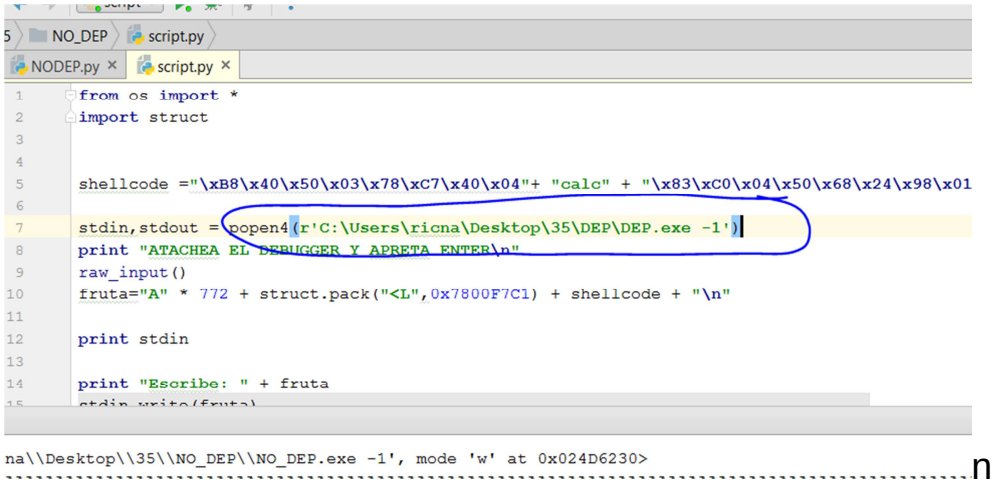
PART 36

We'll start working with the version that has DEP on. We know that it is similar, but what happens if we use the script of the no DEP version?

Change the path in the script to point to DEP.exe.

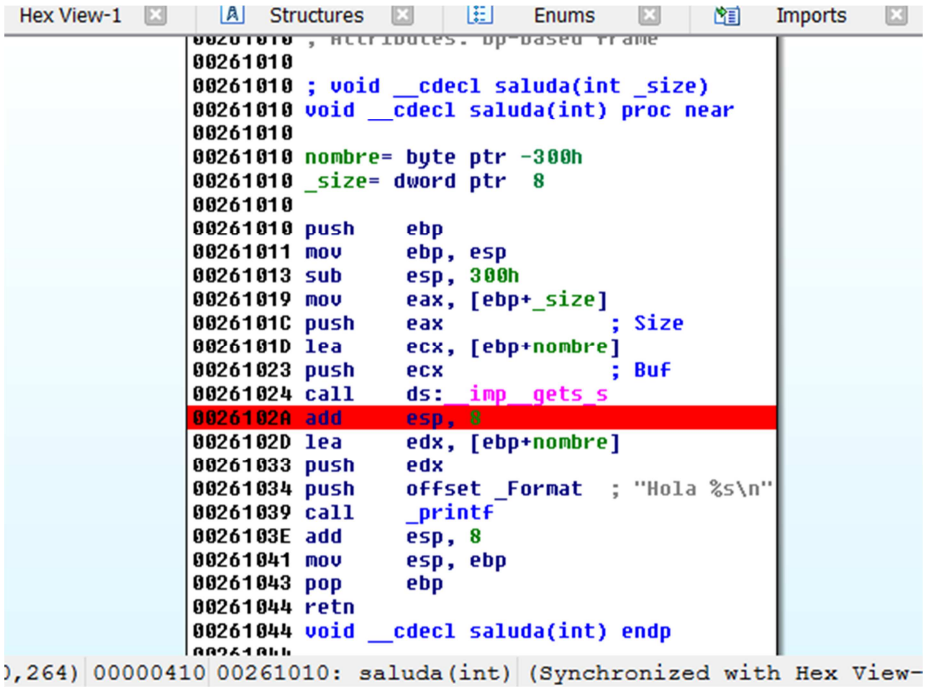


```
5 NO_DEP script.py
NODEP.py × script.py ×
1  from os import *
2  import struct
3
4
5  shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01
6
7  stdin,stdout = popen4(r'C:\Users\ricna\Desktop\35\DEP\DEP.exe -1')
8  print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
9  raw_input()
10 fruta="A" * 772 + struct.pack("<L",0x7800F7C1) + shellcode + "\n"
11
12 print stdin
13
14 print "Escribe: " + fruta
15 stdin.write(fruta)
```

```
na\\Desktop\\35\\NO_DEP\\NO_DEP.exe -1', mode 'w' at 0x024D6230>
```

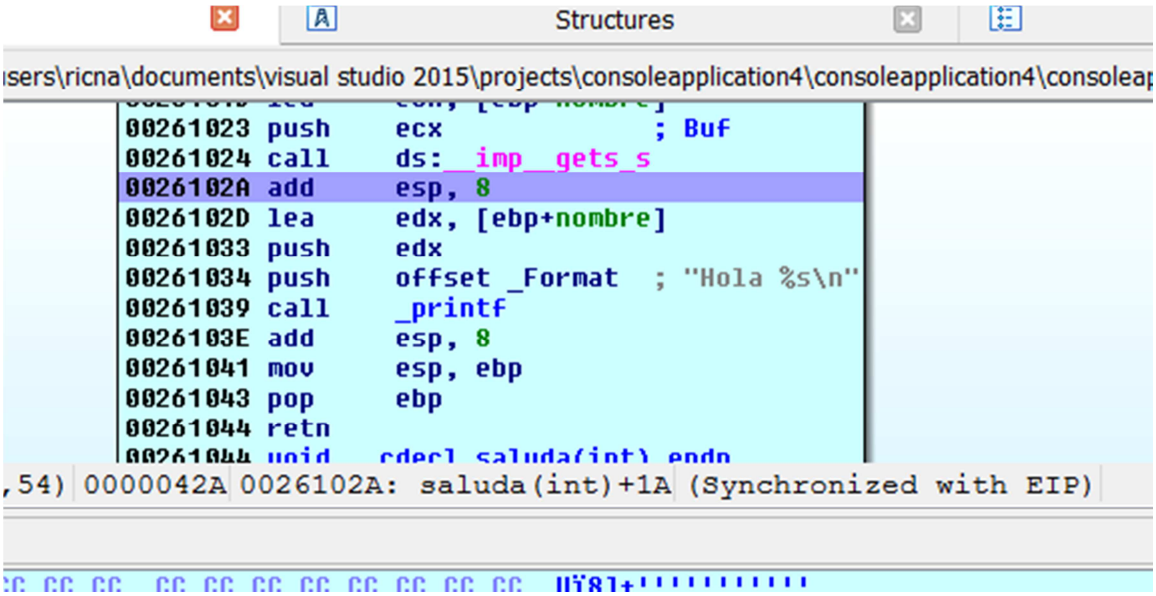
n

I run and attach it with IDA that has the DEP.exe analysis.

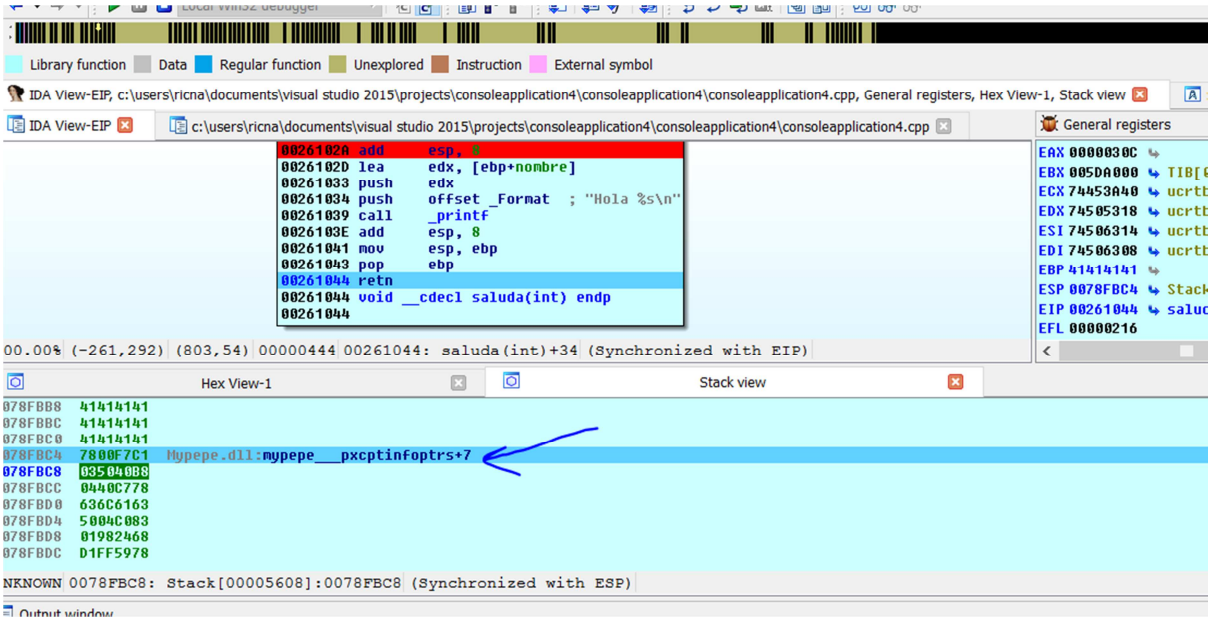


```
Hex View-1 | Structures | Enums | Imports
00261010 ; Attributes: bp-based frame
00261010
00261010 ; void __cdecl saluda(int _size)
00261010 void __cdecl saluda(int) proc near
00261010
00261010 nombre= byte ptr -300h
00261010 _size= dword ptr  8
00261010
00261010 push    ebp
00261011 mov     ebp, esp
00261013 sub     esp, 300h
00261019 mov     eax, [ebp+_size]
0026101C push    eax             ; Size
0026101D lea     ecx, [ebp+nombre]
00261023 push    ecx             ; Buf
00261024 call    ds:__imp__gets_s
0026102A add     esp, 8
0026102D lea     edx, [ebp+nombre]
00261033 push    edx
00261034 push    offset _Format  ; "Hola %s\n"
00261039 call    _printf
0026103E add     esp, 8
00261041 mov     esp, ebp
00261043 pop     ebp
00261044 retn
00261044 void __cdecl saluda(int) endp
),264) 00000410 00261010: saluda(int) (Synchronized with Hex View-
```

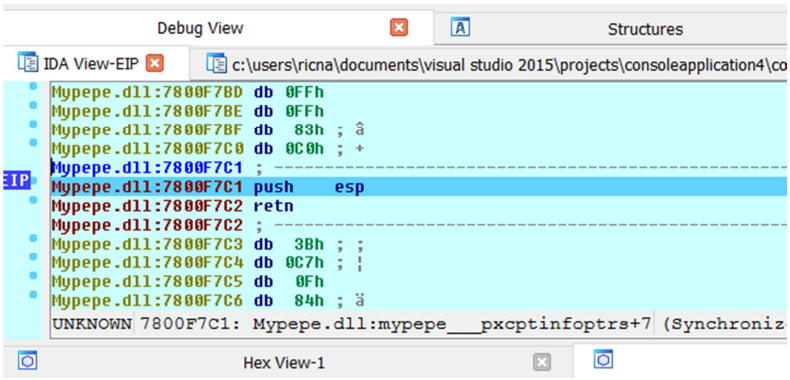
I will set a Breakpoint in the **saluda** function to see what happens just after the gets because the process is waiting inside the Api when we attach it.



