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.

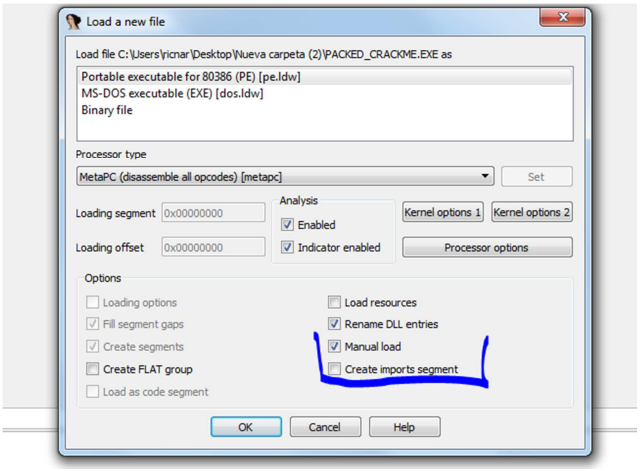
Packed files

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.

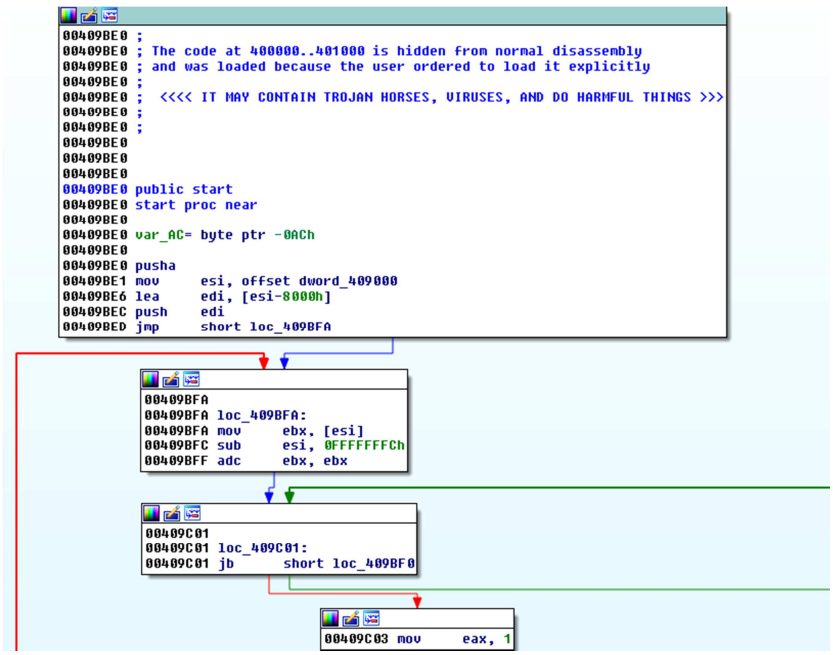
There are many packer variants and most of them are protectors that break the IAT or import table and the HEADER. They add anti-debugging code to avoid unpacking and rebuilding of the original file.

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

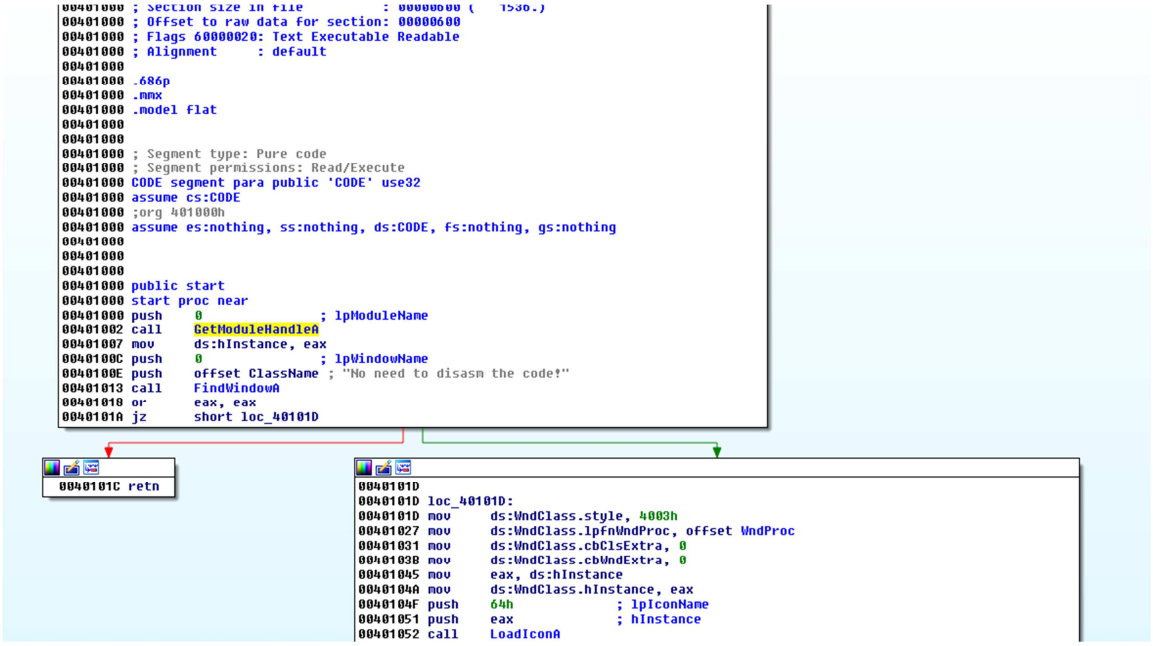
The PACKED_CRACKME.EXE is attached to this tutorial.



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.

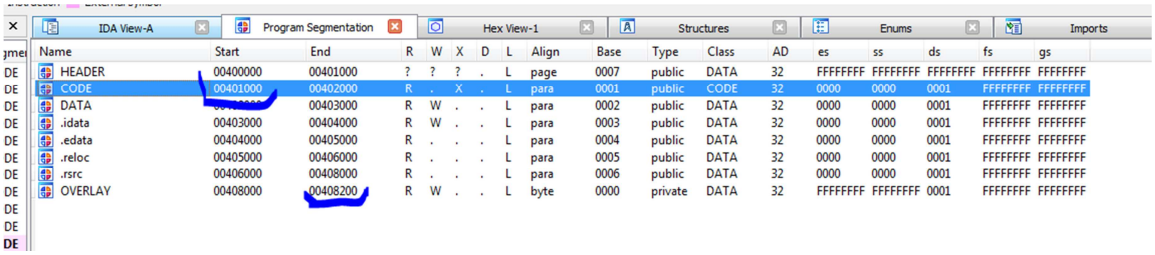


That's the start or ENTRY POINT of the PACKED_CRACKME.exe. It is in the address **0x409BE0** while it was in **0x401000** in the original file.

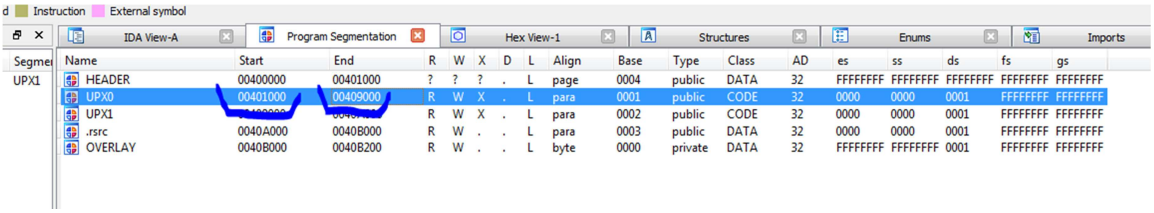


Comparing their segments, we see that after the packed file header, it has a segment called **UPX0** whose size in memory is longer than the original one.

