

REVERSING WITH IDA PRO FROM SCRATCH

PART 14

This course will be mixed and it will have different reversing topics: static reversing, debugging, unpacking and exploiting.

We will unpack the included CRACKME.exe packed with the last version of UPX. It doesn't mean we will have a lot of parts with just unpacking in a row. There will be a mix of different topics to avoid boredom. So that, we will have unpacking lessons mixed with other topics once in a while.

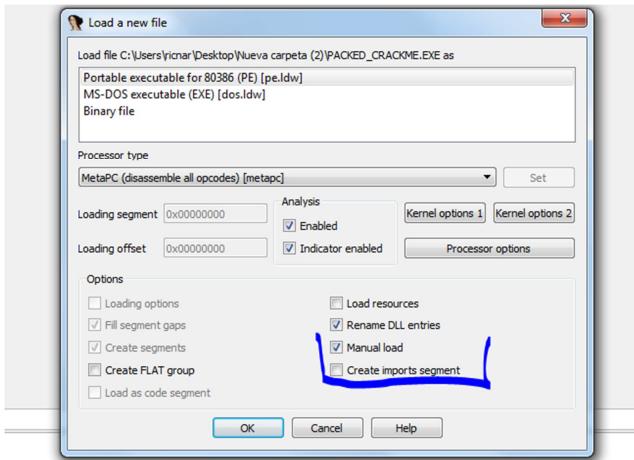
Packed files

It means that a file hides a program executable code saving it with a type of compression or encryption to avoid being reversed easily. It also adds a **STUB** or section where it runs. It takes the encrypted code, in runtime, and it decrypts it in memory saving it in any other section or the same to jump there and execute it.

There are many packer variants and most of them are protectors that break the IAT or import table and the HEADER. They add anti-debugging code to avoid unpacking and rebuilding of the original file.

The easiest case is the UPX packer that doesn't have antidebuggers or any dirty trick, but it helps us start from the basics.

The PACKED_CRACKME.EXE is attached to this tutorial.



Let's check **MANUAL LOAD** and uncheck **CREATE IMPORTS SEGMENT** because we need all sections loaded. IDA recommends to uncheck that when working with packed files.

```

00409BE0 ; The code at 400000..401000 is hidden from normal disassembly
00409BE0 ; and was loaded because the user ordered to load it explicitly
00409BE0 ;
00409BE0 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00409BE0 ;
00409BE0 ;
00409BE0 ;
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov    esi, offset dword_409000
00409BE6 lea    edi, [esi-8000h]
00409BEC push   edi
00409BED jmp    short loc_409BFA

```



```

00409BFA
00409BFA loc_409BFA:
00409BFA mov    ebx, [esi]
00409BFC sub    esi, 0FFFFFFFCh
00409BFF adc    ebx, ebx

```



```

00409C01
00409C01 loc_409C01:
00409C01 jb    short loc_409BF0

```



```

00409C03 mov    eax, 1

```

That's the start or **ENTRY POINT** of the **PACKED_CRACKME.exe**. It is in the address **0x409BE0** while it was in **0x401000** in the original file.

```

00401000 ; SECTION SIZE IN FILE : 00000000 ( 1530 )
00401000 ; Offset to raw data for section: 00000600
00401000 ; Flags 60000020: Text Executable Readable
00401000 ; Alignment : default
00401000
00401000 .686p
00401000 .mnx
00401000 .model flat
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 CODE segment para public 'CODE' use32
00401000 assume cs:CODE
00401000 ;org 401000h
00401000 assume es:nothing, ss:nothing, ds:CODE, fs:nothing
00401000
00401000
00401000 public start
00401000 start proc near
00401000 push 0 ; lpModuleHandle
00401002 call GetModuleHandleA
00401007 mov ds: hWndClass, eax
0040100C push 0 ; lpWindowName
0040100E push offset ClassName ; "No need to disasm the code!"
00401013 call FindWindowA
00401018 or eax, eax
0040101A jz short loc_40101D

```

```

0040101D loc_40101D:
0040101D mov ds: hWndClass.style, 4003h
00401027 mov ds: hWndClass.lpFnWndProc, offset WndProc
00401031 mov ds: hWndClass.cbWndExtra, 0
0040103B mov ds: hWndClass.cbWndExtra, 0
00401045 mov eax, ds: hWnd
0040104A mov ds: hWndClass.hInstance, eax
0040104F push 64h ; lpIconName
00401051 push eax ; hInstance
00401052 call LoadIconA

```

Comparing their segments, we see that after the packed file header, it has a segment called **UPX0** whose size in memory is longer than the original one.

ORIGINAL

	Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
DE	HEADER	00400000	00401000	? ? ? .	L	page	0007	public	DATA	32	FFFFFF	FFFFFF	FFFF	FFFF	FFFF	FFFF	FFFF	
DE	CODE	00401000	00402000	R . X .	L	para	0001	public	CODE	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	DATA	00403000	00404000	R W . .	L	para	0002	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	.idata	00404000	00405000	R . . .	L	para	0003	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	.edata	00405000	00406000	R . . .	L	para	0004	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	.reloc	00406000	00407000	R . . .	L	para	0005	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	.rsrc	00407000	00408000	R . . .	L	para	0006	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	
DE	OVERLAY	00408000	00408200	R W . .	L	byte	0000	private	DATA	32	FFFFFF	FFFFFF	0001	FFFFFF	FFFFFF	FFFF	FFFF	

PACKED

	Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
Segment	Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
UPX0	HEADER	00400000	00401000	? ? ? .	L	page	0004	public	DATA	32	FFFFFF	FFFFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
UPX0	UPX0	00401000	00409000	R W X .	L	para	0001	public	CODE	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	FFFF
UPX0	UPX1	00409000	0040A000	R W X .	L	para	0002	public	CODE	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	FFFF
UPX0	.rsrc	0040A000	0040B000	R W . .	L	para	0003	public	DATA	32	0000	0000	0001	FFFFFF	FFFFFF	FFFF	FFFF	FFFF
UPX0	OVERLAY	0040B000	0040B200	R W . .	L	byte	0000	private	DATA	32	FFFFFF	FFFFFF	0001	FFFFFF	FFFFFF	FFFF	FFFF	FFFF

The packed file UPX0 section ends in 0x409000 while all the original file sections are in memory from 0x401000 to hasta 0x408200.

