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?

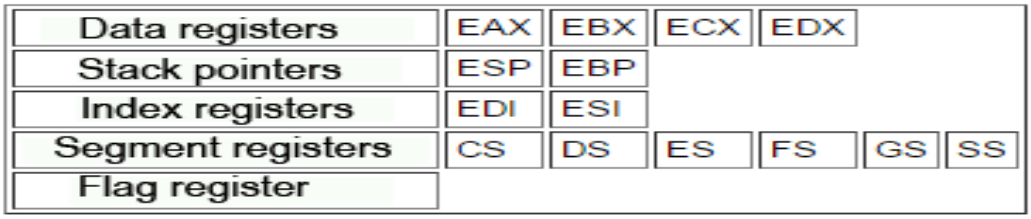
The processor needs assistants when executing programs.

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.



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.

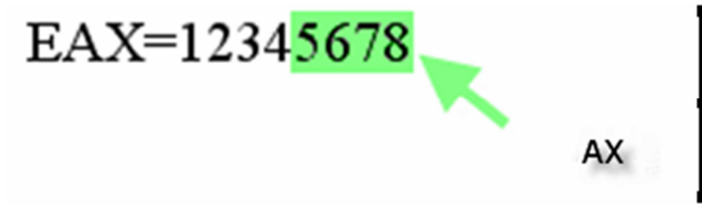
Summarizing what I copied from a guy.

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

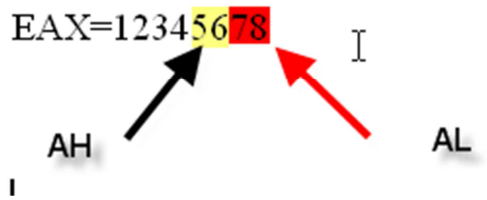
There are 16-bit registers and 8-bit registers that are part of the previous registers.

If EAX equals 12345678

AX is the last four digits (16 bits)

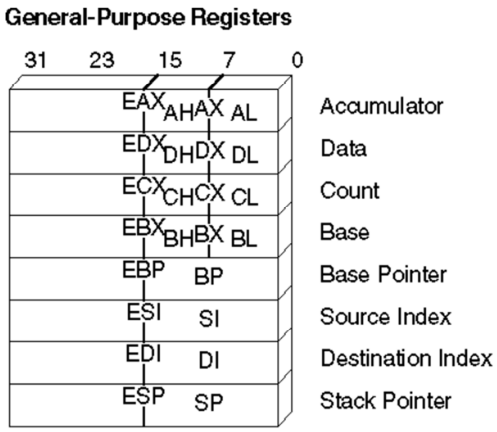


AH is the 5th and 6th digit and AL is the two last digits (8 bits each one)

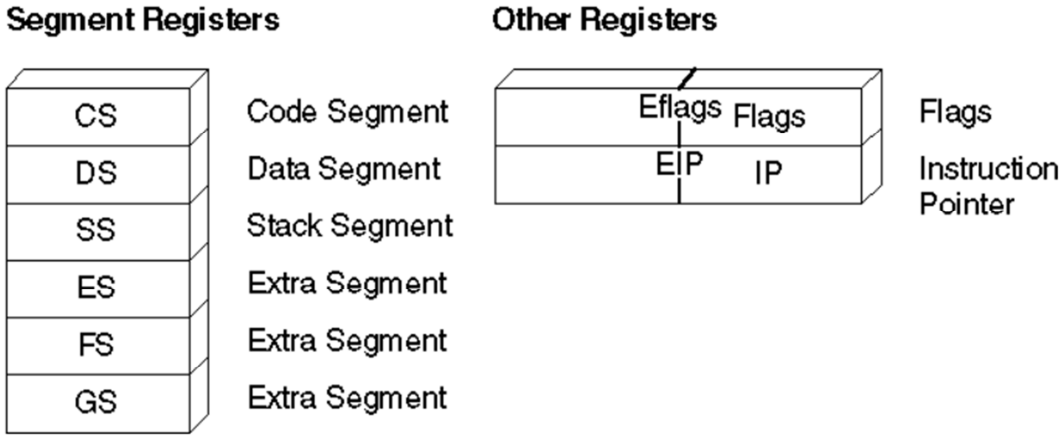


There is a 16-bit register for the low part of EAX called AX and two 8-bit registers called AH and AL. There are not special registers for the high part of any register.

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.



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

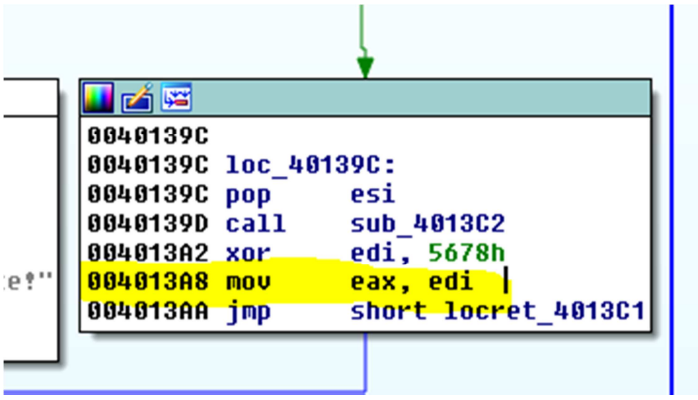
MOV

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



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.