ORIGINAL



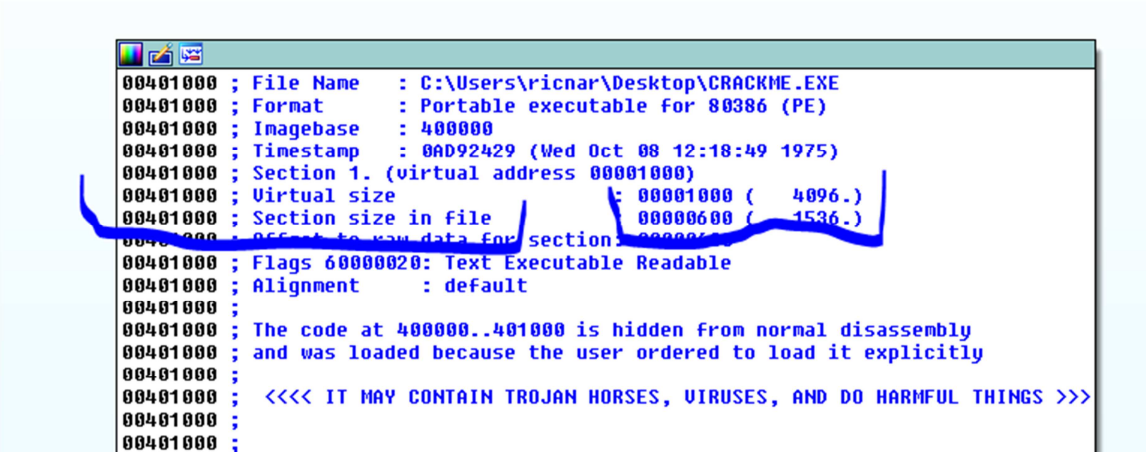
PACKED



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

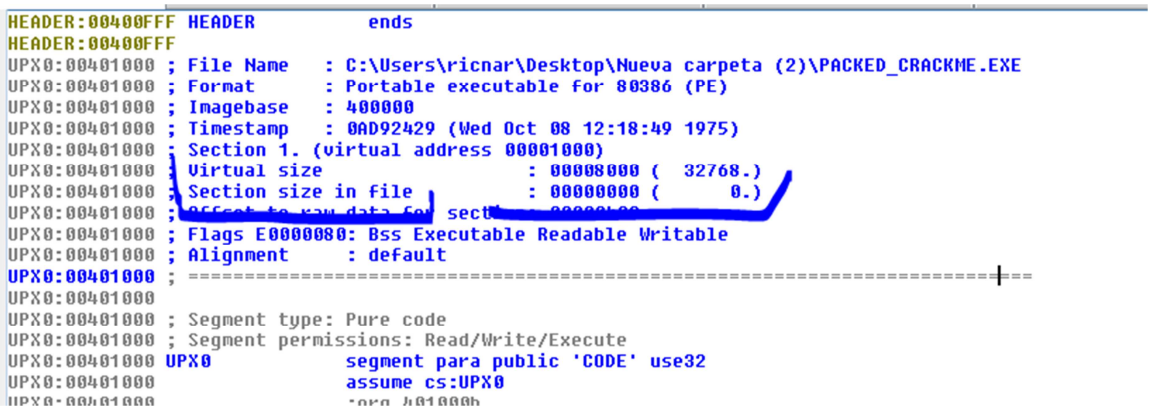
We are talking about virtual memory. When a program runs, it can have 1k in the HHDD and reserve 20k or whatever in memory.

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

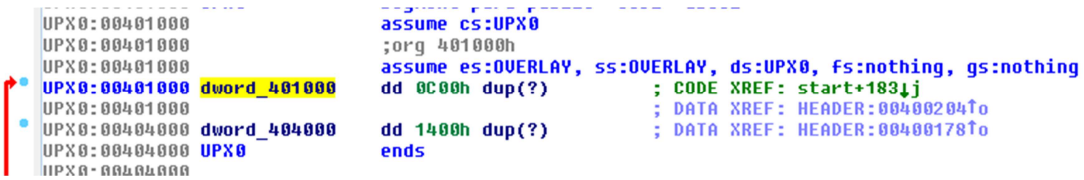


That section (SECTION SIZE IN FILE) takes 0x600 bytes, while (VIRTUAL SIZE) takes 0x1000 in memory.

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

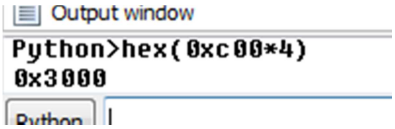


We see that it is section with 0-byte length in disc, but it takes 0x8000 in memory. This reserves empty space to create, here, the original program code and then, it jumps to execute it. It has enough space to do that.

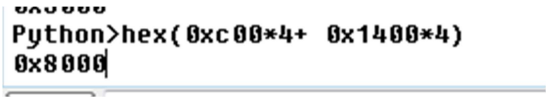


The address **0x401000** has the **dword**_ tag before meaning that its content is a DWORD.

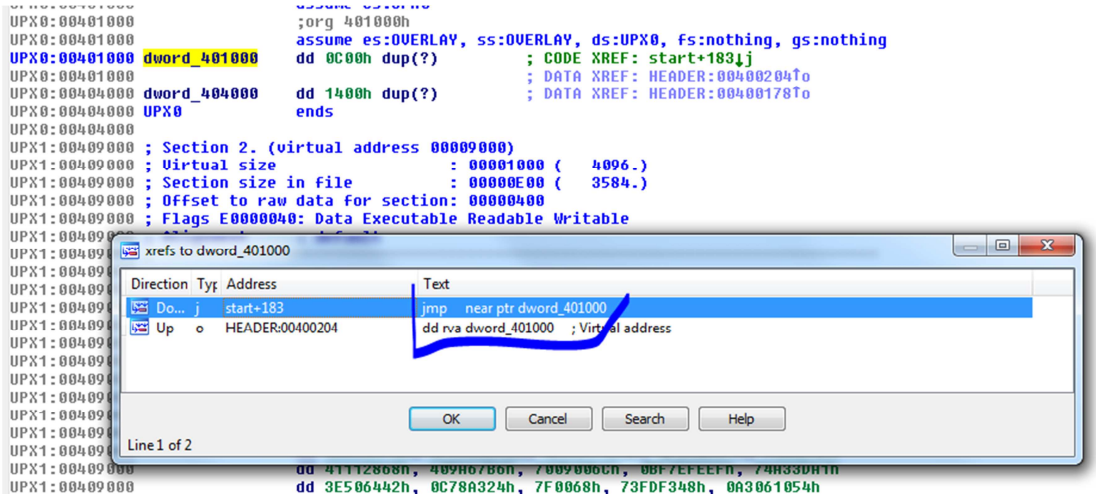
The (?) sign means that it is just reversed and it doesn't have any content and the dup o multiply means that dword multiplies by **0xC00**, that is to say, **0x3000** reserved bytes.



