PART 5

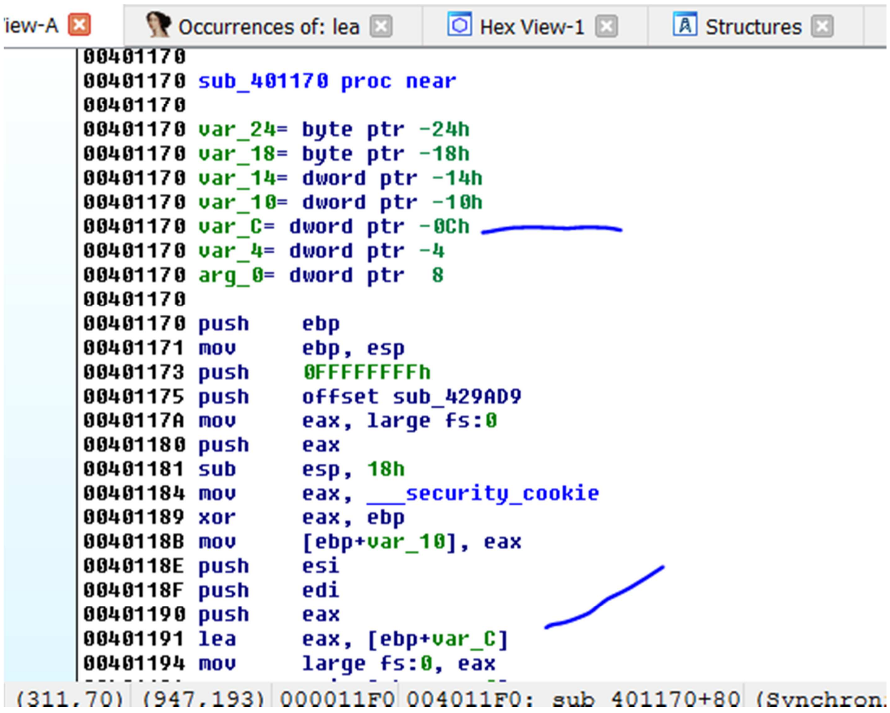
We will continue with the main instructions and their use in the code. Logically, when we see the debugger section, we will have more examples to apply the results in real time.

LEA (LOAD EFFECTIVE ADDRESS)

LEA A,B

The instruction LEA moves the specified address from **B** to **A**. It never accesses the content of **B**. It will always be the address or result of the operation in brackets of the second operand. It is used so much to get the memory addresses of variables and arguments.

Let's see some examples:

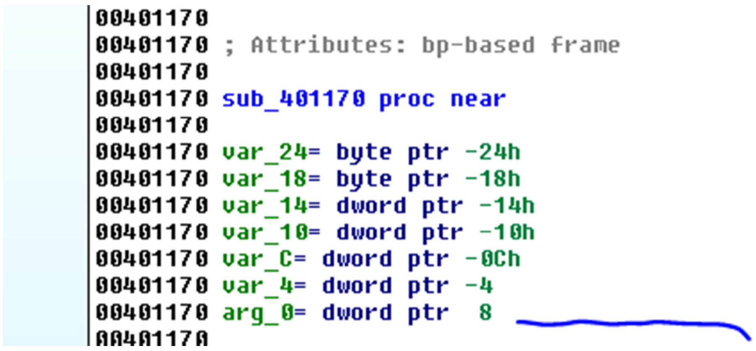


```
00401170
00401170 sub_401170 proc near
00401170
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr  8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFFh
00401175 push    offset sub_429AD9
0040117A mov     eax, large fs:0
00401180 push    eax
00401181 sub     esp, 18h
00401184 mov     eax, ___security_cookie
00401189 xor     eax, ebp
0040118B mov     [ebp+var_10], eax
0040118E push    esi
0040118F push    edi
00401190 push    eax
00401191 lea     eax, [ebp+var_C]
00401194 mov     large fs:0, eax
```

```
(311,70) (947,193) 000011F0 004011F0: sub_401170+80 (Synchron
```

Normally, in the functions detected by IDA, there are arguments received by them. Most of the times, through a PUSH before calling the function.

The PUSH saves those values on the stack. These values are called arguments.