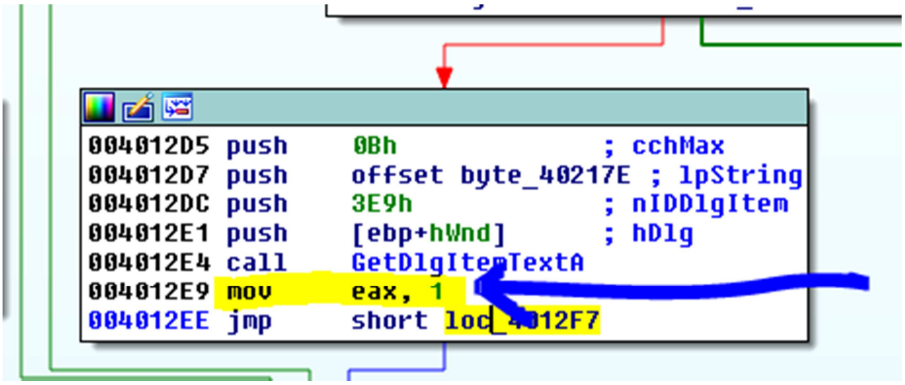
We cannot do the following:

MOV EIP, EAX

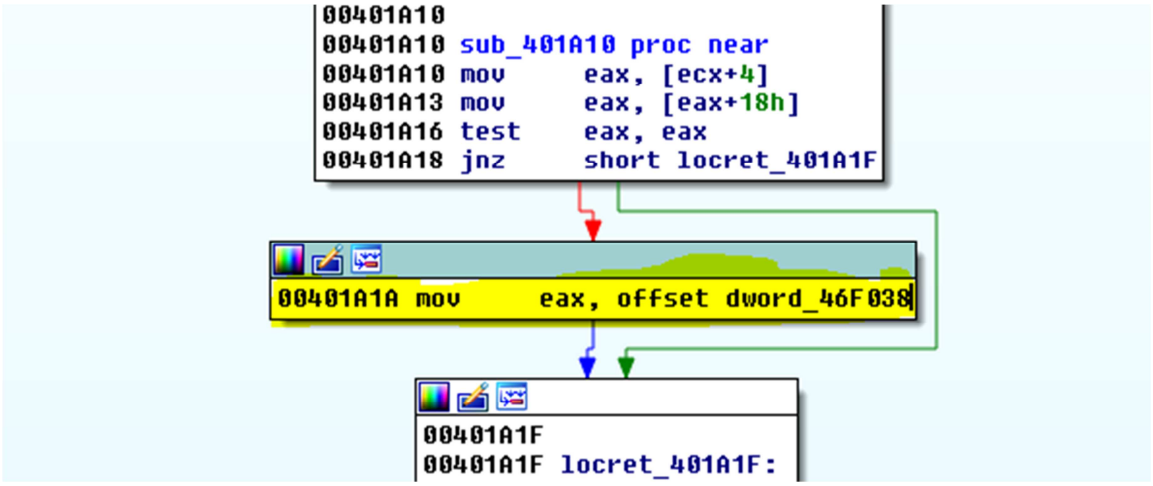
That's not valid.

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

MOV EAX, 1

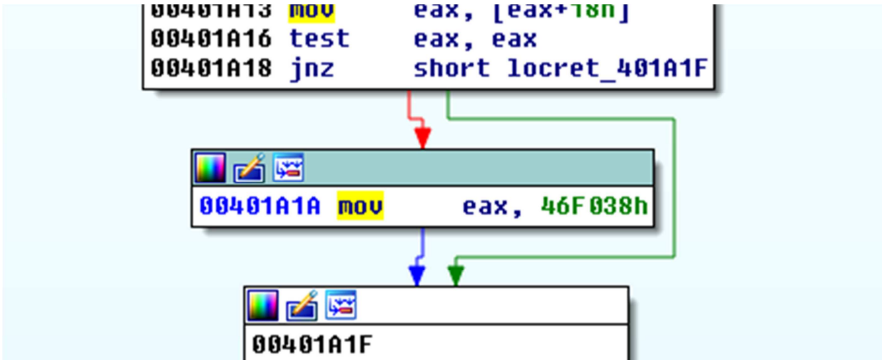


We can also move the value of a memory address not its content. (These instructions in the image belong to another executable not to the CRACKME.exe because they were not there but in VEViewer.exe attached to this part 3)