We are talking about virtual memory. When a program runs, it can have 1k in the HDD and reserve 20k or whatever in memory.

We can see that in IDA, for example, in the start address: 0x401000 of the original file section code.

```
00401000 ; File Name : C:\Users\ricnar\Desktop\CRACKME.EXE
00401000 ; Format : Portable executable for 80386 (PE)
00401000 ; Imagebase : 400000
00401000 ; Timestamp : 0AD92429 (Wed Oct 08 12:18:49 1975)
00401000 ; Section 1. (virtual address 00001000)
00401000 ; Virtual size : 00001000 ( 4096.)
00401000 ; Section size in file : 00000600 ( 1536.)
00401000 ; offset to raw data for section
00401000 ; Flags 60000020: Text Executable Readable
00401000 ; Alignment : default
00401000 ;
00401000 ; The code at 400000..401000 is hidden from normal disassembly
00401000 ; and was loaded because the user ordered to load it explicitly
00401000 ;
00401000 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00401000 ;
00401000 ;
```

That section (SECTION SIZE IN FILE) takes 0x600 bytes, while (VIRTUAL SIZE) takes 0x1000 in memory.

If we go, in the packed file, to 0x401000 that is the UPX0 section start...

```
HEADER:00400FFF HEADER      ends
HEADER:00400FFF
UPX0:00401000 ; File Name : C:\Users\ricnar\Desktop\Nueva carpeta (2)\PACKED_CRACKME.EXE
UPX0:00401000 ; Format : Portable executable for 80386 (PE)
UPX0:00401000 ; Imagebase : 400000
UPX0:00401000 ; Timestamp : 0AD92429 (Wed Oct 08 12:18:49 1975)
UPX0:00401000 ; Section 1. (virtual address 00001000)
UPX0:00401000 ; Virtual size : 00008000 ( 32768.)
UPX0:00401000 ; Section size in file : 00000000 ( 0.)
UPX0:00401000 ; offset to raw data for section
UPX0:00401000 ; Flags E0000080: Bss Executable Readable Writable
UPX0:00401000 ; Alignment : default
UPX0:00401000 ; =====
UPX0:00401000 ; Segment type: Pure code
UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0      segment para public 'CODE' use32
UPX0:00401000      assume cs:UPX0
UPX0:00401000      org 401000h
UPX0:00401000      dd 0C00h dup(?)      ; CODE XREF: start+183↓j
UPX0:00404000 UPX0      dd 1400h dup(?)      ; DATA XREF: HEADER:00400204↑o
UPX0:00404000      ends
UPX0:00404000
```

We see that it is section with 0-byte length in disc, but it takes 0x8000 in memory. This reserves empty space to create, here, the original program code and then, it jumps to execute it. It has enough space to do that.

```
UPX0:00401000      dd 0C00h dup(?)      ; CODE XREF: start+183↓j
UPX0:00401000      dd 1400h dup(?)      ; DATA XREF: HEADER:00400204↑o
UPX0:00404000 UPX0      ends
UPX0:00404000
```

The address **0x401000** has the **dword_** tag before meaning that its content is a DWORD.

The (?) sign means that it is just reversed and it doesn't have any content and the dup o multiply means that dword multiplies by **0xC00**, that is to say, **0x3000** reserved bytes.

```
Output window
Python>hex(0xc00*4)
0x3000
Python
```

Then, in **0x404000**, there are **0x1400** dwords plus (?) It means just reserved.

```
Python>hex(0xc00*4+ 0x1400*4)
0x8000
Python
```

There are **0x8000** bytes reserved in memory to place the program there.

```
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:OVERLAY, ss:OVERLAY, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000 dd 0C00h dup(?) ; CODE XREF: start+1831j
UPX0:00401000 ; DATA XREF: HEADER:00400204To
UPX0:00404000 dd 1400h dup(?) ; DATA XREF: HEADER:00400178To
UPX0:00404000 UPX0 ends
UPX0:00404000
UPX1:00409000 ; Section 2. (virtual address 00009000)
UPX1:00409000 ; Virtual size : 00001000 ( 4096.)
UPX1:00409000 ; Section size in File : 0000E000 ( 3584.)
UPX1:00409000 ; Offset to raw data for section: 00000400
UPX1:00409000 ; Flags E0000040: Data Executable Readable Writable
UPX1:00409000
UPX1:00409000 xrefs to dword_401000
UPX1:00409000
UPX1:00409000 Direction Typ Address
UPX1:00409000 Do... j start+183
UPX1:00409000 Up o HEADER:00400204
UPX1:00409000
UPX1:00409000
UPX1:00409000 Line1 of 2
UPX1:00409000 dd 41112868h, 409H67B0h, 7009000Ch, 0BF7EFEEFn, 74H33DHn
UPX1:00409000 dd 3E506442h, 0C78A324h, 7F0068h, 73FDF348h, 0A3061054h
```

In **0x401000**, there is a reference to executable code. Later, we will see what it does.

