

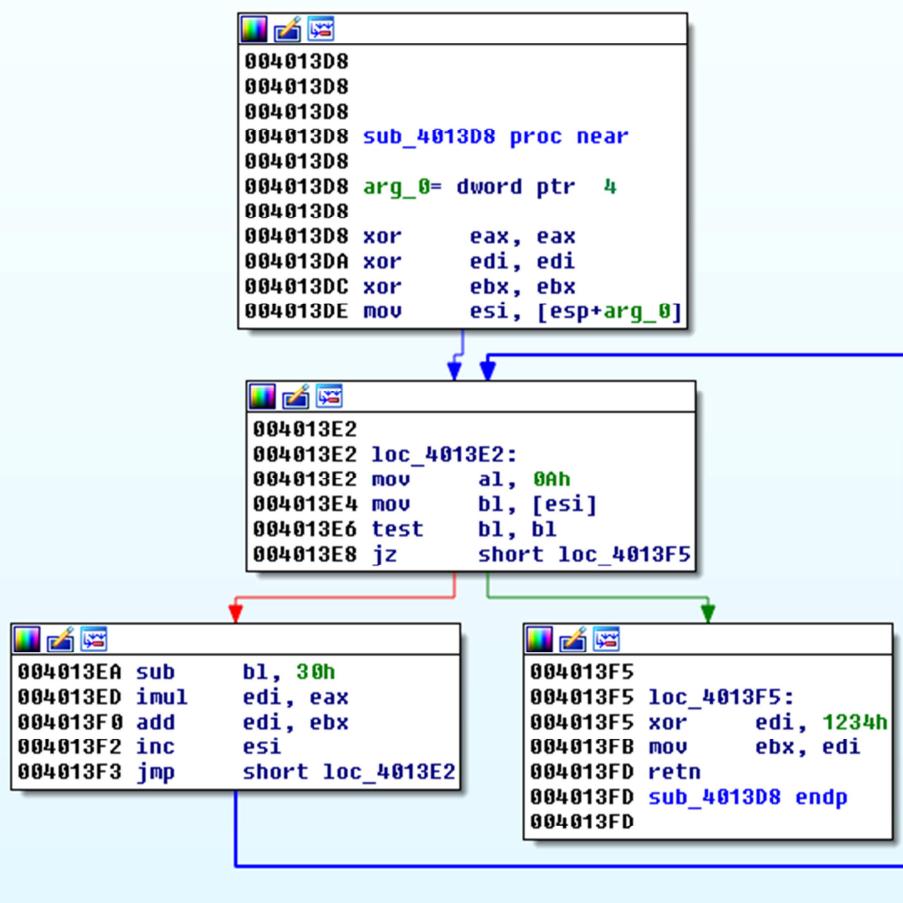
REVERSING WITH IDA PRO FROM SCRATCH

PART 4

Let's continue practicing the data transfer instructions in IDA.

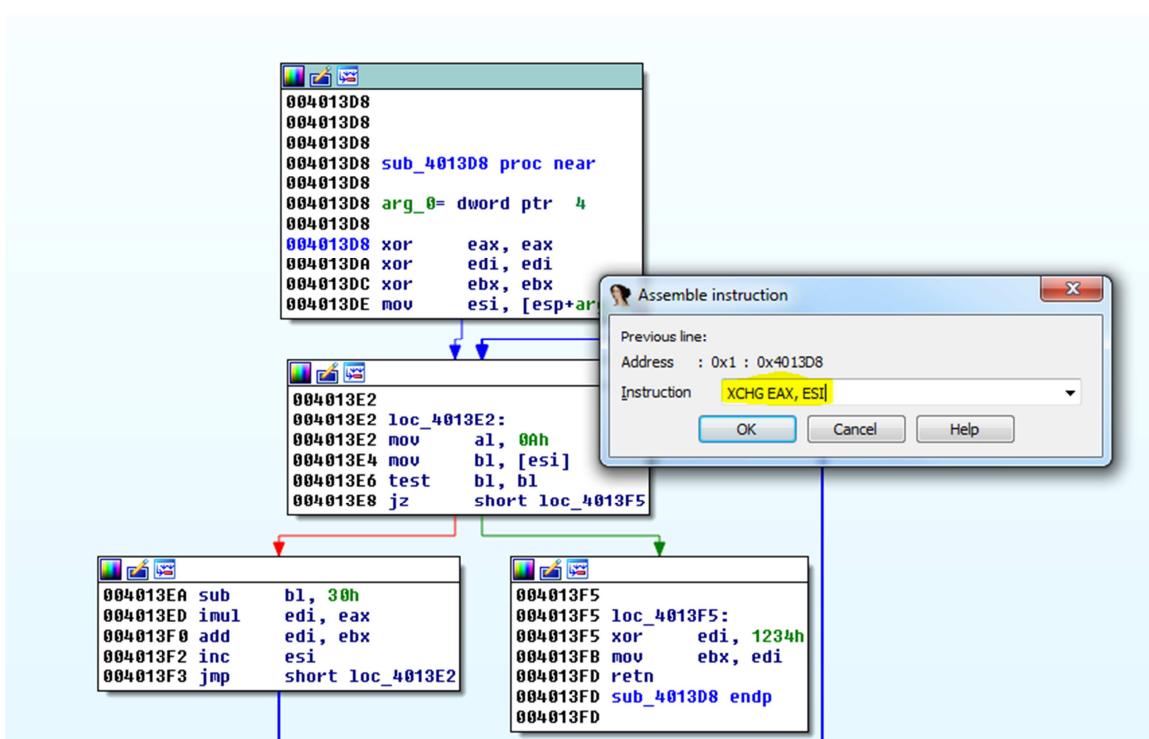
XCHG A, B

Interchange the value of A with the value of B. Let's see some example.