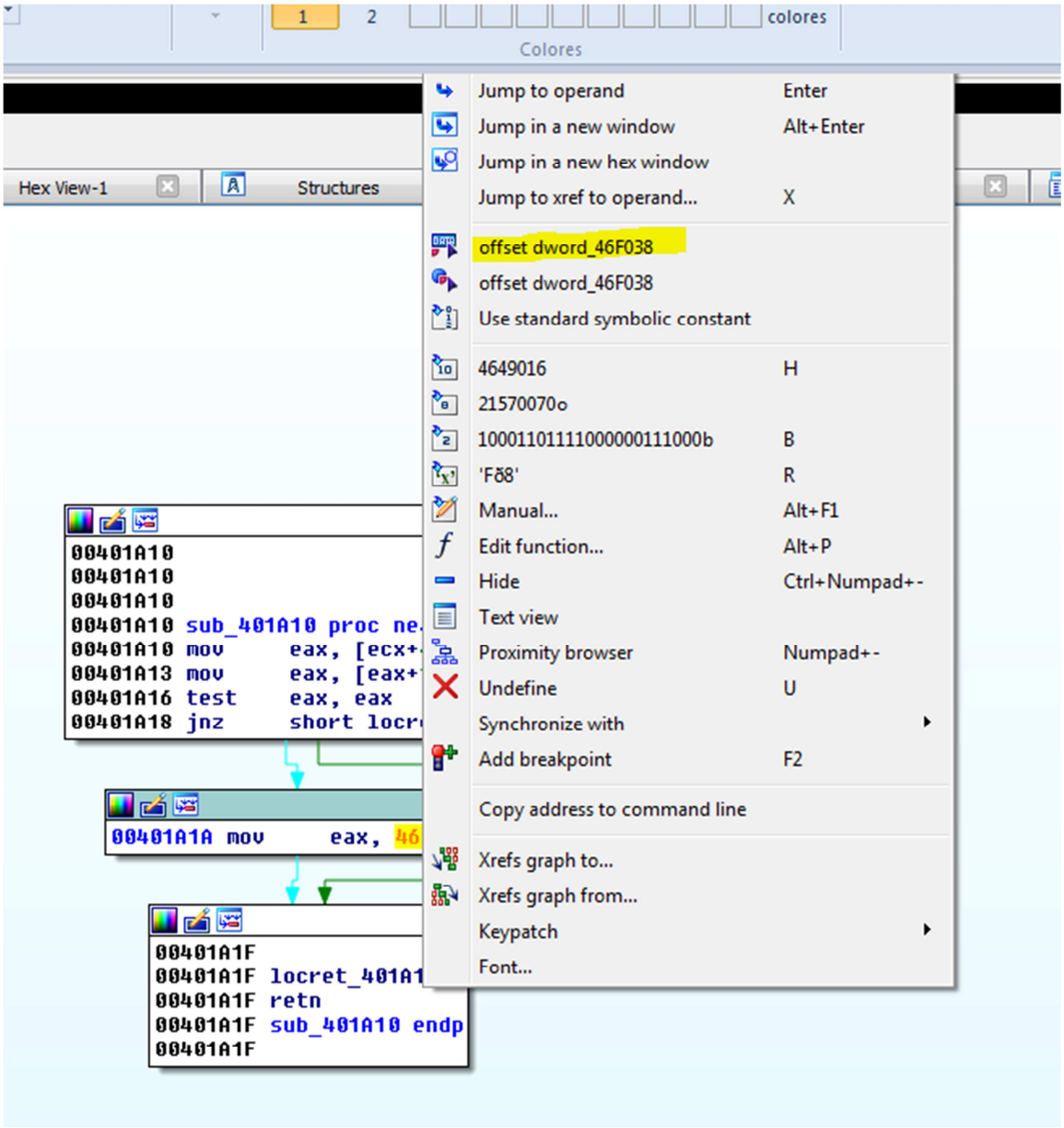
In that case, when the value to be moved is a memory address, the word OFFSET before it tells us that we must use the address not its content.

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



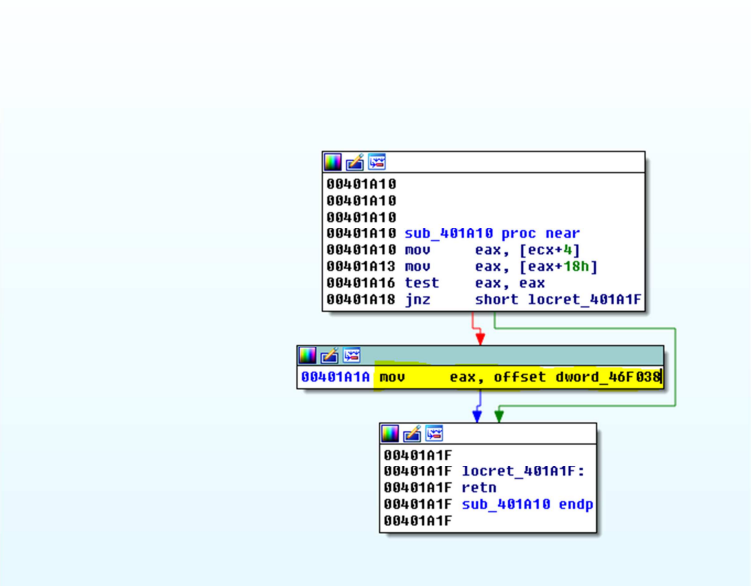
MOV EAX, 46F038

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.