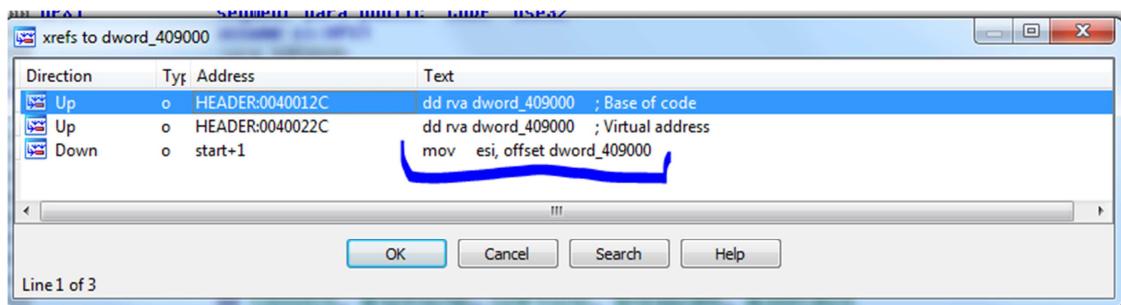
```

UPX1:00409000 ; Section 2. (virtual address 00009000)
UPX1:00409000 ; Virtual size           : 00001000 ( 4096.)
UPX1:00409000 ; Section size in file   : 00000E00 ( 3584.)
UPX1:00409000 ; Offset to raw data for sections: 00000000
UPX1:00409000 ; Flags E0000000: Data Executable Readable Writable
UPX1:00409000 ; Alignment        : default
UPX1:00409000 ; =====
UPX1:00409000 ; Segment type: Pure code
UPX1:00409000 ; Segment permissions: Read/Write/Execute
UPX1:00409000 UPX1      segment para public 'CODE' use32
UPX1:00409000 assume cs:UPX1
UPX1:00409000 ;org 409000h
UPX1:00409000 assume es:OVERLAY, ss:OVERLAY, ds:UPX0, fs:nothing, gs:nothing
UPX1:00409000 dword_409000 dd 0FFF6EE77h, 0E8006Ah, 0A3020500h, 4020CAh, 6F4680Bh
UPX1:00409000 ; DATA XREF: HEADER:0040012Ct0
UPX1:00409000 ; HEADER:0040022Ct0 ...
UPX1:00409000 dd 0B8A0410h, 0FEEEB6D9Bh, 0C30174C0h, 0F6405C7h, 9000203h
UPX1:00409000 dd 41112868h, 409A67B6h, 7009006Ch, 0BF7EEFEh, 74A33DA1h
UPX1:00409000 dd 3E506442h, 0C780324h, 7F0068h, 73DF348h, 0A3061054h
UPX1:00409000 dd 8038207Ch, 0BE673605h, 211084C0h, 7C778813h, 0B766EE64h
UPX1:00409000 dd 0FF338441h, 929035h, 7DB70080h, 0B04C5BEh, 0B4686Eh
UPX1:00409000 dd 0E704CF00h, 9F70DAF30h, 905F350Dh, 10004A3h, 60B70737h
UPX1:00409000 dd 66117D97h, 51177E0Ah, 0DD0B0875h, 48ECEC1Eh, 1B48556Ah
UPX1:00409000 dd 53D66D2h, 0F6C91674h, 5A0F9364h, 0D4EB0209h, 0EDDD05042h
UPX1:00409000 dd 0E05F77Fh, 575679C8h, 0C7D08353h, 815E7402h, 36020405h
UPX1:00409000 dd 0DDDCD365h, 120090B9h, 185D7405h, 1287401h, 0DDCD9A7Eh
UPX1:00409000 dd 7424274Ah, 6C01114Fh, 9BBB814EBh, 2B1F7339h, 90B73EBh
UPX1:00409000 dd 67149669h, 26CB60Fh, 969F0C10h, 3C0153EBh, 67F67FEEh
UPX1:00409000 dd 43EB4523h, 3A0A41EBh, 0C7145D8Bh, 51181843h, 6F3D9CECh
UPX1:00409000 dd 0A01C06h, 25240D20h, 0EDF7737Fh, 67107A14h, 65051574h
UPX1:00409000 dd 7466B574h, 5B80EB25h, 0F0F8605Fh, 0C2C95EF7h, 1300F710h
UPX1:00409000 dd 1F680A40h, 0FB02727Ah, 9C6C649Bh, 531BDDDEBh, 0D8FD1512h
UPX1:00409000 dd 0F8831787h, 8E68BE01h, 7A030E21h, 7F7E6850h, 0A83FF7Fh
UPX1:00409000 dd 4C483D4h, 74C33B58h, 0EB5E0C07h, 0EB49069Ah, 0AE9E193h
UPX1:00409000 dd 2B532A0Fh, 53410FCh, 0F777043h, 0F118335h, 271E8184h

```

The packed has this second section whose size in disc is 0xE00 and 0x1000 in memory. That is possibly the program saved with some simple encryption method to hide the original code.

If we see the references in the start address section: 0x409000...



There is a reference (Down) in an executable part. Let's click there.

```

00409BE0 ;
00409BE0 ; The code at 400000..401000 is hidden from normal disassembly
00409BE0 ; and was loaded because the user ordered to load it explicitly
00409BE0 ;
00409BE0 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
00409BE0 ;
00409BE0 ;
00409BE0 ;
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov     esi, offset dword_409000
00409BE6 lea     edi, [esi-8000h]
00409BEC push    edi
00409BED jmp    short loc_409BFA

```

↓

```

00409BFA
00409BFA loc_409BFA:
00409BFA mov     ebx, [esi]
00409BFC sub    esi, 0FFFFFFFCh
00409BFF adc    ebx, ebx

```

↓

In the STUB after the entry point it loads the address 0x409000 (remember the OFFSET before)

If we press the space bar, we see there...

```

UPX1:00409BE0 dd 002CB8495h, 00646309h, 0072380h, 58EDF Eh, 4C094C5h
UPX1:00409BE0 dd 39800001h, 0C100092h, 0E092A3FFh, 0001000h, 100201h
UPX1:00409BE0 dd 3B2610Ah, 3258EF8h, 0B40C20h, 09211E02h, 3016410h
UPX1:00409BE0 dd 0B1853E0h, 4E2136h, 0E5333856h, 1099E92FBh, 7646424h
UPX1:00409BE0 dd 364AC30h, 4067788Eh, 6055147Ah, 0DC9A42C0h, 9708591h
UPX1:00409BE0 dd 0A000000h, 00000000h, 00000000h, 00000000h, 00000000h
UPX1:00409BE0 dd 0C708BCDh, 0C627616h, 61972E2Ch, 0E2208C27h, 0E08303h
UPX1:00409BE0 dd 008325365h, 164FA080h, SECEA140h, 0DE3F2EB0h, 7B365027h
UPX1:00409BE0 dd 50189299h, 0AC637273h, 97656810h, 89271A30h, 3290h
UPX1:00409BE0 dd 79807070h, 24Eh, 0FF0h, 0
UPX1:00409BE0 ; The code at 400000..401000 is hidden from normal disassembly
UPX1:00409BE0 ; and was loaded because the user ordered to load it explicitly
UPX1:00409BE0 ; <<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO HARMFUL THINGS >>>
UPX1:00409BE0
UPX1:00409BE0 ; ===== SUB R O U T I N E =====
UPX1:00409BE0 ; DATA XREF: HEADER:00h00128To
UPX1:00409BE0 public start
UPX1:00409BE0 start proc near ; DATA XREF: HEADER:00h00128To
UPX1:00409BE0
UPX1:00409BE0 var_AC = byte ptr -0ACh
UPX1:00409BE0
UPX1:00409BE0 pusha
UPX1:00409BE0 mov     esi, offset dword_409000
UPX1:00409BE0 lea     edi, [esi-8000h]
UPX1:00409BE0 push    edi
UPX1:00409BE0 jmp    short loc_409BFA
UPX1:00409BED align 10h
UPX1:00409BE0
UPX1:00409BF0 loc_409BFA: ; CODE XREF: start:loc_409C01h

```

That the STUB code is in the same UPX1 section below the original file packed code. In the UPX1 section, we have the original program encrypted saved bytes and the STUB code from 0x409BE0.

