TIC4303 Lab 2

Background

The purpose of this lab is to investigate memory errors in Linux which occur because of the operating system, e.g. "*segmentation fault*". It also looks at the memory layout of a Linux process.

A reminder about labs. In general, we have labs so that students can try out practical experiments and hands-on exercises. Lab exercises are mainly straightforward (but may require more work if you are not familiar with C). A Lab can have a mix of graded and non-graded exercises. Each exercise has a **specific "learning purpose"**. You should do all exercises **regardless** of whether they are graded or not.[1] The intent is that after doing the lab, you should understand the "learning purpose" of the lab. We will usually discuss the exercises in a subsequent lab. **All labs should be run on our Ubuntu lab setup from Lab 0**.

In general, labs require programming but not a large amount. Lab 2 only requires understanding C and how C programs run in Linux (except for the bonus section which is more complex). *If you do not know how to use C in Linux, you can approach me (the assumption is that you know basic C or will pick it up quickly).* Tutorials 1 and 2 also contain a C refresher mainly focusing on pointers and basic types.

Lab 2 has graded and non-graded exercises. Please submit **only** the graded portion.

Exercise 1: Exploring Memory Regions in Linux (not graded)

Every process in Linux can be thought of as being composed of a number of memory regions. Most of these regions can be thought of as being of a particular type, e.g. text (the code), heap, stack, etc.

Read the code for the `L2-mem-region.c` program. It prints out the addresses of objects in memory. It also makes use of the special `/proc` filesystem in Linux. The directories which are numbered in `/proc` correspond to processes and are numbered by the process ID. Various information about a process can be found in `/proc` (see the `proc` man page and how `L2-mem-region.c` outputs the `/proc` information). Compile and run `L2-mem-region.c`, e.g.

```
$ gcc -o memory-region -no-pie L2-mem-region.c
$ ./memory-region
```

On Ubuntu 20, `gcc` by default generates a *Position Independent Executable* (PIE) to support the use of ASLR.[2] In order to simply the memory layout, we will disable the use of PIE. Thus, Lab 2 changes the `gcc` default behavior with the "`-no-pie`" option.[3]

To understand the `proc` filesystem, see the `proc(4)` man under `/proc[pid]/maps`.[4] By reading the

[1] Ungraded exercises are simply because there isn't something to grade and does not mean its optional.

[2] We will cover Address Space Layout Randomization later in the module. Note that in older versions of Ubuntu, PIE was not the default. It is also possible that some Linux distributions may not be using PIE as PIE has some cost to it.

[3] `-no-pie`: Do not produce a position independent executable.

[4] The `proc(4)` man page is quite long. For this question, you can just focus on the `maps` portion. However, a wealth of information on the process can be found through `proc(4)`.

program and correlating the results of the execution, answer (for yourself) the following questions:[5]

- Why are the addresses printed in hex rather than decimal?
- What is "`rostr`", and the difference with "`&rostr`"?
- Which direction does the runtime stack grow?
- Explain which memory regions correspond to: code (text), globals, stack and heap. Note that the answer can vary depending on the program.
- Each region has permissions (`rwx`, can ignore `p`) – what is the rationale for a region having a particular set of permissions.
- Some regions are associated with files and some are not? Can you explain the distinction?

Note: you can ignore the following regions which are Linux kernel specific: `vvar`, `vdso`, `vsyscall`.

Run the program multiple times to see if the addresses change. Notice which addresses change and which don't. We will return to this observation when we cover memory defenses (Hint: ASLR).

Try to see if you are able to answer the above questions yourself first. As they are ungraded, we will discuss in a subsequent lab. Please try Exercise 1 before Exercise 2 as they are related.

Exercise 2: Exploring Segmentation Fault in Linux (Graded: 12 Marks)

Exercise 2 is **graded**.[6] The program you should use is `L2-memerr.c`. Please Read , Compile "`L2-memerr.c`" (with the `-no-pie` option as in Exercise 1) and Run the code:[7]
The program takes one option on the command line from 1 to 8, e.g.

```
$ ./a.out 2
```

Some choices will lead to a memory error and some will not. Memory errors are sometimes - but not always - caught by Linux and the default action is a runtime exception. In Unix, a runtime exception gives rise to a *signal* in the process, see the man page `signal(7)`. The default action to many signals is to crash.[8] This exercise explores some of the memory errors which lead to a "*segmentation fault*" (segfault) in Linux.

For each choice (1 to 5 and 8) in `L2-memerr.c`, explain the following:

a. What is the behavior?
b. Is there a memory error? If there is a memory error, explain the rationale for the memory error. If there is no memory error, explain why it isn't a memory error. **You should also explain the reason for the result and what happened**. Your explanation should be clear, refer to the code and justify the behavior observed.

Note that an explanation simply saying it segfaults is not sufficient as a reason—the explanation needs to go deeper and be more specific. Your explanation should also consider the semantics of C, what the code is trying to do and also what is the choice made by Linux (the OS). Some of the choices in the code involve specific constants, try to explain the reason for the particular constant in the cases where the value is important to the specific behavior (note: the specific choices are not highlighted as this is an

[5] Exercise 1 is not graded, so you should try to explore and answer the questions yourself. The philosophy of this module is to learn through your own experimentation and augmented by lectures and reading material.

[6] A note on the marks, marks are relative to the lab and the numeric value is for grading convenience but cannot be directly compared between different labs. For example, labs are not simply summed to get a final mark for the total lab marks.

[7] `gcc -no-pie L2-memerr.c`

[8] Note that not all signals are due to errors.

exploratory part of the exercise). **Choices Six and Seven are not required but you can investigate (by uncommenting the code) if you want but do not submit them**.

You may find to get a good understanding of the behavior may require some enhancements to `L2-memerr.c`. You can also use the debugger, e.g. `gdb`, and compile for debugging with `-g`. Please be careful that the question is on the original `L2-memerr.c` and not any changed version.

Submit a report with the above explanations for each choice. **Order** your choices in sequence from One to Five and Eight.

Extras: (this is not part of the report)
Try compiling with the `gcc` option `-Wall` which turns on most (but not all) warnings. Did any warnings occur? Note there are some type casts in the code, understand why they are there. You can experiment with what happens with `-Wall` once the casts are removed. Be careful to not change how the program works. Does the program behave the same?

Comments:
Some comments on runtime exceptions. Although we only use Linux, runtime exceptions from memory errors also occurs in other operating systems. In Windows, a form of segmentation fault is *exception access violation* or *general protection fault*, but there are also others, such as when the Windows kernel gets a "blue screen". In Java, many runtime exceptions are possible, e.g. `NullPointerException`, `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`, etc.

Submission

In general, make sure that reports contain the following details: name, student number, email. Reports can be in any format but **must be a PDF file**. Do not submit DOC/DOCX files.

Labs are individual work and Lab 2 is an **individual submission**. Submit the following for Lab 2:

- Report for Exercise 2 in PDF format and the filename in the (usual submission) format:
  Surname-ID`-lab2-ex2.pdf`
  e.g. `yap-A123X-lab2-ex2.pdf`.

Please check that you are using the correct file name convention.