Software Security & Secure Programming
Lab session 2: buffer overflow

Before you start:

To start the virtual machine: `lance-vm-5MMMSSI.sh`
Virtual machine account: `securimag / 123456`

You should write a **report** on the work you did during this lab sessions and give it back during the next class on **tuesday 25/10**. Only the first exercise below is mandatory.

1. Writting an exploit for a buffer overflow vulnerability (mandatory)

The program to exploit is named **bof** (on the VM desktop). First we explain how to find the bug and then how to exploit it.

1) Run this program (with a short string argument).

2) Run it again with a **long** string argument (see appendix "Using python") to produce easily large strings. The goal here is to get a program crash ...

3) Open this program with IDA (see appendix "Using IDA") and look briefly at the disassembled code. Just using this information can you easily locate the bug ? Reading the code of function `main` you can see that it calls a function `vuln`, which calls itself the function `strcpy` ...

4) Start gdb on this program (see appendix « Using gdb »). Under gdb, run this program with a long input leading to a crash (e.g.. "AAA ... A"). Why does this crash happen?
Clue: Knowing that the Ascii code of 'A' is 0x41, look at the content of the EIP register after the crash (why is this register used for?).

5) Using gdb you can see in more details how the program run. Put a breakpoint before the call to function `vuln` and another one when this fonction exits (on its `ret` instruction). Execute one instruction more after the breakpoint (`stepi`). What do you observe?

6) The next step is now to find which part of the input string controls the content of register EIP.
A first solution to do that is to increment the size of the input string as long as EIP is not controlled .
A more efficient solution is to use the python script « pattern.py » (see appendix "using pattern.py").

7) Your are now able to contrrol the program execution flow by assigning to register EIP a value of your choice. The next step is therefore to hijack the program bof (making it executing some unforeseen piece of code). To do so, we are going to use a *shellcode* written in assembly code. You can find several examples of shellcodes on the Internet, for instance at the following URL:
http://shell-storm.org/shellcode/).
For this lab session you can choose this one, those purpose is to start /bin/sh shell interpreter:
http://shell-storm.org/shellcode/files/shellcode-827.php

If the shellcode is given as a program argument it will be copied inside the array where the buffer overflow takes place, and hence inside the program execution stack. The goal is then to re-direct the execution flow by overwritting EIP with the "correct" shell code address in the stack. Your input string should then look like:

AAA ...AxxxxYYYYYYY...

where AAA are padding characters, xxxx is the shell-code address, and YY...Y is the shell-code itself. To find this exact address in the stack a solution is to use as input string:

AAA ...ABBBBCCCCCCC

Then, by dumping the memory from register ESP (the stack pointer) when the crash occurs, you can find the address of the first character C in the stack. This will be the address of your shell-code. When you know this address you can now execute your shell-code under gdb ...

Indication:

- the target machine architecture is little-endian, which means that addresses should be written from right to left (e.g. : `0xabcdefgh` should be written `\xgh\xef\xcd\xab`);
- to build the complete input string don't hesitate to use the python facilities, e.g.:
  ```
  $( python -c ' print "A"*10 + "\xgh\xef\xcd\xab" + "..." ' )
  ```

Draw the content of the execution stack when the crash takes place ...

8) If you try to execute bof with the same input string outside gdb you may not manage to open a shell (but get a crash instead ...). The reason is that gdb slightly shift the stack addresses. A possible solution is therefore to add a sequence of NOP instructions (opcode 0x90 in x86 assembly) at the beginning of the shellcode. Thus, it is no longer necessary to find the *exact* starting address of the shellcode: starting its execution somewhere in the middlle of this NOP sequence is enough.

Using this information, try to run the exploit outside gdb ...

2. Reverse engineering (optional)

crackme1 :

start the program crackme1 with IDA. Find the call to function strcmp().

crackme2 :

Open it with IDA. What the `magic` function does ?

You have two options to answer this question:

- Using IDA (static analysis)
- Using gdb (dynamic analysis)

Appendix

Using IDA

To start IDA, type « ida » from a terminal, or use the script `ida.sh` which is on the desktop.

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

breakpoint * ADDR : put a breakpoint at address ADDR

(e.g., "breakpoint * 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

<u>Using pattern.py</u>

python pattern.py make 200
 build a a 200 characters string with no repetitions

python pattern.py find xyzt
 indicate the position of pattern xyzt in the string produced

<u>Using python</u>

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

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

./programme $(python -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 the following command runs program with the content of file "file.txt" as argument:
run $(python cat file.txt)