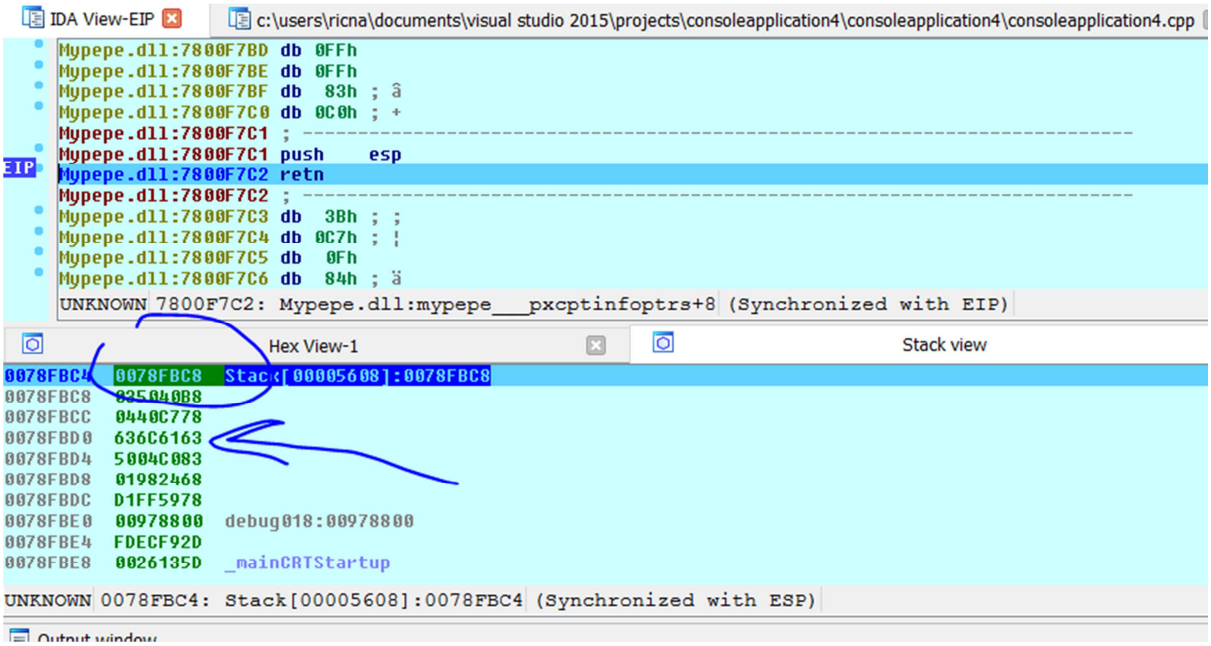
It stopped there. I will trace it with F8 until the RET.



There, we see the stack overwritten. It will jump to the PUSH ESP – RET that is in 0x7800f7C1. It should have no problem here because it is a code section instruction of a module and these sections have always execution permission. Press F7.

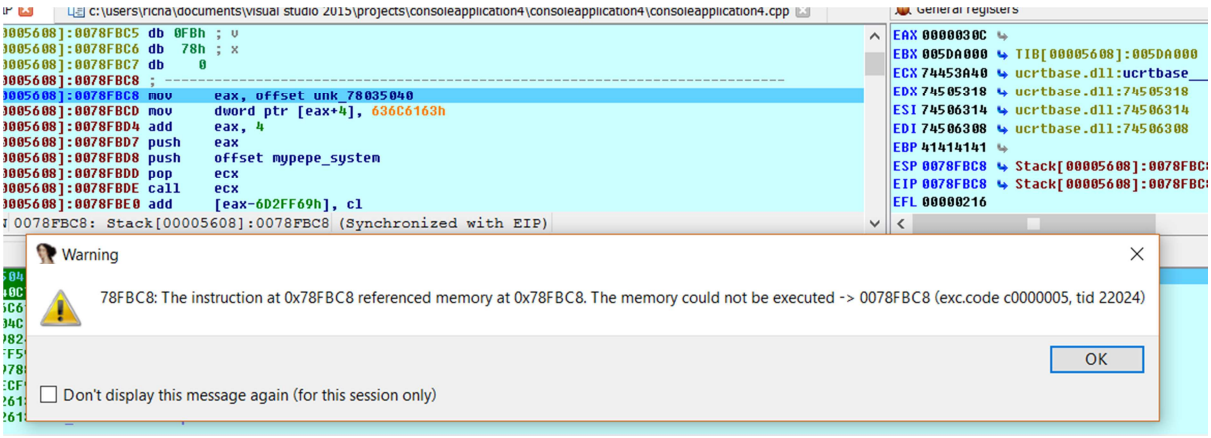


Trace it with F7.



It PUSHes the ESP register value and then, it comes to the RET. It will jump there to execute 0x78fbc8 as if it were a Return Address and there it is my shellcode in the stack. In the NO_DEP, it jumped and executed that shellcode I sent, but what happens here? Press F7.

0078FBC4 0078FBC8 Stack[00005608]:0078FBC8

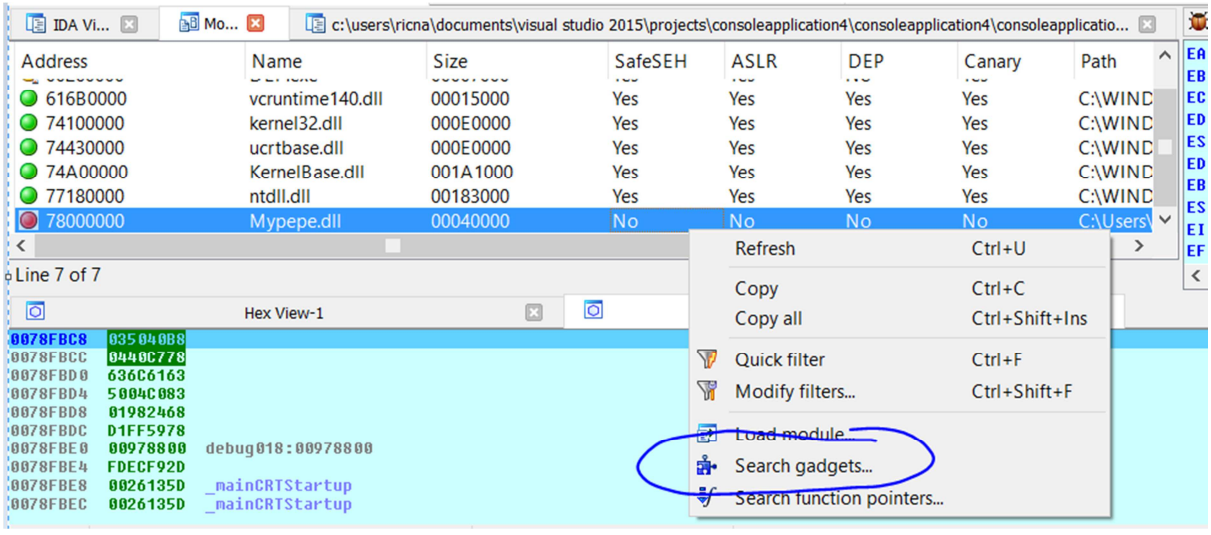


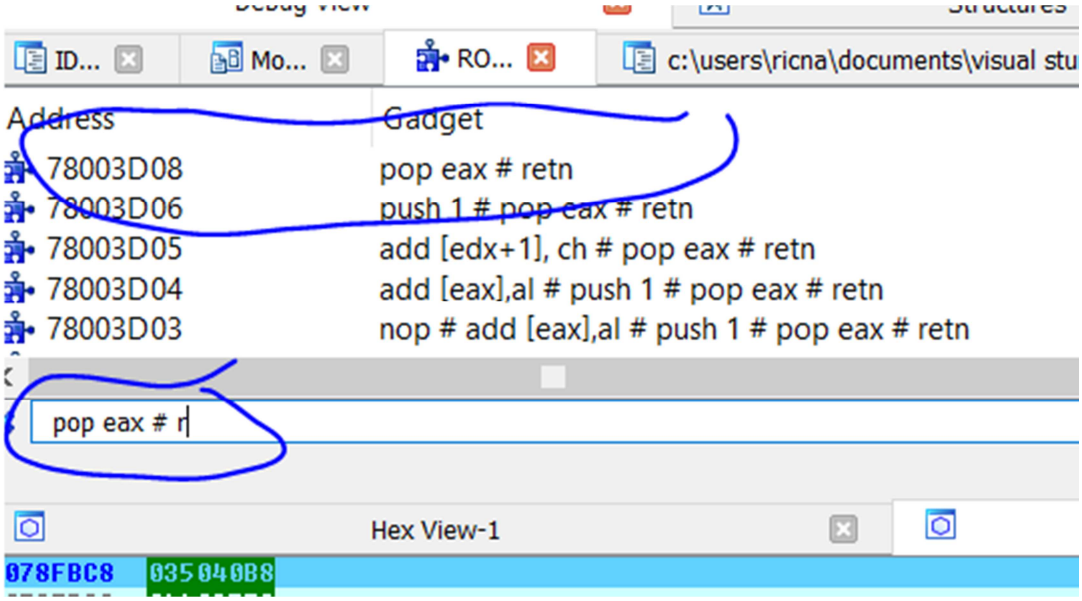
The code that it executed in the NO_DEP in the stack without problems can't be executed here because the DEP doesn't allow execution permission to the stack (the heap, etc) and it only has read and write permission.

What can we do here?

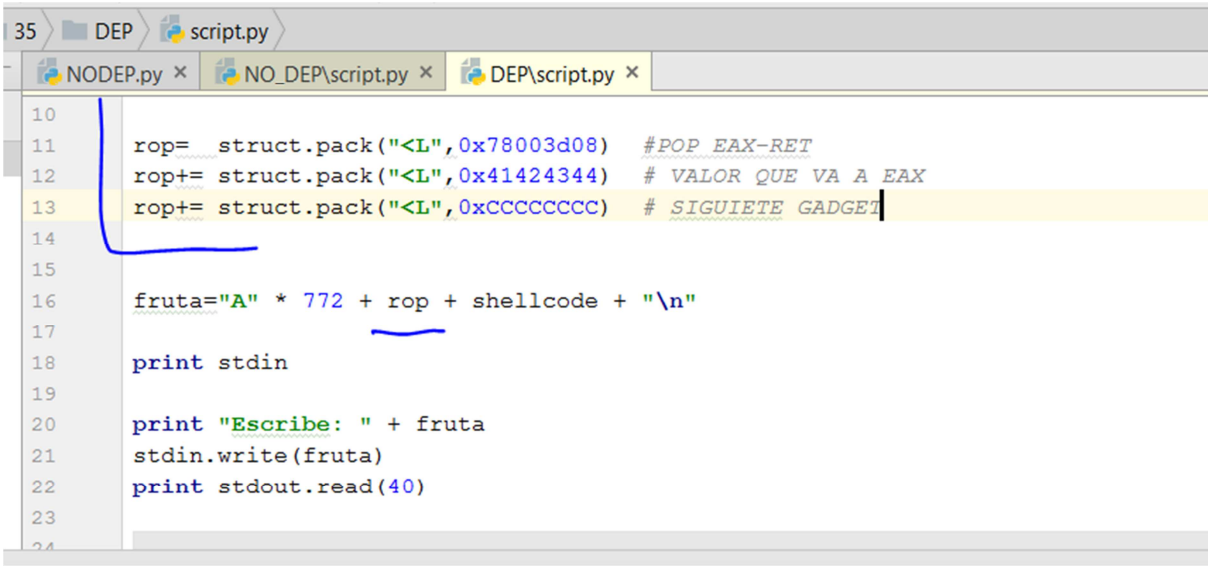
When it jumps to a DLL code as we did with the PUSH ESP-RET can be done and that's the ROP idea. Put gadgets together that are code chunks ended in RET to give the stack, heap or whatever execution permission.

