UGA **M2 CSI**

Lab 1: about software vulnerabilities
(starting Wednesday October the 1st)

Indications:

- You can work either with your own laptop or with the computers of Room F103;
- The material required for this lab can be downloaded here from the Moodle course web page;
- You are (of course) free to look for information you need in the web, in particular the **slides and resources** provided in the course web page ..... but querying some help from an LLM assistant on these topics would definitely be a waste of time (and energy!).

Objectives:

The ultimate goal is to **answer all the exercises**... However this objective **is shared** between all of you. In practice you can split your class into groups, each groups working on distinct exercises (and you can get help from one group to another). Note also that your answers should be precised enough, you are definitely **not asked** to do all the exercises within half a day !

Write your (common) answer on the dedicated "drop zones" associated to each exercise. We will discuss together these answers later in the week.

Exercise 1: "unexpected" code behaviors

Answer the questions provided in file ***exercise-1.md*** ...

Exercise 2: CVE 2025-46713

Browsing the the web, e.g., the site CVE.org, write a brief presentation of CVE-2025-46713, telling:

- which product is concerned, with which programming language(s);
- the nature of this vulnerability (CWE numbers, severity score), with some explanation of the faulty code (do not hesitate to reproduce this faulty pattern, to better understand it);
- how it could be exploited by an attacker, what could be the gain obtained
- how to patch this vulnerability (looking for instance at the CERT secure coding standards guides)

Do you think such a vulnerability could occur in a Java code? Explain you answer.

Exercise 3: Winloose

Look at the source code of the C program ***winloose.c***. Note that this program takes as input two integer arguments on the command line (`argv[1]` and `argv[2]`).

Question 1
Compile this programm with gcc using the following command:

```
gcc -fno-stack-protector -o winloose winloose.c
```

(Have a look at the gcc manual to know the meaning of `-fno-stack-protector`).

Question 2
Execute it with some random arguments, e.g.:

```
./winloose 5 10
./winloose 2 17
```

This program may lead to several possible behaviors:

- print `You loose`
- infinite loop
- crash

Find the program input allowing to print `You win` !
Explain each of theses different result you get, drawing the execution stack.

Hint: to you can try to find the addresses of the local variables either using `gdb`, or modifying the source code to print them, or even using a disassembler like `objdump` ...

Question 3
Compile now the C program winloose.c with the ``stack protection'' enabled:

```
gcc -fstack-protector -o winloose winloose.c
```

What do you obtain now when running this new executable code with the inputs you provided for question **Q2**? Why ?

Exercise 4: Optimize

Look at the source code of ***optimise.c***.

Question 1
Compile it, execute it and explain the result obtained in each following case:

- no option:

```
gcc -o optimise1 optimise.c
```

- optimization option:

```
gcc -O2 -o optimise2 optimise.c
```

- overflow detection option:

```
gcc -fno-strict-overflow -o optimise3 optimise.c
```

- optimization and overflow detection option:

```
gcc -02 -fno-strict-overflow -o optimise4 optimise.c
```

(Look at the gcc manual to know the meaning of `-O2` and `-fno-strict-overflow`)

Question 2
Propose a solution to make this function secure. You can use the following rule.

Exercise 5: Open a shell (1)

The objective of this exercise is to show how a use-after-free vulnerability can be exploited by an attacker to get *an arbitrary code execution*.
The commands to be executed in the following questions can be copy-pasted from the file ***exploit-uaf2.txt***.

Question 1
Have a look at the file ***uaf2.c*** to spot the occurrence of a *use-after-free*.
Compile this file using option `-z execstack` to set the stack as "executable"' (which is not a default option, we will come back to this point later in the course):

```
gcc -z execstack -o uaf2 uaf2.c
```

Question 2
What do you obtain when executing the following command: `./uaf2 foo`
Explain why you get this result ...

Question 3
Execution of `uaf2` can be hijacked by an attacker and may lead to an arbitrary code execution by giving as input a sequence of processor instructions (called a *shellcode*). An example of such a shellcode, allowing to **open a shell** under Linux x86/64 is given for instance here:
https://www.exploit-db.com/exploits/46907

Run ***uaf2*** with this shellcode as a command line argument (can be copy-pasted from file ***exploit-uaf2.txt***):

```
./uaf2  $(printf "\x48\x31\xf6\x56\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\
x54\x5f\x6a\x3b\x58\x99\x0f\x05")
```

Question 4
Re-compile now your code using Adress-Sanitizer in order to enforce memory safety:

```
gcc -g -fsanitize=address -z execstack -o uaf2 uaf2.c
```

Check that the previous ``exploit'' does not work anymore ...

Exercise 6: Open a shell (2)

Look at the source code of the C program ***exec_shell.c***. This program takes as input one directory name and prints its content (like the Linux `ls` command).
Compile this program with gcc:

```
gcc -o exec_shell exec_shell.c
```

Run it: `./exec_shell /tmp`

If the argument string is too long, then an error message is printed and the user is requested to enter a character string.

Question 1
Explain why this program is **vulnerable**.

Question 2
Find how you can use this program to execute any shell command of your choice (e.g., `/bin/sh`, `xcalc`, etc.) using the two following attack patterns:

1. shell command injection
2. use-after-free vulnerability

Exercise 7: a vulnerable python library

The file ***code.py*** contains a tiny library simulating an e-commerce server able to receive customer orders (either product commands, payments, or refund requests).
Its main functionality is to check if a given sequence of orders is valid, and to print the resulting customer balance (typically wether it is null, positive, or negative).

The file ***test.py*** gives some examples of interactions with this library (creating sequences of orders and checking them).

Question 1.
The current version of this library is vulnerable in the sense that it is possible for a user, ***using only some sequence of valid orders***, to fake the computation of the resulting balance (earning money in a dishonest way :-).

The objective is therefore to change the content of file ***test.py*** to exploit these vulnerabilities, either by getting for free a non-free product, or by ending with a greater balance than the regular one. To do so, **you should not** modify the content of ***code.py***.

Hint: think about using **floating-point** values in Python, which may lead to overflow or precision errors ...

Question 2.
Update ***code.py*** in order to get rid of the vulnerabilities you found at Question 1.

Exercise 8: a vulnerable C library

The file ***code.h*** contains a (very) lightweight C library allowing to create and manage a set of user accounts.

Each user account consists in:

- a user id (a strictly positive integer), which uniquely identifies a user;
- a boolean value telling if these user is "administrator" or not;
- a user name (possibly non unique, we don't care)
- a sequence of dummy user information called "setting", stored as pairs (index,value), where both index and values are integers.

File ***test.c*** is an example of code using this library.

Question 1
The first objective is to pawn this library by successfully promoting as admin a user initially created as non-admin (i.e, *privilege escalation* vulnerability).

To do so, you should ***only*** use the API functions provided in ***code.h***, as they are, without changing them nor adding anything else in this file!
In practice you should therefore **only** change the contents of ***test.c***.

Rk: succeeding in downgrading as non-admin an admin user is a valuable exploit as well!

Question 2
Update (in a minimal way) the content of **file.h** to correct the vulnerability, since preserving the initial functional behavior. **Hint:** think about possible **buffer overflows** ...