# CS-E4760: Platform Security (2023)
# Exercise 3: Run-time security

## Part 1: Theory

Answer the following questions:

1. **Undefined behaviour.**

   Secure programs generally avoid triggering undefined behaviour, as this means that the language standard allows anything to occur. Each time an action triggers undefined behaviour, it could be ignored, or it could hand control of the user's machine to an attacker: both are equally legal.

   a. If an application in its source code representation triggers undefined behaviour, can we conclude that it is insecure when run? Why [not]? [2 points]

   b. Undefined behaviour in a language standard is often contrasted with *implementation-defined behaviour*, defined in Section 3.4.1 of the C99 standard as "*unspecified behavior where each implementation documents how the choice is made*".

      Must developers avoid implementation-defined behaviour to develop a secure program? Why [not]? [2 points]

   c. Formal methods can be used to mathematically prove that one representation of a program is a *refinement* of another (that is, that its behaviour is equivalent after transformation from a higher-level form to a lower-level form).

      Can we prove the correctness of a program by using formal methods to show that each step in the program transformation model described in the lecture slides is a refinement of the previous one? Why [not]? [2 points]

# Part 2: Vulnerabilities

Write the following code:

2. **Triggering undefined behaviour.**

   For each of the following kinds of undefined behaviour, write a program in C that triggers it and demonstrates how it can cause data corruption (e.g. by crashing or modifying data). Build your programs in a Docker container for ease of reproduction.
   a. Stack array overflow [4 points]
   b. Type confusion [4 points]

# Part 3: Defences

Attempt the following steps, and answer the associated questions.

3. **Address space layout randomization.**

   **a.** Modify one of your programs from Question 3 to corrupt a function pointer so that it will contain a user-supplied value from the terminal; then, call the function pointer.

   Add a function `attack` to the program that produces some output, so that you can see the results of your attack. [5 points]

   **b.** Compile the program with the `-fno-pie` option to prevent the use of ASLR. Write a script that will print an input for the program that causes your new function to be called in place of the function pointer's original target. [5 points]

   Hints:
   - You may find the `nm` or `objdump` programs to be useful.
   - You may need to pay attention to the endianness of your platform.

   **c.** Compile your program with `-fpie -pie` instead of `-fno-pie`. Verify that your original attack no longer works.

   i. Why does the attack no longer work? [1 point]

   ii. What other information does the attacker need in order to carry out the attack? Modify your vulnerable program to leak this information, and demonstrate a new attack. [5 points]

   NB: Leaking the address of `attack` directly (e.g. by printing it with `printf("%p\n", attack)`) is too trivial and will not receive any points. The address of anything else, or any other information, is fair game.

# Submission

Submit to MyCourses your answers to the questions above as a **text or PDF file (not ZIP)**. Attach a copy of your programs and scripts, along with the Dockerfile used to build them.