For example, if I want to set a value in EAX, instead of jumping to the PUSH ESP-RET, I will jump to a POP EAX-RET. I will find a POP EAX-RET in the idasploiter gadgets in mypepe DLL with no ASLR.

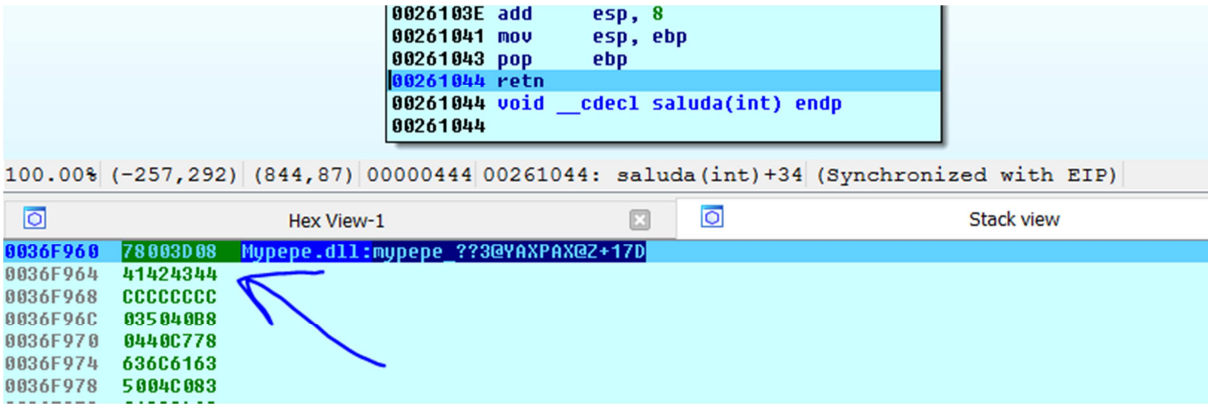




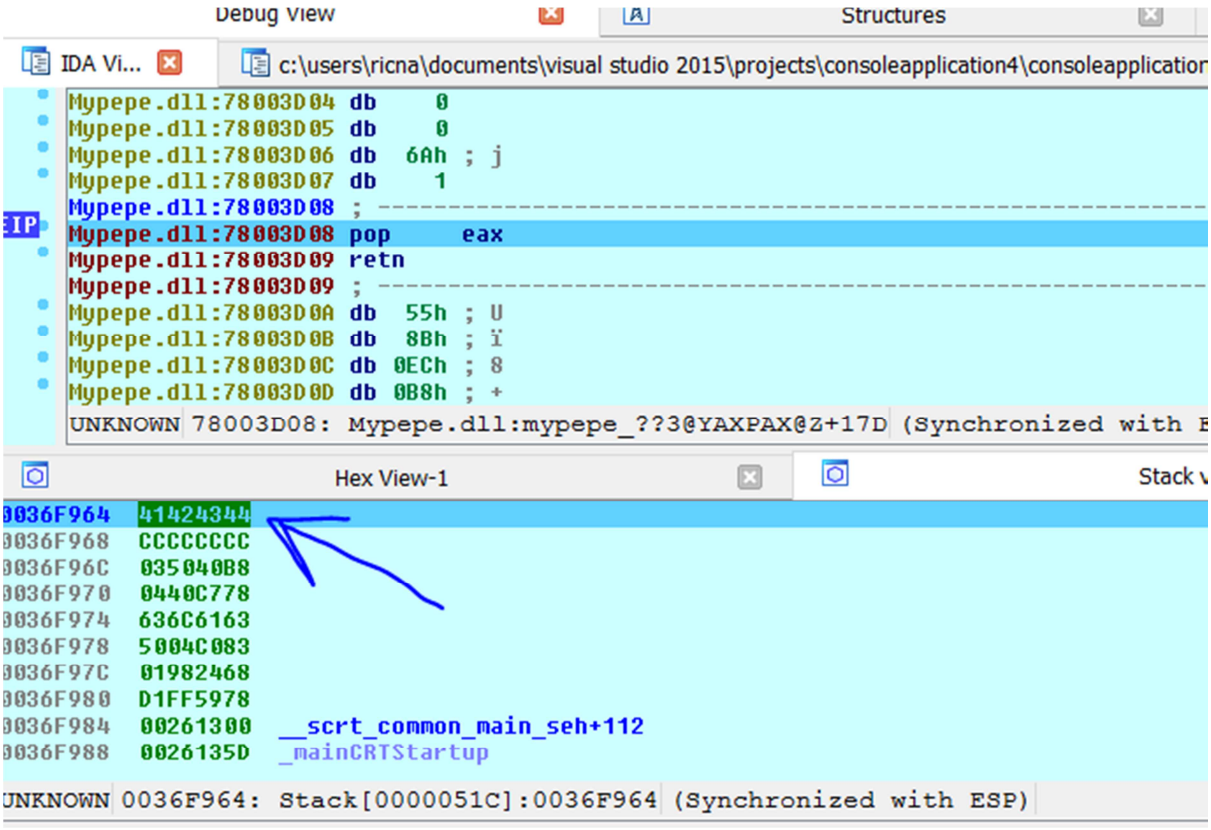
There, it is the gadget in 0x78003d08.



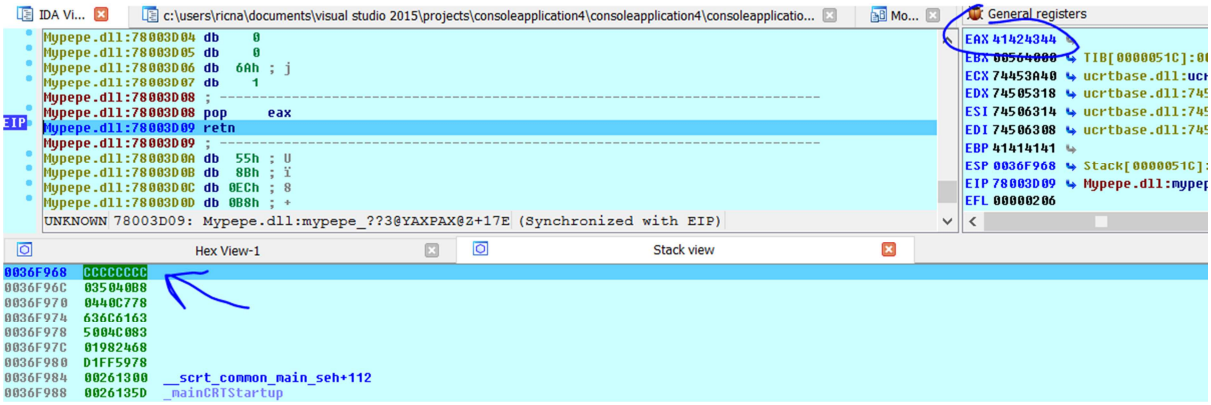
There, we replace the jump to the PUSH ESP - RET by our ROP that starts with a POP EAX that moves the 0x41424344 value that is below to EAX and then, when it comes to the RET (remember that all or almost all gadgets end in RET), it jumps to the next gadget in 0xCCCCCCCC. I we'll see what it is later, but let's execute this to see what happens. Attach it again and trace it as before.



When coming to the RET, I see the POP EAX-RET jump in my ROP. The 0x41424344 that will end up in EAX and the 0xCCCCCCCC where I should add the pointer to the next gadget. Trace it with F7.



We jump to my first gadget. It is the POP EAX-RET. We know that the POP takes a value out of the stack and moves it, in this case, to EAX. If I execute it with F7.



There, it was moved to EAX and as it came to a RET, it will jump to the next gadget, in this case, 0xCCCCCCCC. It doesn't have it yet, but we should add it there.

And this is the ROP. Putting different gadgets together to do what I want. One after the other. That's why it is called ROP (RETURN ORIENTED PROGRAMMING) because we are executing code without assembling our own instructions but pointing to code chunks that work.

There are the manual and automatic ways. We'll do it manually first. If you don't need it, skip it and go to the next part where it is done automatically.

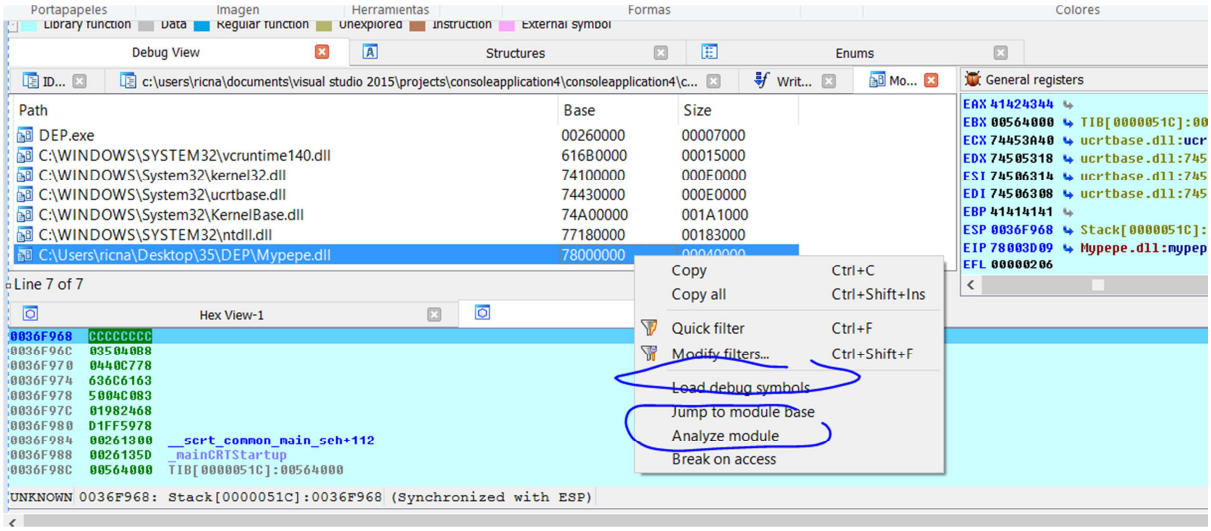
The general method is ordering certain values in the registers and then, jumping to a PUSHAD RET. We'll see that it will rearrange everything, although there are thousands of ways to do a ROP. This is the most common.

We have to decide which API we'll use first to unprotect the stack, in this case, it could be VirtualAlloc or VirtualProtect which are the most used, although there are more.

We know there are two modules now. The idasploiter's and IDA's. The last one is in

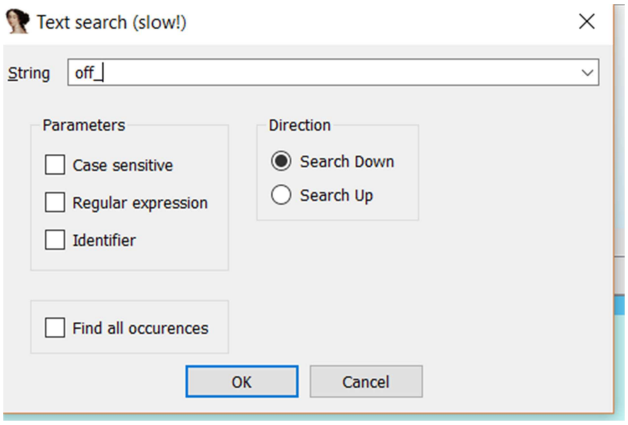
Sabemos que hay dos pestañas modules ahora la del idasploiter y la del IDA mismo, esta última está en DEBUGGER-DEBUGGER WINDOWS-MODULE LIST.