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



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

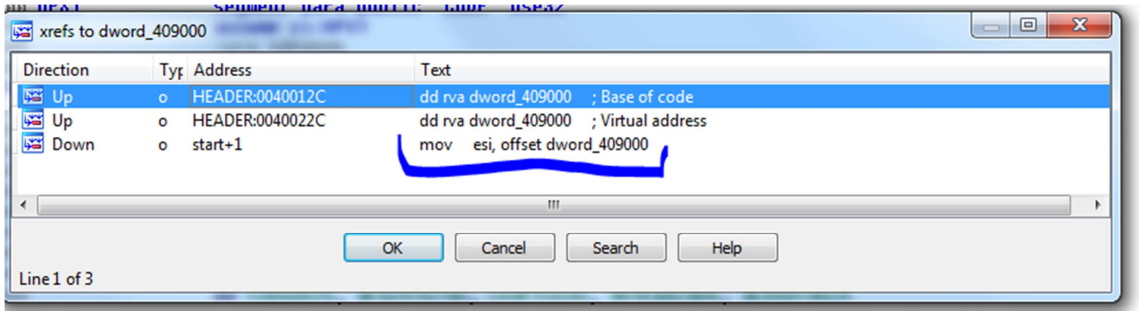


In 0x401000, there is a reference to executable code. Later, we will see what it does.

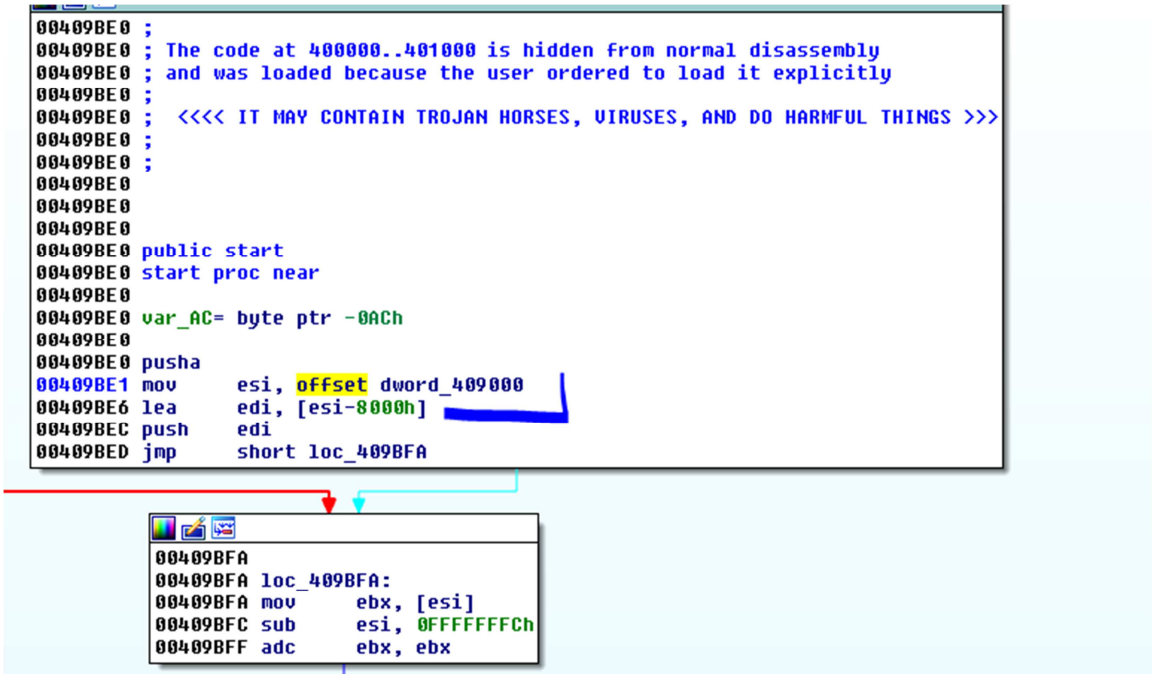
```
UPX1:00409000 ; Section 2. (virtual address 00009000)
UPX1:00409000 ; Virtual size                  : 00001000 (   4096.)
UPX1:00409000 ; Section size in file          : 00000E00 (   3584.)
UPX1:00409000 ; Offset to raw data for section: 00000400
UPX1:00409000 ; Flags E0000040: Data Executable Readable Writable
UPX1:00409000 ; Alignment     : default
UPX1:00409000 ; ===========================================================================
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, 0A30205D0h, 4020CAh, 6F4680Bh
UPX1:00409000                                         ; DATA XREF: HEADER:0040012C↑o
UPX1:00409000                                         ; HEADER:0040022C↑o ...
UPX1:00409000                 dd 0BBA0410h, 0FEEB6D9Bh, 0C30174C0h, 0F6405C7h, 9000203h
UPX1:00409000                 dd 41112868h, 409A67B6h, 7009006Ch, 0BF7EFEEFh, 74A33DA1h
UPX1:00409000                 dd 3E506442h, 0C78A324h, 7F0068h, 73FDF348h, 0A3061054h
UPX1:00409000                 dd 803B207Ch, 0BE673605h, 211084CDh, 7C778813h, 0B766EE64h
UPX1:00409000                 dd 0FF338441h, 929035h, 7DB70080h, 0B04C5BEh, 0B4686Eh
UPX1:00409000                 dd 0E704CF00h, 9F7DAF30h, 905F35DDh, 10B04A3h, 6BD70737h
UPX1:00409000                 dd 66117D97h, 51177E0Ah, 0DD0B0875h, 48ECEC1Eh, 1B48556Ah
UPX1:00409000                 dd 53D66D2h, 0F6C91674h, 5A0F9364h, 0D4EBA209h, 0EDDD5042h
UPX1:00409000                 dd 0E05F77Fh, 575679C8h, 0C7D8353h, 815E7402h, 36020405h
UPX1:00409000                 dd 0DDDCD365h, 120090B9h, 185D7405h, 1287401h, 0DDCD9A7Eh
UPX1:00409000                 dd 7424274Ah, 6C01114Fh, 9BB814EBh, 2B1F7339h, 90B73EBh
UPX1:00409000                 dd 67149669h, 26CB60Fh, 969F0C10h, 3CD153EBh, 67F67FEEh
UPX1:00409000                 dd 43EB4523h, 3A0A41EBh, 0C7145D8Bh, 51181843h, 6F3D9CECh
UPX1:00409000                 dd 0A01C06h, 25240D20h, 0EDF7737Fh, 67107A14h, 65051574h
UPX1:00409000                 dd 7466B574h, 5B00EB25h, 0F0F8605Fh, 0C2C95EF7h, 130AF710h
UPX1:00409000                 dd 1F680A40h, 0FBB2727Ah, 9C6C649Bh, 531BDDEBh, 0D8FD1512h
UPX1:00409000                 dd 0F88317B7h, 8E68BEC1h, 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.



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…

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.

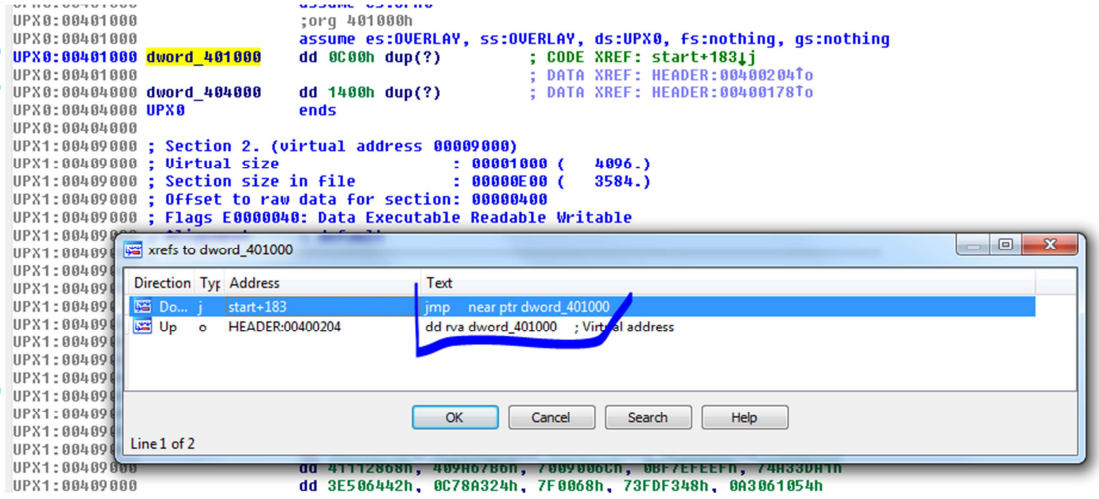
We don’t have to be a genius to know that it will read bytes from 0x409000, it will apply some operations and save them in 0x401000. EDI = ESI-0x8000.

