

## REVERSING WITH IDA PRO FROM SCRATCH

---

### PART 3

#### THE LOADER

We realized that when opening an executable, that action is done by the loader that is the static analyzer of it.

Its parts and characteristics will be analyzed. Some stuff we've seen will be applied to the LOADER and the DEBUGGER. When there is something different, I will let you know.

Obviously, the fact that in the LOADER the program is not executed but is analyzed and a database is created with its info makes a great significant difference with respect to the DEBUGGER.

In the LOADER, there are no REGISTER, STACK, and MODULE list windows loaded in memory. Those are things present when running and debugging the program.

After loading the Cruehead Crackme (CRACKME.EXE) if we see in the process list, it is not running and never executes unless we open the IDA debugger.

This is very useful for certain uses like malware analysis, exploit, etc. because we won't always be able to access to the function we need to study debugging while in the LOADER we can analyze any program function. It doesn't matter if we don't know how to call it.

Of course, to talk about function analysis we need to know the use of registers and instructions because in spite of not being debugging and not having a register window with the values at each moment, the instructions use them and we need to understand them to know what the program does.

#### **What are registers and what are they used for?**

The processor needs assistants when executing programs.

Registers help the processor with that. When we see ASM instructions, we will realize that two memory contents cannot be added directly. The processor has

to move one of them to a register and then add it with the other memory position. This is an example, but some registers have more specific uses.

32-bit registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI and EIP

At the end of this course, there will be a part dedicated to 64 its.

<b>Data registers</b>	EAX	EBX	ECX	EDX		
<b>Stack pointers</b>	ESP	EBP				
<b>Index registers</b>	EDI	ESI				
<b>Segment registers</b>	CS	DS	ES	FS	GS	SS
<b>Flag register</b>						

### General purpose registers

**EAX** (accumulator): The accumulator is used for instructions like division, multiplication and some format instructions. Also as general purpose registers.

**EBX** (Base index): It can direct memory data and it's also a general purpose register logically.

**ECX** (counter): It's a general purpose register that can be used as a counter for the different instructions. It can also have the memory data offset address. The instructions that use a counter are those with repeated strings, offset instructions, rotation and LOOP/LOOPD.

**EDX** (data): It's a general purpose register that has part of a multiplication product or dividend of a division. It can also direct memory data.

**EBP** (base pointer): EBP points to a memory place. Almost always as a base of arguments and variables location in a function apart from being a general purpose register.

EDI (destination index): EDI often directs data from strings destination for string instructions. It's a general purpose register too.

ESI (source index): It often directs data from strings source for string instructions. Like EDI and ESI, it also works as a general purpose register too.

EIP: Index that points to the next instructions that will be executed.

ESP: Index that points to the top of the stack.

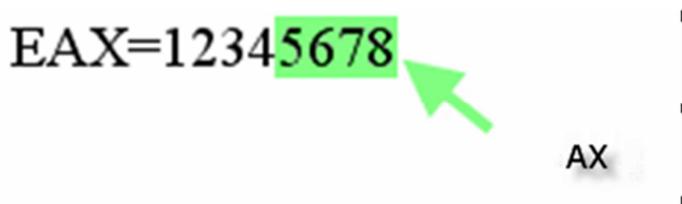
Summarizing what I copied from a guy.

The eight registers are EAX (accumulator), EBX (base), ECX (counter), EDX (data), ESP (stack pointer), ESI (source), and EDI (destination).

There are 16-bit registers and 8-bit registers that are part of the previous registers.

If EAX equals 12345678

AX is the last four digits (16 bits)

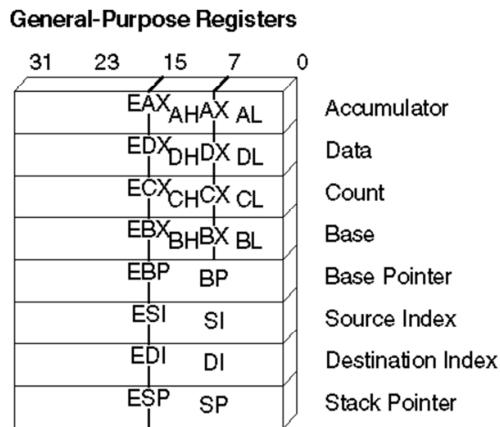


AH is the 5th and 6th digit and AL is the two last digits (8 bits each one)



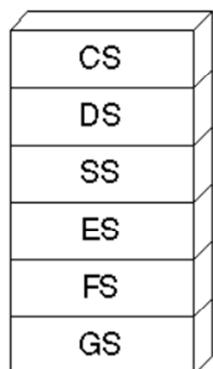
There is a 16-bit register for the low part of EAX called AX and two 8-bit registers called AH and AL. There are not special registers for the high part of any register.

For EBX there are BX, BH and BL. For ECX we have CX, CH and CL and so on for almost all registers. ESP just has SP of 16 bits and SL of 8 bits.