Right click there on mypepe and analyze it. Then, make it load its symbols if it has them.

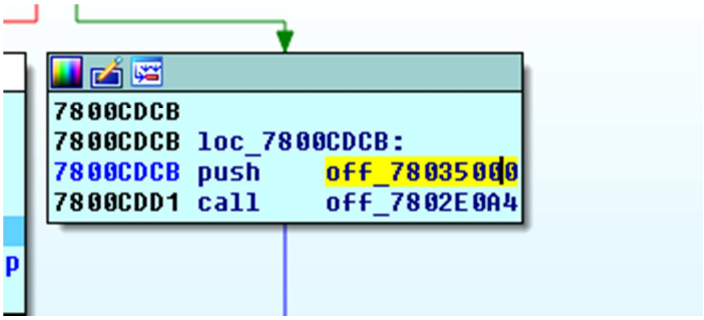


As we don't have mypepe symbols, it looks good, but it doesn't show some info as the imported functions used in the function list just mypepe's.

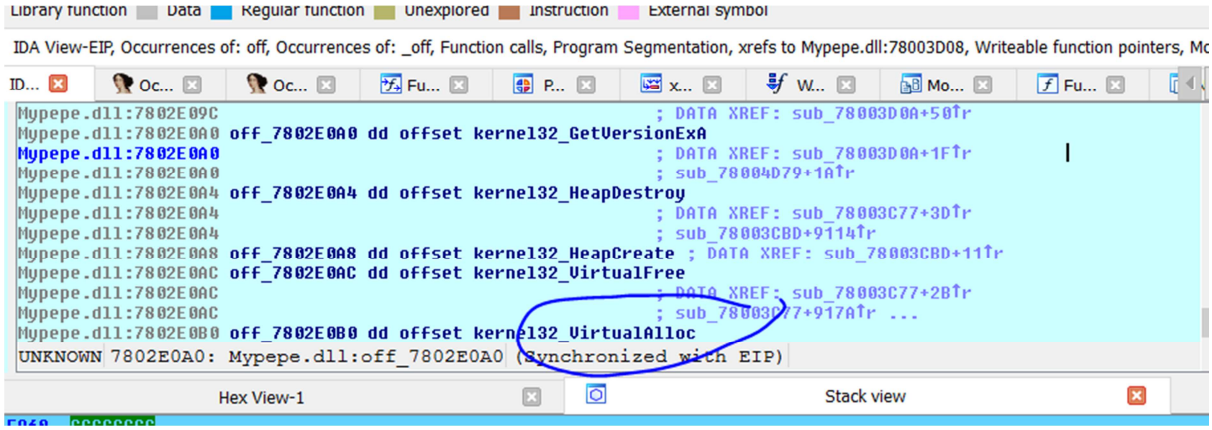
Let's search for the **off**_ string.



I don't need them all. Just one.



The CALL will be the typical jump to the IAT imported function because off_ as a prefix (don't confuse it with OFFSET that is the address), is the content of that address where it will jump is a pointer as in the IAT case. Let's go there.



Seeing a Little among the IAT functions, it is VirtualAlloc. Let's prepare a ROP for it (we already know that 0x7802e0b0 is the IAT start. From now on, I will call it VA)

The idea is to order these values in each register and then, with some PUSHAD RET, push them into the stack and they will be ordered as VirtualAlloc arguments. It is not magic. ☺

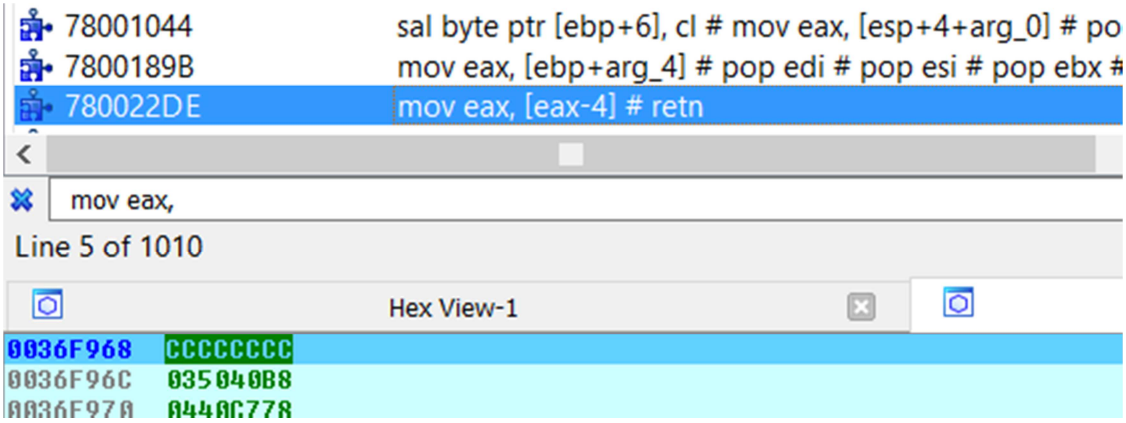
```
#                                                                    #
# EAX 90909090 => Nop                                                #
# ECX 00000040 => flProtect                                          #
# EDX 00001000 => flAllocationType                                   #
# EBX 00000001 => dwSize                                             #
# ESP ???????? => Leave as is                                        #
# EBP ???????? => Call to ESP (jmp, call, push,..)                   #
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR of 0x1005d060      #
# EDI 10019C60 => ROP-Nop same as EIP                                #
#                                                                    #
```

We have to add 0x90909090 in EAX. We have added that gadget. We just need to change that value to 0x90909090, but EAX has to be set with the last one because it can be necessary to set other values. It's recommended to add the most difficult ones.

The VirtualAlloc address must be in ESI and we only have the IAT location, but just the address will change. So, basing on that location, we will find the address and it will always work.

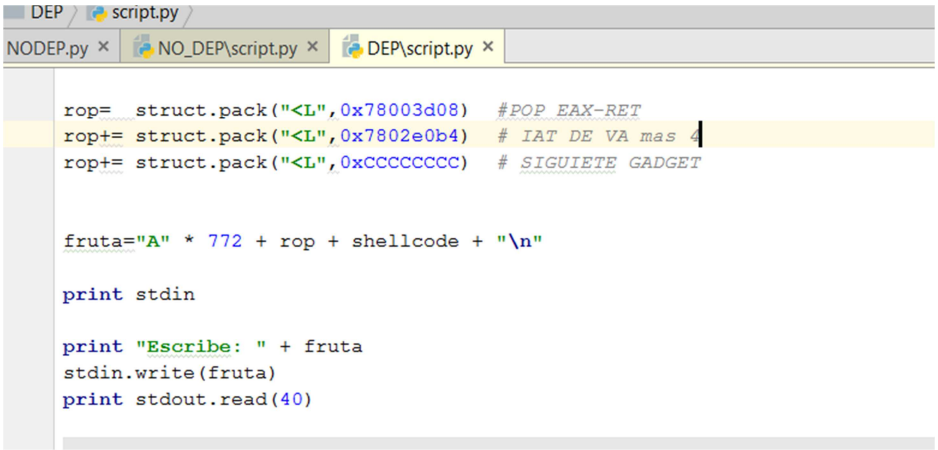
We have to find the easiest if there is a MOV ESI, [register] – RET. Let's search in the gadgets.

There isn't any, but we have:



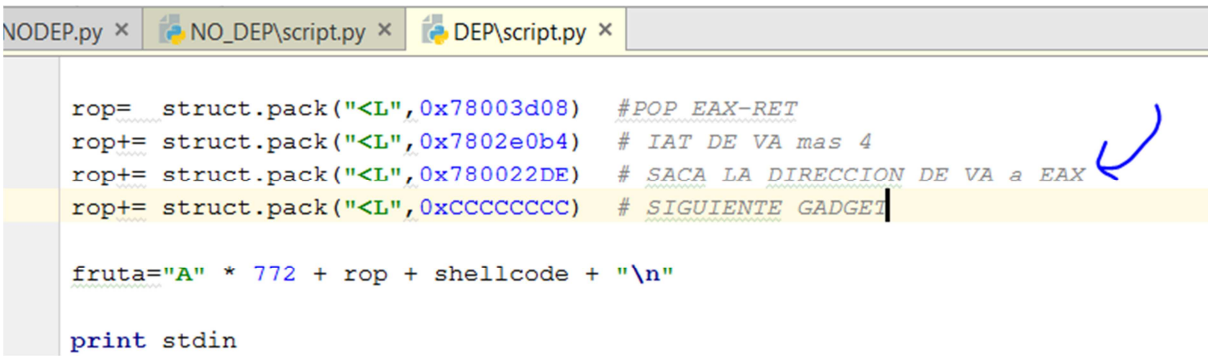
We move the IAT address + 4 to EAX and with that instruction, we move the API address to EAX. We'll have to move it from EAX to ESI with other gadget. Let's order all this and try it.

The IAT location was 0x7802E0B0. We add it 4 and move it to EAX with the gadget we already had.



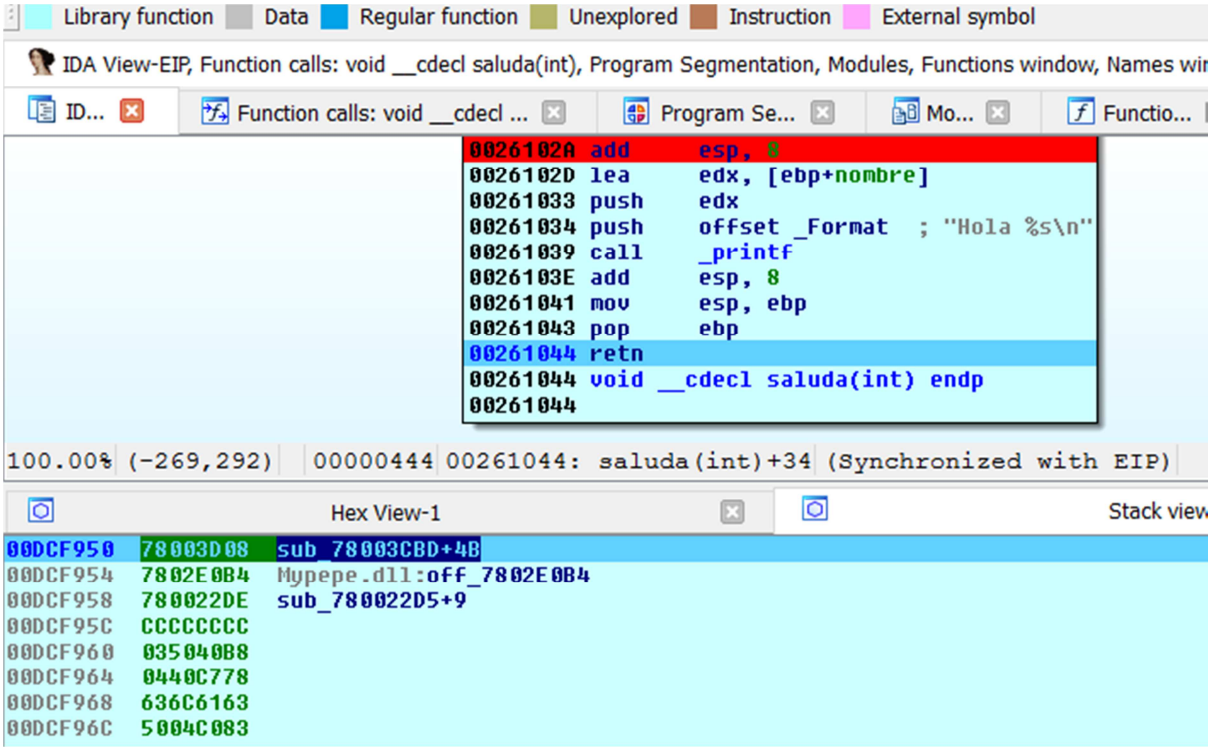
With that, we'll have the IAT location + 4 in EAX. The next gadget will be the one we found.