We don't have to be a genius to know that it will read bytes from 0x409000, it will apply some operations and save them in 0x401000. EDI = ESI-0x8000.

```

00409BE0
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov     esi, offset dword_409000
00409BE6 lea     edi, [esi-8000h]
00409BEC push    edi
00409BED jmp    short loc_409BFA

```

Python>hex(0x409000-0x8000)
0x401000

It will use the ESI content as SOURCE where it will read the data; it will apply some operations and save them in the EDI content to create the program code.

If we click on 0x401000, (the reference we talked about before)

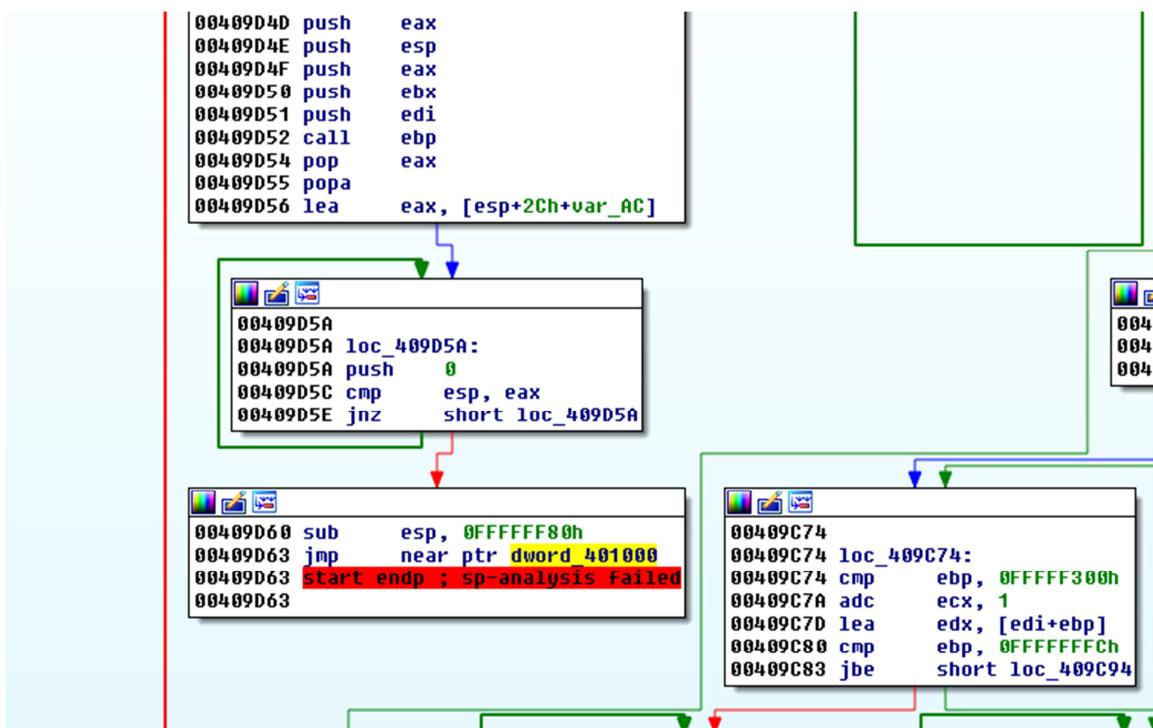
```

UPX0:00401000      assume cs:[...]
UPX0:00401000      ;org 401000h
UPX0:00401000      assume es:OVERLAY, ss:OVERLAY, ds:UPX0, Fs:nothing, gs:nothing
UPX0:00401000      dd 0C00h dup(?)      ; CODE XREF: start+1834j
UPX0:00401000      ; DATA XREF: HEADER:00400204To
UPX0:00404000      dd 1400h dup(?)      ; DATA XREF: HEADER:00400178To
UPX0:00404000      UPX0
UPX0:00404000      ends
UPX1:00409000      ; Section 2. (virtual address 00009000)
UPX1:00409000      ; Virtual size           : 00001000 ( 4096.)
UPX1:00409000      ; Section size in file   : 00000E00 ( 3584.)
UPX1:00409000      ; Offset to raw data for section: 00000400
UPX1:00409000      ; Flags E0000040: Data Executable Readable Writable
UPX1:00409000
UPX1:00409000      xrefs to dword_401000
UPX1:00409000
UPX1:00409000      Direction Typ Address          Text
UPX1:00409000      Do... j  start+183      jmp    near ptr dword_401000
UPX1:00409000      Up   o  HEADER:00400204  dd rva dword_401000 ; Virtual address
UPX1:00409000
UPX1:00409000
UPX1:00409000      Line1 of 2
UPX1:00409000      dd 41112808h, 409H07B0h, 70890000h, 0BF7EFEEFDh, 74H330H1h
UPX1:00409000      dd 3E506442h, 0C78A324h, 7F0068h, 73FDF348h, 0A3061054h

```

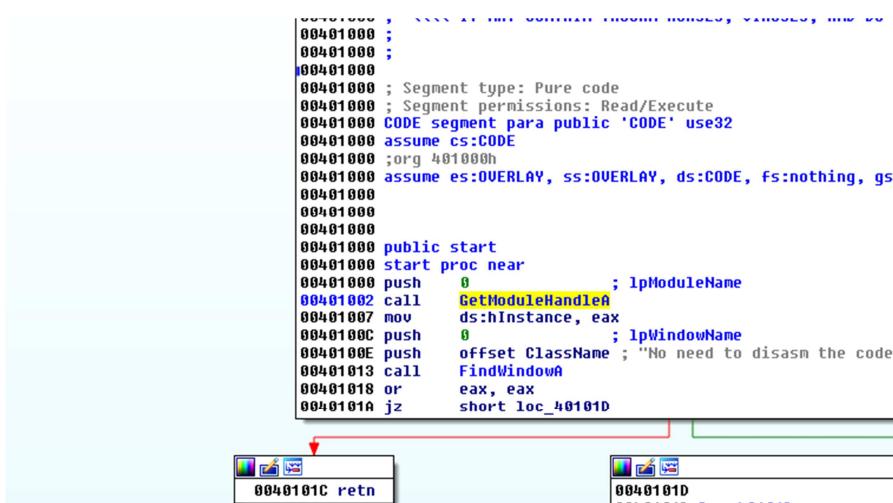
The screenshot shows a debugger interface with assembly code at the top and memory dump at the bottom. A call instruction at address 0x409000 is highlighted with a red bracket, pointing to the address 0x401000. The assembly code shows a jump to 0x401000. A blue bracket highlights the destination address 0x401000 in the assembly code.