```
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
```

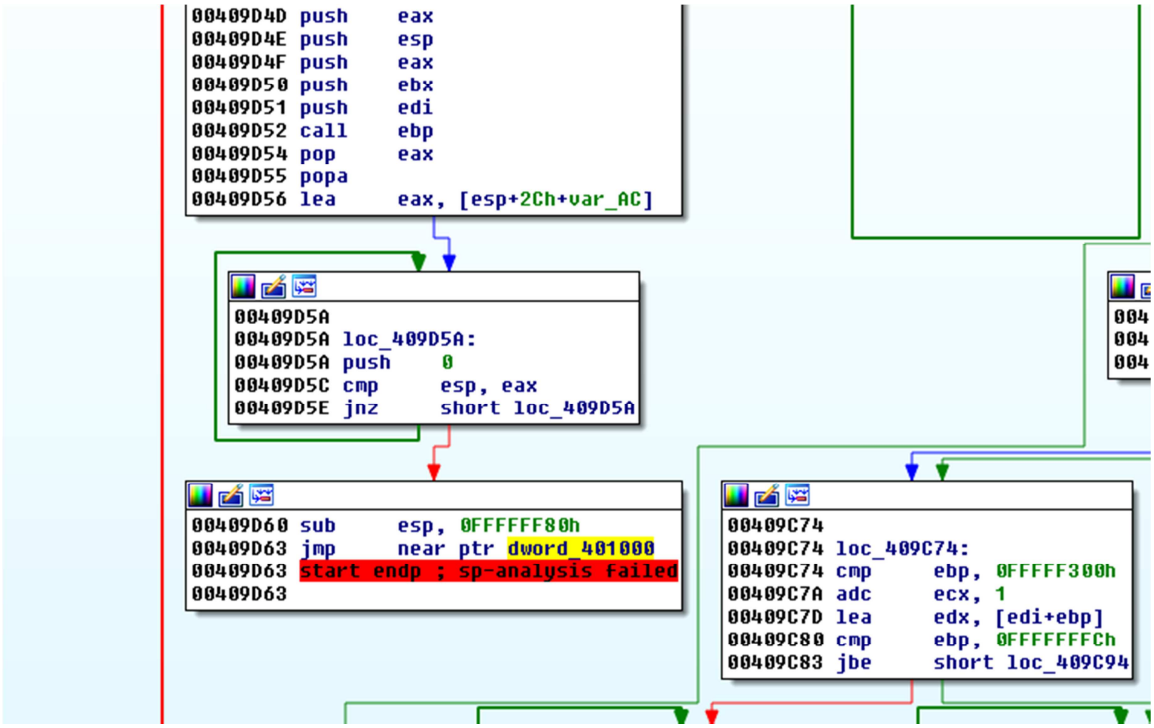
```
Python>hex(0x409000-0x8000)
0x401000
```

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)

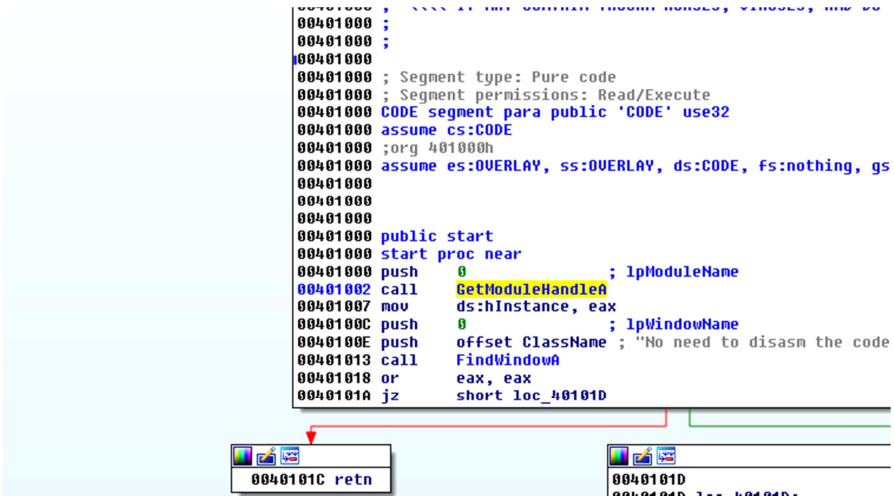


We see a JMP to 0x401000.



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.

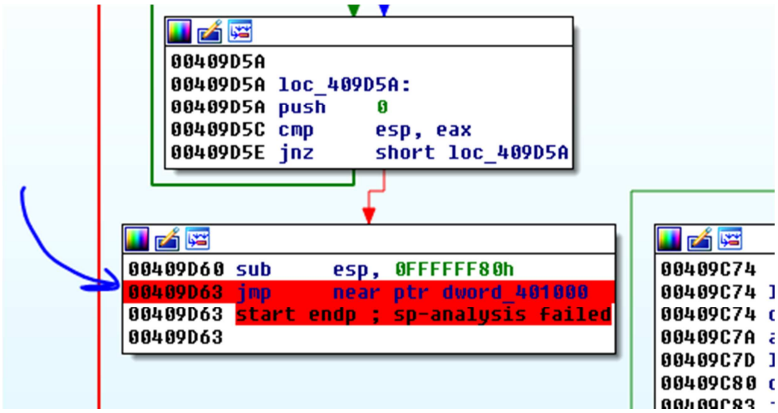
We will call OEP or ORIGINAL ENTRY POINT to the original program ENTRY POINT. As it is a packed program, we don't know where it is and as we have the original one, we can know that it was in 0x401000.



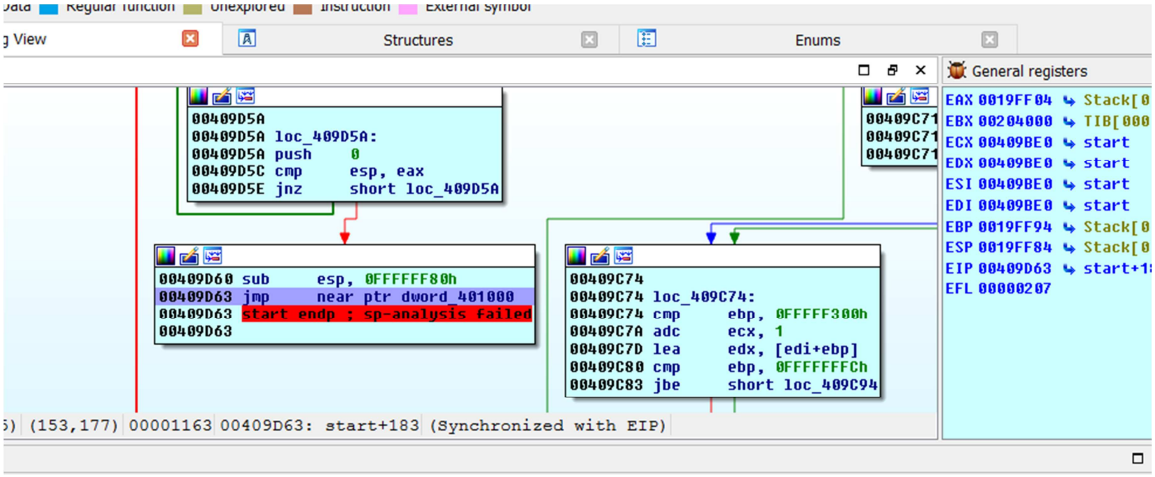
Most of the times, we have a packed program and we don't know the OEP because we don't have the original one. So, let's learn how to find it.

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.

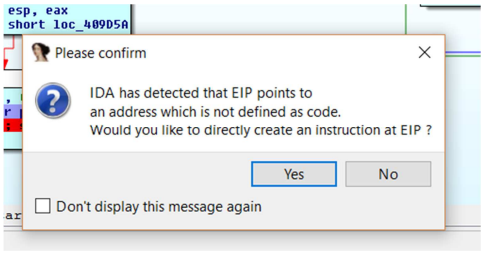
We could set a Breakpoint in that JMP to OEP to see if the original program is already created there. Let's try it.