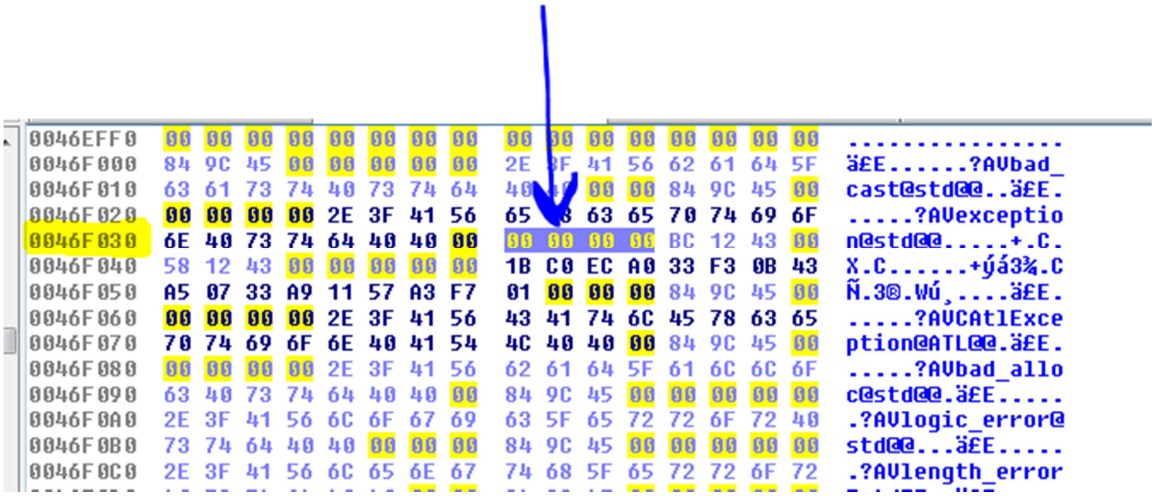


And it will remain like before.

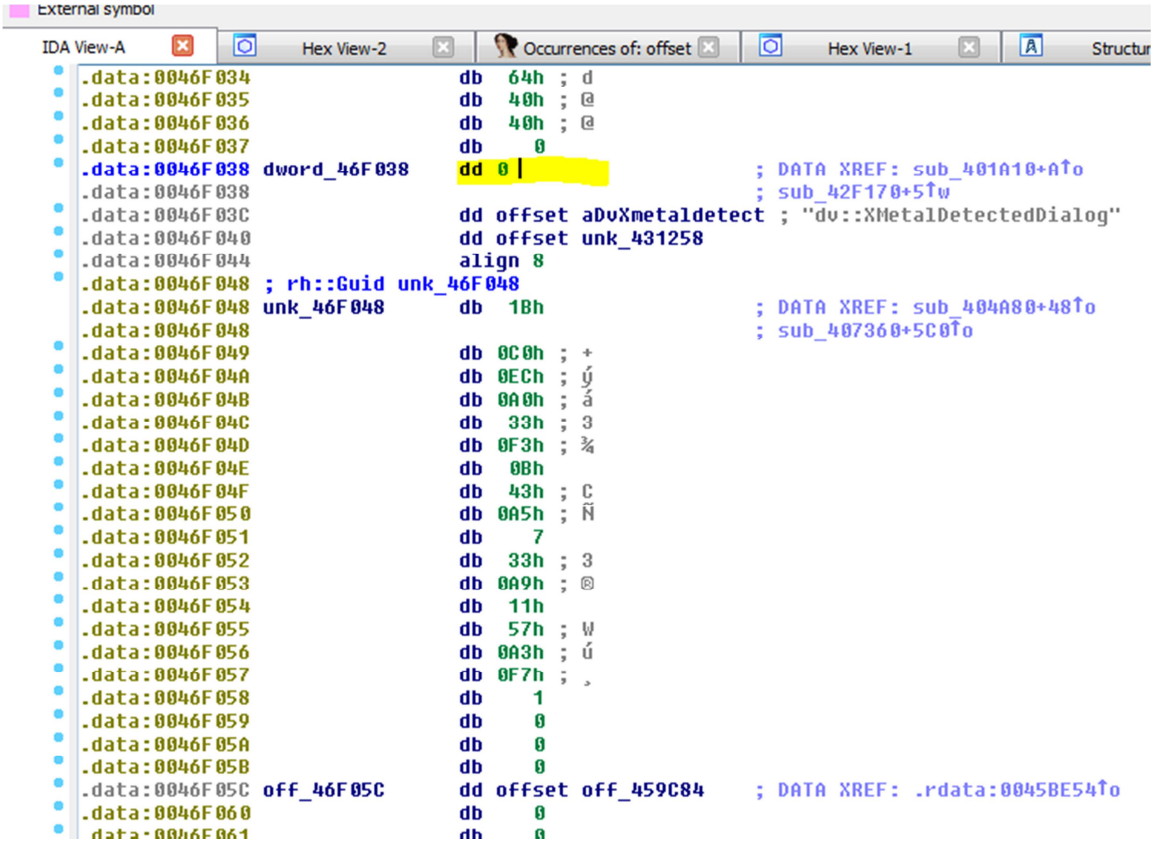
What does that IDA extra info tell me about that memory address?



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.