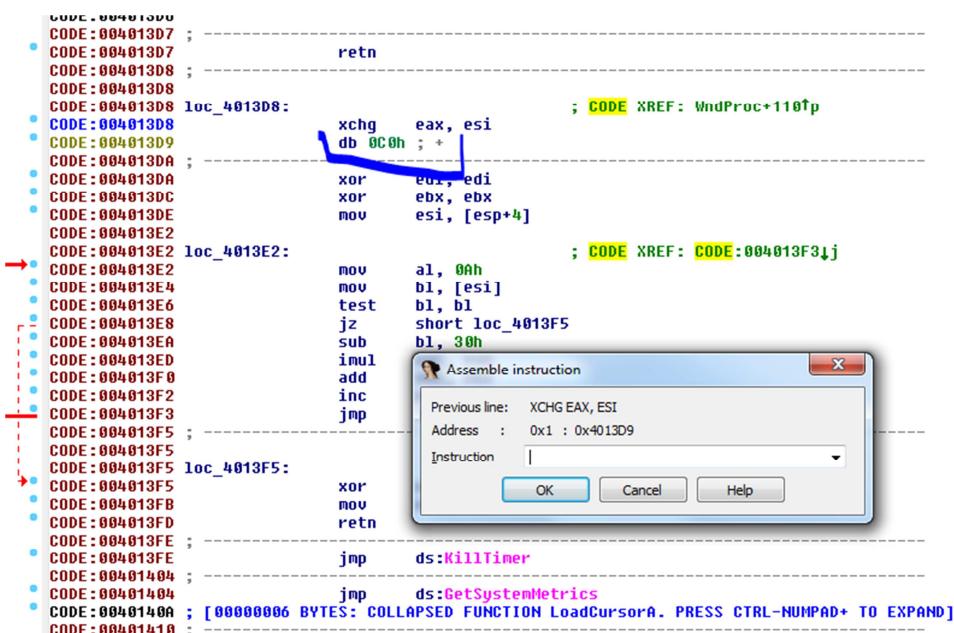


As there is no XCHG in VEWIEWER. I load Cruehead's CRACKME.exe and I will change the instruction in 0x4013D8.

I place the cursor on that line and go to Menu EDIT-PATCH PROGRAM-ASSEMBLE



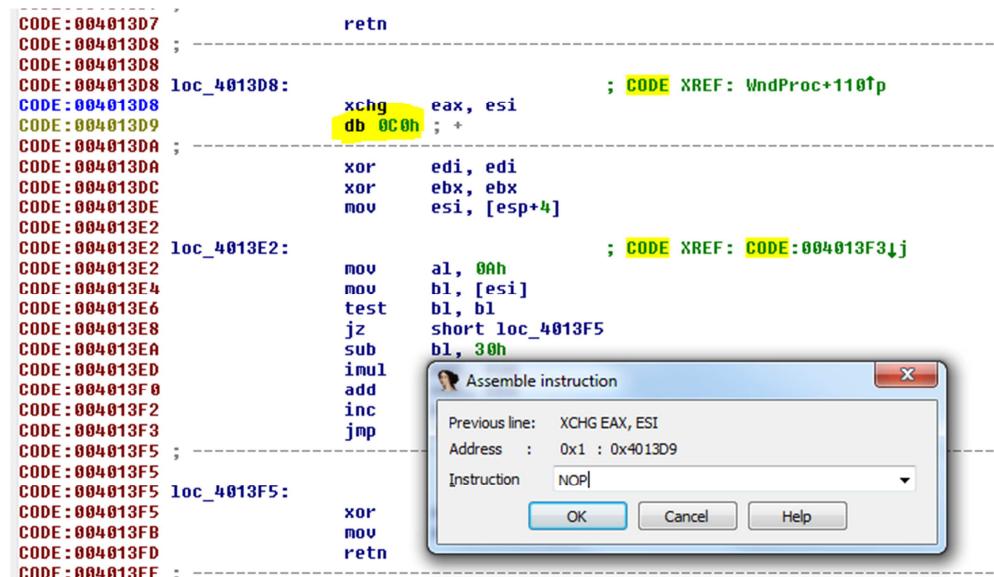
The function broke down.



Because we have already seen that if there is no a known instruction, that zone is shown as data.

In this case, it is just 1 byte. As there are no references in any part of the program, there is no info on the right side of the address. Then, the data type that is **db**. Just 1 byte and its value is **0xC0** here.

If we write the NOP instruction, that means NO OPERATION or a filling instruction that does nothing, to replace that **0xC0**.



The screenshot shows a debugger interface with assembly code. The code is as follows:

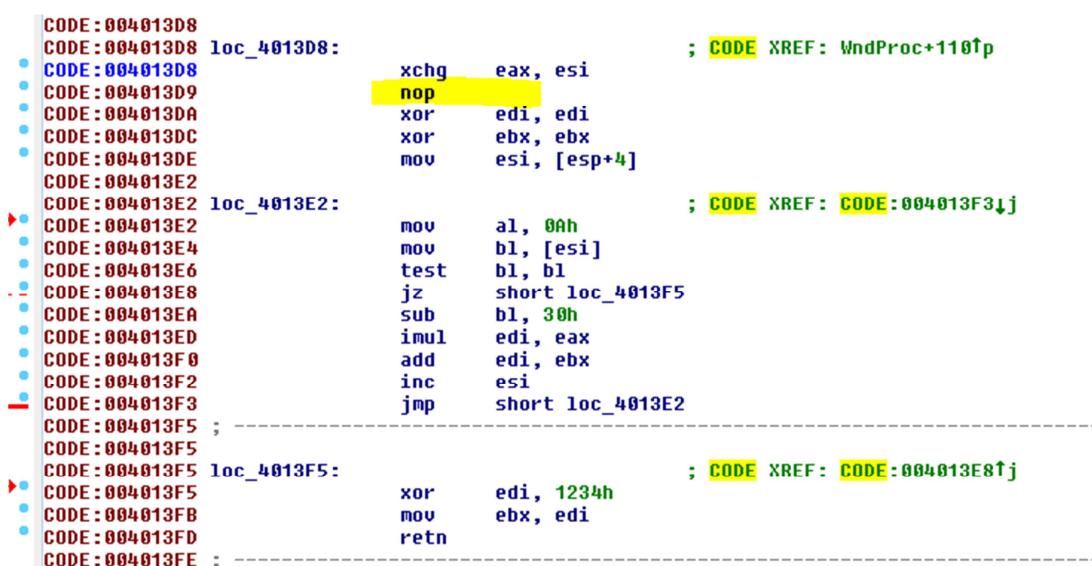
```
CODE:004013D7          ret
CODE:004013D8 ; -----
CODE:004013D8
CODE:004013D8 loc_4013D8:      xchg    eax, esi      ; CODE XREF: WndProc+110↑p
CODE:004013D8             db 0C0h ; +
CODE:004013D9 ; -----
CODE:004013DA             xor     edi, edi
CODE:004013DC             xor     ebx, ebx
CODE:004013DE             mov     esi, [esp+4]
CODE:004013E2
CODE:004013E2 loc_4013E2:      mov     al, 0Ah      ; CODE XREF: CODE:004013F3↓j
CODE:004013E2             mov     bl, [esi]
CODE:004013E4             test    bl, bl
CODE:004013E6             jz     short loc_4013F5
CODE:004013E8             sub     bl, 30h
CODE:004013EA             imul    edi, eax
CODE:004013F0             add     edi, ebx
CODE:004013F2             inc     esi
CODE:004013F3             jmp     short loc_4013E2
CODE:004013F5 ; -----
CODE:004013F5 loc_4013F5:      xor     edi, 1234h    ; CODE XREF: CODE:004013E8↑j
CODE:004013F5             mov     ebx, edi
CODE:004013FD             ret
CODE:004013FE ; -----
```