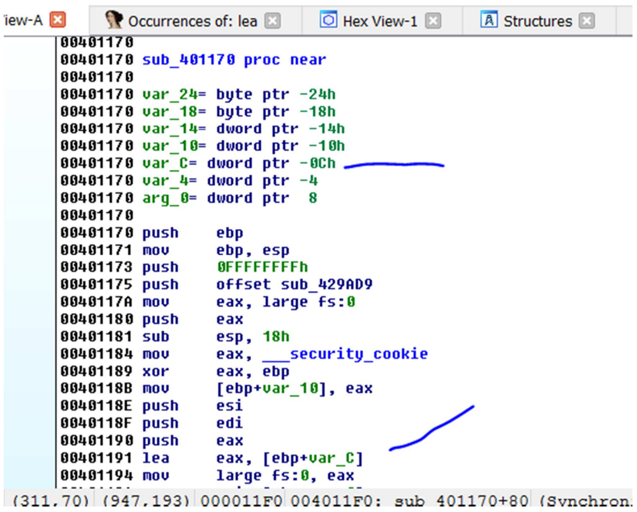


```
00401170
00401170 ; Attributes: bp-based frame
00401170
00401170 sub_401170 proc near
00401170
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr  8
00401170
```

In the list of variables and arguments of the header in each function, there is just an argument on the stack as in the list. In this case, **arg_0**.

The function also has local variables to reserve space on the stack above the arguments.

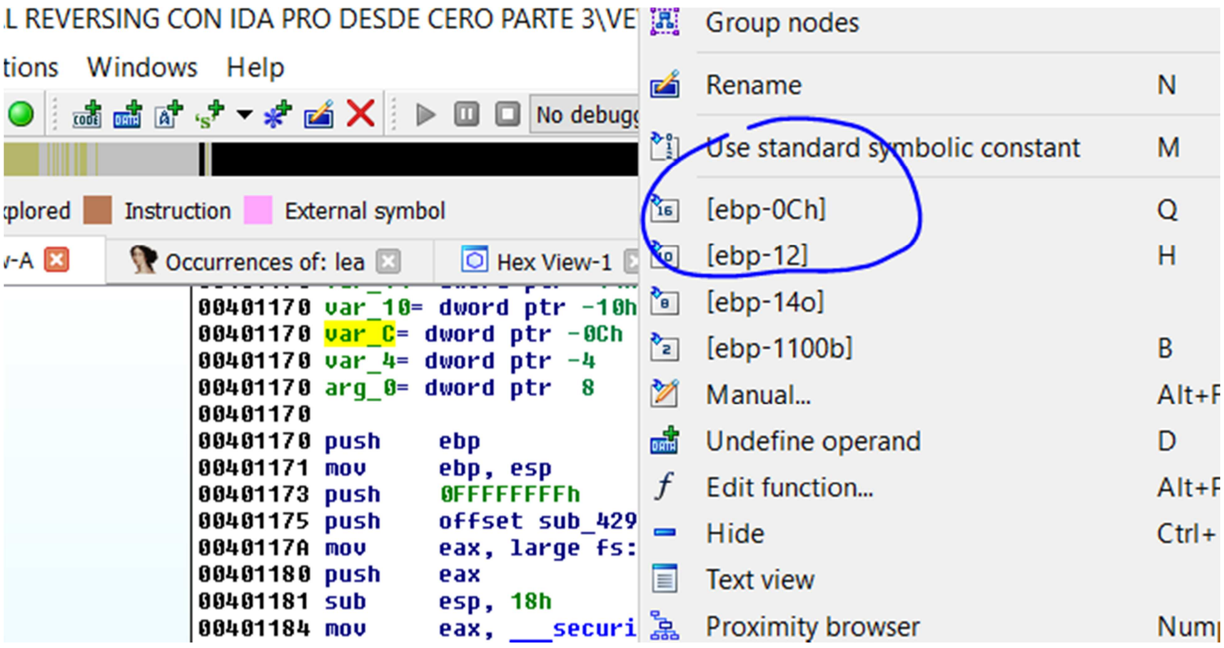
More ahead, I will explain the exact location of arguments and variables on the stack. By now, it is just important that each argument or variable used by a function has an address where it is located and a value as any other variable does anywhere in the memory.



```
00401170
00401170 sub_401170 proc near
00401170
00401170 var_24= byte ptr -24h
00401170 var_18= byte ptr -18h
00401170 var_14= dword ptr -14h
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr  8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFFh
00401175 push    offset sub_429AD9
0040117A mov     eax, large fs:0
00401180 push    eax
00401181 sub     esp, 18h
00401184 mov     eax, ___security_cookie
00401189 xor     eax, ebp
0040118B mov     [ebp+var_10], eax
0040118E push    esi
0040118F push    edi
00401190 push    eax
00401191 lea     eax, [ebp+var_C]
00401194 mov     large fs:0, eax
(311,70) (947,193) 000011F0 004011F0: sub_401170+80 (Synchron
```

In this picture, we see when the program, in 0x401191, uses LEA. It moves just the address from the stack. If it were a MOV, it would move the content or value saved in that variable.