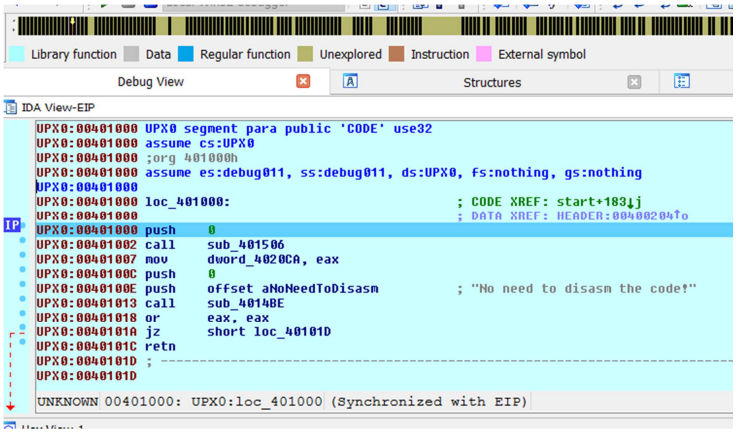
Select LOCAL WIN32 DEBUGGER and press START DEBUGGER.



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

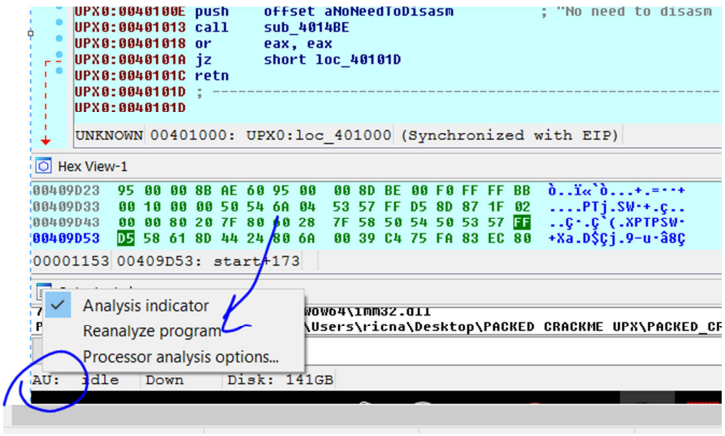


Press YES to interpret the first UPX0 section as CODE that was defined as DATA.

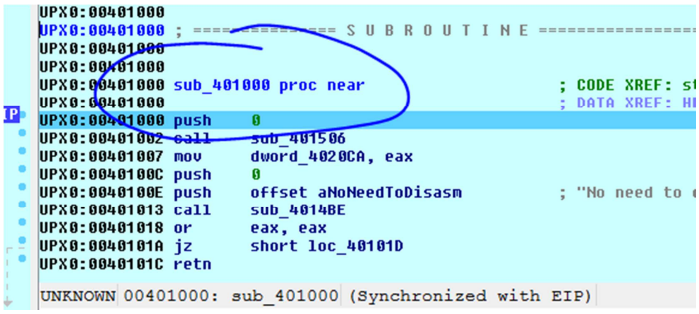


It unpacked the code and jumped to execute it. The code is very similar to the 0x401000 of the original, although, we can't change IDA to graphical mode because it is not defined as function (loc_401000), but we will do it automatically.

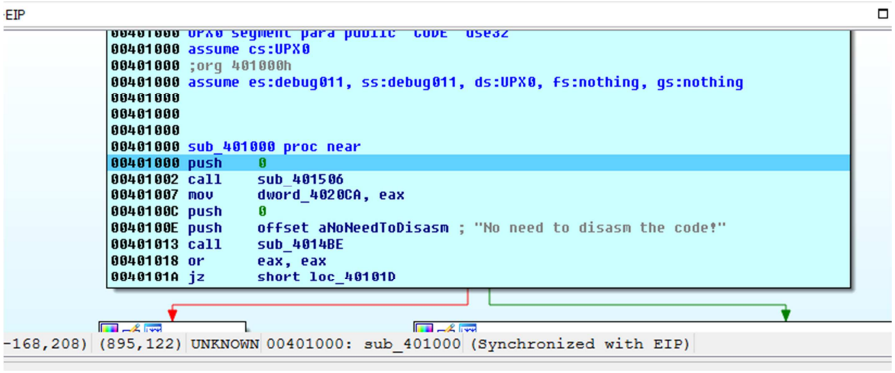
There is a almost hidden menu on the upper left corner. Right click- REANALYZE PROGRAM.



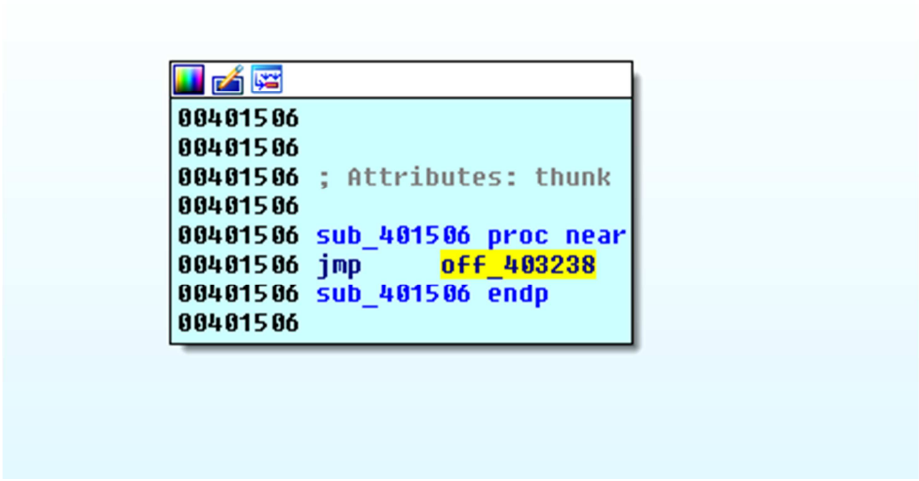
When clicking on the address loc_401000, it changed to sub_401000 which indicates that it is a function now. Then, we can change it to graphical mode with the space bar.



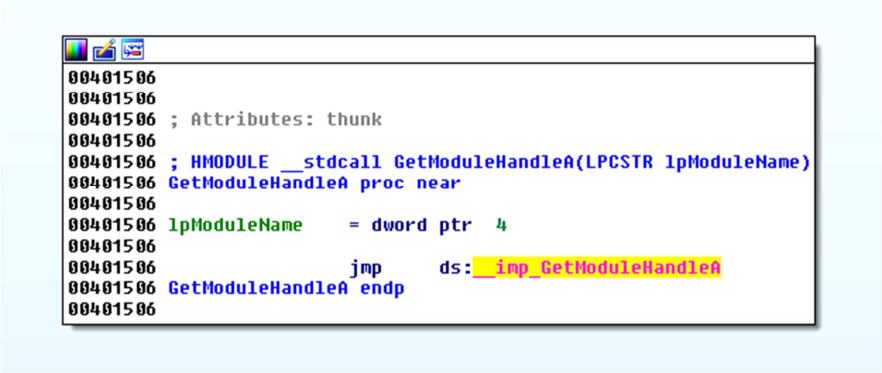
Now, it looks better.



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.