There, you can see registers like EAX EDX ECX and EBX that have 16 bits and 8 bits subregisters and EBP, ESI, EDI and ESP that only have 16-bit subregisters.

**Segment Registers**



**Other Registers**



Flags

Instruction  
Pointer

We will see the other registers. As important auxiliary registers we have the EFLAGS that according to their value it will activate flags that make decisions in some moments in the program that we'll see later. The segments registers direct to different parts of the executable like CS=CODE, DS=DATA, etc.

Another important detail is the size of the most used data type.

The size attribute associated with each data type is:

Data Type	Bytes
BYTE, SBYTE	1
WORD, SWORD	2
DWORD, SDWORD	4
FWORD	6
QWORD	8
TBYTE	10

IDA handles more data types that we'll see more ahead step by step. The most important thing is to know that BYTE is 1 byte, WORD is 2 bytes and DWORD is 4 bytes in memory.

## INSTRUCTIONS

IDA works with an instruction syntax that is not the easiest one of world. Most of the people are used to OllyDBG disassembly that is easier and decaffeinated (easy to understand. ☺) Although Olly gives us less info.

### ***Data transfer Instructions***

#### **MOV**

**MOV dest,src:** Copies the content of the source operand (src) to the destination (dest).

Operation: dest <- src

Here, we have some possibilities. For example, we can move the value of a register to another one.

#### **MOV EDI, EAX**

```
0040139C loc_40139C:
0040139C pop    esi
0040139D call   sub_4013C2
004013A2 xor    edi, 5678h
004013A8 mov    eax, edi |
004013AA jmp    short locret_4013C1
```

In general, we can move from or to a register directly taking into account that EIP cannot be either DESTINATION or SOURCE of any direct operation.

We cannot do the following:

**MOV EIP, EAX**

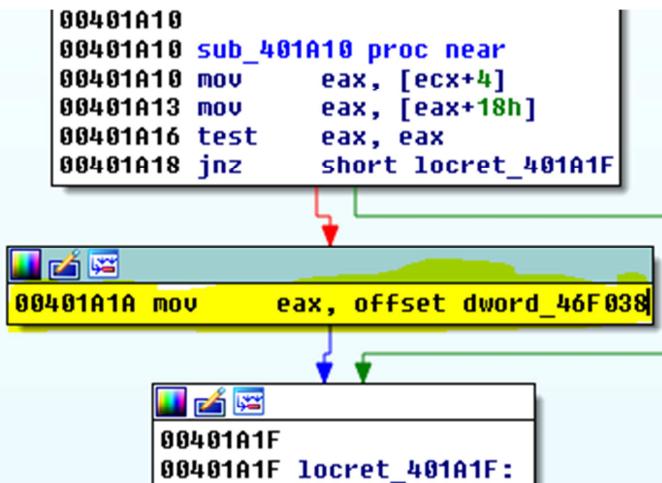
That's not valid.

Another option is moving a constant to a register, for example:

**MOV EAX, 1**

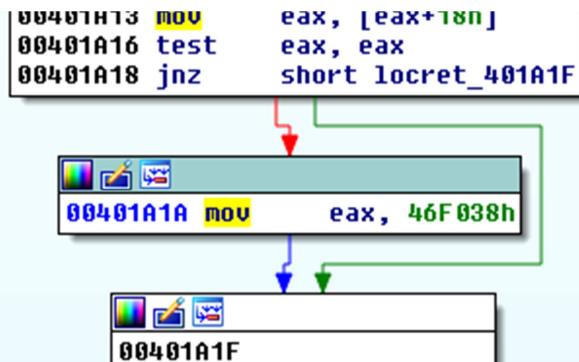
```
004012D5 push 0Bh ; cchMax
004012D7 push offset byte_40217E ; lpString
004012DC push 3E9h ; nIDDlgItem
004012E1 push [ebp+hWnd] ; hDlg
004012E4 call GetDlgItemTextA
004012E9 mov eax, 1 |
004012EE jmp short loc_4012F7
```

We can also move the value of a memory address not its content. (These instructions in the image belong to another executable not to the CRACKME.exe because they were not there but in VEViewer.exe attached to this part 3)