LEA, in spite of using brackets, it just moves the address without accessing its content because it just solves the operations in brackets and as EBP is normally used as a base of variables and arguments of the stack in each function, what it really does is adding or subtracting a constant to the EBP value that points to an address of the stack taken as a base for that function.



```
00401170 var_10= dword ptr -10h
00401170 var_C= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr  8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFFh
00401175 push    offset sub_429
0040117A mov     eax, large fs:
00401180 push    eax
00401181 sub     esp, 18h
00401184 mov     eax, ___securi
```

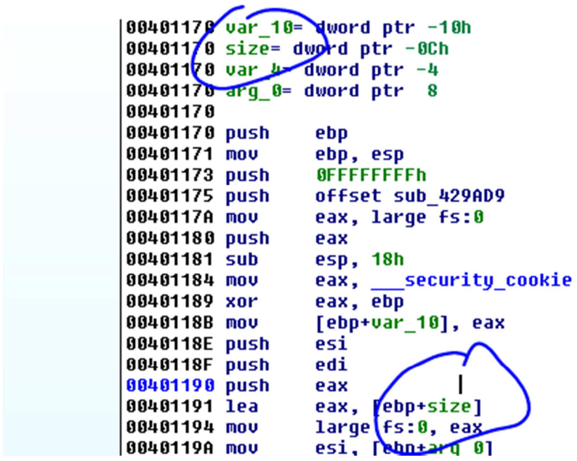
If we right click on that variable, we see that the mathematical way to write it and modify it using the Q key is **[EBP-0C]**.

That’s why **LEA** just solves that **EBP-0C** operation as **EBP** has a stack address that will be the base in this function subtracting **0C**, I get the address of that variable.

Here, many people wonder if it is not easier that IDA uses the pure math notation for variables and arguments instead of [EBP - or + a tag.]

In reversing, it is very important for us to be able to rename the variables and arguments in real time to know what they do and have a better orientation.

It’s not the same to use a variable called EBP-0C than a variable that I can rename as EBP+SIZE, for example. (Using the N key. Try it.) If I know that it saves a size. If I need to see the original value, I right click on it.



```
00401170 var_10= dword ptr -10h
00401170 size= dword ptr -0Ch
00401170 var_4= dword ptr -4
00401170 arg_0= dword ptr  8
00401170
00401170 push    ebp
00401171 mov     ebp, esp
00401173 push    0FFFFFFFFh
00401175 push    offset sub_429AD9
0040117A mov     eax, large fs:0
00401180 push    eax
00401181 sub     esp, 18h
00401184 mov     eax, ___security_cookie
00401189 xor     eax, ebp
0040118B mov     [ebp+var_10], eax
0040118E push    esi
0040118F push    edi
00401190 push    eax
00401191 lea     eax, [ebp+size]
00401194 mov     large fs:0, eax
0040119A mov     esi, [ebp+arg_0]
```

LEA is also used to solve operations in brackets moving the result to a destination register without accessing its content.

Example:

LEA EAX,[4+5]

It will move 9 to EAX, not the content of the address 0x9 as a MOV would do.

MOV EAX,[4+5]

That’s why…

LEA EAX,[EBP - 0C]

…moves the result of **EBP-0C** that is the result of the variable memory address gotten when solving **EBP-0C** and moves it to **EAX**.