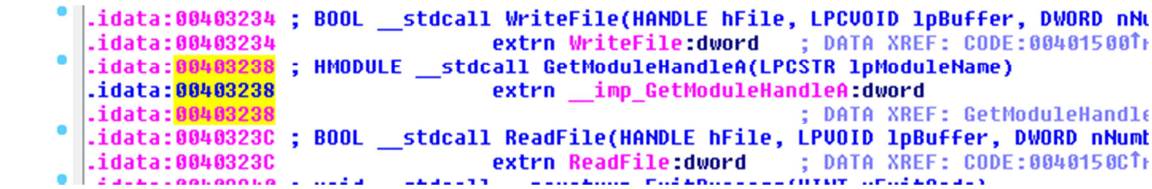


Let's see the difference with the original. If we enter the CALL GetModuleHandleA in the original.



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?

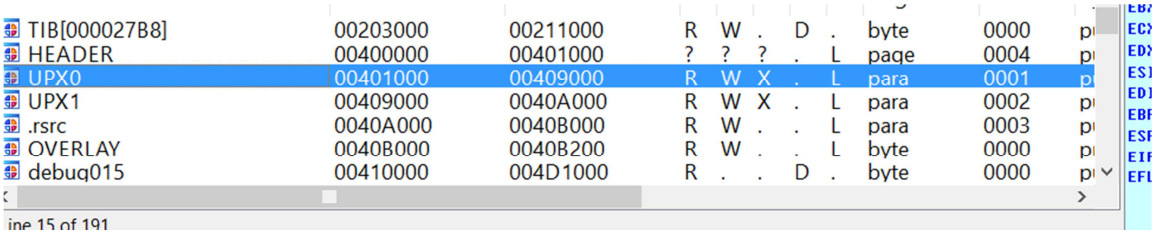
The 0x403028 content is an offset (off_). In the GetModuleHandleA API address and in the original, we see the same address in the **idata** section and it has the address of the same API.



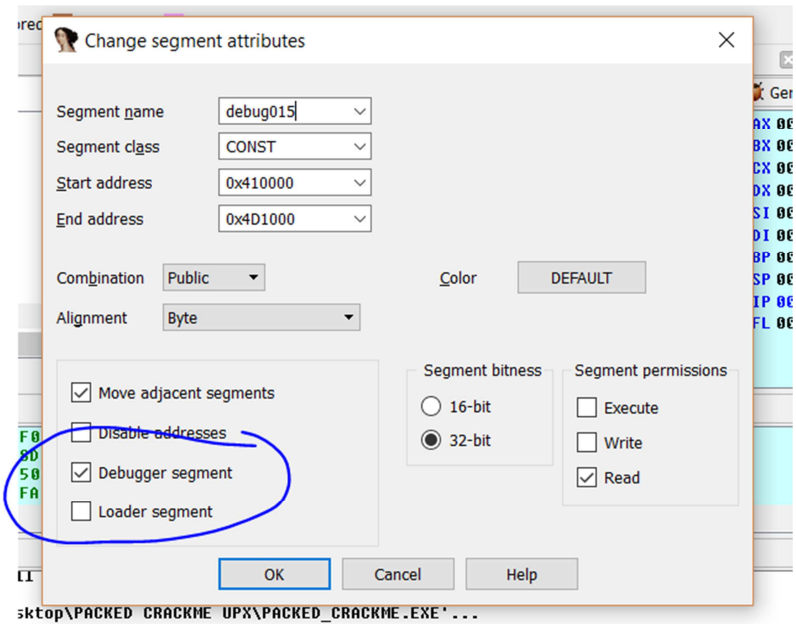
In spite of that they end up jumping to the same place, there is a very important difference we will see later.

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.

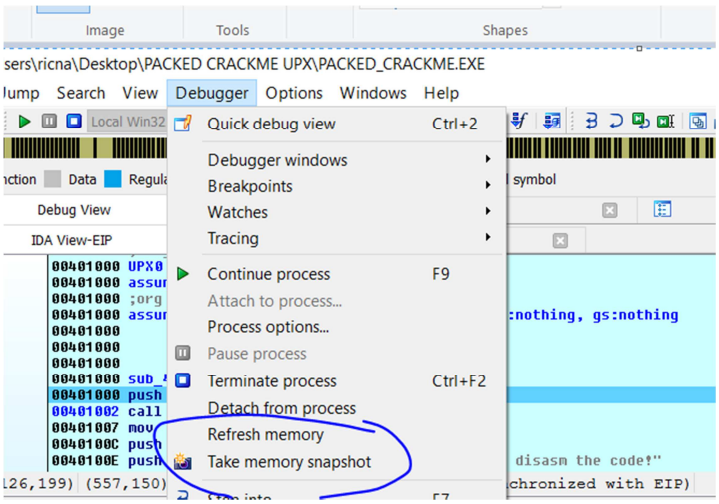
First in SEGMENTS.

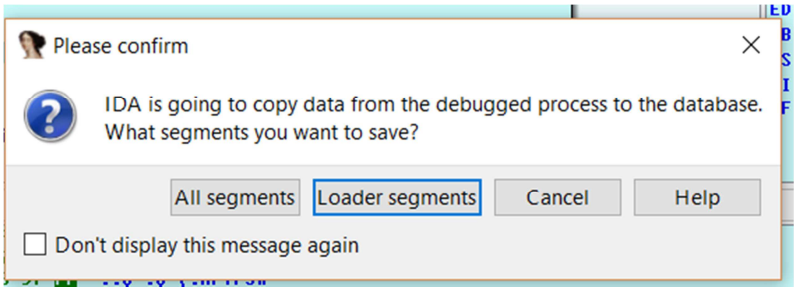


I verify that all the packers sections have the letter **L** that means they will be loaded by the LOADER. I can add some DLL or segment I want to be in the static analysis. Right click- EDIT SEGMENT on the line we want to add to the LOADER.



