Laboratory 6

Daniel Felipe Rambaut Lemus

September 2020

1. The mean of the a code construct is a code abstraction level that defines a functional property but not the details of its implementation. The impacts that can be generated when analyzing the assembly code are caused by the versions of the computers that are used to generate it.

2. One difference that occurs in the use of global variables is the fact that in memory it does not need to be assigned to within the execution of a function, if not by default it is in a global context. It can be seen in the mov of more that is used in the compilation of the code with local variables.

```
.text:00401410                push    ebp
.text:00401411                mov     ebp, esp
.text:00401413                and     esp, 0FFFFFFF0h
.text:00401416                sub     esp, 10h        ; char *
.text:00401419                call    sub_4019A0
.text:0040141E                mov     edx, dword_404004
.text:00401424                mov     eax, dword_404008
.text:00401429                add     eax, edx
.text:0040142B                mov     dword_404004, eax
.text:00401430                mov     eax, dword_404004
.text:00401435                mov     [esp+10h+var_C], eax
.text:00401439                mov     [esp+10h+var_10], offset aTotalD ; "Total = %d\n"
.text:00401440                call    printf
.text:00401445                nop
.text:00401446                leave
.text:00401447                retn
.text:00401447 sub_401410     endp
```

Figure 1: Global variable

```
.text:00401410                push    ebp
.text:00401411                mov     ebp, esp
.text:00401413                and     esp, 0FFFFFFF0h
.text:00401416                sub     esp, 20h        ; char *
.text:00401419                call    sub_4019A0
.text:0040141E                mov     [esp+20h+var_4], 1
.text:00401426                mov     [esp+20h+var_8], 2
.text:0040142E                mov     eax, [esp+20h+var_8]
.text:00401432                add     [esp+20h+var_4], eax
.text:00401436                mov     eax, [esp+20h+var_4]
.text:0040143A                mov     [esp+20h+var_1C], eax
.text:0040143E                mov     [esp+20h+var_20], offset aTotalD ; "Total = %d\n"
.text:00401445                call    printf
```

Figure 2: Local variable

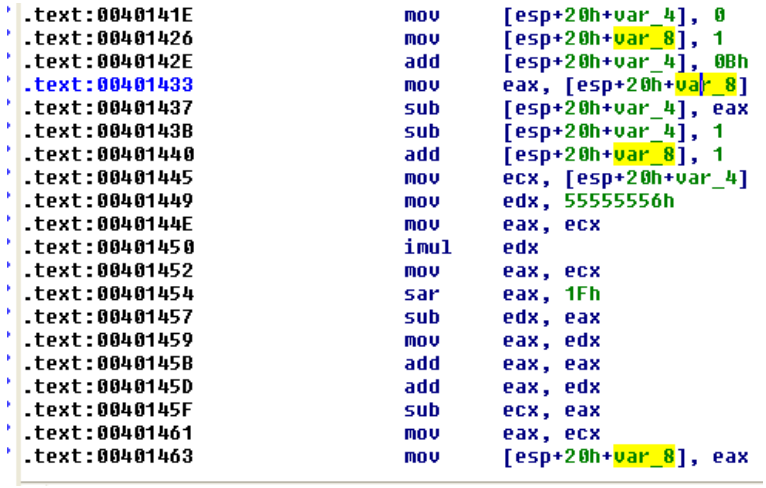

Figure 3: Operations

3. In the previous image we can see to perform the addition, what the add function is performed, which takes the variable a and adds $0Bh$ to it. Now to perform the subtraction, the variable of the value of b is moved and saved in eax, then we proceed to perform the sub of the variable a and eax. To perform the $a--$ and $b++$ operations, it is similar to the above, since the add and sub command is used to update the value of the variables. Finally we have the operation of the module, for this case the imul and sar functions are presented, which allow us to perform the multiplicative inverse and the respective subtraction to obtain the module.

4. In order to identify the if conditional, we can see it with assembly instructions such as *jnz*, which allows us to make a comparison between the values, which in our case are numbers. The *jnz* function makes a jump when the values are not equal and in this way we can identify when they are not equal.

Figure 4: Conditional

5. For the realization of the two nested ifs, we can analyze them as we saw

previously that if we use the jnz function and the cmp function, which allow us to compare the values. The cmp function this instruction subtracts the source operand from the destination operand but without it storing the result of the operation, only the state of the flags is affected. With this present we can see in the image that there are several branches for the established conditions.

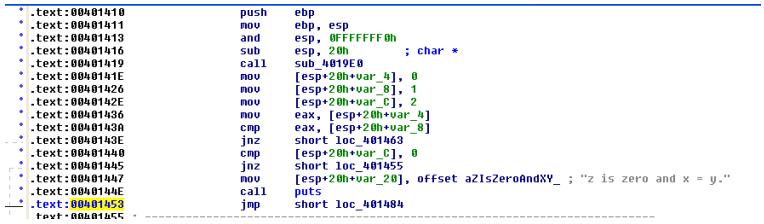

Figure 5: Conditional

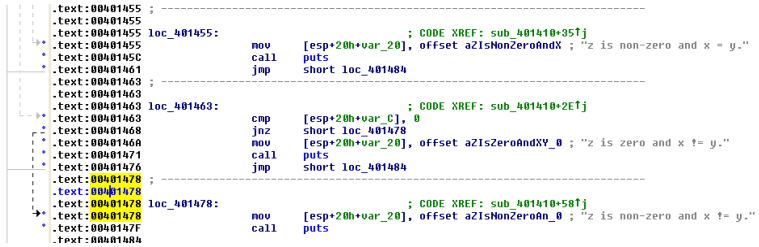

Figure 6: Conditional

6. The way to identify the loops is to be able to see how the jmp is used to perform the validations of the variable that is increased in each iteration. Also in the image below we can see how the code prints the variables and also redoes the validation with cmp.

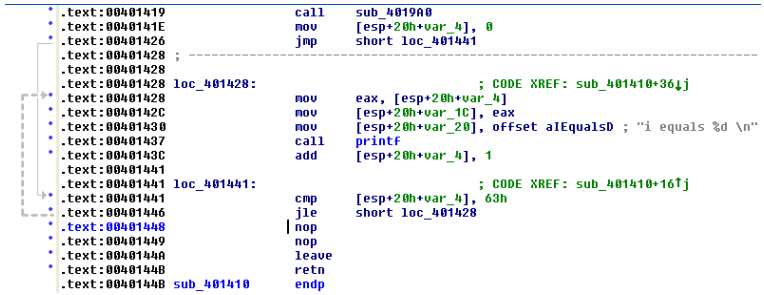

Figure 7: Loop

7. In order to identify the while we can see that the declaration of the comparison variables is presented to stop the while. It is worth mentioning that I have modified the example in the book for my own and in we can see how the result variable is assigned the value plus one in each iteration. The while can be recognized at the time of comparison before entering the while.

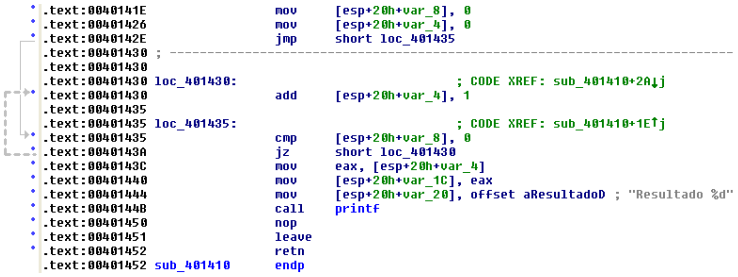

Figure 8: While