780022DE mov eax, [eax-4] # retn

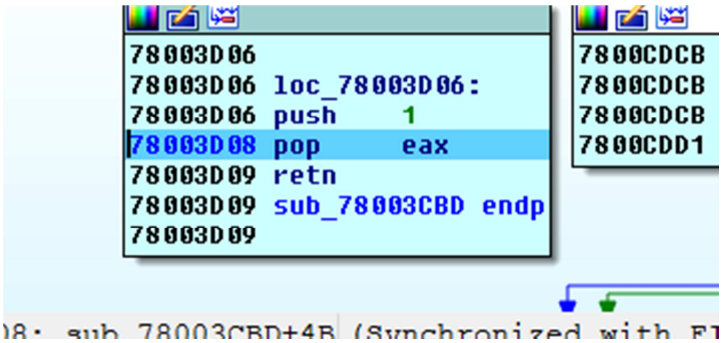


There, we put both gadgets together. Let's try it to see if it does what we think and the VA address is in EAX.

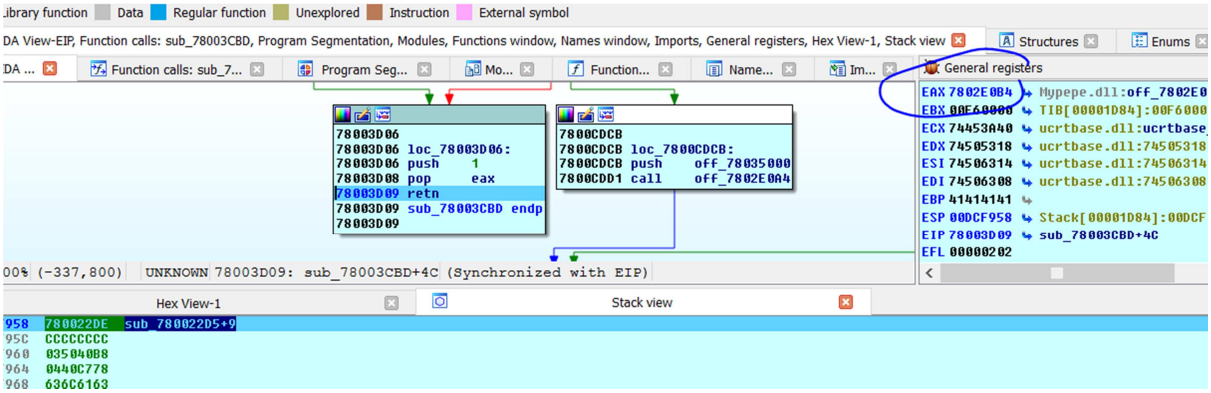
One of the uncomfortable things we'll have is that idasploiter just runs in debugging mode. So, if we need to debug it, it will close after restarting it, but we can have other IDA instance opened with the process stopped debugging to find in the IDA SPLOITER and debug it in the other.



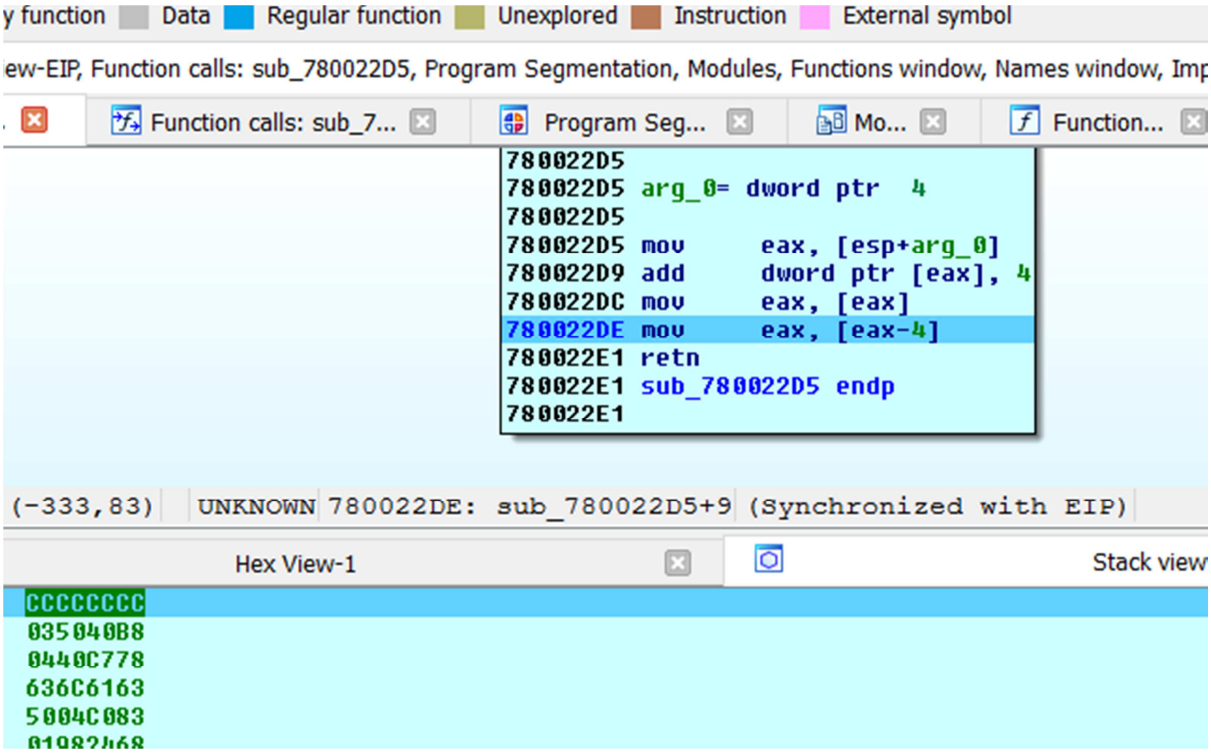
We will trace the ROP we did until now with F7.



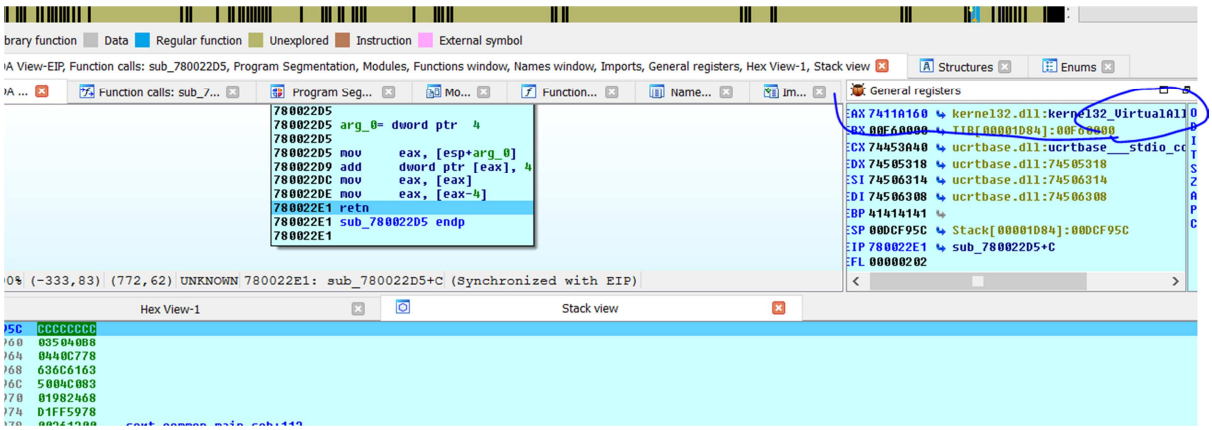
Now, it will move the IAT entry address from VA + 4 to EAX.



Now, it will jump to the second gadget. I continue with F7.



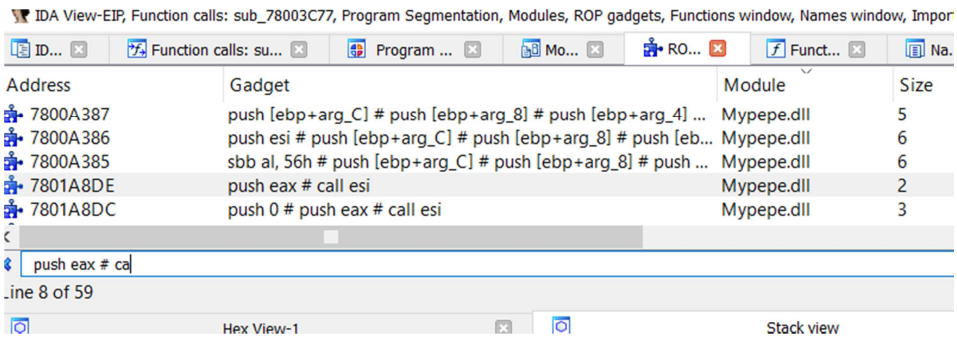
Execute it with F7.