We see a JMP to 0x401000.



JMP NEAR is a direct jump to the address that is next to it. It will jump to 0x401000, obviously, here, after executing all the STUB and creating the original code, it will jump to the OEP in 0x401000 that would be the ORIGINAL ENTRY POINT in contrast to the STUB ENTRY POINT that is in 0x00409BE0.

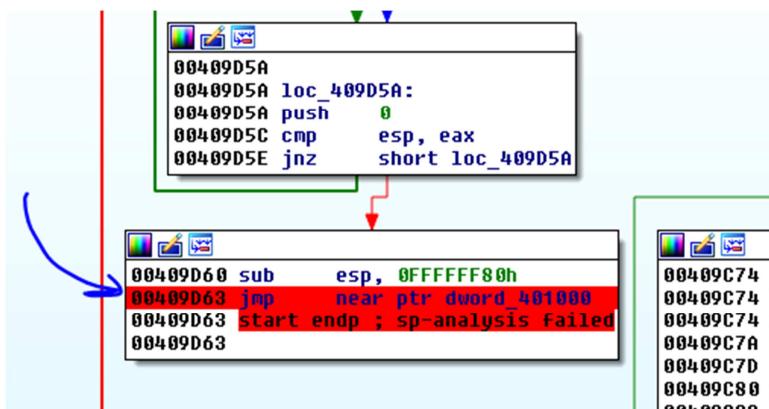
We will call OEP or ORIGINAL ENTRY POINT to the original program ENTRY POINT. As it is a packed program, we don't know where it is and as we have the original one, we can know that it was in 0x401000.



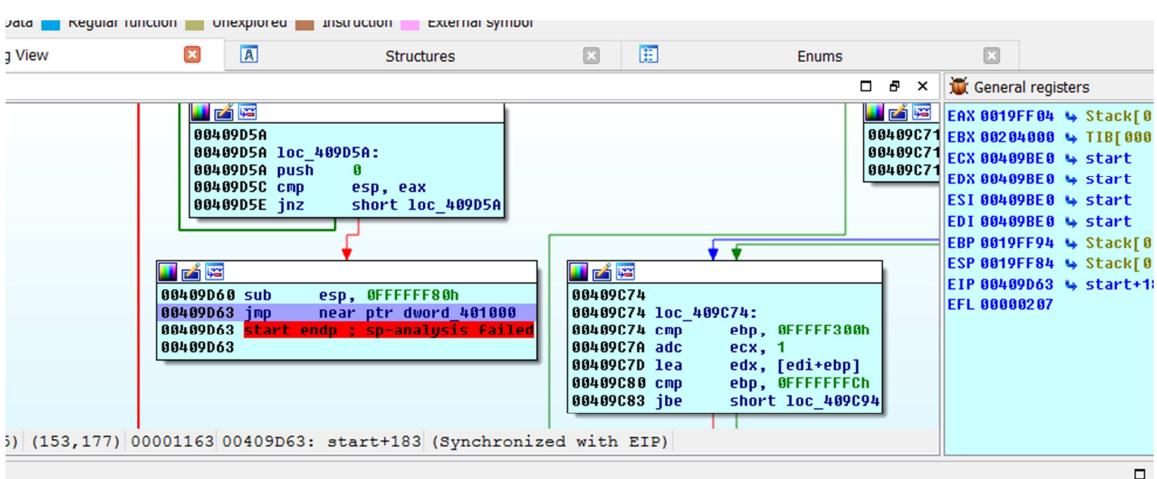
Most of the times, we have a packed program and we don't know the OEP because we don't have the original one. So, let's learn how to find it.

When the STUB does all its tricks and ends creating the original code, it will jump to execute the program. Almost always, the first code line that it executes in the created section will be the OEP.

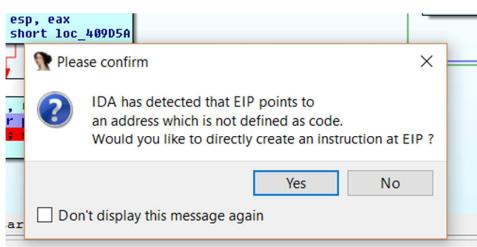
We could set a Breakpoint in that JMP to OEP to see if the original program is already created there. Let's try it.



Select LOCAL WIN32 DEBUGGER and press START DEBUGGER.



It stopped in the jump to the OEP. Trace it with F8.



Press YES to interpret the first UPX0 section as CODE that was defined as DATA.

```

UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000
UPX0:00401000 loc_401000: ; CODE XREF: start+183$J
UPX0:00401000
IP UPX0:00401000 push 0
UPX0:00401002 call sub_401506
UPX0:00401007 mov dword_4020CA, eax
UPX0:0040100C push 0
UPX0:0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!"
UPX0:00401013 call sub_4014BE
UPX0:00401018 or eax, eax
UPX0:0040101A jz short loc_40101D
UPX0:0040101C ret
UPX0:0040101D ;
UPX0:00401000 UNKNOWN 00401000: UPX0:loc_401000 (Synchronized with EIP)

```

It unpacked the code and jumped to execute it. The code is very similar to the 0x401000 of the original, although, we can't change IDA to graphical mode because it is not defined as function (loc_401000), but we will do it automatically.