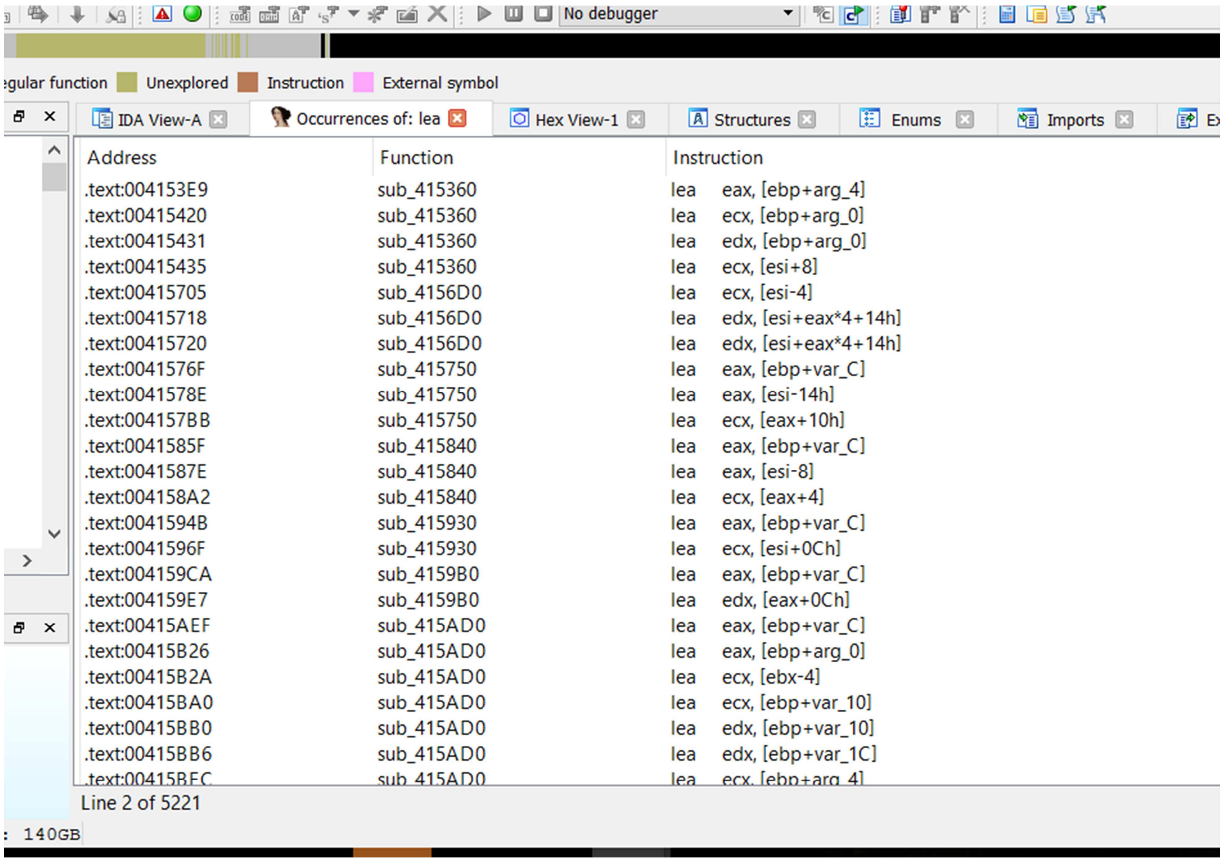
MOV EAX, [EBP - 0C]

Apart from solving EBP-0C and getting the address, it looks for its content (the saved value in that variable) and moves it to EAX.

It is very important to differentiate a LEA from MOV and their use.

LEA gets variable addresses.

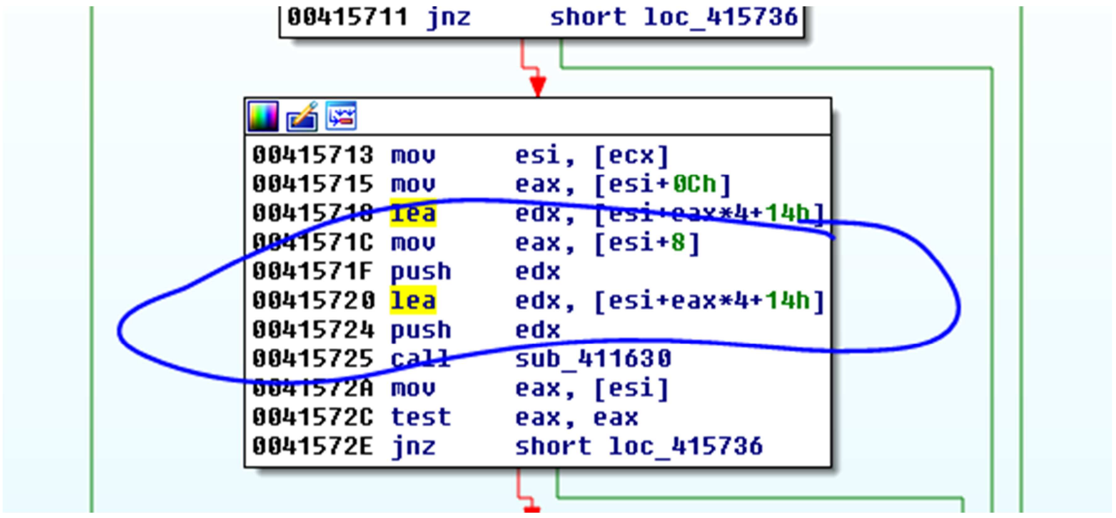
MOV gets the values saved in the variable addresses.



```
Address            Function      Instruction
.text:004153E9     sub_415360    lea   eax, [ebp+arg_4]
.text:00415420     sub_415360    lea   ecx, [ebp+arg_0]
.text:00415431     sub_415360    lea   edx, [ebp+arg_0]
.text:00415435     sub_415360    lea   ecx, [esi+8]
.text:00415705     sub_4156D0    lea   ecx, [esi-4]
.text:00415718     sub_4156D0    lea   edx, [esi+eax*4+14h]
.text:00415720     sub_4156D0    lea   edx, [esi+eax*4+14h]
.text:0041576F     sub_415750    lea   eax, [ebp+var_C]
.text:0041578E     sub_415750    lea   eax, [esi-14h]
.text:004157BB     sub_415750    lea   ecx, [eax+10h]
.text:0041585F     sub_415840    lea   eax, [ebp+var_C]
.text:0041587E     sub_415840    lea   eax, [esi-8]
.text:004158A2     sub_415840    lea   ecx, [eax+4]
.text:0041594B     sub_415930    lea   eax, [ebp+var_C]
.text:0041596F     sub_415930    lea   ecx, [esi+0Ch]
.text:004159CA     sub_4159B0    lea   eax, [ebp+var_C]
.text:004159E7     sub_4159B0    lea   edx, [eax+0Ch]
.text:00415AEF     sub_415AD0    lea   eax, [ebp+var_C]
.text:00415B26     sub_415AD0    lea   eax, [ebp+arg_0]
.text:00415B2A     sub_415AD0    lea   ecx, [ebx-4]
.text:00415BA0     sub_415AD0    lea   ecx, [ebp+var_10]
.text:00415BB0     sub_415AD0    lea   edx, [ebp+var_10]
.text:00415BB6     sub_415AD0    lea   edx, [ebp+var_1C]
.text:00415BEC     sub_415AD0    lea   ecx, [ebp+arg_4]
Line 2 of 5221
: 140GB
```