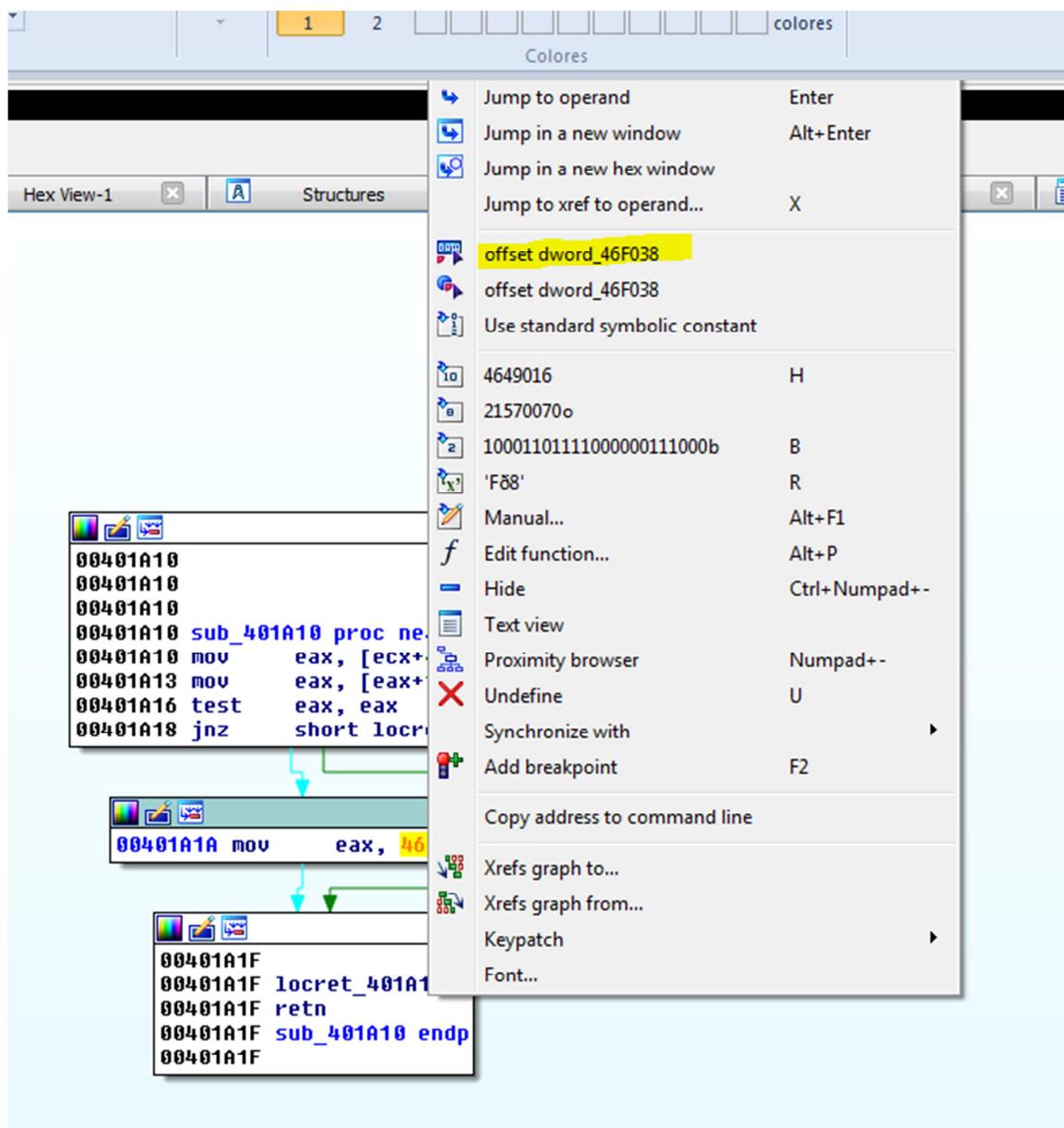
In that case, when the value to be moved is a memory address, the word OFFSET before it tells us that we must use the address not its content.

If I press **Q**, it shifts to:



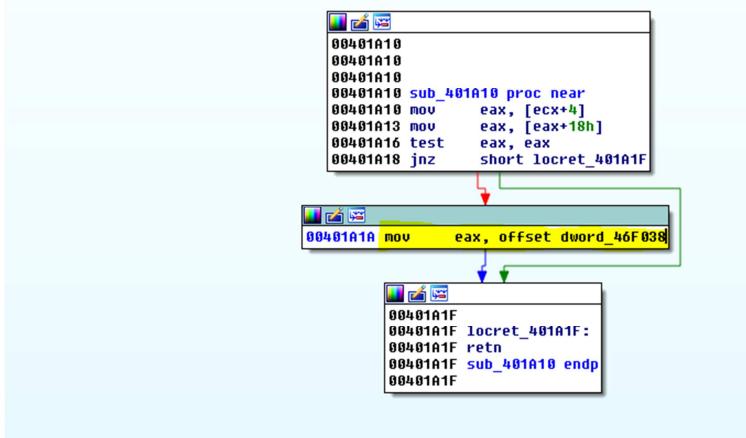
## MOV EAX, 46F038

That is an instruction like in OLLY, but it doesn't give us any info about its content of that address. If we right click on the address 46F038, we can go back to the original instruction.



And it will remain like before.

What does that IDA extra info tell me about that memory address?



If I open the HEX DUMP window and look for that address, I see that there are 0's initially. I could know that it is a DWORD, but I don't really know that because it depends on where it uses that value in the program what define the variable type.

