# Teamwork for Computer and Network Fundamentals

Winter 2024

## Teamwork Goals

The teamwork has these goals:

* Applying the knowledge acquired in the course in a realistic, albeit simplified environment.
* Knowing a real architecture, specifically, the Intel x86-32 architecture. In addition, by knowing a second architecture, the student can appreciate how general concepts can be applied in different ways in concrete implementations.
* Practicing C/C++ programming, needed in other parts of the course and for future courses, apart from being a highly used language in the industry. In addition, we will see how to mix it with C.
* Developing teamwork and documentation skills essential for a Computer Engineer.

## Teamwork Development

The assignment will have two phases.

### Phase 1 (5 points)

#### Phase 1 – first part (2.5 points)

In this phase, each team must develop a program in C/C++ to practice concepts such as bit masks, bit shifts and strings. To create this program, you must use the Visual Studio solution **Teamwork** that you can download from the Virtual Campus. This solution is configured with certain compilation and linking options that make the second phase of the teamwork easier; the student must not change these options.

The program to create simulates a simple license control system. It is parametrized for each team. You must use a number that will be called ID and must be the lowest UO of all of the team members. We will use the notation ID[position] to indicate each figure inside the number, i.e., if ID=43298, then ID[0]=4, ID[1]=3, etc. Therefore, if the instructions say something like “if the bit ID[0] is 1”, it means “if the bit 4 is 1” for the example given.

You must write in a comment at the beginning of the code the ID that you have used.

You must implement the following functions:

**PasswordControl\_T3()**

The function uses strings, which must be declared as arrays of characters of 20 positions. It is not valid to use the String class. When reading a string, it can be assumed to have fewer than 20 characters. Error checking will not be carried out. Keep in mind that this is a bad practice, but since we are starting with C, we will skip these important checks.

The function should follow these steps:

* Read a string from the terminal. Using the function strcmp(), check that the string is v31cQD4fA. If not, it should print the message "You do not have access here" and call exit().
* Read another string from the terminal. Check with strlen() that the length of the string is at least 15 and that the character at index ID[4] is equal to the character at index ID[3]. If not, it should print the message "There was some failure" and call exit().

**MaskAccess\_T3()**

First, it must request two unsigned 32-bit integer numbers. Next, it has to check:

* That the bit at position ID[4] of the first number is equal to the bit at position ID[3] of the second number. If not, it should print the message "Intruder detected" and call exit().
* That the number formed with the ID[2] most significant bits of the first number and the remaining least significant bits of the second number is greater than the value of ID[1]\*347. If not, it should print the message "These are not the droids you're looking for" and call exit().
* That taking the bits between position ID[3] and position ID[1] (both inclusive) of the second number entered by the user, the resulting bit sequence interpreted as natural binary is equal to 0. If not, it should print the message "Halt!" and call exit().

**AsmBasedControl\_T3()**

You must read three 32-bit integer numbers from the terminal and pass them to the function IsValidAssembly(). Then, you must verify the result of the call, and if it's 0, print the message "Intruder detected" and call exit().

The IsValidAssembly() function must be implemented in assembly in the file Assembly-code.asm and should return 1 if these two conditions on the numbers read from the terminal are true, and 0 otherwise:

* The bit at position ID[5] of the second number is equal to the bit at position ID[3] of the third number.
* The value of the least significant ID[1] bits of the first number, interpreted in natural binary, is greater than 11.

**CheckArray\_T3()**

This function should first declare an array of 3 elements, each of 16 bits. Next, it should read the values of the three elements from the terminal. Finally, it should verify that the logical AND bitwise is equal to 369. If not, it should print the message "Stop" and call exit().

**main()**

It should call the functions **PasswordControl\_T3()**, **MaskAccess\_T3()**, **AsmBasedControl\_T3()**, and **CheckArray\_T3()**. If all these functions execute without calling **exit()**, it should display the message "Access granted".

#### Phase 1 – second part (2.5 points)

You must answer the following questions using the Visual Studio disassembler with the project uploaded by the team. **You must indicate the process to obtain each solution and include screenshots to check where the answer comes from:**

* (1 point) Memory address where the code to pass the parameters to the function **IsValidAssembly()** begins and the code itself, as machine code and as mnemonics.
* (0.5 points) Memory address where the string read in the first function is stored.
* ***Imagen que contiene Texto

  Descripción generada automáticamente***(0.5 points) Stack frame of the last function indicated in the instructions, in its state after reading the elements of the vector. You should take a screenshot and indicate on it what the different elements in memory are, as shown in the example for this function test():



Variable a

EBP copy

Return address

* (0.5 points) Explanation of the read access to an element of the vector in the last function indicated in the instructions. You must indicate which mnemonic is used to read the value of an element of the vector and explain what the different parts that appear mean.

You must upload a ZIP file in task **Phase 1** of resource **Teamwork** of the Virtual Campus before **Friday March 22nd**. The ZIP file must include the **.cpp** and **.asm** source code, and a report in PDF with the names and UO of the students, an example of valid input and another of invalid input for each function, and the answer to the questions above. In addition, at the end of the report, you must indicate how the work has been divided between team members and how many hours have each one of them used for the work.

### Phase 2 (5 points)

The world needs you. Alien threaten humanity. They have placed several bombs in critical infrastructures that can only be defused with programs called “binary bombs”. These programs have 4 stages. In each of them, they ask for some input. If the input follows some criteria, the stage is defused and the program goes to the next stage; otherwise, the bomb explodes. If all stages are defused, the bomb will be defused.

The teachers of the course do not have time to deal with all the bombs, so we have decided to ask students for help.