The result of the search of the **LEA** text in the VEWIEVER shows us that it is used most of the times to get variable addresses or stack arguments. There are more instructions using **[EBP+something]**.

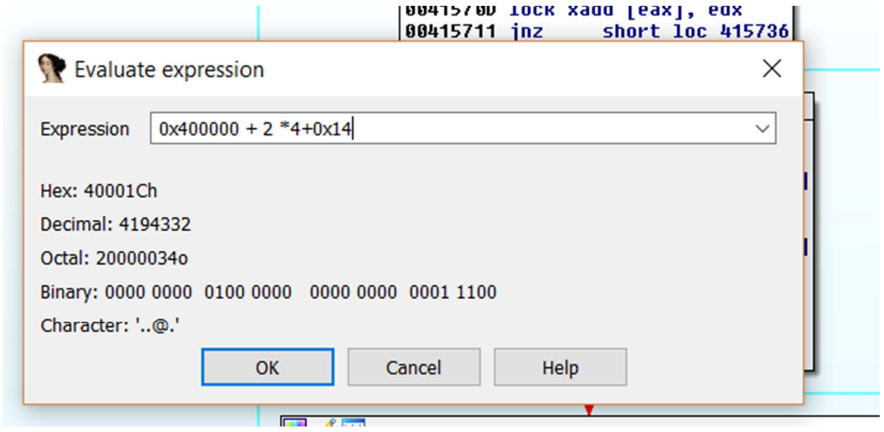
The rest is combined operations between registers and constants whose result is moved to the first operand that can have numerical results or some address also depending on the registers value.



```
00415711 jnz     short loc_415736

00415713 mov     esi, [ecx]
00415715 mov     eax, [esi+0Ch]
00415718 lea     edx, [esi+eax*4+14h]
0041571C mov     eax, [esi+8]
0041571F push    edx
00415720 lea     edx, [esi+eax*4+14h]
00415724 push    edx
00415725 call    sub_411630
0041572A mov     eax, [esi]
0041572C test    eax, eax
0041572E jnz     short loc_415736
```

When solving the operation, if ESI equals 400000, for example, and EAX equals 2, the result of **0x400000 + 2 *4+0x14** will be moved to EDX.



```
0041570D lock xadd [eax], edx
00415711 jnz     short loc_415736
```

Evaluate expression

Expression 0x400000 + 2 *4+0x14

```
Hex: 40001Ch
Decimal: 4194332
Octal: 20000034o
Binary: 0000 0000  0100 0000   0000 0000  0001 1100
Character: '..@.'
```

That means it will move **0x40001C**.

Ricardo Narvaja

Translated by: @IvinsonCLS