There is a almost hidden menu on the upper left corner. Right click- REANALYZE PROGRAM.

```

UPX0:0040100E push offset aNoNeedToDisasm ; "No need to disasm"
UPX0:00401013 call sub_4014BE
UPX0:00401018 or eax, eax
UPX0:0040101A jz short loc_40101D
UPX0:0040101C ret
UPX0:0040101D ;
UPX0:00401000 UNKNOWN 00401000: UPX0:loc_401000 (Synchronized with EIP)

```

Hex View 1

```

00409D23 95 00 00 8B AE 68 95 00 00 8D BE 00 F0 FF FF BB 0..Í«`ò...+.=+.
00409D33 00 10 00 00 50 54 6A 04 00 53 57 FF D5 8D 87 1F 02 ...PTj.SW+*.ç..
00409D43 00 00 80 20 7F 80 00 28 7F 58 59 54 50 53 57 F7 ..ç-.ç(.XPTPSW-
00409D53 00 58 61 8D 44 24 80 6A 00 39 C4 75 FA 83 EC 80 +Xa.D$çj.9-u-a8ç
00001153 00409D53: start+173

```

Analysis indicator
Reanalyze program
Processor analysis options...

When clicking on the address loc_401000, it changed to sub_401000 which indicates that it is a function now. Then, we can change it to graphical mode with the space bar.

```

UPX0:00401000 ; ====== S U B R O U T I N E ======
UPX0:00401000
UPX0:00401000
UPX0:00401000 sub_401000 proc near
UPX0:00401000
UPX0:00401000 push 0
UPX0:00401002 call sub_401506
UPX0:00401007 mov dword_4020CA, eax
UPX0:0040100C push 0
UPX0:0040100E push offset aNoNeedToDisasm ; "No need to
UPX0:00401013 call sub_4014BE
UPX0:00401018 or eax, eax
UPX0:0040101A jz short loc_40101D
UPX0:0040101C retn

UNKNOWN 00401000: sub_401000 (Synchronized with EIP)

```

Now, it looks better.

```

00401000 ;DATA SEGMENT PARA PUBLIC CODE USE32
00401000 assume cs:UPX0
00401000 ;org 401000h
00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push 0
00401002 call sub_401506
00401007 mov dword_4020CA, eax
0040100C push 0
0040100E push offset aNoNeedToDisasm ; "No need to disasm the code!"
00401013 call sub_4014BE
00401018 or eax, eax
0040101A jz short loc_40101D

-168,208) (895,122) UNKNOWN 00401000: sub_401000 (Synchronized with EIP)

```

A difference we see is that the original showed CALL GetModuleHandleA in 0x401002 while this one shows CALL sub_401056. Let's see what there is inside that CALL.

```

00401506
00401506
00401506 ; Attributes: thunk
00401506
00401506 sub_401506 proc near
00401506 jmp off_403238
00401506 sub_401506 endp
00401506

```

Let's see the difference with the original. If we enter the CALL GetModuleHandleA in the original.

```

00401506
00401506
00401506 ; Attributes: thunk
00401506
00401506 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
00401506 GetModuleHandleA proc near
00401506
00401506 lpModuleName      = dword ptr 4
00401506
00401506     jmp    ds:_imp_GetModuleHandleA
00401506 GetModuleHandleA endp
00401506

```

There is also an indirect jump, but here, it detects that it jumps to the API, the other one doesn't, but where does the packed go?

```

• UPX0:00403228 off_403228 dd offset kernel32_GlobalAlloc ; DATA XREF: UPX0:004014EE↑r
• UPX0:0040322C off_40322C dd offset kernel32_lstrlen ; DATA XREF: UPX0:004014F4↑r
• UPX0:00403230 off_403230 dd offset kernel32_CloseHandle ; DATA XREF: UPX0:004014FA↑r
• UPX0:00403234 off_403234 dd offset kernel32_WriteFile ; DATA XREF: UPX0:00401500↑r
• UPX0:00403238 off_403238 dd offset kernel32_GetModuleHandleA ; DATA XREF: sub_401506↑r
• UPX0:0040323C off_40323C dd offset kernel32_ReadFile ; DATA XREF: UPX0:0040150C↑r
• UPX0:00403240 off_403240 dd offset kernel32_ExitProcess ; DATA XREF: sub_401512↑r
• UPX0:00403244 align 8
• UPX0:00403248 off_403248 dd offset comctl32_InitCommonControls ; DATA XREF: UPX0:00401518↑r
• UPX0:0040324C off_40324C dd offset comctl32_CreateToolbarEx ; DATA XREF: UPX0:0040151E↑r

```

The 0x403028 content is an offset (off_). In the GetModuleHandleA API address and in the original, we see the same address in the **idata** section and it has the address of the same API.

```

• .idata:00403234 ; BOOL __stdcall WriteFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNt
• .idata:00403234           extrn WriteFile:dword ; DATA XREF: CODE:00401500↑r
• .idata:00403238 ; HMODULE __stdcall GetModuleHandleA(LPCSTR lpModuleName)
• .idata:00403238           extrn _imp_GetModuleHandleA:dword
• .idata:00403238           ; DATA XREF: GetModuleHandleA
• .idata:0040323C ; BOOL __stdcall ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNt
• .idata:0040323C           extrn ReadFile:dword ; DATA XREF: CODE:0040150C↑r
• .idata:0040323C           ; DATA XREF: ReadFile

```

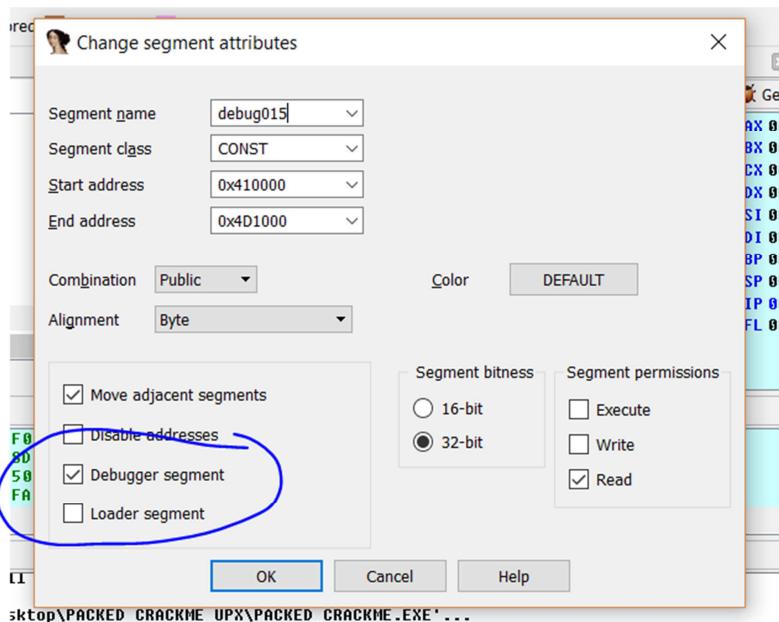
In spite of that they end up jumping to the same place, there is a very important difference we will see later.