A modal dialog box titled "Assemble instruction" is open, showing the following settings:

- Previous line: XCHG EAX, ESI
- Address : 0x1 : 0x4013D9
- Instruction: NOP

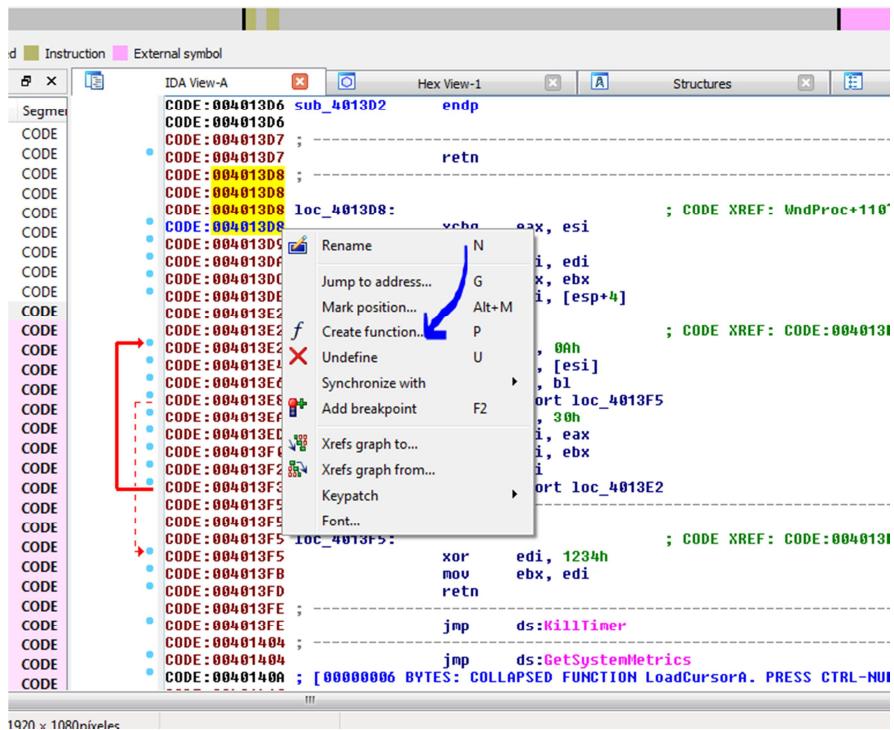
Buttons: OK, Cancel, Help.

There's the NOP.



The screenshot shows the assembly code with the NOP instruction highlighted in yellow. The code is identical to the one in the previous screenshot, but the NOP instruction at address 0x4013D9 is highlighted.

Everything's nice, but the function broke down. If there are unknown parts like code in the middle, it won't create it, but now that the code is OK, we right click on where it was the start of the function 0x4013D8 and there we select CREATE FUNCTION.