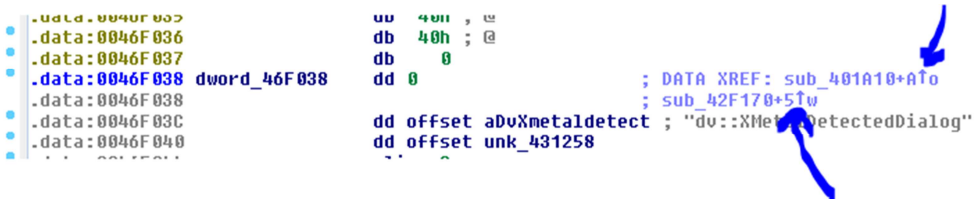
If I come back to the disassembly and double click on the address…



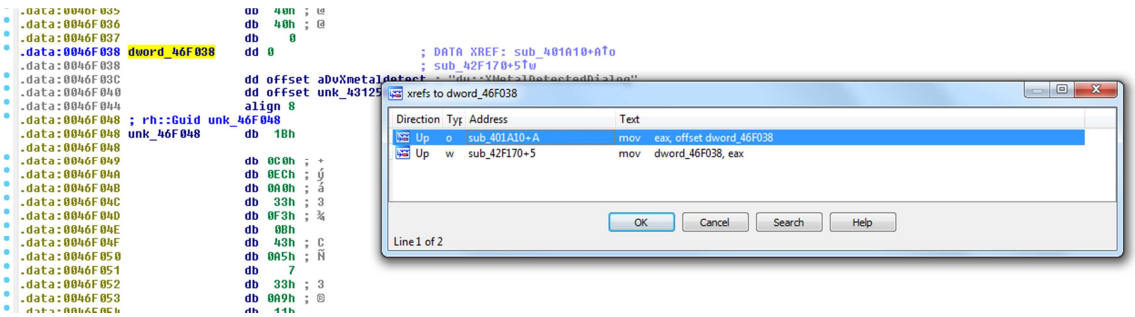
There, I will see IDA tells me that the content of that address is a DWORD. In IDA disassembly list, when we see memory zones that are not code like in this case, they belong to the data section. Of course, the first column is the addresses.

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.



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 uses 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@
```

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.

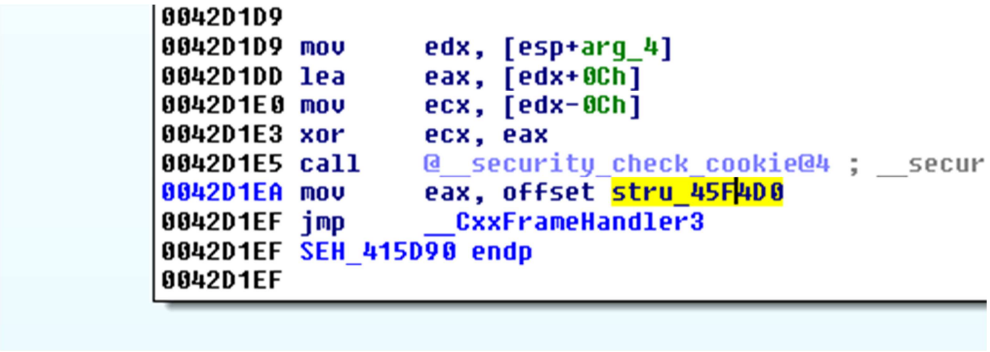
This just happens when we refer to numerical addresses if we work with registers only.