I have the unpacked file code. Although, it is not functional yet and if I just have to analyze the code created in the first section statically, I do the following.

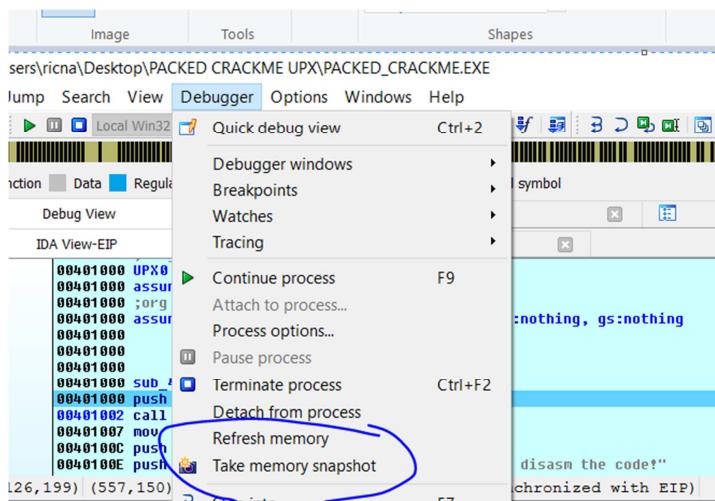
First in SEGMENTS.

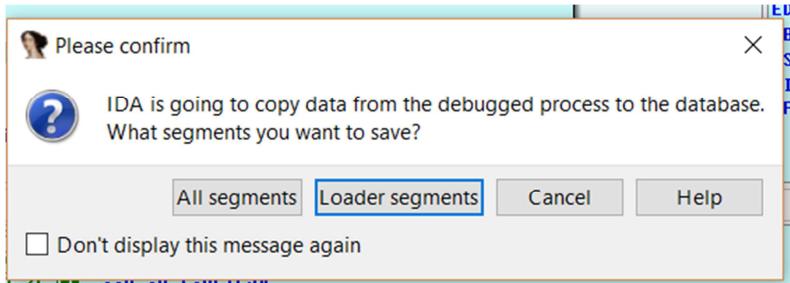
• TIB[000027B8]	00203000	00211000	R	W	.	D	.	byte	0000
• HEADER	00400000	00401000	?	?	?	.	L	page	0004
• UPX0	00401000	00409000	R	W	X	.	L	para	0001
• UPX1	00409000	0040A000	R	W	X	.	L	para	0002
• .rsrc	0040A000	0040B000	R	W	.	.	L	para	0003
• OVERLAY	0040B000	0040B200	R	W	.	.	L	byte	0000
• debug015	00410000	004D1000	R	.	.	D	.	byte	0000

I verify that all the packers sections have the letter **L** that means they will be loaded by the LOADER. I can add some DLL or segment I want to be in the static analysis. Right click- EDIT SEGMENT on the line we want to add to the LOADER.



We check the segment we want to be added in the LOADER SEGMENT. In this case, we will just keep the packer segments, but it's good to know we can add others.





Then, the TAKE MEMORY SNAPSHOT option will save the segments we have marked as LOADER with the code they have. Don't confuse it with the other FILE-TAKE DATABASE SNAPSHOT option that we studied before.

The debugger stopped and of course, it is in the LOADER. I go to 0x401000 and I see that now we see the code we copied when we were in the OEP and it is available to do static reversing as the other segments with the letter **L**.

If we run the debugger again, it would disappear because it would overwrite it with the bytes that would be there when the section initializes in the debugger. So, if we need the database with the static analysis, we must copy it to other folder and open it with other IDA to work comfortably.

If I run the debugger again and stop at its EntryPoint before executing the STUB, I see that the 0x401000 zone is empty again.

```

00409BE0 public start
00409BE0 start proc near
00409BE0 var_AC= byte ptr -0BACH
00409BE0 pusha
00409BE1 mov     esi, offset dword_409000
00409BE6 lea     edi, [esi-800h]
00409BEC push    edi
00409BED jmp    short loc_409BFA

5,149) (913,128) 00000FE0 00409BE0: start (Synchronized with EIP)

0C 20 4D 33 30 08 0E 65 53 32 D8 80 40 4F 16 .-M30..e$2+G00.
41 CE 5E 8D 2E 3F DE 27 50 30 78 99 92 18 50 @A+^.?;`P@{0E.P
72 63 AC 18 68 65 97 43 1A 27 89 90 32 00 00 srck,heuC.'ë.2..
7D 0D 79 EA 24 00 00 00 FF 00 00 00 00 00 p}.yo$...-.....
D9BB0: UPX1:00409BB0

```

```

UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000 unk_401000 db 0 ; CODE XREF: start+183↓j
UPX0:00401000 ; DATA XREF: HEADER:00400204↑o
UPX0:00401001 db 0
UPX0:00401002 db 0
UPX0:00401003 db 0
UPX0:00401004 db 0
UPX0:00401005 db 0
UPX0:00401006 db 0

```

UNKNOWN 00401000: UPX0:unk_401000 (Synchronized with EIP)

Day View 1

What we saved in the database is lost because, in the debugger, the LOADER info was overwritten with the bytes that initialized the UPX0 section. So, if we need it for static reversing, as I said before, after selecting TAKE MEMORY SNAPSHOT, we have to copy it to another folder before running the debugger.

As I am annoying, I will find another way of getting the OEP. That's finding the first instruction executed in the first section. This method can work sometimes.

I run the packed file again in the debugger stopping at its EntryPoint.

```

00409BE0
00409BE0
00409BE0 public start
00409BE0 start proc near
00409BE0
00409BE0 var_AC= byte ptr -0ACh
00409BE0
00409BE0 pusha
00409BE1 mov    esi, offset dword_409000
00409BE6 lea    edi, [esi-8000h]
00409BEC push   edi
00409BED jmp    short loc_409BFA

```

5,149 | (908,149) 00000FEO 00409BE0: start (Synchronized with EIP)

I go to the first section where it starts. This is 0x401000.