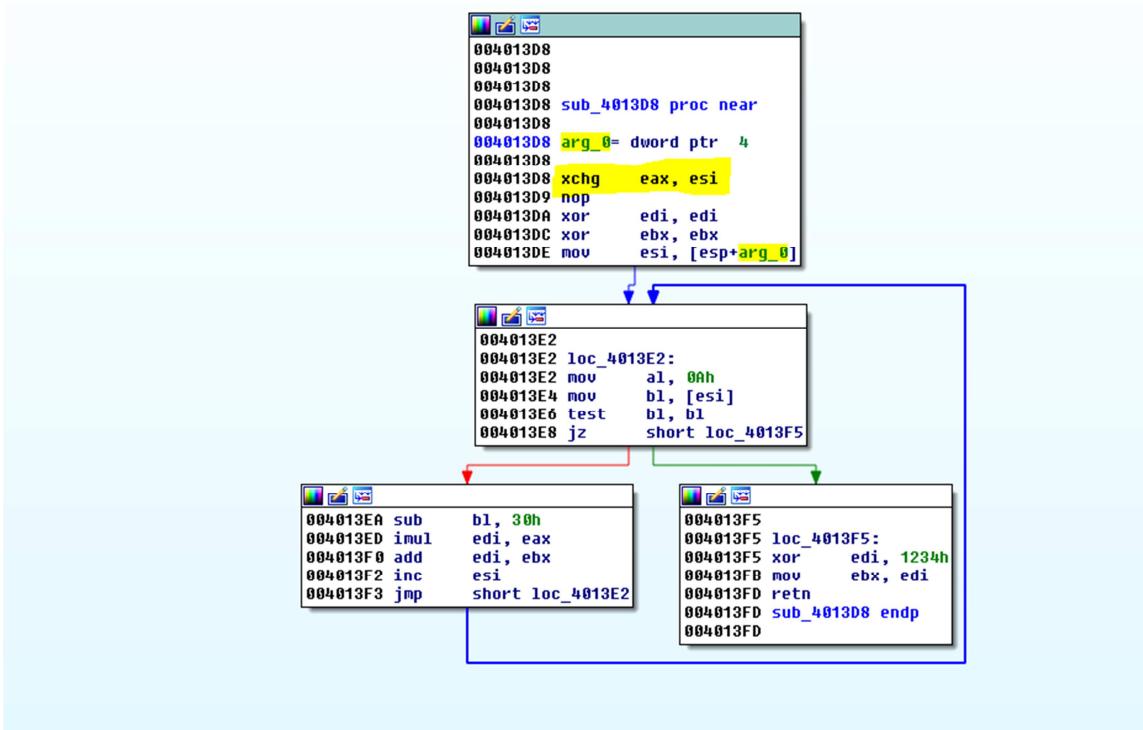


And it changes the prefix **loc_** that means “common instruction” for the prefix **sub_** that means “start of a subroutine or function.”

```

CODE:004013D8
CODE:004013D8 sub_4013D8 proc near ; CODE XREF: WndProc+110Fp
CODE:004013D8 arg_0 = dword ptr 4
CODE:004013D8
CODE:004013D8 xchg eax, esi
CODE:004013D9 nop
CODE:004013D9 xor edi, edi
CODE:004013D9 xor ebx, ebx
CODE:004013D9 mov esi, [esp+arg_0]
CODE:004013D9
CODE:004013D9 loc_4013E2: ; CODE XREF: sub_4013D8+1B+j
CODE:004013D9 mov al, 0Ah
CODE:004013D9 mov bl, [esi]
CODE:004013D9 test bl, bl
CODE:004013D9 jz short loc_4013F5
CODE:004013D9 sub bl, 30h
CODE:004013D9 imul edi, eax
CODE:004013D9 add edi, ebx
CODE:004013D9 inc esi
CODE:004013D9 jmp short loc_4013E2
CODE:004013D9
CODE:004013D9 loc_4013F5: ; CODE XREF: sub_4013D8+10+j
CODE:004013D9 xor edi, 1234h
CODE:004013D9 mov ebx, edi
CODE:004013D9 retn
CODE:004013D9 sub_4013D8 endp
CODE:004013D9
CODE:004013D9 ; jnp ds:KillTimer
CODE:00401404 ; jnp ds:GetSystemMetrics
CODE:00401404 ; [00000006 BYTES: COLLAPSED FUNCTION LoadCursorA. PRESS CTRL+NUMPAD+ TO EXPAND]
CODE:00401410 ; jnp ds:LoadAcceleratorsA
CODE:00401416 ; [00000006 BYTES: COLLAPSED FUNCTION MessageBeep. PRESS CTRL+NUMPAD+ TO EXPAND]
CODE:0040141C ; jnp ds:GetWindowRect
CODE:00401422 ; jnp ds:LoadStringA
CODE:00401428 ; [00000006 BYTES: COLLAPSED FUNCTION LoadIconA. PRESS CTRL+NUMPAD+ TO EXPAND]
CODE:0040142E ;
```

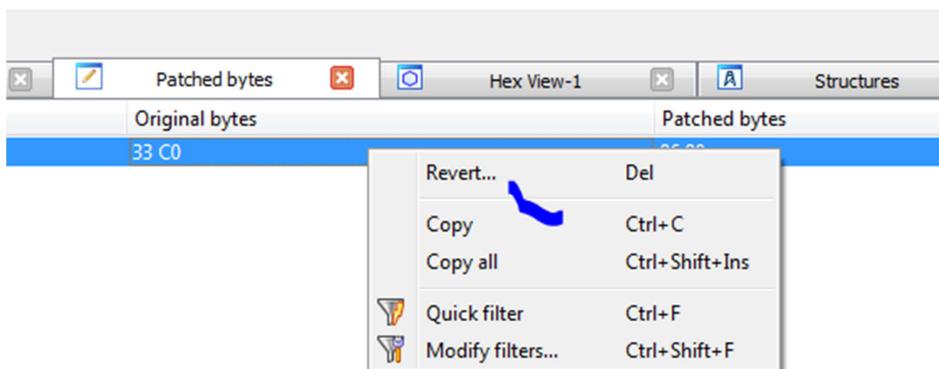
Now that it is a function, if it is not in graphic mode, we press the space bar.



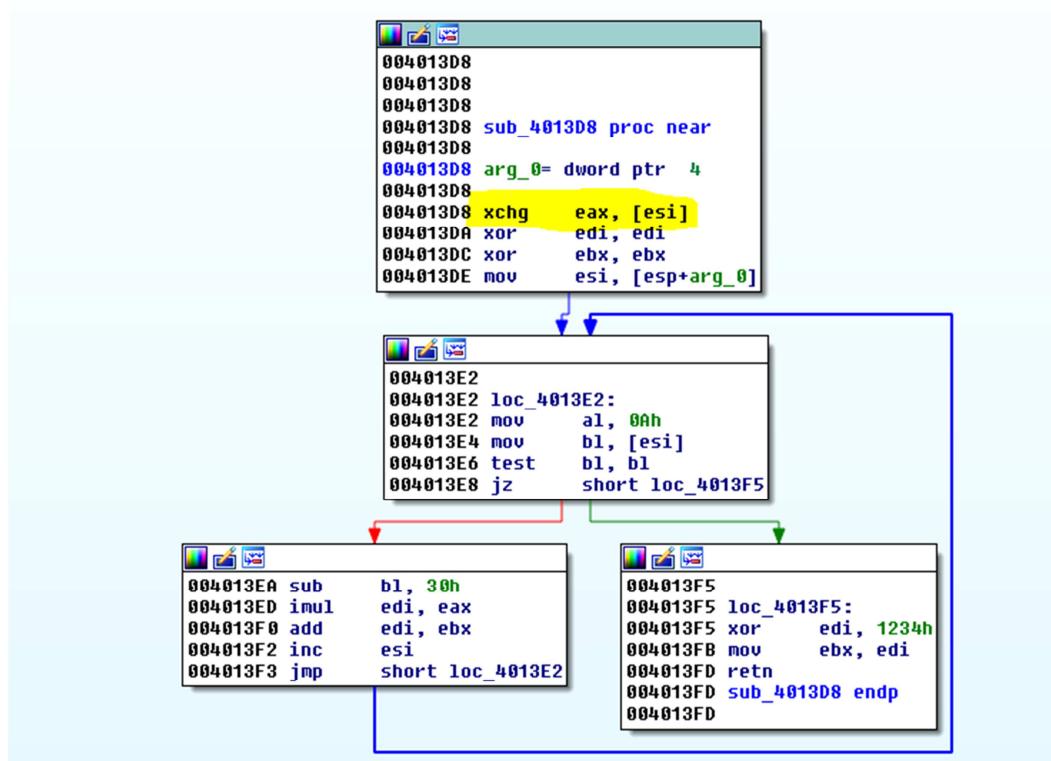
It looks better now and with our XCHG there.