```
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
```

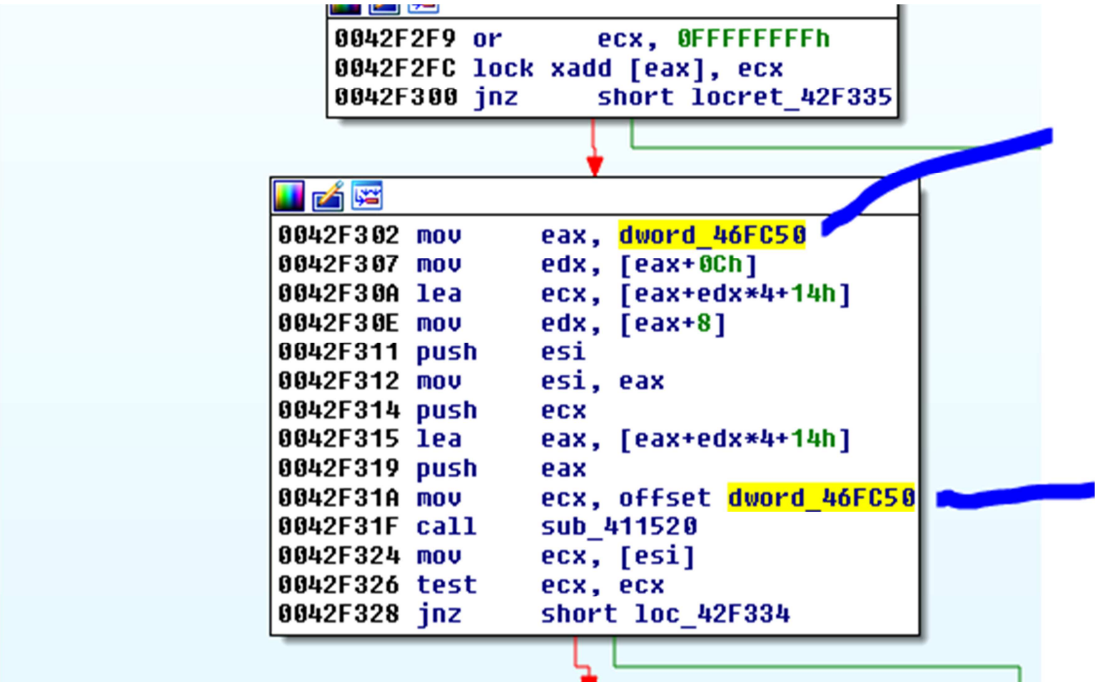
There, it uses brackets because, obviously, it doesn't know statically what value the register has at that moment and what address it will point to get more info.

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.

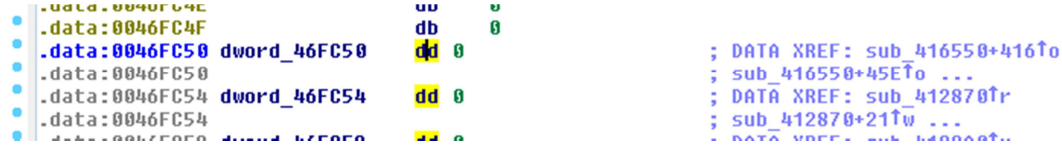


There, EAX moves the value 0x45F4d0 because it has the word OFFSET and tells me that address has a **stru**_ that is a structure.



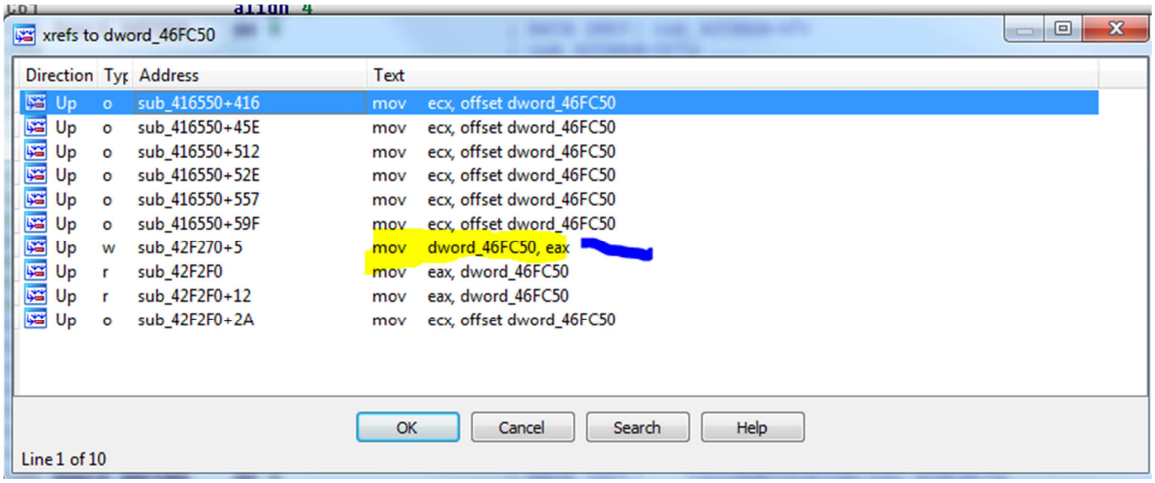
In this other case, in the first marked instruction it moves the content of 0x46FC50 that is a DWORD and in the next one it moves just the address that is the number 0x46FC50.

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.



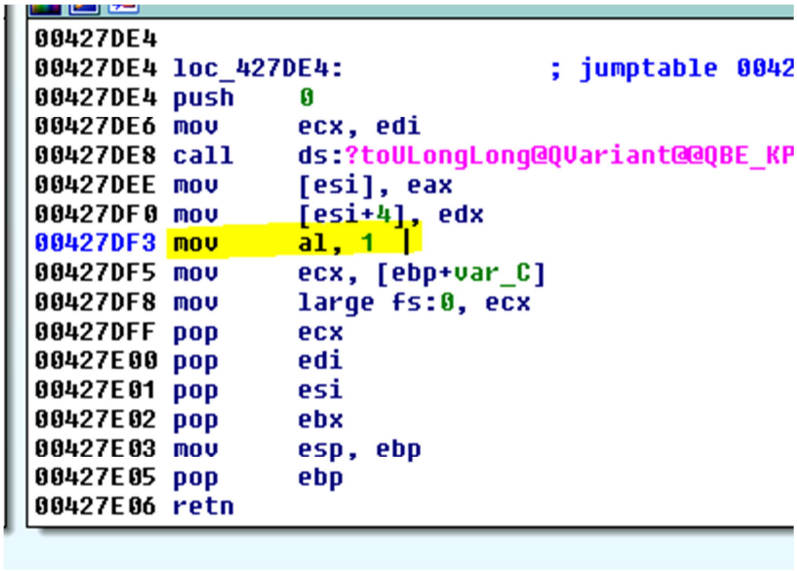
It will move a 0 if another instruction was not executed and saved any value there to modify it. As the references show us.

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

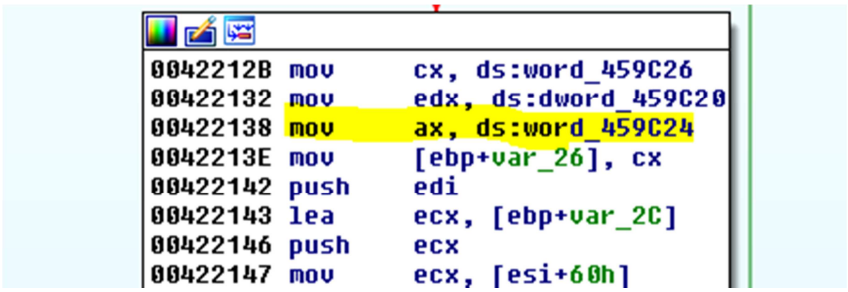


Just in that yellow marked instruction it saves a DWORD in that memory address. The others read the address with OFFSET or just read the value.

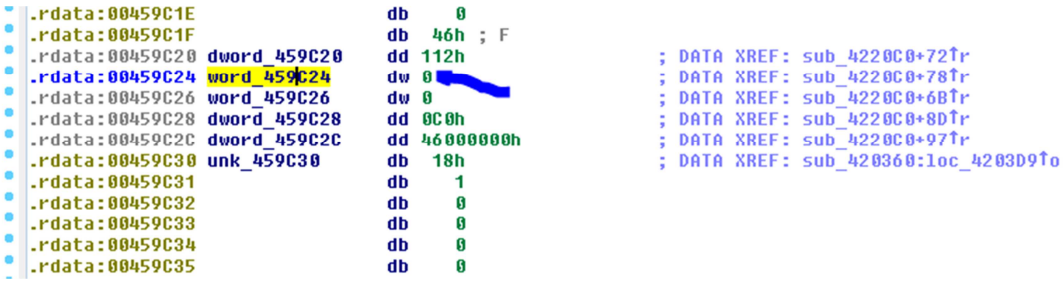
Of course, constants can also be moved to 16-bit or 8-bit registers.



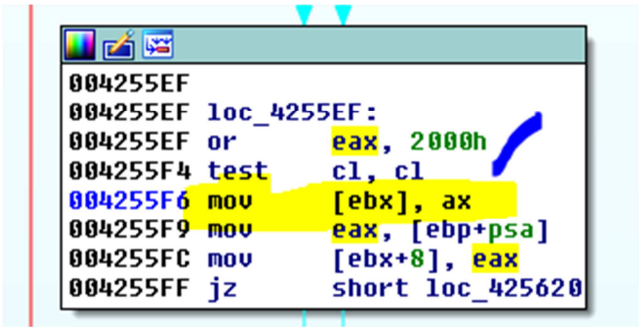
It moves 1 to AL leaving the rest of EAX with the value it had before. It just changes the lowest byte.



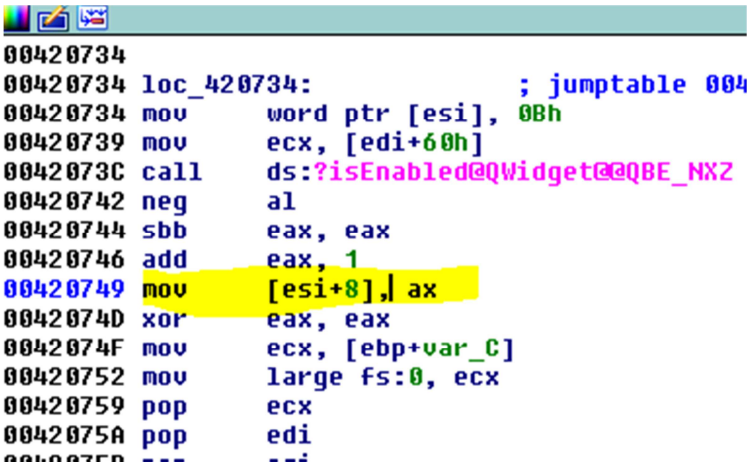
There, it moves the content of the memory address 0x459C24 to AX and tells us it's a DWORD.



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



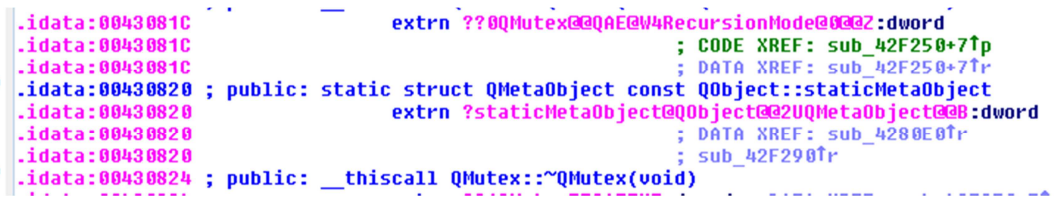
There, it moves AX to the content of EBX. As it is a register and it doesn't know what value it will have, it cannot tell more and it uses a bracket to indicate that it writes in the content of EBX.



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

Another example:

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

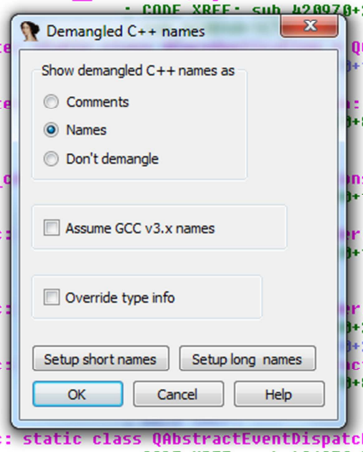


We know that the **IAT** (table that saves the imported function addresses when running the executable) is almost always in the **idata** section.

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.