0046EFF8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....?AVbad_
0046F000	84 9C 45 00 00 00 00 00 00 00 2E 0F 41 56 62 61	64 5F .....?AVbad_
0046F010	63 61 73 74 40 73 74 64 40 00 00 84 9C 45 00	cast@std@@..?EE.
0046F020	00 00 00 00 2E 3F 41 56 65 00 00 63 65 70 74 69	6F .....?AVexception
0046F030	6E 40 73 74 64 40 40 00 00 00 00 00 00 00 00 00	BC 12 43 00 n@std@@.....+..C.
0046F040	58 12 43 00 00 00 00 00 00 1B C0 EC A0 33 F3 0B 43	X.C.....+ý3%.C
0046F050	A5 07 33 A9 11 57 A3 F7 01 00 00 00 84 9C 45 00	N.3@.Wú,....?EE.
0046F060	00 00 00 00 2E 3F 41 56 43 41 74 6C 45 78 63 65	.....?AVCAt1Exce
0046F070	70 74 69 6F 6E 40 41 54 4C 40 40 00 84 9C 45 00	ption@ATL@@.?EE.
0046F080	00 00 00 00 2E 3F 41 56 62 61 64 5F 61 6C 6C 6F	.....?AVbad_alloc@std@@.?EE.
0046F090	63 40 73 74 64 40 40 00 84 9C 45 00 00 00 00 00	c@std@@.?EE.....
0046F0A0	2E 3F 41 56 6C 6F 67 69 63 5F 65 72 72 6F 72 40	.?AVlogic_error@
0046F0B0	73 74 64 40 40 00 00 00 84 9C 45 00 00 00 00 00	std@@...?EE.....
0046F0C0	2E 3F 41 56 6C 65 6E 67 74 68 5F 65 72 72 6F 72	.?AVlength_error@

If I come back to the disassembly and double click on the address...

External symbol

IDA View-A    Hex View-2    Occurrences of: offset    Hex View-1    Structure

```

.data:0046F034      db 64h ; d
.data:0046F035      db 40h ; @
.data:0046F036      db 40h ; @
.data:0046F037      db 0
.data:0046F038      dword_46F038  dd 0 ; DATA XREF: sub_401A10+A10
.data:0046F038
.data:0046F03C      dd offset aDvXmetaldetect ; "dv::XMetalDetectedDialog"
.data:0046F040      dd offset unk_431258
.data:0046F044      align 8
.data:0046F048      ; rh::Guid unk_46F048
.data:0046F048      unk_46F048   db 18h ; DATA XREF: sub_404A80+48To
.data:0046F048
.data:0046F049      db 0C0h ; +
.data:0046F04A      db 0ECh ; Ú
.data:0046F04B      db 0A0h ; á
.data:0046F04C      db 33h ; 3
.data:0046F04D      db 0F3h ; %4
.data:0046F04E      db 0Bh
.data:0046F04F      db 43h ; C
.data:0046F050      db 0A5h ; Ñ
.data:0046F051      db 7
.data:0046F052      db 33h ; 3
.data:0046F053      db 0A9h ; @
.data:0046F054      db 11h
.data:0046F055      db 57h ; W
.data:0046F056      db 0A3h ; Ú
.data:0046F057      db 0F7h ; ,
.data:0046F058      db 1
.data:0046F059      db 0
.data:0046F05A      db 0
.data:0046F05B      db 0
.data:0046F05C      off_46F05C  dd offset off_459C84 ; DATA XREF: .rdata:0045BE54To
.data:0046F060      db 0
.data:0046F061      db 0

```

There, I will see IDA tells me that the content of that address is a DWORD. In IDA disassembly list, when we see memory zones that are not code like in this case, they belong to the data section. Of course, the first column is the addresses.

Just next to it, it says **dword\_46F038** that means that the content of that address is a DWORD. It is like a clarification of the address on the left. Then we have the data type **dd** that is a DWORD and then the value that contains that memory position that is 0.