We saw that if EAX equals 12345678 and ESI equals 55 when executing the XCHG the register EAX equals 55 and ESI equals 12345678 in this example.

As a comment, we see that in the PATCH menu there is an item called PATCHED BYTES that shows the changes and it can be reverted to the original ones.



XCHG can also interchange values between a register and the content of a register that points to a memory position.



In this case, the ESI value and it will search for the content in that memory position and it will interchange it for the EAX value that will be saved in that memory position if it has write permission.

If EAX equals 55 and ESI equals 0x10000, it will check what there is in that memory position and if it is writable, it will save there the 55 and it will read the value that was there and it will put it in EAX.

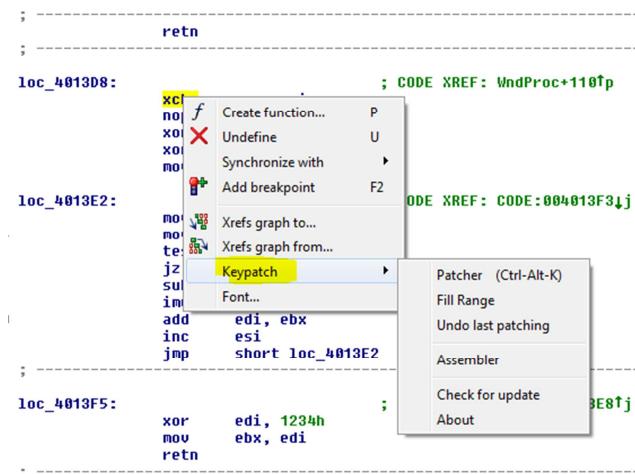
What happens if we do the same thing but instead of using a register we use a numerical memory position as we did in the MOV?

As the ASSEMBLE command doesn't work completely for all instructions, we should change the bytes there in the menu where it says PATCH BYTES, but it is better to download a plugin like the keystone that let us write all the instruction in an easy way.

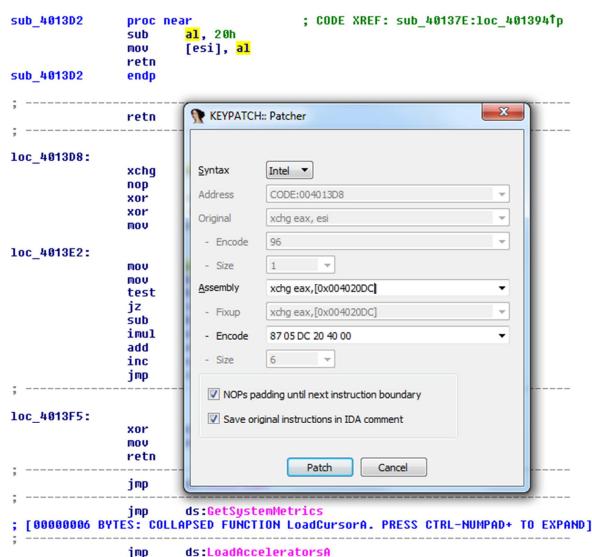
<https://github.com/keystone-engine/keypatch>

[https://drive.google.com/file/d/0B13TW0I0f8O2eU1VdUJzVjdYTWs/vie
w?usp=sharing](https://drive.google.com/file/d/0B13TW0I0f8O2eU1VdUJzVjdYTWs/view?usp=sharing)

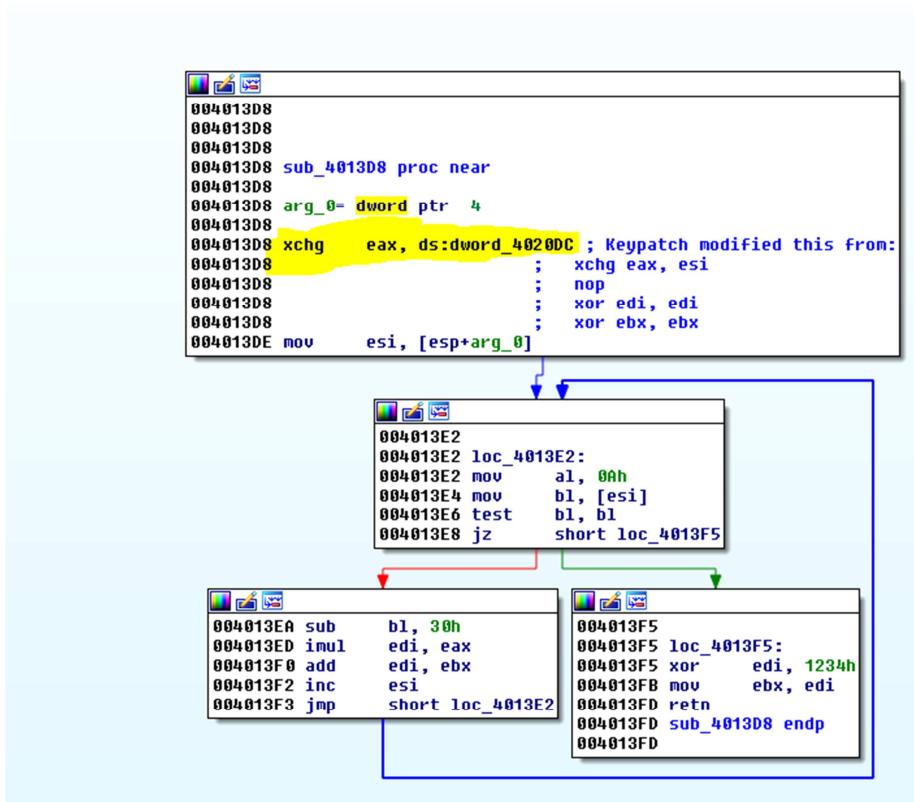
In the second link is the `keypatch.py` file that we must copy in the IDA plugin folder and the `keystone-0.9.1-python-win32.msi` setup we have to install.