Fortunately, a heroic raid in the alien servers has allowed us to obtain the source code of the main program of each bomb and its debugging information. Your teacher will give your team the binary bomb that you have to defuse. In order to achieve this, you must deduce valid inputs by debugging the binary bomb.

Every stage that your team defuses gives you one point. The final point will be obtained if you can modify the bomb code so that it is defused without needing any input.

You must upload a ZIP file in task **Phase 2** of resource **Teamwork** of the Virtual Campus before **Monday April 29th**. The ZIP file must include the **.exe** file modified so that the bomb is defused without requiring any input. The ZIP file must also include a report in PDF indicating which inputs defuse each stage and the process that you have followed for obtaining them. You must include screenshots in the explanation. The report must also include a section indicating how you modified the code so that the bomb is defused without requiring any input. Finally, it must include how the work has been shared between team members and how many hours have each one of them used for the work.

Assessment

The grades will be obtained using the points detailed above. The total result will be weighted with a factor between 0.5 and 1 depending on the quality of the code, the writing and the presentation of the report. This grade can be weighted with another factor depending on the contribution of each student to the work. Each day late in any of the phases takes 2 points from the grade.

If the work is carried out using illicit ways, the grade in the **continuous assessment** for **all the members** of the team will be **zero**.

## Guides for Developing the Work

For phase 1, lab session 1 of the unit 0, which introduces the programming language C/C++, is required. In addition, you must also read the annex of this document and the description of the x86-32 architecture that you can find in the Virtual Campus.

For the second phase, lab session 1 of the unit 1, which explains how to debug in Visual Studio, will also be useful. However, this session explains how to debug when the source code is available, and in phase 2 of this work the students will not have the source code. For overcoming this problem, you must select in Visual Studio **Open->Project or solution** and choose the **.exe** file provided. Pressing **F11** begins the debugging.

## Annex: masks and shifts

A bit mask is a bit pattern used in bitwise operations. Masks have several uses depending on which logic operation is used:

* AND: it is used to detect if a bit is 0 or 1. For instance, if we have an 8-bit variable **v** and we apply an AND operation with the mask 0x4 (0000 0100b), the result will be 0 if bit 2 of **v** is 0 and 1 otherwise, independently of the value of the rest of the bits in **v**. This operation is indicated in C/C++ with the symbol **&**. Example: **v & 0x4**. The AND operation is also used to set a bit to 0 when it is applied with a mask with all the bits set to 1 except the bit we want to set to 0. For instance, if we want to set bit 2 of **v** to 0, we can use the mask 0xFB (1111 1011b) this way: **v = v & 0xFB**.
* OR: it is used to set a bit to 1, because the OR operation of any bit with 1 always generates 1. In C/C++ this operation is indicated with the symbol **|**. For instance, **v | 0x4** sets bit 2 of **v** to 1 without affecting the rest of the bits.
* XOR: it is used to invert a bit. In C/C++ it is indicated with the symbol **^**. For instance, **v = v ^ 0x4** inverts bit 2 of **v**.

Another operation commonly used is shifting the bits of a variable to the right or to the left. For instance, the color of a pixel is often defined in a 32-bit variable where the highest 8 bits indicate the amount of transparency (alpha), the next 8 bits the amount of red, the next 8 bits the amount of green and the 8 least significant bits, the amount of blue, as shown in this figure:

|  |  |  |  |
| --- | --- | --- | --- |
| Transparency | Red | Green | Blue |
| 1111 1111 | 1000 0101 | 0011 1101 | 1001 1001 |

To obtain the amount of red of a variable called **color**, a mask with all the bits set to 0 except the ones that we want to extract (bits 8 to 15) can be applied; after that, a shift of 16 bits to the right will be required. Therefore, if the variable **color** is 0xFF853D99, the amount of red (0x85) can be obtained with this operation: **red = (color & 0x00FF0000) >> 16**. The AND operation with 0x00FF0000 sets to zero all bits except the bits we are interested in, and the operation **>> 16** shifts the result 16 positions to the right. To shift to the left, the symbol **<<** is used.

Another interesting use of the bitwise OR operation is combining bits from two variables. For instance, if in the lowest 8 bits of the variable **alpha** we have the amount of transparency and in variable **rgb** we have the amount of red, green and blue, we can obtain a variable including all the information with this sentence: **color = (alpha << 24) | rgb**. This operation shifts 24 bits to the left the variable **alpha**, so that the transparency is in the 8 highest bits, and then combines the result using a bitwise OR operation with the variable **rgb** (we are assuming that the 8 highest bits of **rgb** are zero; if not, we would need to apply first a bitwise AND operation with the mask 0x00FFFFFF to set those bits to zero and leave the rest unchanged).

In x86 assembly, the bitwise operations with masks are carried out using the instructions **and**, **or** and **xor**. To carry out shift operations, there are two types of instructions:

* Logical shifts: **shl** to shift left and **shr** to shift right.
* Arithmetic shifts: **sal** to shift left and **sar** to shift right.

Logical and arithmetic shifts to the left are the same: they shift the bits to the left and they introduce zeros on the right. The shifts to the right are different: the logical shift introduces zeros from the left, but the arithmetic shift introduces the value from the most significant bit before carrying out the shift, which preserves the sign in numbers represented in two’s complement. Examples with 8-bit numbers:

* Initial value: 10110111
  + 3-bit logical right shift: 00010110
  + 3-bit arithmetic right shift: 11110110
  + 3-bit logical/arithmetic left shift: 10111000
* Initial value: 00110111
  + 3-bit logical right shift: 00000110
  + 3-bit arithmetic right shift: 00000110
  + 3-bit logical/arithmetic left shift: 10111000