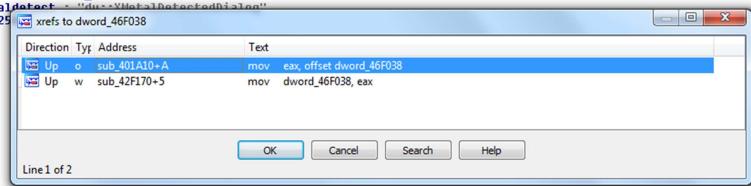
IDA is telling me the program uses that address like a DWORD and on the right side, I see the reference where that DWORD is being used.

```

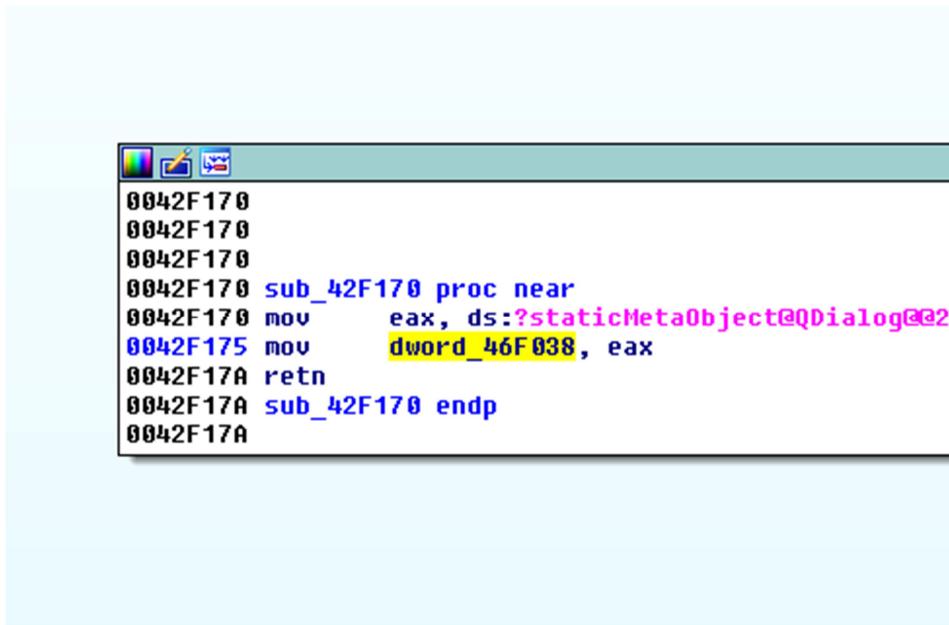
.udld 00400F032
.data:0046F036      db 40h ; @
.data:0046F037      db 0
.data:0046F038      dword_46F038  dd 0 ; DATA XREF: sub_401A10+A10
.data:0046F038
.data:0046F03C      dd offset aDvXmetaldetect ; "dv::XMetalDetectedDialog"
.data:0046F040

```

There are two references. Each arrow is one of them and hovering the cursor on that, I can see the code. Also, if I press X on that address, I see where that is referred to.



The first one is when it reads the address where we were before. The second one writes a DWORD in the content of 0x46F038. That is why IDA, in the first instruction, told us that that address pointed to a DWORD because there was another reference that accessed to it and there, it wrote a DWORD and everything is clear now.



So that, IDA in the original instruction not only told me that was going to move an address to a register, but it had a DWORD. That's something extra.

When it is about numerical addresses, IDA marks the address as OFFSET and when we look for the content of it, as in this case, it would be 0. It doesn't use brackets if it is a numerical address,

```
mov eax, offset dword_46F908
```

It moves the address 0x46F908 to EAX.

```
mov eax, dword_46F908
```

It moves the content or value found in that address.

```
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
0040119A mov esi, [ebp+arg_0]
0040119D push 0
0040119F lea ecx, [ebp+var_18]
004011A2 call ds:??0_Lockit@std@@QAE@H@
004011A8 mov [ebp+var_4], 0
004011AF mov eax, dword_46F908
004011B4 mov ecx, ds:?id@?$ctype@G@std@
004011BA mov [ebp+var_14], eax
004011BD call ds:??Bid@locale@std@@QAEI@_
004011C3 push eax
004011C4 mov ecx, esi
004011C6 call ds:_GetFacet@locale@std@@_
004011CC mov esi eax
```

This instruction would have brackets in OLLY for those who are used to that debugger.

MOV EAX, DWORD PTR DS:[46F908]

When an address has the word OFFSET before, it refers about the numerical value of that address. When it doesn't have that word it refers to the content of that address.

This just happens when we refer to numerical addresses if we work with registers only.

```
00401514 mov edi, eax
00401516 call ds:?begin@?$basic_string@GU?$c
0040151C mov eax, [edi+4]
0040151F mov ecx, [edi]
00401521 mov edx, [ebp+var_18]
00401524 push eax
00401525 pop
```

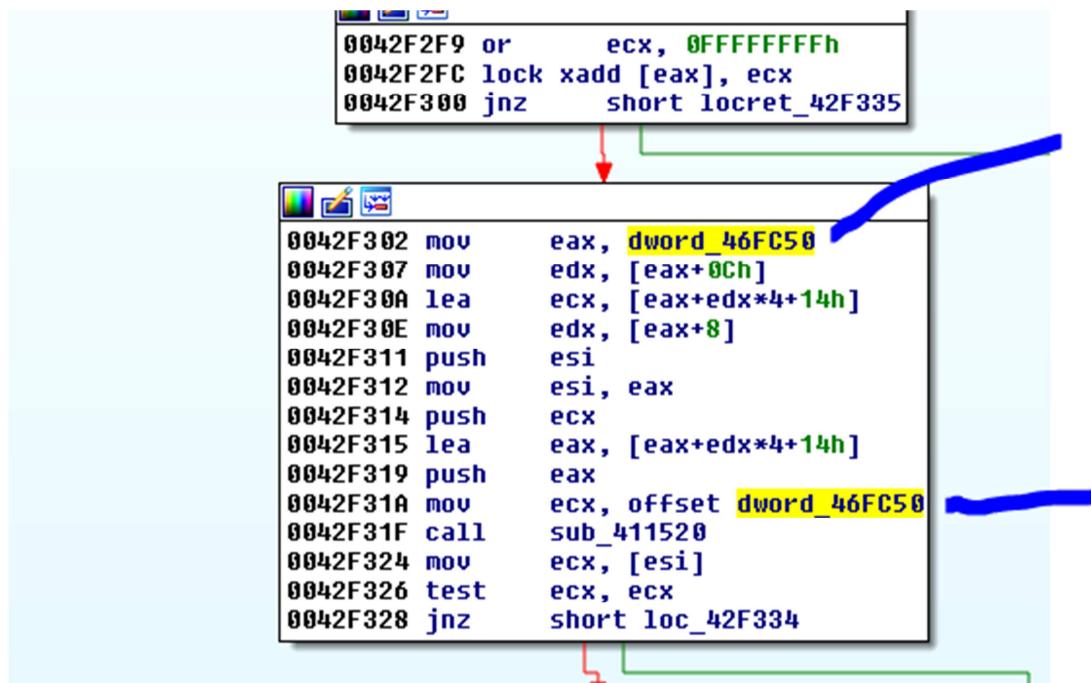
There, it uses brackets because, obviously, it doesn't know statically what value the register has at that moment and what address it will point to get more info.