If we select PATCHER, we see that if we write the instruction in the easiest way and with brackets, it will write it and transform it into IDA syntax.



And here is the result...



As in the MOV when it shows the prefix dword_ without OFFSET before it means that it interchanges the content of 0x4020DC for the EAX value.

STACK BASED TRANSFER SPECIFIC INSTRUCTIONS

WHAT IS THE STACK?

A stack is a memory section in which the access mode is FILO that means **First In, Last Out**. It permits to save and recover data.

.For the data handling, there are two basic operations: **PUSH** that places an object on the stack and its inverse operation **POP** that takes out the last pushed element.

At each moment, it only has access to the top of the stack or the last pushed object.

The **POP** operation gets this element that is popped from the stack permitting the access to the one that was below (pushed previously) that comes to be the last pushed object.

In the CRACKME.EXE, we see examples of both instructions.

```
0040101D mov ds:WndClass.style, 4003h
00401027 mov ds:WndClass.lpfnWndProc, offset WndProc
00401031 mov ds:WndClass.cbClsExtra, 0
00401038 mov ds:WndClass.cbWndExtra, 0
00401045 mov eax, ds:hInstance
0040104A mov ds:WndClass.hInstance, eax
0040104F push 64h ; lpIconName
00401051 push eax ; hInstance
00401052 call LoadIconA
00401057 mov ds:WndClass.hIcon, eax
0040105C push 7F00h ; lpCursorName
00401061 push 0 ; hInstance
00401063 call LoadCursorA
00401068 mov ds:WndClass.hCursor, eax
0040106D mov ds:WndClass.hbrBackground, 5
00401077 mov ds:WndClass.lpszMenuName, offset aMenu ; "MENU"
00401081 mov ds:WndClass.lpszClassName, offset ClassName ; "No need to di:
00401088 push offset WndClass ; lpWndClass
00401090 call RegisterClassA
00401095 push 0 ; lpParam
00401097 push ds:hInstance ; hInstance
0040109D push 0 ; hMenu
0040109F push 0 ; hWndParent
004010A1 push 8000h ; nHeight
004010A6 push 8000h ; nWidth
004010AB push 6Eh ; Y
004010AD push 0B4h ; X
004010B2 push 0CF0000h ; dwStyle
004010B7 push offset WindowName ; "CrackMe v1.0"
004010BC push offset ClassName ; "No need to disasm the code!"
004010C1 push 0 ; dwExStyle
004010C3 call CreateWindowExA
004010C8 mov ds:hWnd, eax
004010CD push 1 ; nCmdShow
004010CF push ds:hWnd ; hWnd
004010D5 call ShowWindow
004010DA push ds:hWnd ; hWnd
004010E0 call UpdateWindow
004010E5 push 1 ; bErase
004010E7 push 0 ; lpRect
004010E9 push dword ptr [ebp+8] ; hWnd
004010EC call InvalidateRect
```

Commonly, in 32 bits, the **PUSH** is used to send function arguments to the stack before calling it with a **CALL**. In the picture above, we see that example in 0x40104F.

PUSH 64 places the dword 64 on the top of the stack, then **PUSH EAX** places the **EAX** value on the dword **64** saving it and so the **EAX** value will be the top of the stack now.

ESP → EAX value

→ 0x64

There, we see different PUSH types. We can push constants, but we can also push memory addresses as in this case.

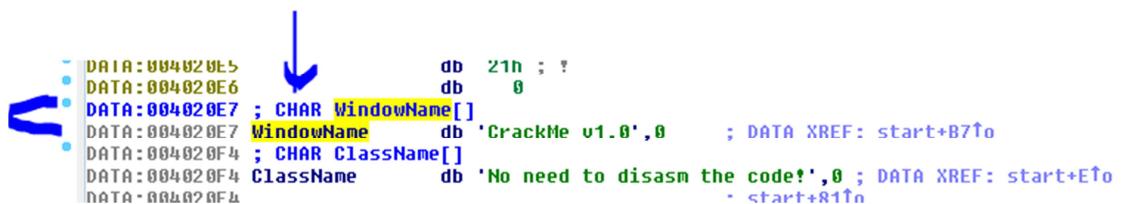
```
00401000 push    00000000 ; 0
00401002 push    0CF0000h   ; dwStyle
00401007 push    offset WindowName ; "CrackMe v1.0"
0040100C push    offset ClassName ; "No need to disasm the code!"
00401001 push    0           ; dwExStyle
00401002 call    CreateWindowExA
```

We see the word **OFFSET** before the TAG that belongs to a string. There, it would push the address whose content is a string or character array.

If we double click on the tag that represents the string name **WindowName**.

