Software Vulnerabilities: Exploitation and Mitigation

—

Lab 2

Prof. Jacques Klein & Pedro Jesús Ruiz Jiménez
(inspired from Prof. Alexandre Bartel's course)

2 Architecture and Use of GDB (47 P.)

2.1 Setup Labs Environment

2.1.1 Install tools

Install *VirtualBox* or *QEMU* (required for MacBooks with M1, M2, or M3 chip) and *Vagrant* by following the instructions here (*VirtualBox*) or here (*QEMU*) and here respectively. Additionally, you have to install *Git* using the instructions here.

Additionally, for *QEMU* users, you have to install two *Vagrant* plugins by using the following commands:

```
$ vagrant plugin install vagrant-vbguest
$ vagrant plugin install vagrant-qemu
```

2.1.2 Clone resources

This is the private token needed to clone the labs resources:

```
github_pat_11AKN732Q0qHLNFj2KZi3h_YVKBI0s2XKz1atbbemNZAlsJ4jVeUG
↪  dGC5F9HcpeBGhREZMG32KAPoVqYm8
```

Use this command to clone the labs resources into your local machine (make sure to replace `<token>` with the above token):

```
$ git clone https://PJRJ:<token>@github.com/PJRJ/SVEM.git
```

Note that you can simply copy this link after the `git clone` command.

2.2 Launch Emulated Environment

On your host machine, go to the `Lab02` directory found inside the repository that you just cloned. Then, create the virtual machine (vm) and connect to it by using the following commands:

```
$ vagrant up
$ vagrant ssh
```

Other important vagrant commands that you might need are:

```
$ vagrant halt  # Stops vm
$ vagrant reload  # Resets vm
$ vagrant destroy  # Destroys vm
$ vagrant provision  # Rerun provision
$ vagrant global-status  # Status about vms
```

Finally, once you are connected to the vm, you'll need to move to the work directory: `lab`

Question 2.1 Can you connect to the virtual image? Provide a screenshot. 2 P.

2.3 First Code

The following program is `hello.c`:

```c
#include <stdio.h>

int main(int argc, char** argv){
    printf("Helloworld!\n")
}
```

Question 2.2 What is wrong in the above code? Describe the syntactic and semantic errors. 2 P.

Correct the above code snippet in file `hello.c` (in your *local* machine or in the *virtual* one), compile it, and run it using the following commands:

```
$ gcc -Wall -o hello hello.c
$ ./hello
```

Question 2.3 Provide a screenshot of the terminal in which you have successfully compiled and run the program. 1 P.

The following program is `first_input.c`:

```c
#include <stdio.h>
#include <string.h>

int main (int argc, char** argv){
  printf ("argv[1]: %s\n",argv[1]);
  return 0;
}
```

Compile it and run it to check that it works. Then, use the following command to start the **gdb** debugger on the `first_input` program:

```
$ gdb first_input
```

It is highly advised to have a look at the gdb cheat sheet and keep it close to you from now on. This will be your best friend for the following labs :-).

To run the program from within gdb use the following command:

```
(gdb) run programArgumentsHere
```

2.4 Program Location

The program is already loaded in memory. To see where, execute the following command:

```
(gdb) x/20i main
```

- 20: number of elements to display
- i: type of elements to display, here instructions (could also be `b` for byte, cf gdb cheat sheet)

Question 2.4 At what address is the code of the main function written? Provide a screenshot. 1 P.

To see more and around, one could explore with `x/30i main-15` (change the numbers and see how it evolves).

Question 2.5 In what language is the code you see in gdb written? 1 P.

Question 2.6 Which instruction in the `main` function is related to printing the inputs? 1 P.

Other than giving the name of a function, one could provide an address as follows (where Z is the address):

```
(gdb) x/XY 0xZ
```

Question 2.7 Using a command of the style **x/$\alpha\beta$ 0xγ** or **x/$\alpha\beta$ name_func** (as explained above) ask gdb to show the assembly instructions of the function located just above the `main` function. Provide a screenshot. 1 P.

Question 2.8 Comparing the results given by **x/αi main** and **x/αb main** make a correspondence table between hexadecimal opcodes (operational codes) and assembly instructions for the `main` function. 4 P.

Your correspondence table should look like the following:

0x55	push %rbp
0x48 89 e5	move %rsp,%rbp

2.5 Breakpoints

Type the following commands to set a breakpoint at the main function and print the list of breakpoints:

```
(gdb) b main
(gdb) info breakpoints
```

Question 2.9 Is the breakpoint set exactly on the first assembly instruction of the `main` function? If not, list the instructions executed nonetheless (aka before the breakpoint). What is the role of these instructions and can you guess why the breakpoint is set **after** them? 4 P.

Execute the following command:

```
(gdb) run hello
```

Question 2.10 What happens? What is the purpose of a breakpoint? 2 P.