Of course, in this case, If EDI points to 0x10000, that instruction will look for the content in the memory address and it will copy it in ECX.

It is important to understand when IDA uses the word OFFSET before an instruction it refers to the numerical value and not the content. Let do another example.

```
0042D1D9
0042D1D9 mov     edx, [esp+arg_4]
0042D1DD lea     eax, [edx+8Ch]
0042D1E0 mov     ecx, [edx-8Ch]
0042D1E3 xor     ecx, eax
0042D1E5 call    @_security_check_cookie@4 ; __secur
0042D1EA mov     eax, offset stru_45F4D0
0042D1EF jmp     _CxxFrameHandler3
0042D1EF SEH_415D90 endp
0042D1EF
```

There, EAX moves the value 0x45F4d0 because it has the word OFFSET and tells me that address has a **stru\_** that is a structure.



```
0042F2F9 or      ecx, 0xFFFFFFFFh
0042F2FC lock xadd [eax], ecx
0042F300 jnz    short locret_42F335

0042F302 mov     eax, dword_46FC50
0042F307 mov     edx, [eax+8Ch]
0042F30A lea     ecx, [eax+edx*4+14h]
0042F30E mov     edx, [eax+8]
0042F311 push   esi
0042F312 mov     esi, eax
0042F314 push   ecx
0042F315 lea     eax, [eax+edx*4+14h]
0042F319 push   eax
0042F31A mov     ecx, offset dword_46FC50
0042F31F call    sub_411520
0042F324 mov     ecx, [esi]
0042F326 test   ecx, ecx
0042F328 jnz    short loc_42F334
```

In this other case, in the first marked instruction it moves the content of 0x46FC50 that is a DWORD and in the next one it moves just the address that is the number 0x46FC50.

We can see what value the address has to see what it will move in 0x42F302. If I click on 0x46FC50. Let's see what is in there.

.data:0046FC4E	uu	u
.data:0046FC4F	db	0
.data:0046FC50 dword_46FC50	dd	0
.data:0046FC50	dd	0
.data:0046FC54 dword_46FC54	dd	0
.data:0046FC54	dd	0
.data:0046FC54	dd	0
.data:0046FC54	dd	0

; DATA XREF: sub\_416550+416↑o  
; sub\_416550+45E↑o ...  
; DATA XREF: sub\_412870↑r  
; sub\_412870+21↑w ...  
; DATA XREF: sub\_412870+21↑w ...

It will move a 0 if another instruction was not executed and saved any value there to modify it. As the references show us.

Pressing X on the references, we see that it saves a DWORD in that address.

Direction	Type	Address	Text
Up	o	sub_416550+416	mov ecx, offset dword_46FC50
Up	o	sub_416550+45E	mov ecx, offset dword_46FC50
Up	o	sub_416550+512	mov ecx, offset dword_46FC50
Up	o	sub_416550+52E	mov ecx, offset dword_46FC50
Up	o	sub_416550+557	mov ecx, offset dword_46FC50
Up	o	sub_416550+59F	mov ecx, offset dword_46FC50
Up	w	sub_42F270+5	mov dword_46FC50, eax
Up	r	sub_42F2F0	mov eax, dword_46FC50
Up	r	sub_42F2F0+12	mov eax, dword_46FC50
Up	o	sub_42F2F0+2A	mov ecx, offset dword_46FC50

Just in that yellow marked instruction it saves a DWORD in that memory address. The others read the address with OFFSET or just read the value.