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.



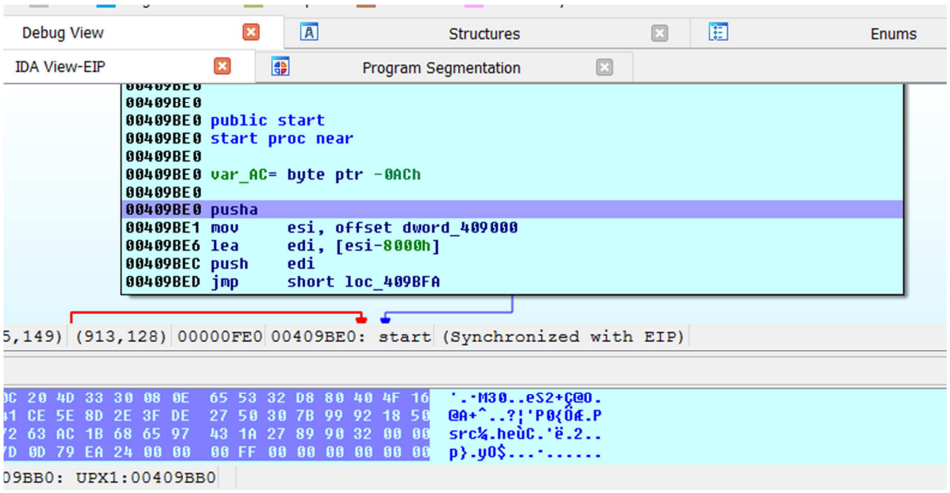


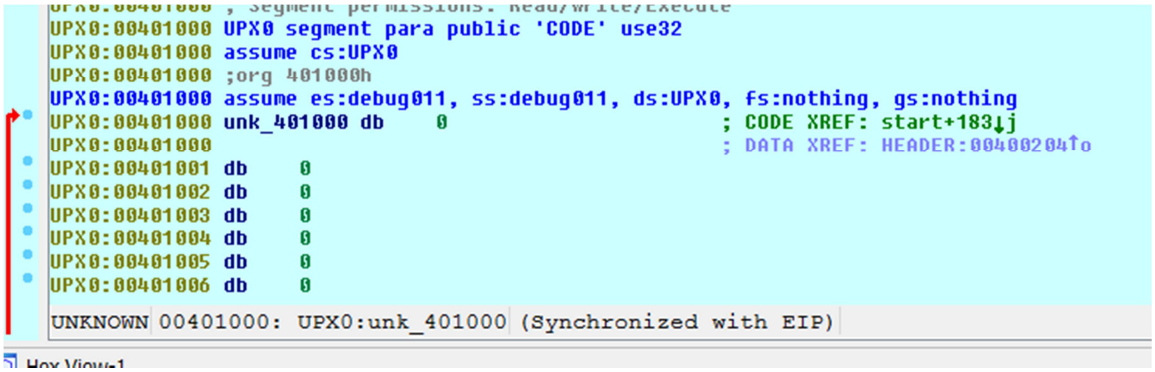
Then, the TAKE MEMORY SNAPSHOT option will save the segments we have marked as LOADER with the code they have. Don't confuse it with the other FILE-TAKE DATABASE SNAPSHOT option that we studied before.

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.

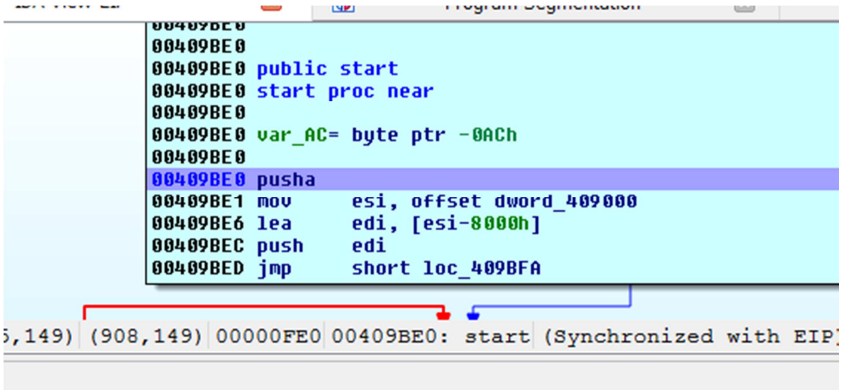




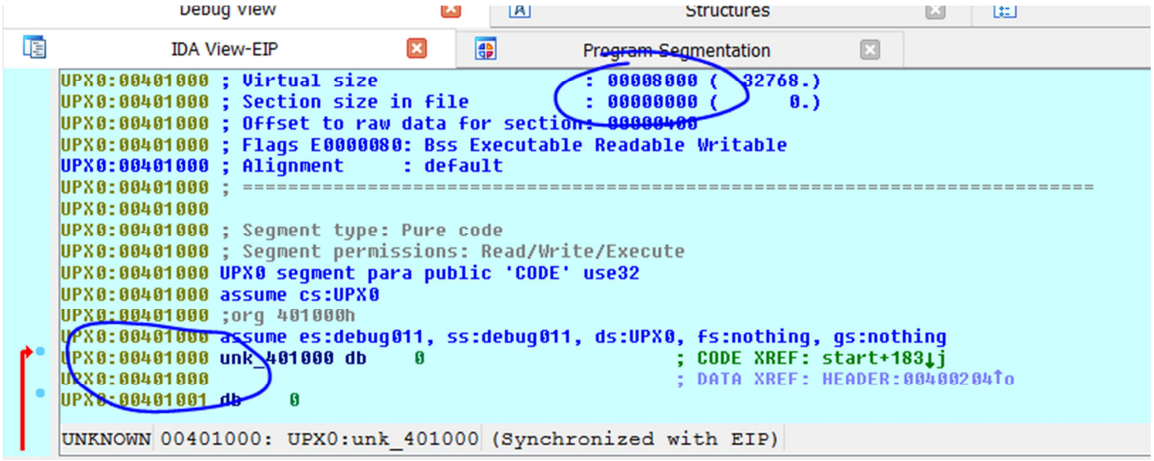
What we saved in the database is lost because, in the debugger, the LOADER info was overwritten with the bytes that initialized the UPX0 section. So, if we need it for static reversing, as I said before, after selecting TAKE MEMORY SNAPSHOT, we have to copy it to another folder before running the debugger.

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.

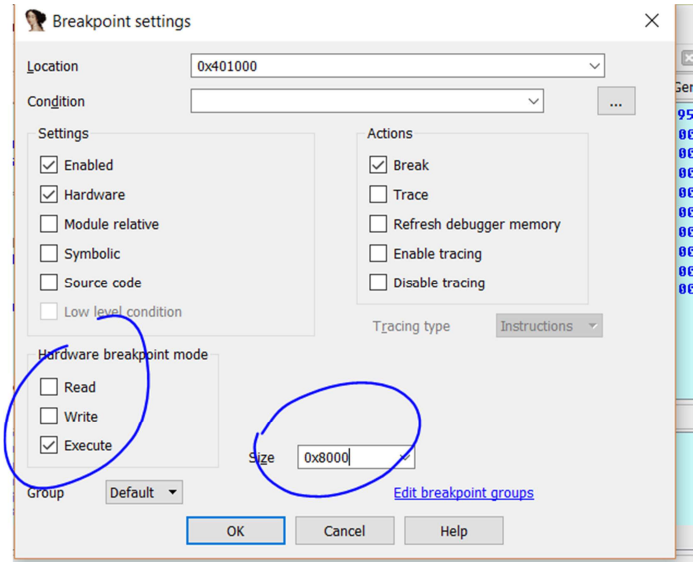
I run the packed file again in the debugger stopping at its EntryPoint.