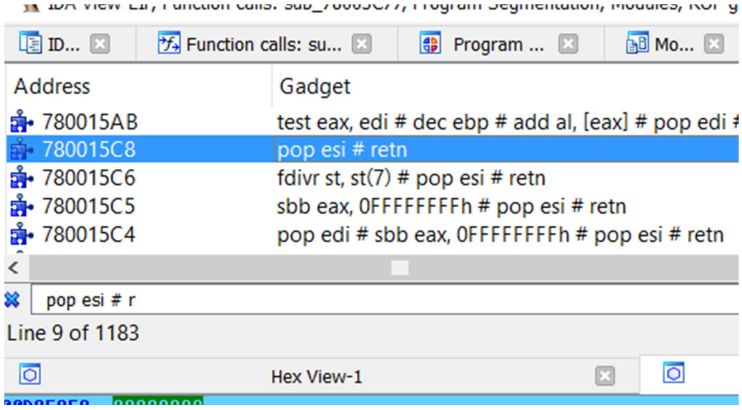
We got our goal. The API address is in EAX and we took it out from the IAT. So, it will work in any PC. The next gadget should move that address from EAX to ESI where it corresponds.

```
# skipping ESP leaving it intact.
#
# EAX 90909090 => Nop
# ECX 00000040 => flProtect
# EDX 00001000 => flAllocationType
# EBX 00000001 => dwSize
# ESP ???????? => Leave as is
# EBP ???????? => Call to ESP (jmp, call
# ESI ???????? => PTR to VirtualAlloc -
# EDI 10019C60 => ROP-Nop same as EIP
#-------------------------------------
```

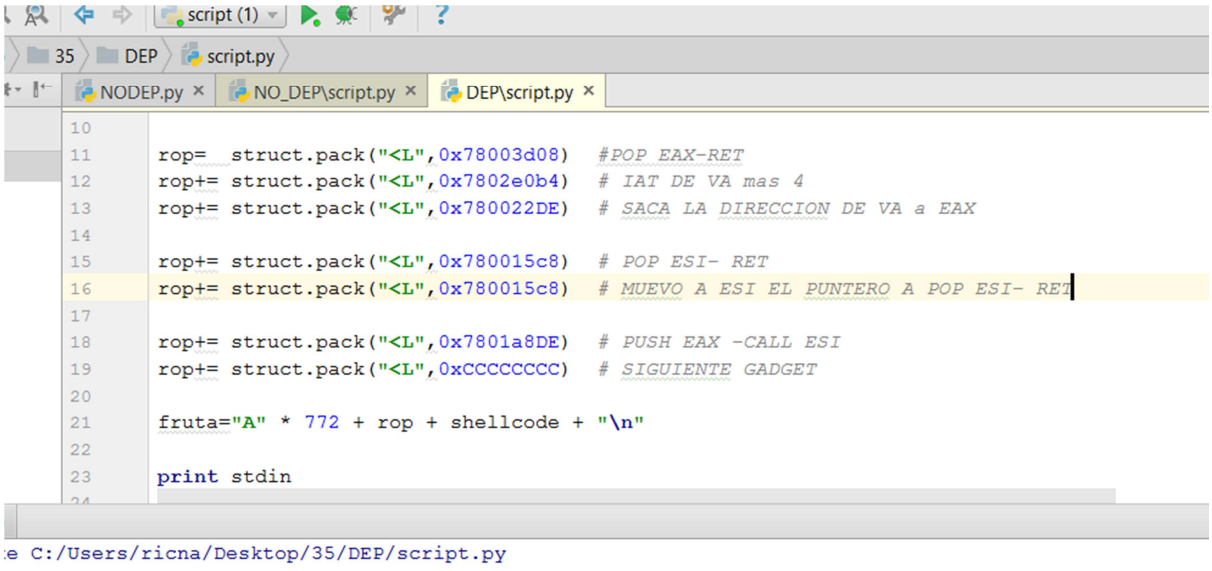
There is no MOV ESI. So, we need to use our imagination. Despite that the gadgets end in a RET normally, any code even if it doesn't have a RET, lets me continue and take control. It would be a gadget although less traditional, it will work.

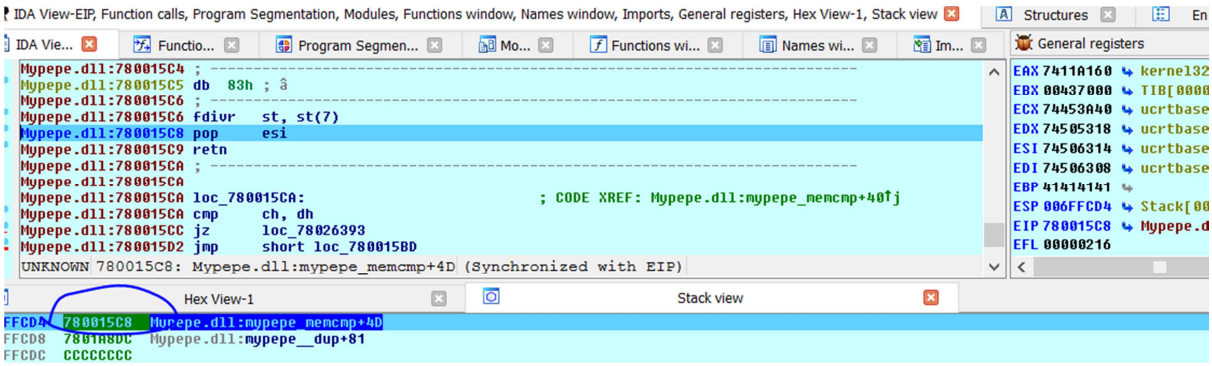


If I push the EAX value to the stack and prepare ESI to have a POP ESI - RET, I could move EAX to ESI using the stack. Let's see.

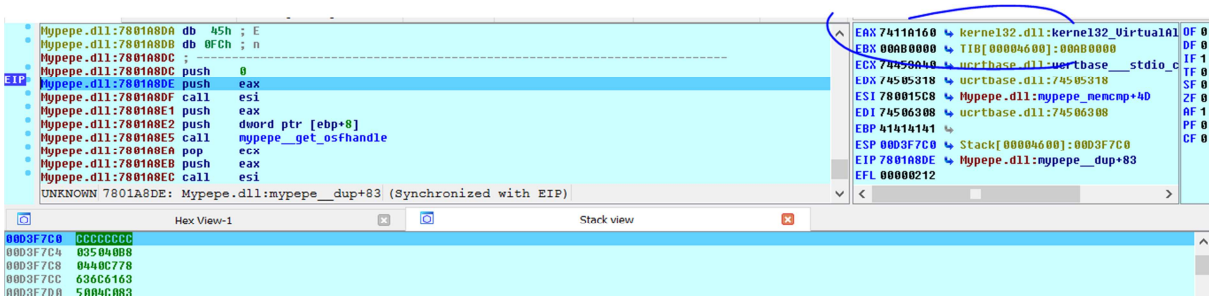


I will add the gadget to POP ESI-RET in ESI before.

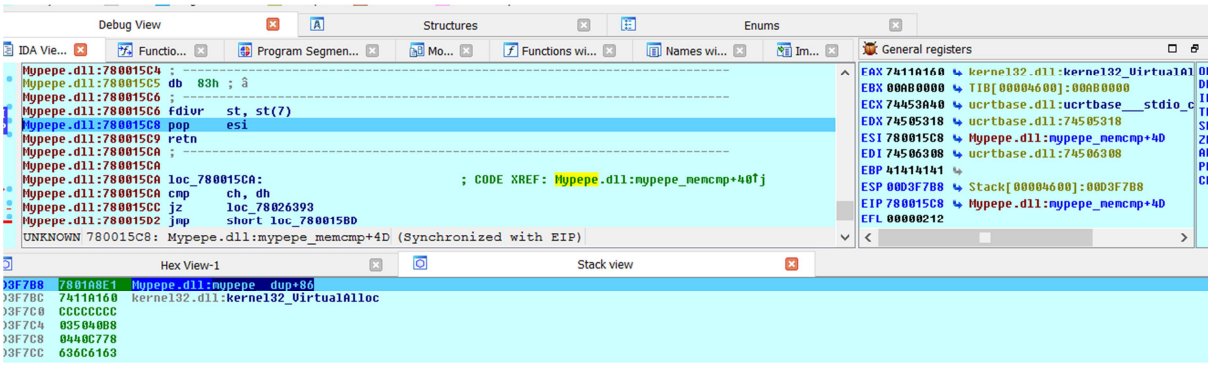




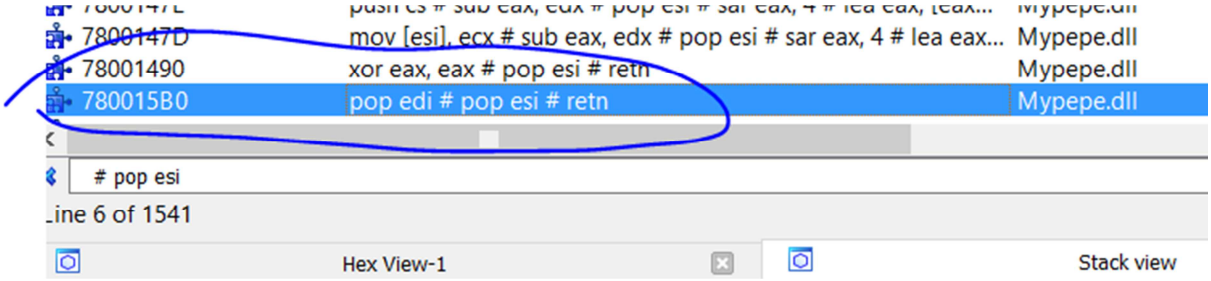
POP ESI moves the same pointer to POP ESI - RET to ESI to retake control after the next gadget.



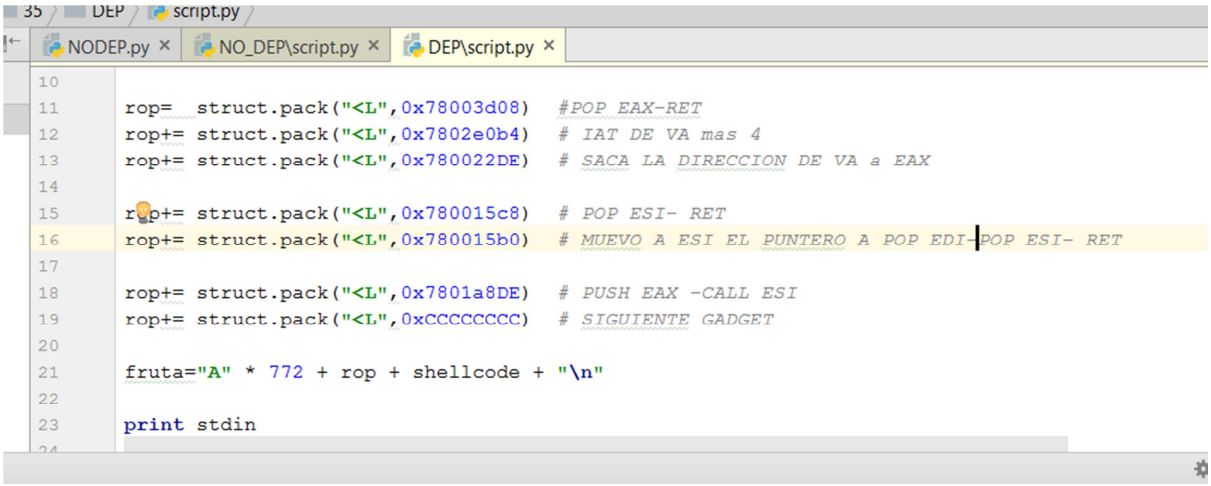
It will push the VA address to the stack and jumps to the POP ESI – RET with CALL ESI again because we had saved the POP ESI-RET address in ESI.



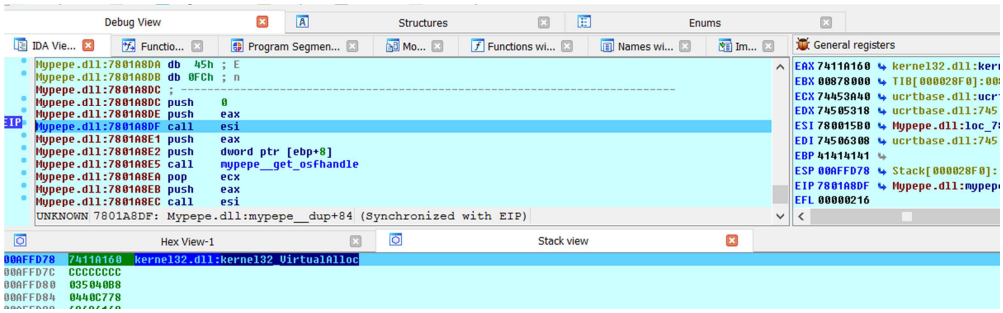
It fails because it moves the Return Address that it saved to ESI and below, it is the VA address. So, instead a POP ESI-RET, this last one should be a POP XXX, POP ESI – RET to take the first value of the stack out to other place and then, it pops the VA value to ESI.