In C source code, an array of characters could be this way:

```
char mystring[] = "Hello";
```



```
DATA:004020E5 db 21h ; ?
DATA:004020E6 db 0
DATA:004020E7 ; CHAR WindowName[]
WindowName db 'CrackMe v1.0',0 ; DATA XREF: start+B7
DATA:004020F4 ; CHAR ClassName[]
ClassName db 'No need to disasm the code!',0 ; DATA XREF: start+E0
                                         - start+81
```

Here, it uses two lines for the variable description.

char **WindowName**[] is there because it detect from the api **CreateWindow** that receives the argument that must be an **LPCTSTR** that is a **char[]** and that argument is a string called **WindowName**.

CreateWindow function

Creates an overlapped, pop-up, or child window. It specifies the window class, window style, and extended window style. The function also specifies the window's parent or owner, if any, and the window's rectangle.

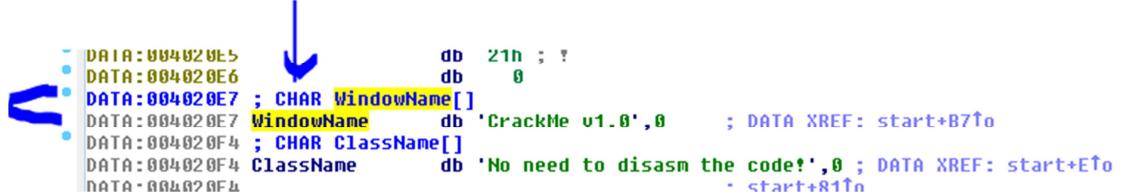
To use extended window styles in addition to the styles supported by [CreateWindow](#), call [CreateWindowEx](#).

Syntax

C++

```
HWND WINAPI CreateWindow(
    _In_opt_ LPCTSTR    lpClassName,
    _In_opt_ LPCTSTR    lpWindowName,
    _In_      DWORD     dwStyle,
    _In_      int       x,
    _In_      int       y,
    _In_      int       nWidth,
    _In_      int       nHeight,
    _In_opt_ HWND      hWndParent,
    _In_opt_ HMENU     hMenu,
    _In_opt_ HINSTANCE hInstance,
    _In_opt_ LPVOID    lpParam
);
```

Anyways, it is an array of characters or bytes and IDA adds it a little more info that it gets from the api. After 0x4020E7, the next address in the disassembly is 0x4020F4. There are some consecutive bytes that belong to the string characters of “Crackme v1.0” and the 0 that delimits the final of the string.

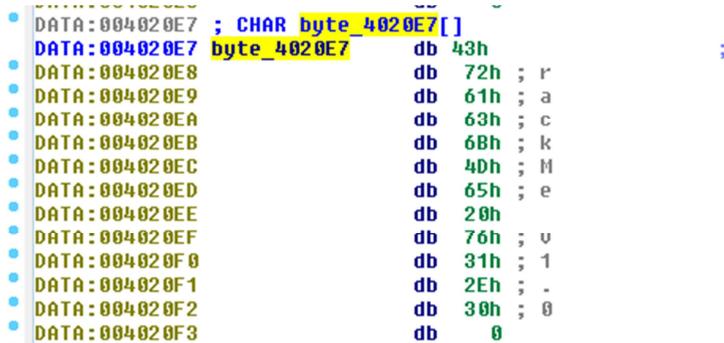


```

DATA:004020E5 db 21h ; ?
DATA:004020E6 db 0
DATA:004020E7 ; CHAR WindowName[]
WindowName db 'CrackMe v1.0',0 ; DATA XREF: start+B7$0
DATA:004020F4 ; CHAR ClassName[]
ClassName db 'No need to disasm the code!',0 ; DATA XREF: start+E1$0
DATA:004020F4 db 0

```

If we press D to change the data type on WindowName, we can make it stop detecting it as a character array and stay as **db** o byte.



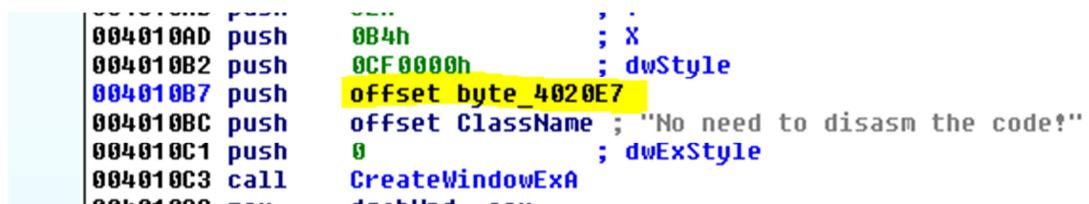
```

DATA:004020E7 ; CHAR byte_4020E7[]
byte_4020E7 db 43h ; r
DATA:004020E8 db 72h ; a
DATA:004020E9 db 61h ; a
DATA:004020EA db 63h ; c
DATA:004020EB db 68h ; k
DATA:004020EC db 40h ; M
DATA:004020ED db 65h ; e
DATA:004020EE db 20h
DATA:004020EF db 76h ; v
DATA:004020F0 db 31h ; 1
DATA:004020F1 db 2Eh ; .
DATA:004020F2 db 30h ; 0
DATA:004020F3 db 0

```

They are the same bytes that belong to the string Crackme v1.0...

The original instruction that it changed is in the reference, obviously, the word offset before keeps **0x4020E7** as a pushable value, but now that the content stopped being a character array and it is a byte now, the instruction changed to...



```

004010AD push 0B4h ; X
004010B2 push 0CF0000h ; dwStyle
004010B7 push offset byte_4020E7
004010BC push offset ClassName ; "No need to disasm the code!"
004010C1 push 0 ; dwExStyle
004010C3 call CreateWindowExA
004010C9 ret