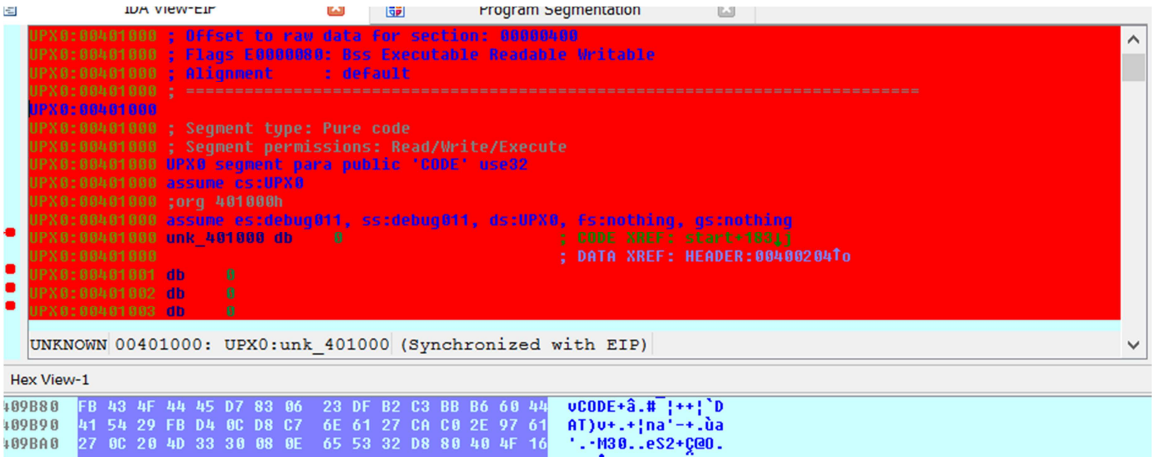
I go to the first section where it starts. This is 0x401000.



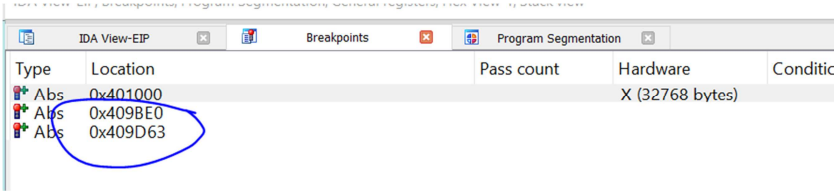
I set a BreakPoint with F2. I set it to stop by execution or EXECUTE. It will stop just when executing not reading or writing. As it will stop when copying the code and creating it I don't want that. I just want it to jump when it is already created and it stops in the first instruction it executes and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



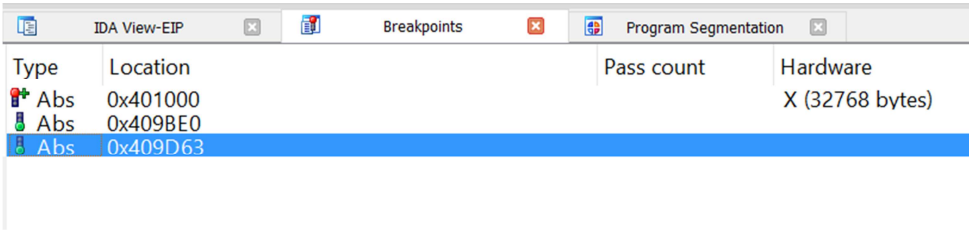
All instructions are marked in red.



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

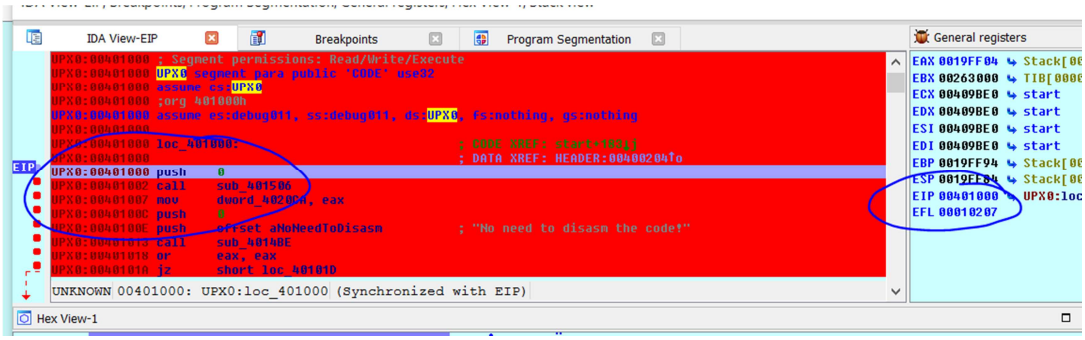


Right click-DISABLE.



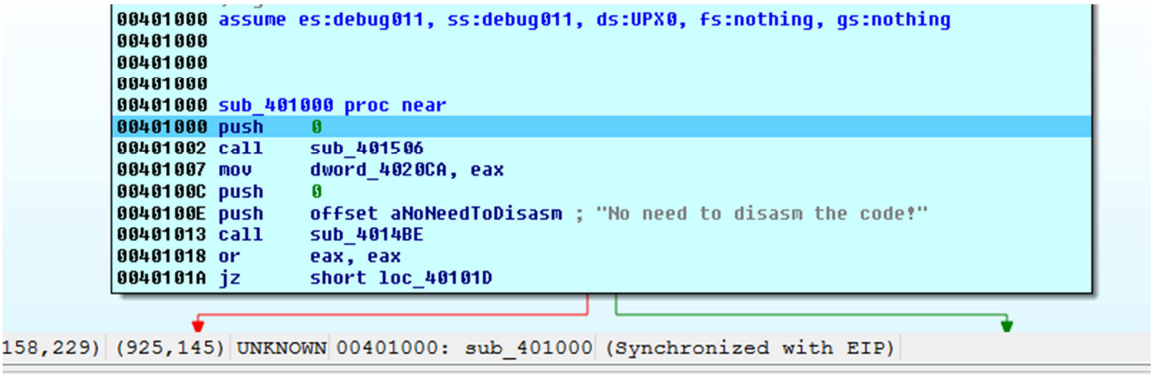
And now, RUN.

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



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

After reanalyzing, it becomes a function again.



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