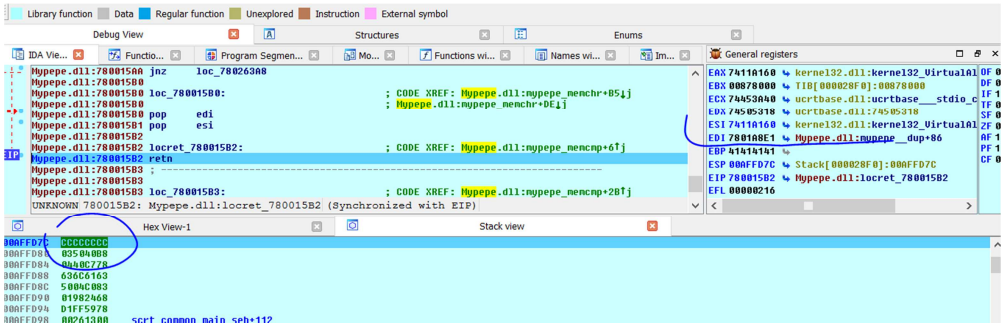
There, it is. So, I change just this that I move to ESI. The other should be the same.



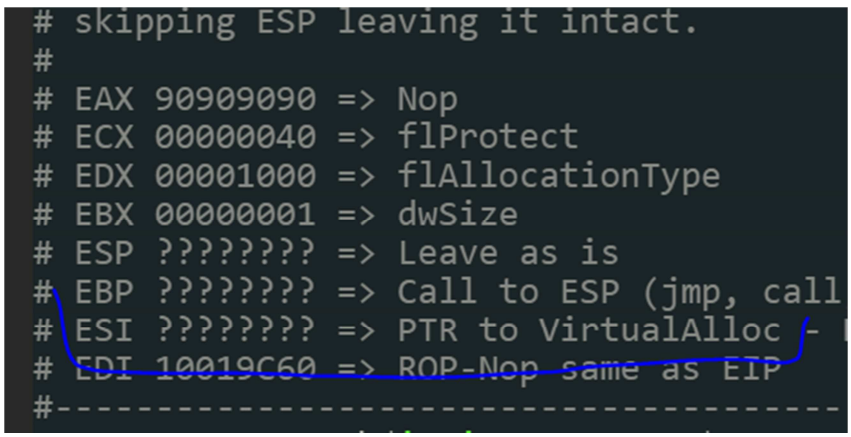
Let’s try it now.



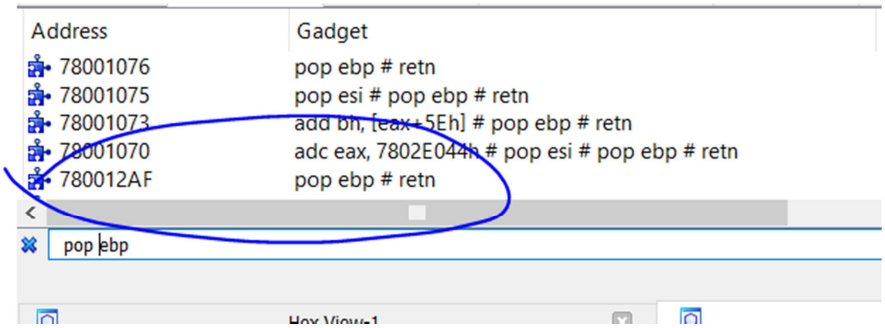
I enter it with F7.

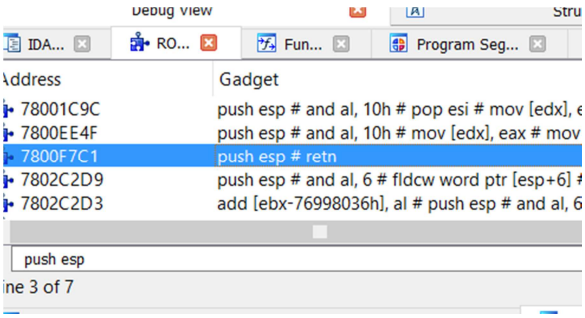


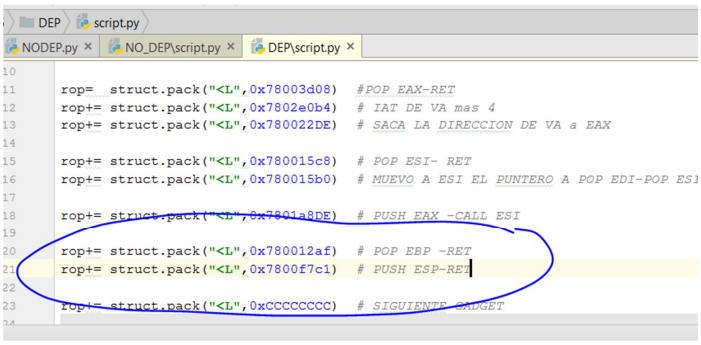
Ready. I got the first goal. The VA address is in ESI and it will jump to 0xCCCCCCCC that is the next gadget. So, I retook control.



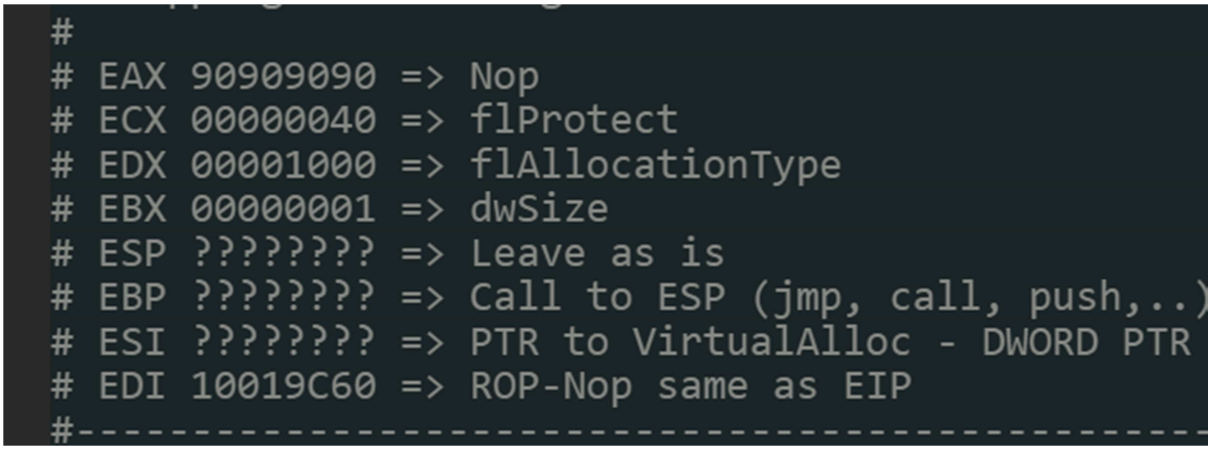
The next one will be easy. We have to set a JMP ESP. CALL ESP or PUSH ESP – RET in EBP. We already had a PUSH ESP-RET. We only have to find a POP EBP-RET to move it to EBP.



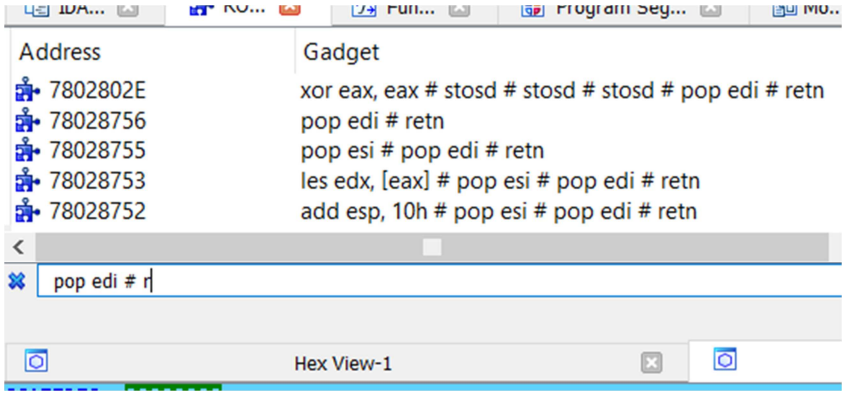


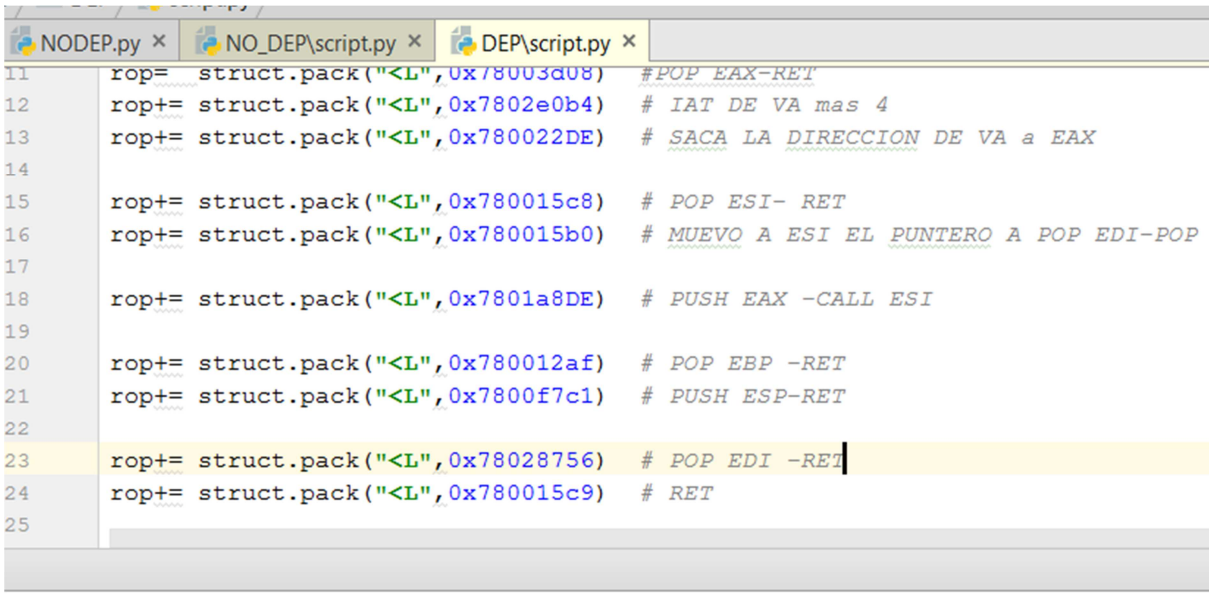


This is the EBP setting. With this, it will be set with its pointer to PUSH ESP-RET. Let's continue.