```

push offset byte_4020e7

Because when it looks for the content of **0x4020E7** to tell us what it is, there is a **db**, that is to say, a variable changed by us into just a byte.

```

0000:004020E0
DATA:004020E7 ; CHAR byte_4020E7[1] 0
DATA:004020E7 byte_4020E7 db 43h
DATA:004020E8 db 72h ; r
DATA:004020E9 db 61h ; a
DATA:004020EA db 63h ; c
DATA:004020EB db 6Bh ; k
DATA:004020EC db 4Dh ; M
DATA:004020ED db 65h ; e
DATA:004020EE db 20h
DATA:004020EF db 76h ; v
DATA:004020F0 db 31h ; 1
DATA:004020F1 db 2Eh ; .
DATA:004020F2 db 30h ; 0
DATA:004020F3 db 0
DATA:004020F4 ; CHAR ClassName[]
DATA:004020F4 ClassName db 'No need to disasm thi
DATA:004020F5

```

Pressing A that is an ASCII string it shows it as before.

```

0000:004020E0 uu 0
DATA:004020E7 ; CHAR aCrackmeV1_0[1]
DATA:004020E7 aCrackmeV1_0 db 'CrackMe v1.0',0 ; DATA XREF: start+B7
DATA:004020F4 : CHAR ClassName[]1

```

We do the same if when we work we see any string as separated bytes.

```

0000:004020D1 db 0
0000:004020D2 db 0
0000:004020D3 db 0
0000:004020D4 db 0
0000:004020D5 db 0
0000:004020D6 db 54h ; T
0000:004020D7 db 72h ; r
0000:004020D8 db 79h ; y
0000:004020D9 db 20h
0000:004020DA db 74h ; t
0000:004020DB db 6Fh ; o
0000:004020DC db 20h
0000:004020DD db 63h ; c
0000:004020DE db 72h ; r
0000:004020DF db 61h ; a
0000:004020E0 db 63h ; c
0000:004020E1 db 6Bh ; k
0000:004020E2 db 20h
0000:004020E3 db 60h ; m
0000:004020E4 db 65h ; e
0000:004020E5 db 21h ; !
0000:004020E6 db 0
0000:004020F7 - CHAR aCrackmeV1_0[1]

```

We go to the start of it and press A and it will look better.

```
DATA:004020D4          uu      u
DATA:004020D5          db      0
DATA:004020D6 aTryToCrackMe db 'Try to crack me!',0
```

In this case, there is a string that is not defined in two lines as the previous one and it doesn't say it is a CHAR[], but it is just defined with a tag that starts with the letter **a** for being an ASCII string. In the previous case, it showed extra info because it detected that it was an api argument or system function and it told it that argument should be a **char[]** and that's why it add it there, but a normal string will look like this last one.

There, we see other string.

```
DATA:00402110 aMenu          db 'MENU',0          ;_Search.c... ...
; DATA XREF: start+77↑o
```

In 0x402110, the first byte of it can be separated pressing D in **aMenu**.

```
DATA:004020F4          -----
DATA:00402110 byte_402110      db 40h          ; start+81↑o ...
DATA:00402111          db 45h ; E          ; DATA XREF: start+77↑o
DATA:00402112          db 4Eh ; N
DATA:00402113          db 55h ; U
DATA:00402114          db 0
```

If we press A, we revert the changes.

If I press X to look for the reference and I see where it uses that string...

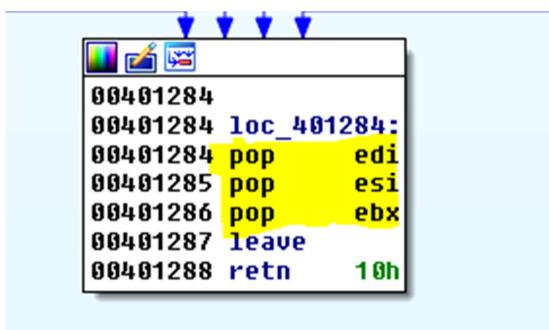
```
00401068 mov    ds:WndClass.hCursor, eax
0040106D mov    ds:WndClass.hbrBackground, 5
00401077 mov    ds:WndClass.lpszMenuName, offset aMenu ; "MENU"
00401081 mov    ds:WndClass.lpszClassName, offset ClassName ; "No need to disasm the code!"
0040108B push   offset WndClass ; lpWndClass
00401090 call   RegisterClassA
```

We realize that it saves the address 0x402110 because it adds OFFSET before it.

Normally, when adding arguments to functions, we'll always see the **PUSH offset xxxx** because the idea is adding the address where the string is if it did not have the word offset, we would push the content of the address 0x402110 that is the bytes 55 4E 45 4D of the same string and the api's don't work like that. They always receive the pointer or address to the start or where the string starts.

In the above instruction, we see the prefix DS:TAG that indicates it is going to save it in a memory address of the data section (DS=DATA). When we see structures, we will study that case. Now, the important thing is that it saves the address of the string start in the DATA section.

POP



```
00401284
00401284 loc_401284:
00401284 pop    edi
00401285 pop    esi
00401286 pop    ebx
00401287 leave
00401288 retn   10h
```

It is the operation that reads the top of the stack and moves it to the destination register, in this case, POP EDI will read the first value or top of the stack and it will copy in EDI and then it will point ESP to the value that was below and it will become the top of the stack.

If we search as text the word POP, we see that there are no many variants in spite of that there is the possibility of POPPING to a memory address instead of a register, this option is not that used.

Segment	Address	Function	Instruction
CODE	CODE:004011E6	WndProc	pop ebx
CODE	CODE:004011E7	WndProc	pop edi
CODE	CODE:004011E8	WndProc	pop esi
CODE	CODE:00401240	WndProc	pop eax
CODE	CODE:00401284	sub_401253	pop edi
CODE	CODE:00401285	sub_401253	pop esi
CODE	CODE:00401286	sub_401253	pop ebx
CODE	CODE:00401325	DialogFunc	pop edi
CODE	CODE:00401326	DialogFunc	pop esi
CODE	CODE:00401327	DialogFunc	pop ebx
CODE	CODE:0040139C	sub_40137E	pop esi
CODE	CODE:004013AC	sub_40137E	pop esi
CODE	.idata:00403280		; BOOL __stdcall GetSaveFileNameA(LPOPENFILENAMEA)

We will continue with more instructions in part 5 to be able to study the LOADER function.

Ricardo Narvaja

Translated by: **@lvinsonCLS**