Of course, constants can also be moved to 16-bit or 8-bit registers.

```
00427DE4
00427DE4 loc_427DE4:          ; jumptable 0042
00427DE4 push    0
00427DE6 mov     ecx, edi
00427DE8 call    ds:__toULongLong@QVariant@@QBE_KP
00427DEE mov     [esi], eax
00427DF0 mov     [esi+4], edx
00427DF3 mov     al, 1
00427DF5 mov     ecx, [ebp+var_C]
00427DF8 mov     large fs:0, ecx
00427DFF pop    ecx
00427E00 pop    edi
00427E01 pop    esi
00427E02 pop    ebx
00427E03 mov     esp, ebp
00427E05 pop    ebp
00427E06 retn
```

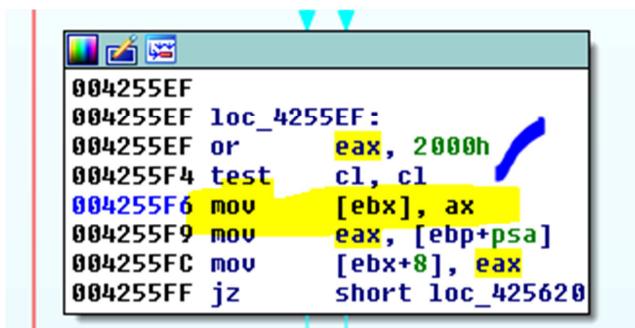
It moves 1 to AL leaving the rest of EAX with the value it had before. It just changes the lowest byte.

```
0042212B mov     cx, ds:word_459C26
00422132 mov     edx, ds:dword_459C20
00422138 mov     ax, ds:word_459C24
0042213E mov     [ebp+var_26], cx
00422142 push   edi
00422143 lea    ecx, [ebp+var_2C]
00422146 push   ecx
00422147 mov     ecx, [esi+60h]
```

There, it moves the content of the memory address 0x459C24 to AX and tells us it's a DWORD.

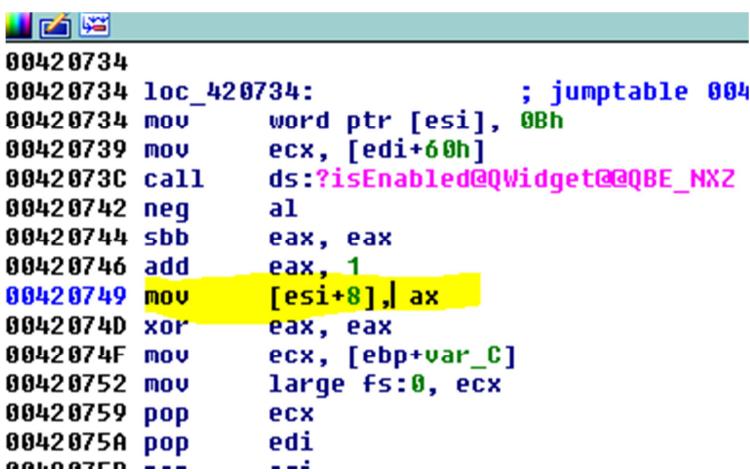
```
.rdata:00459C1E db    0
.rdata:00459C1F db    46h ; F
.rdata:00459C20 dword_459C20 dd 112h ; DATA XREF: sub_4220C0+72tr
.rdata:00459C24 word_459C24 dw 0 ; DATA XREF: sub_4220C0+78tr
.rdata:00459C26 word_459C26 dw 0 ; DATA XREF: sub_4220C0+6Btr
.rdata:00459C28 dword_459C28 dd 0C0h ; DATA XREF: sub_4220C0+8Dtr
.rdata:00459C2C dword_459C2C dd 46000000h ; DATA XREF: sub_4220C0+97tr
.rdata:00459C30 unk_459C30 db 18h ; DATA XREF: sub_420360:loc_4203D9tr
.rdata:00459C31 db 1
.rdata:00459C32 db 0
.rdata:00459C33 db 0
.rdata:00459C34 db 0
.rdata:00459C35 db 0
```

Initially, it is a 0. Maybe, more ahead when running the program it could be modified.



```
004255EF
004255EF loc_4255EF:
004255EF or    eax, 2000h
004255F4 test  cl, cl
004255F6 mov    [ebx], ax
004255F9 mov    eax, [ebp+psa]
004255FC mov    [ebx+8], eax
004255FF jz    short loc_425620
```

There, it moves AX to the content of EBX. As it is a register and it doesn't know what value it will have, it cannot tell more and it uses a bracket to indicate that it writes in the content of EBX.



```
00420734
00420734 loc_420734:          ; jumptable 004
00420734 mov    word ptr [esi], 0Bh
00420739 mov    ecx, [edi+60h]
0042073C call   ds:isEnabled@QWidget@@QBE_NXZ
00420742 neg    al
00420744 sbb    eax, eax
00420746 add    eax, 1
00420749 mov    [esi+8], ax
0042074D xor    eax, eax
0042074F mov    ecx, [ebp+var_C]
00420752 mov    large fs:0, ecx
00420759 pop    ecx
0042075A pop    edi
0042075D ---
```

There, it will write the value of AX in the content of ESI+8.

