Lab on Buffer overflow

We suggest to use the VirtualBox image available on the VM page on Canvas. This will ensure that:

- you have all the tools ready to use
- you use the same version of the tools that we used, thus the produced binary code is the same
- you use the same operating system and environment, thus memory allocation is the same we used for our experiments

This guide and all the exercises are described as follows.

Requirements for this lab are:

- understanding of C and its memory system
- usage of GDB
- usage of linux shell
- minimal usage of python
- usage of VirtualBox

All you experiments should be done in the virtual machine of VirtualBox.

There are five exercises:

- Exercise 0 contains the program used by the following tutorial. It is not required to report any solution for this exercise.
- Exercise 1 is on buffer overread and requires to report the file solution1.txt
- Exercise 2 is on buffer overflow and requires to report the file solution2.txt
- Exercise 3 is on control flow hijacking and requires to report the file solution3.py
- Exercise 4 is on code injection and requires to report the file solution4.py

READ ME !!!!

Notice that addresses on the stack described below can change, for example if you run the program into different terminals or nested Bash instances

Background GDB

To complete the exercises you must use a debugger or a dissassembler to inspect the code produced by the compiler and find out the memory layout of the application. Here we summarize some basic commands of GDB that are useful to complete your exercises.

The directory `exercise0` contains a small program to demonstrate the usage of GDB. Move in the directory `exercise0`

```
cd exercise0
```

Compile the example program

```
make
```

Explain what the program does

Run the program

```
./main <username>
<enter a random password>
```

To debug the program with gdb, use

```
gdb main
```

Breakpoints, running, stepping, resuming

To stop the program just before `strcpy(local_var, username);` is executed (notice that the `strcpy` is at line 11 of main.c) set a breakpoint at line 11:

```
b main.c:11
```

All Makefiles in this lab compile programs with debug symbols, which allows to set breakpoints by specifying the source code line.

To start the execution of the program, just type

```
run
```

When the program terminates, you can re-execute it using `run`. To provide the command line arguments to the program, use

```
run <args>
```

When the program stops due to a breakpoint, gdb prints the status of the program

```
(gdb) run roberto
Starting program: /home/student/lab-o/exercise0/main roberto

Breakpoint 1, afunction (username=0xbffff36c "roberto") at main.c:11
11          strcpy(local_var, username);
```

In this case the Breakpoint 1 has been activated, the program has been suspended before the string copy, inside function `afunction` at line 11. The string `roberto` has been allocated at `0xbffff36c`, therefore the actual parameter of the function is the pointer `0xbffff36c`. (**Notice that stack addresses can be different, due to different processes executed by linux**) A single C instruction can be executed by firing the command `next`

```
(gdb) next
12          scanf("%s", global_var);
```

GDB prints the line number and the code of the next instruction. The program can be resumed using `continue` and restarted using `run`. Finally, the content of a file can be redirected to the program standard input using `run < <filename>`, e.g.

```
run roberto < pwd.txt
```

Inspecting memory

Debug the program with gdb, set a breakpoint at line 11 of `main.c`, start the program providing the command line argument:

```
~/lab-o/buffer/exercise0$ gdb main

GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...done.

(gdb) b main.c:11
Breakpoint 1 at 0x8048534: file main.c, line 11.

(gdb) run roberto
Starting program: /home/student/lab-o/exercise0/main roberto

Breakpoint 1, afunction (username=0xbffff36c "roberto") at main.c:11
11        strcpy(local_var, username);
```

GDB provides several functions to inspect the content of memory, variables, and expressions. For instance:

```
(gdb) p 16
$4 = 16
(gdb) p 0x10
$5 = 16
(gdb) p (0x10-10)
$6 = 6
```

The following command inspects the actual parameter `username` of `afunction`:

```
(gdb) p username
$7 =  0xbffff36c "roberto"
```

The actual parameter is a pointer to the address `0xbffff36c`, which contains the string `roberto\0`.

The following command prints the address where `username` is stored:

```
(gdb) p &username
$8 = (char **) 0xbffff08c
```

The following command inspects the local variable `local_var` of `afunction`.

```
(gdb) p &local_var
$11 = (char (*)[16]) 0xbffff09c
(gdb) p local_var
$12 = "\001\000\000\000\000\000\000\000\215\a@\000\000\000\000"
```

The local variable is allocated on the stack at `0xbffff09c`, it is a buffer of 16 chars, it has not been initialized, so its content contain random values.

Notice that we cannot access to local variables and parameters of other functions, since the execution is currently suspended inside `afunction`:

```
(gdb) p &argc
No symbol "argc" in current context.
```

However, we can access the global variables.

```
(gdb) p &global_var
$13 = (char (*)[16]) 0x804a031 <global_var>
(gdb) p global_var
$14 = '\000' <repeats 15 times>
```

You can inspect the content of a specific region of memory. For example, since we know that `username` is allocated in`0xbffff08c` we can directly inspect this memory address (** change to username, and show that the address is &username
x/a is used to print bytes as pointers **)

```
(gdb) p username
$16 = 0xbffff36c "roberto"
(gdb) p &username
0xbffff08c
(gdb) x/a 0xbffff08c
0xbffff08c:      0xbffff36c
```

We are using a 32-bit machine, so pointers are 4 bytes.

