UE SCLAM - Software Security

TP "Security weaknesses in C"

Before you start :

Copy the tar file `Examples.tar` availabe on the Moodle web page and and untar it into a directory of your choice :

```
tar -xvf Examples.tar
```

Exercise 1 Optimise

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

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

   no option :
   ```
   gcc -o optimise1 optimise.c
   ```
   optimisation option :
   ```
   gcc -O2 -o optimise2 optimise.c
   ```
   overflow detection option :
   ```
   gcc -fno-strict-overflow -o optimise3 optimise.c
   ```
   optimisation and overflow detection option :
   ```
   gcc -O2 -fno-strict-overflow -o optimise4 optimise.c
   ```
   (Look at the `gcc` manual to know the meaning of `-O2` and `-fno-strict-overflow` ...).

2. Propose a solution to make this function *secure*. You can use the following rule : `https://www.securecoding.cert.org/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow`

Exercise 2 WinLoose

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

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` ...).

   Execute it with some random arguments :

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

   This program may leed to several possible results :
   — print `"You loose"`
   — infinite loop
   — crash
   — etc.
2. Explain each different result you get, drawing the execution stack.

3. Find the program input allowing to print `"You win"` !

4. Disassemble this program using the `objdump` command [1] :
   ```
   objdump -S winloose
   ```

   Look at the assembly code of functions `<main>`. Try to understand this code, and to retrieve the offsets in the stack of the local variables.

   Indication : in this 64-bits architecture registers `ebp` (frame pointer) and `esp` (stack pointer) are called `rbp` end `rsp` ...

   You can have a look to the file `DemoDisassembling.pdf` to see a concrete example.

5. 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 1 ?

6. Disassemble this program using the `objdump` command :
   ```
   objdump -S winloose
   ```

   Look at the assembly code of functions `<main>` to retrieve how the stack protection mechanism is implemented.

1. alternatively you can use `IDA` if it is installed on your machine ...

Exercise 3 Exploiting a Use-After-Free

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`.

1. Have a look at the file `uaf2.c` to spot the *use-after-free*.
   Compile this file using option `-z execstack` to set the stack as "executable" (which is not a default option).

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

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

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 a command line argument (see file `exploit-uaf2.txt`).

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 ...