Another example:



```
004280E0
004280E0
004280E0
004280E8 sub_4280E8 proc near
004280E8 mov    eax, ds:staticMetaObject@QObject@@2UQMetaObject@@ ; QMetaObject const QObject::staticMetaObject
004280E5 retn
004280E5 sub_4280E8 endp
004280E5
```

If I click on the ugly name, it takes me to...

```
idata:0043081C          extr ??0QMutex@QAE@W4RecursionMode@0@@Z:dword
idata:0043081C                      ; CODE XREF: sub_42F250+7↑p
idata:0043081C                      ; DATA XREF: sub_42F250+7↑r
idata:00430820 ; public: static struct QMetaObject const QObject::staticMetaObject
idata:00430820          extr ?staticMetaObject@QObject@@2UQMetaObject@@B:dword
idata:00430820                      ; DATA XREF: sub_428E0@r
idata:00430820                      ; sub_42F290↑r
idata:00430824 ; public: __thiscall QMutex::~QMutex(void)
idata:00430824          extr ??0QMutex@QAE@W4RecursionMode@0@@Z:dword
idata:00430824          ; CODE XREF: sub_42F250+7↑p
idata:00430824          ; DATA XREF: sub_42F250+7↑r
```

We know that the **IAT** (table that saves the imported function addresses when running the executable) is almost always in the **idata** section.

If I look for that address in the HEX DUMP view, it doesn't still have the function value because the IAT is filled when the process starts and it hasn't started yet.

If I go to OPTIONS-DEMANGLE NAMES and I check NAMES, it looks a little better.

The prefix **extrn** means that it is an EXTERNAL imported function to this executable.

If we scroll up, we see that it tells us they are imported functions that belong to DVCORE.dll and above there are others that belong to different functions.

```
0x0000:00000000: Flags 00000000 Data read/write  
.idata:00430000 ; Alignment : default  
.idata:00430000  
.idata:00430000  
.idata:00430000  
.idata:00430000  
.idata:00430000  
.idata:00430000 ; Segment type: Externs  
.idata:00430000 ; _idata  
.idata:00430000 ; LSTATUS __stdcall RegCloseKey(HKEY hKey)  
    extrn RegCloseKey:dword ; CODE XREF: sub_402390+98Tp  
.idata:00430000 ; sub_402390+D7Tp ...  
.idata:00430000 ; LSTATUS __stdcall RegOpenKeyExW(HKEY hKey, LPCTSTR lpSubKey, DWORD dwOptions, REGSAM samDesired, PHKEY phKeyResult)  
    extrn RegOpenKeyExW:dword ; CODE XREF: sub_402390+55Tp  
.idata:00430000 ; sub_402390+55Tp  
.idata:00430000 ; DATA XREF: ...  
.idata:00430000 ; LSTATUS __stdcall RegQueryValueExW(HKEY hKey, LPCTSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)  
.idata:00430000 ; extrn RegQueryValueExW:dword ; CODE XREF: sub_402390+80Tp  
.idata:00430000 ; sub_402390+4CTp  
.idata:00430000 ; DATA XREF: ...  
.idata:00430000  
.idata:00430010 ; Imports From DUCore.dll  
.idata:00430010  
.idata:00430010  
.idata:00430010 ; extrn public: class dv::ActionBase * __thiscall dv::ActionManager::GetActionByIndex(enum dv::ActionIndex) const:dword  
.idata:00430010 ; CODE XREF: sub_407290+80Tp  
.idata:00430010 ; DATA XREF: sub_407290+80Tp ...  
.idata:00430014 ; extrn public: virtual __thiscall dv::SaveDialog::SaveDialog(void):dword  
.idata:00430014 ; CODE XREF: sub_407360+672Tp  
.idata:00430014 ; sub_407360+604Tp  
.idata:00430014 ; DATA XREF: ...  
.idata:00430018 ; extrn public: virtual __thiscall dv::SceneSaver::SceneSaver(void):dword  
.idata:00430018 ; CODE XREF: sub_407360+662Tp  
.idata:00430018 ; sub_407360+6C4Tp  
.idata:00430018 ; DATA XREF: ...  
.idata:00430010 ; extrn public: class std::basic_string<unsigned short, struct std::char_traits<unsigned short>, class std::allocator<unsigned short> > __thiscall dv::SaveDialog::SaveDialog(class QWidget *, class QString const &):dword  
.idata:00430010 ; CODE XREF: sub_406E90+87Tp  
.idata:00430010 ; DATA XREF: sub_406E90+87Tp  
.idata:00430010  
.idata:00430020 ; extrn public: __thiscall dv::SaveDialog::SaveDialog(class QWidget *, class QString const &):dword  
.idata:00430020 ; CODE XREF: sub_406E90+87Tp  
.idata:00430020 ; DATA XREF: sub_406E90+87Tp
```

We've seen different examples of MOV that you can practice and see in IDA with the executable I attached.

In part 4, we will continue with more instructions.

**Ricardo Narvaja**

**Translated by: @IvinsonCLS**