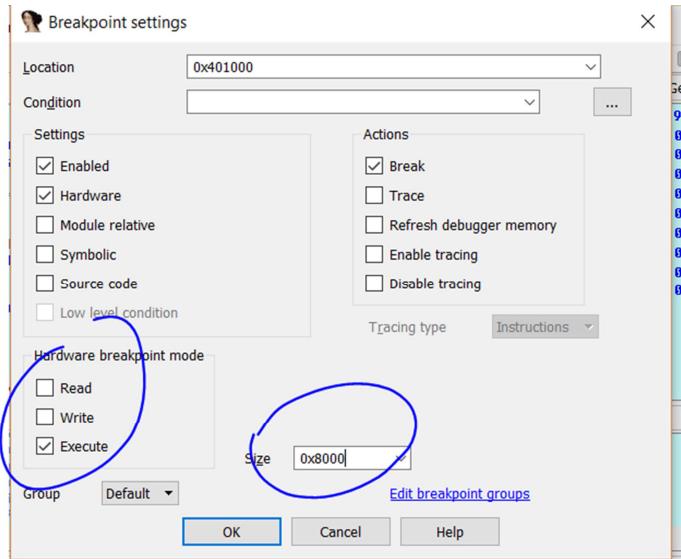
```

UPX0:00401000 ; Virtual size : 00000000 ( 32768.)
UPX0:00401000 ; Section size in file : 00000000 ( 0.)
UPX0:00401000 ; Offset to raw data for section: 00000400
UPX0:00401000 ; Flags E0000080: Bss Executable Readable Writable
UPX0:00401000 ; Alignment : default
UPX0:00401000 ; =====
UPX0:00401000
UPX0:00401000 ; Segment type: Pure code
UPX0:00401000 ; Segment permissions: Read/Write/Execute
UPX0:00401000 UPX0 segment para public 'CODE' use32
UPX0:00401000 assume cs:UPX0
UPX0:00401000 ;org 401000h
UPX0:00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
UPX0:00401000 unk_401000 db 0 ; CODE XREF: start+183↓j
UPX0:00401000 ; DATA XREF: HEADER:00400204↑o
UPX0:00401001 db 0

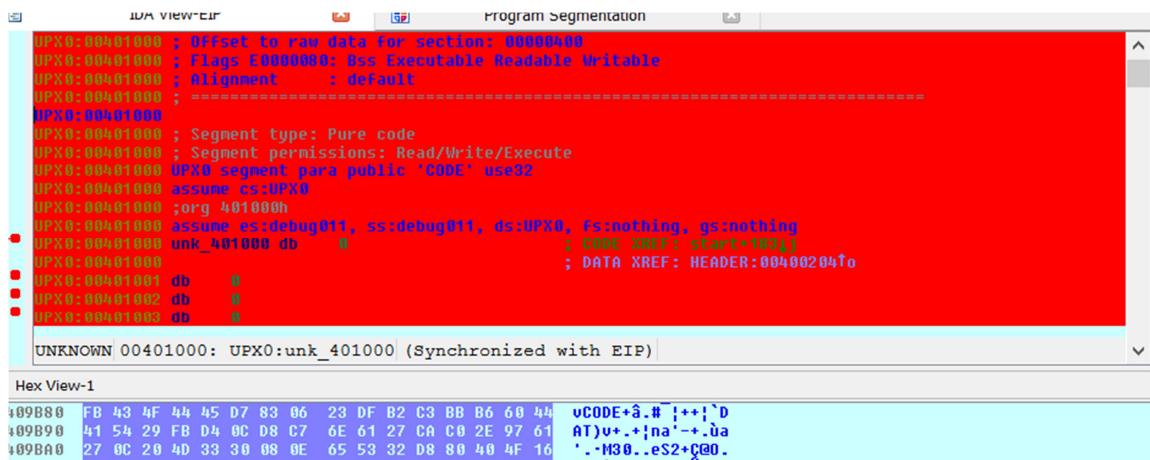
```

UNKNOWN 00401000: UPX0:unk_401000 (Synchronized with EIP)

I set a BreakPoint with F2. I set it to stop by execution or EXECUTE. It will stop just when executing not reading or writing. As it will stop when copying the code and creating it I don't want that. I just want it to jump when it is already created and it stops in the first instruction it executes and as I don't know what it is, I set a BreakPoint on Execute that covers all the section (0x8000 bytes).



All instructions are marked in red.



I disable the other BreakPoints in DEBUGGER-BREAKPOINT-BREAKPOINT LIST.

Type	Location	Pass count	Hardware	Condition
Abs	0x401000		X (32768 bytes)	
Abs	0x409BE0			
Abs	0x409D63			

Right click-DISABLE.

Type	Location	Pass count	Hardware
Abs	0x401000		X (32768 bytes)
Abs	0x409BE0		
Abs	0x409D63		

And now, RUN.

I see that the first instruction where it stopped, in the section created recently, in this case, 0x401000 is my found OEP.

The screenshot shows the IDA Pro interface with the assembly view active. The assembly code at address 0x401000 is highlighted in red:

```

00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push    0
00401002 call    sub_401506
00401007 mov     dword_4020CA, eax
0040100C push    0
0040100E push    offset aNoNeedToDisasm ; "No need to disasm the code!"
00401013 call    sub_4014BE
00401018 or     eax, eax
0040101A jz     short loc_401010

```

The registers pane on the right shows several registers starting with 0x00409BE0, indicating they are part of the current process context.

When this method works and we find the OEP that is 0x401000, I disable the BreakPoint.

After reanalyzing, it becomes a function again.

The screenshot shows the assembly view after reanalysis. The code is now correctly identified as a function:

```

00401000 assume es:debug011, ss:debug011, ds:UPX0, fs:nothing, gs:nothing
00401000
00401000
00401000
00401000 sub_401000 proc near
00401000 push    0
00401002 call    sub_401506
00401007 mov     dword_4020CA, eax
0040100C push    0
0040100E push    offset aNoNeedToDisasm ; "No need to disasm the code!"
00401013 call    sub_4014BE
00401018 or     eax, eax
0040101A jz     short loc_401010

```

Red arrows point from the original assembly code to the reanalyzed version, indicating the transformation.

Until now, we got the OEP and stopped there using two different methods. We could do a SNAPSHOT of the created code. The only thing we need is to DUMP and REBUILD the IAT to have an unpacked and functional executable.

Ricardo Narvaja

Translated by: @IvinsonCLS