To finish the program, type:

```
(gdb) continue
```

Question 2.11 Type `(gdb) x/42i main` and select the address of an instruction. Type `(gdb) b *0x@address_selected`. We here set the a breakpoint at a precise address in the code. Type **run hello** and hit every breakpoint by typing `(gdb) continue` as many times as required. You can also type `(gdb) ni` to execute instruction after instruction. Provide a screenshot of the program that has finished executing. 2 P.

Re-run until reaching a breakpoint and type:

```
(gdb) info reg
```

Question 2.12 3 P.
a) What do you see? Provide a screenshot.
b) What does the value in **rip** represent? Can you guess/explain its meaning? Is its value logical?
c) What does the value in **rsp** represent? Can you guess/explain/find in the course its meaning?

2.6 Reusable input

Rather than manually typing the same input several times, one useful skill is to use (from within gdb) a script where the input is generated.

The following shell script is `inputGenerator.sh`:

```
printf "\xAA""\xAA""\xAA"
```

Call the script from gdb (using the back quotes) by typing:

```
(gdb) run `bash inputGenerator.sh`
```

Question 2.13 If you want to generate the string **hello** in ASCII, what hexadecimal codes should you write instead of "\xAA" "\xAA" etc.? 2 P.

Question 2.14 Update `yourInputGenerator.sh` accordingly, run it, and provide a screenshot of this working altogether. 1 P.

2.7 Input inspection

Read this and this

Question 2.15 In the context of a Linux process, what is the stack ? Does it grow up or down ? 1 P.

Follow this instructions:

1. Open a new terminal
2. Move to the `Lab02` directory
3. Connect to the vm using `vagrant ssh`
4. In one terminal enter `gdb first_input`
5. In the other terminal enter `pgrep gdb` and get the `PID`
6. Enter `cat /proc/PID/maps` (replace `PID` with the output from step 5)
7. In the gdb terminal set a breakpoint **before** the instruction calling the printf function.

Question 2.16 Where is the stack located? (Help yourself with any register that could be related to the stack) When hitting the breakpoint, where does the stack starts? Where does it stop? Provide a screenshot of the registers. 2 P.

Question 2.17 Explore the stack and try to find your input (Using **x** as unit, instead of **b** or **i** in x/Yi or x/Yb)? At what address? Provide a screenshot. (You can change the input to make it more obvious) 2 P.

2.8 The Call Stack

To `first_input.c` we added a function called **new_function** (printing something else) making `main` call this function at some point. This can be seen in `first_input_mod.c`.

Question 2.18 4 P.
a) Once compiled and launched with gdb (`gdb first_input_mod`), where is the code of `new_function` located in memory?
b) From what instruction of the `main`, will the code jump to the `new_function`?
c) What instruction, from `new_function` jumps back to the main function?
d) Explain how the program knows where (i.e. at which instruction) to go back in main?

Question 2.19 After setting a breakpoint in the new function compare values rbp and rsp when reaching a breakpoint in `main` and a breakpoint in `new_function`. Draw the corresponding stacks, as done on slide 28 of the course. 4 P.

2.9 Altering Values in Memory

The following program is `change_path.c`:

```c
#include <stdio.h>
#include <string.h>

int main (int argc, char** argv){
    int c = 0;
    if (1 == c) {
        printf("Wait, that was not an offering for the gods?");
    } else {
        printf("Troy shall never fall.");
    }
    return 0;
}
```

Question 2.20 What should be the output at every execution of the program? 1 P.

Compile the code using the option `-O2`.

Question 2.21 Can you explain the differences in the machine code of the main between using -O2 and not using it? 1 P.

Recompile the code **without optimizations**, run `gdb change_path` and set a breakpoint at the beginning of the `main` function. Run the program until this first breakpoint and display the `main`.

Question 2.22 What are the 2 assembly instructions enabling to decide what branch to execute (aka what string to print)? How does it work in our case? 3 P.

Now we want to get the first branch printed, even though `c` is set to the wrong value. Set a breakpoint on the `cmpl` instruction, and fetch (in the stack) where is the value written. Use the following command to change the value to 0x1 at the corresponding address:

```
(gdb) set *0xaddress=0x00000001
```

Verify if the value has been changed in the stack and type the following instruction to verify that we have hijacked the course of the execution:

```
(gdb) continue
```

Question 2.23 Provide a screenshot demonstrating the rerouting. 2 P.

Exit and **stop** the emulated environment by using the following commands:

```
$ exit
$ vagrant halt
```

Note on plagiarism

Plagiarism is the misrepresentation of the work of another as your own. It is a serious infraction. Instances of plagiarism or any other cheating will at the very least result in failure of this course. To avoid plagiarism, always cite the source from which you obtained the text.