```
.idata:00430818                                         ; DATA XREF: ...
.idata:0043081C                 extrn public: __thiscall QMutex::QMutex(enum  QMutex::RecursionMode):dword
.idata:0043081C                                         ; CODE XREF: sub_42F250+7↑p
.idata:0043081C                                         ; DATA XREF: sub_42F250+7↑r
.idata:00430820                 extrn public: static struct QMetaObject const QObject::staticMetaObject:dword
.idata:00430820                                         ; DATA XREF: sub_4280E0↑r
.idata:00430820                                         ; sub_42F290↑r
.idata:00430824                 extrn public: __thiscall QMutex::~QMutex(void):dword
.idata:00430824                                         ; DATA XREF: sub_42F2E0+5↑r
.idata:00430828                 extrn public: void __thiscall QObject::installEventFilter(class QObject *):dword
.idata:00430828                                         ; CODE XREF: sub_420970+3E↑p
.idata:00430828                                         ; DATA XREF: sub_420970+3E↑r
.idata:0043082C                 extrn public: void __thiscall QObject::removeEventFilter(class QObject *):dword
.idata:0043082C                                         ; CODE XREF: sub_420970+27↑p
.idata:0043082C
.idata:00430830                 extrn private                                    QCoreApplication::self:dword
.idata:00430830                                                                  )+1F↑r
.idata:00430830
.idata:00430834                 extrn private                                    ::notifyInternal(class QObject *, cl
.idata:00430834                                                                  )+84↑p
.idata:00430834
.idata:00430834
.idata:00430838                 extrn int __c                                    nst *):dword
.idata:00430838                                                                  )+1C↑p
.idata:00430838
.idata:0043083C                 extrn public:                                    r(int):dword
.idata:0043083C                                                                  )+108↑p
.idata:0043083C
.idata:0043083C
.idata:00430840                 extrn public:                                    r(int):dword
.idata:00430840                                                                  )+21↑p
.idata:00430840                                                                  )+21↑r
.idata:00430844                 extrn public:                                    ctEventDispatcher::setEventFilter(bo
.idata:00430844                                                                  )+85↑p
.idata:00430844
.idata:00430844
.idata:00430848                 extrn public: static class QAbstractEventDispatcher * __cdecl QAbstractEventDispatc
```



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.

