens when the `mmap` lines are swapped and uncommented appropriately? Explain the reason for the difference. After testing, the changes should be undone.
4. What happened to `f()` at the line with comment LINE3? Explain the result.
   While the code here is intentional, you can consider the general case. Suppose there is a memory error, what could happen with code using function pointers?
5. What happened to `f()` at the line with comment LINE4? Carefully explain the result.
6. What does the `memcpy` do? What happens when `f()` (at LINE5) is called? Is this error different from the segmentation fault (in the `f = NULL`) experiment and why is that the case?

Submit a report with your explanation of each question above in **numerical order**. Good explanations which explain the experiment corresponding to the question more comprehensively can get more marks.

Hint:
It can be useful to use the debugger, e.g. `gdb`, for this exercise. The `gdb` documentation (long) in HTML form can be found at,
`https://sourceware.org/gdb/current/onlinedocs/gdb`
In particular, how to deal with machine code at
`https://sourceware.org/gdb/current/onlinedocs/gdb/Machine-Code.html`
Note that `gdb` is not the only debugger but it is quite standard.

Submission

Submit the following for Lab 6: (please note that labs submissions are individual)

- Report for Exercise 2. It should be in PDF format with the filename in the format

Surname-*StudentNumber*-`lab6-ex2.pdf`.