(four bytes) Before and after the variable `username` there is something else. (it's not important what, we want just demonstrate that you can read arbitrary addresses in memory)

```
(gdb) x/a 0xbffff088
0xbffff088:      0xb7e21c34
(gdb) x/a 0xbffff090
0xbffff090:      0x0
```

Additionally to variables, you can print the addresses of functions (and if you want their binary code)

```
(gdb) p &afunction
$17 = (void (*)(char *)) 0x804851d <afunction>
(gdb) p &main
$18 = (int (*)(int, char **)) 0x8048588 <main>
(gdb) x 0x8048588
0x8048588 <main>:        0x83e58955
```

Changing memory content

Debug the program with gdb, set a breakpoint at line 11 of `main.c`, start the program providing the command line argument:

```
~/lab-o/exercise0$ gdb main
...

(gdb) b main.c:11
Breakpoint 1 at 0x8048534: file main.c, line 11.

(gdb) run roberto
Starting program: /home/student/lab-o/exercise0/main roberto

Breakpoint 1, afunction (username=0xbffff36c "roberto") at main.c:11
```

```
11        strcpy(local_var, username);
```
Execute the string copy

```
(gdb) next
12        scanf("%s", global_var);
```
Print the value of `local_var`
```
(gdb) p local_var
$19 = "roberto\000\215\a@\000\000\000\000"
```
You can change the second character of `local_var` by directly writing into the memory. ** Warning the address 0xbffff09c can be different **
```
(gdb) p local_var
$19 = "roberto\000\215\a@\000\000\000\000"
(gdb) p &local_var
$20 = (char (*)[16]) 0xbffff09c
(gdb) set *((char *)(0xbffff09c + 1)) = 'X'
(gdb) p local_var
$21 = "rXberto\000\215\a@\000\000\000\000"
```
You must specify the data type, for example if you set a pointer (void *) then 4 bytes are changed:

```
(gdb) set *((void **)(0xbffff09c + 1)) = 0
(gdb) p local_var
$22 =  "r\000\000\000\000to\000/\000\000\000\000\240\004\b"
```

Inspecting the stack

Debug the program with gdb, set a breakpoint at line 11 of `main.c`, start the program providing the command line argument:
```
~/lab-o/exercise0$ gdb main
...
(gdb) b main.c:11
Breakpoint 1 at 0x8048534:: file main.c, line 11.

(gdb) run roberto
Starting program: /home/student/lab-o/exercise0/main roberto

Breakpoint 1, afunction (username=0xbffff36c "roberto") at main.c:11
11        strcpy(local_var, username);
```
The stack of frames can be inspected using the command `bt`
```
(gdb) bt
#0  afunction (username=0xbffff36c "roberto") at main.c:11
#1  0x080485ba in main (argc=2, argv=0xbffff174) at main.c:24
```
Here the stack contains two frames, the top (and active) frame #0 (actually in lower memory address) is for the function `afunction`, while the bottom frame #1 (actually in a higher memory address) is for the function `main`, which is also the entry point of the program. The funciton `main` has been invoked with two parameters: `argc=2` and `argv=0xbffff174`. The latter is a pointer to an array of strings allocated by the OS and containing the command line arguments. The funciton `afunction` has been invoked with one parameter: `username=0xbffff36c`. This is the address of a string containing `roberto\0`.
The active frame can be inspected using `info f`
```
(gdb) info f
Stack level 0, frame at 0xbffff0c0:
```

```
eip = 0x8048534 in afunction (main.c:11); saved eip = 0x80485ba
called by frame at 0xbffff0e0
source language c.
Arglist at 0xbffff0b8, args: username=0xbffff36c "roberto"
Locals at 0xbffff0b8, Previous frame's sp is 0xbffff0c0
Saved registers:
 ebp at 0xbffff0b8, eip at 0xbffff0bc
```

The active frame (i.e. the one for `afunction`) ends at address `0xbffff0c0`. Inside the frame (i.e. below its end-address) there are (in order):

- parameters (i.e. `username`)
- local variables (i.e. `local_var`)
- the address where the previous frame started (at address `0xbffff0c0-8`)
- the `saved eip` (at address `0xbffff0c0-4`)

```
p &username
$25 = (char **) 0xbffff08c
(gdb) p &local_var
$24 = (char (*)[16]) 0xbffff09c
x/a (0xbffff0c0-8)
0xbffff0b8:      0xbffff0d8
(gdb) x/a (0xbffff0c0-4)
0xbffff0bc:      0x80485ba <main+50>
```

The `saved eip` is the return address, which is the address where the funciton should jump back after termination. In this case, the invocation of `afunction` should jump back to the address `0x80485ba`. Information about the code located at this address is obtained as follows:

```
(gdb) info line *0x80485ba
Line 25 of "main.c" starts at address 0x80485ba <main+50> and ends at 0x80485bf
<main+55>.
```

This address corresponds to line 25 of `main`: the line that immediately follows the invocation of `afunction`.

References

Look at GDB Manual for more detailed information on GDB.

python

In some exercises you must forge inputs or argument that exploit overflows and redirect the control flow of a program. Forging these inputs can be tricky since they contain characters that represent binary values (e.g. pointers or binary code). For this reason it is usually easier to prepare inputs using python. For example, the following snippet encodes a pointer (integer value of 4 bytes) to a string that can be printed to the stdout.

```
import sys
import struct
pointer = 0x80485ba
sys.stdout.write(struct.pack("@P", pointer))
```

Here, `struct.pack("@P", pointer)` encodes the pointer to a string and `sys.stdout.write` prints the string the standard output (without newline).