```
.idata:00430000 ; Alignment     : default
.idata:00430000 ;
.idata:00430000 ; Imports from ADVAPI32.dll
.idata:00430000 ;
.idata:00430000 ; ===========================================================================
.idata:00430000
.idata:00430000 ; Segment type: Externs
.idata:00430000 ; _idata
.idata:00430000 ; LSTATUS __stdcall RegCloseKey(HKEY hKey)
.idata:00430000                 extrn RegCloseKey:dword ; CODE XREF: sub_402390+9B↑p
.idata:00430000                                         ; sub_402390+D7↑p ...
.idata:00430004 ; LSTATUS __stdcall RegOpenKeyExW(HKEY hKey, LPCWSTR lpSubKey, DWORD ulOptions, REGSAM samDesired, PHKEY phkResult)
.idata:00430004                 extrn RegOpenKeyExW:dword ; CODE XREF: sub_402390+55↑p
.idata:00430004                                         ; sub_40A860+2C↑p
.idata:00430004                                         ; DATA XREF: ...
.idata:00430008 ; LSTATUS __stdcall RegQueryValueExW(HKEY hKey, LPCWSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)
.idata:00430008                 extrn RegQueryValueExW:dword ; CODE XREF: sub_402390+8A↑p
.idata:00430008                                         ; sub_40A860+4C↑p
.idata:00430008                                         ; DATA XREF: ...
.idata:0043000C
.idata:00430010 ;
.idata:00430010 ; Imports from DVCore.dll
.idata:00430010 ;
.idata:00430010                 extrn public: class dv::ActionBase * __thiscall dv::ActionManager::GetActionByIndex(enum  dv::ActionIndex)const:dword
.idata:00430010                                         ; CODE XREF: sub_407290+80↑p
.idata:00430010                                         ; DATA XREF: sub_407290+80↑r ...
.idata:00430014                 extrn public: virtual __thiscall dv::SaveDialog::~SaveDialog(void):dword
.idata:00430014                                         ; CODE XREF: sub_407360+672↑p
.idata:00430014                                         ; sub_407360+6DA↑p
.idata:00430014                                         ; DATA XREF: ...
.idata:00430018                 extrn public: virtual __thiscall dv::SceneSaver::~SceneSaver(void):dword
.idata:00430018                                         ; CODE XREF: sub_407360+662↑p
.idata:00430018                                         ; sub_407360+6CA↑p
.idata:00430018                                         ; DATA XREF: ...
.idata:0043001C                 extrn public: class std::basic_string<unsigned short, struct std::char_traits<unsigned short>, class std::allocator<unsigne
.idata:0043001C                                         ; CODE XREF: sub_406E90+87↑p
.idata:0043001C                                         ; DATA XREF: sub_406E90+87↑r
.idata:00430020                 extrn public: __thiscall dv::SaveDialog::SaveDialog(class QWidget *, class QString const &):dword
```

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