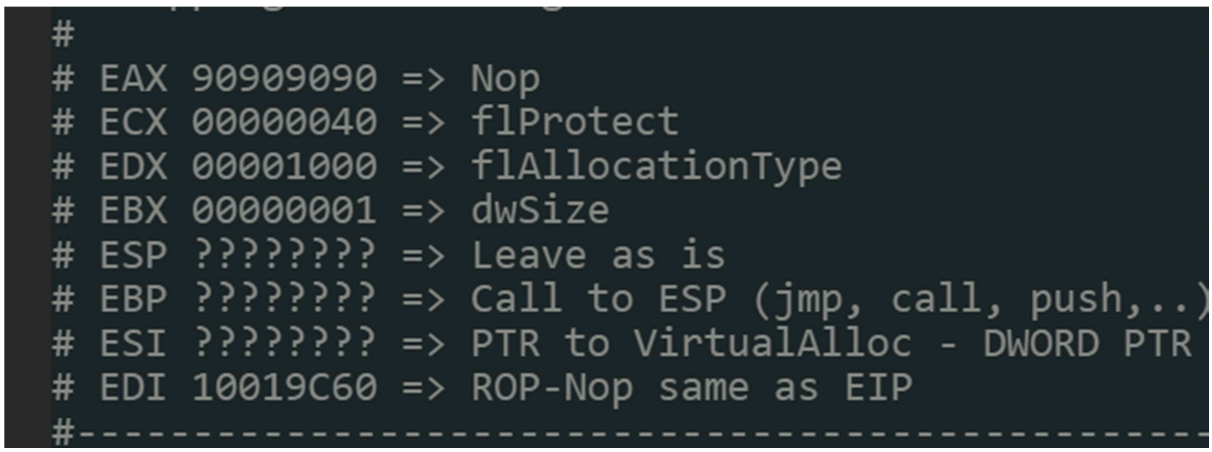


EDI = ROP NOP means it should point to a RET that is the NOP in ROP programming. So, let's find a POP EDI-RET to set EDI.

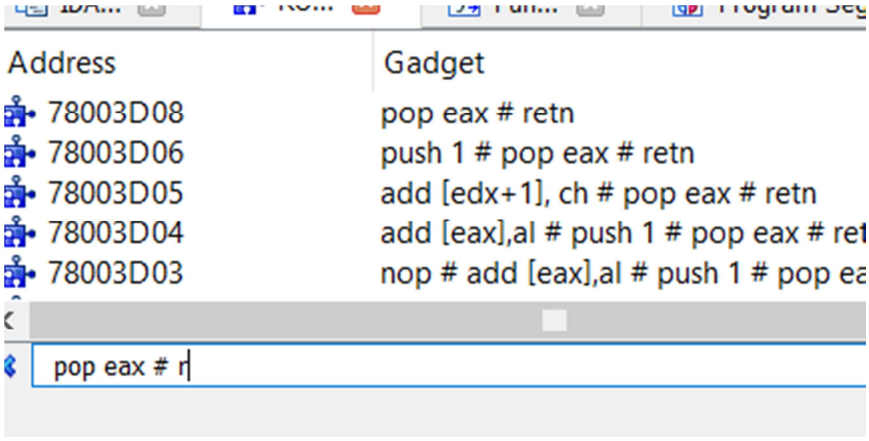


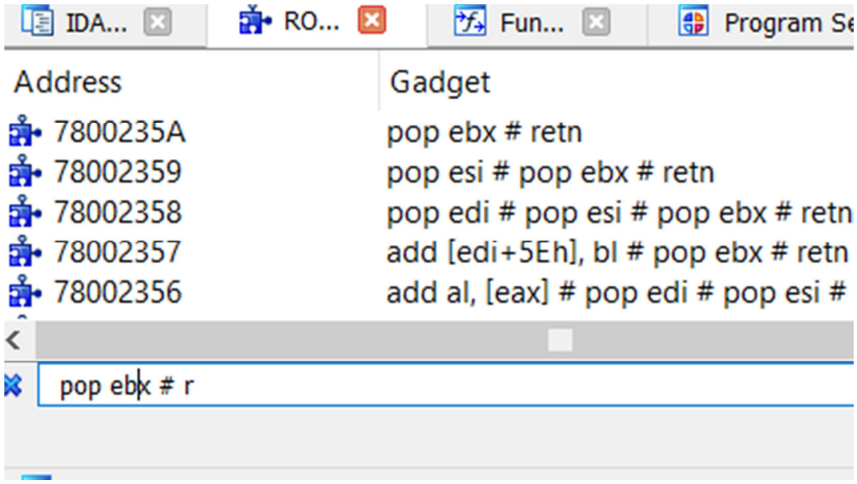


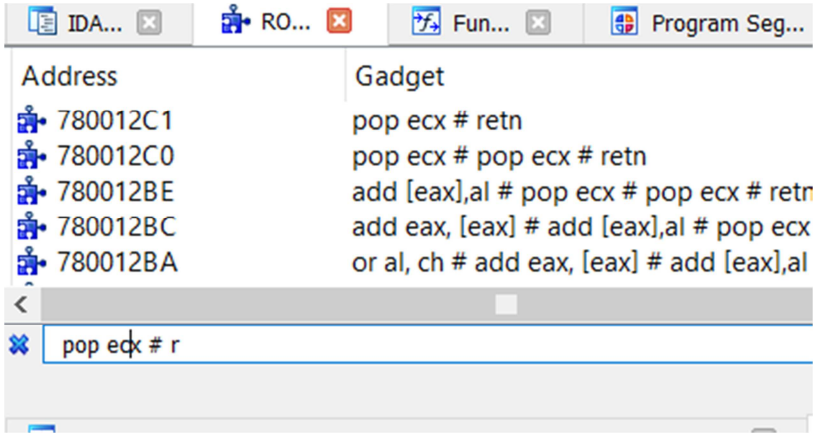
We set any pointer of mypepe to a RET. It will move it to EDI. Let's continue.

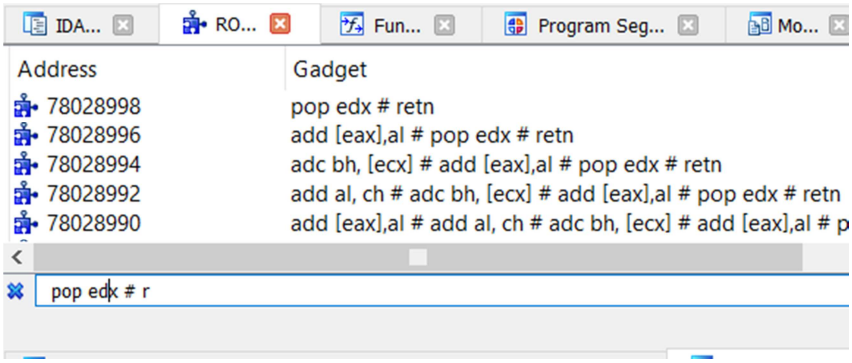


We need to move 4 constants to EAX, 40 to ECX, 1000 to EDX and 1 to EBX. We'll find the correspondent POPs and add the constants.

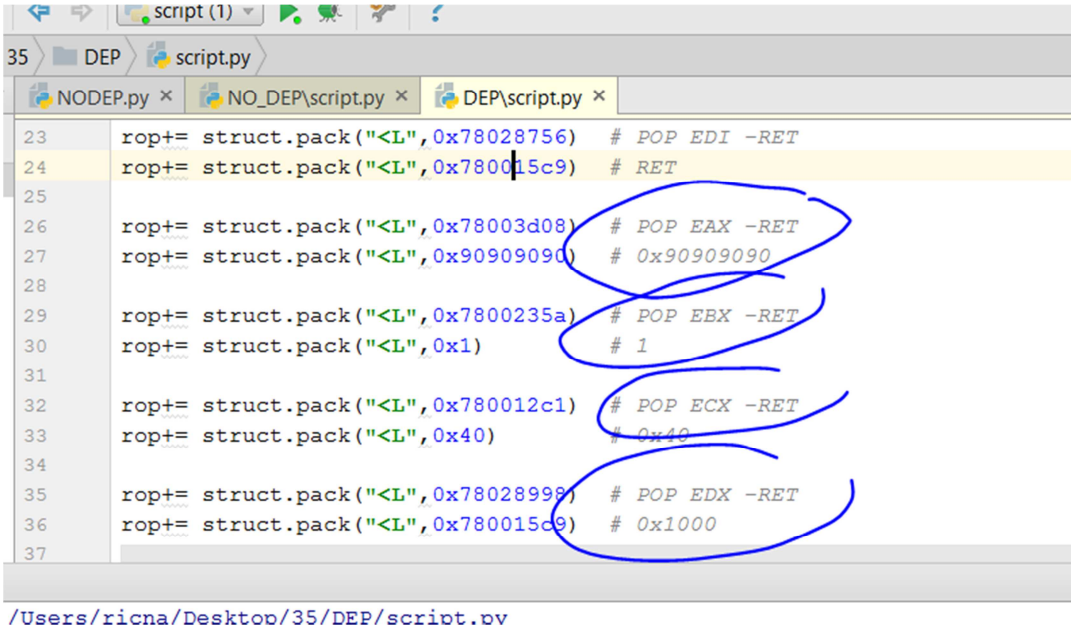




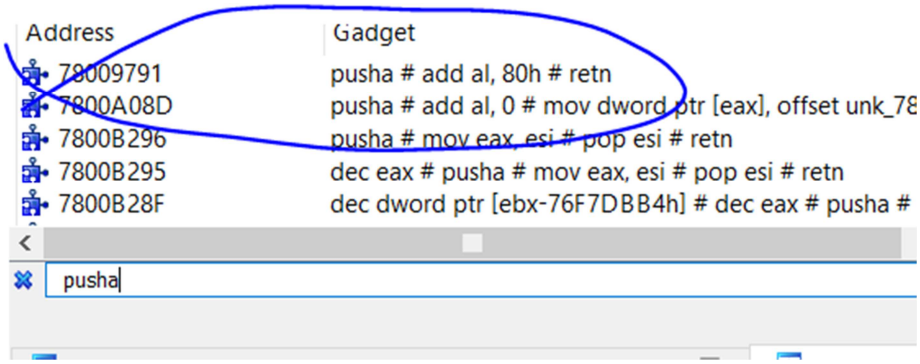




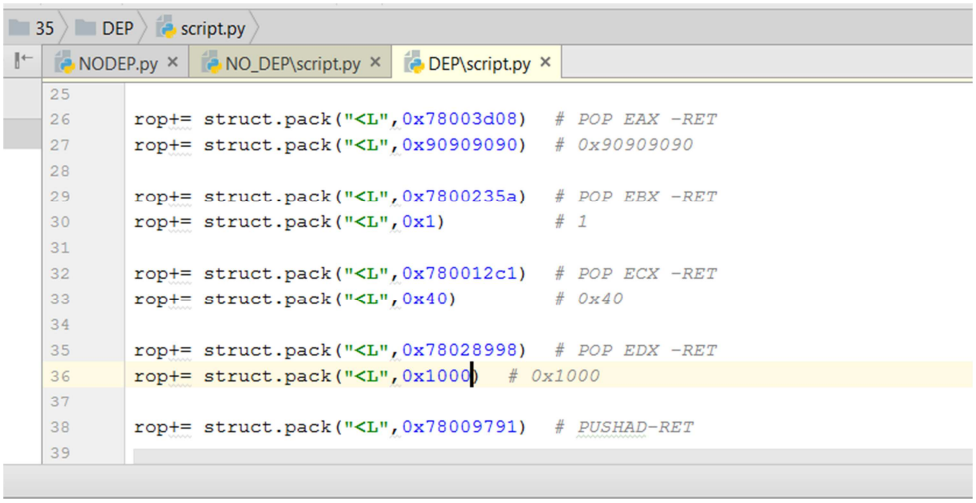
Let’s add them to the ROP.



Ready. We just need the final gadget that orders all. It is a PUSHAD-RET.

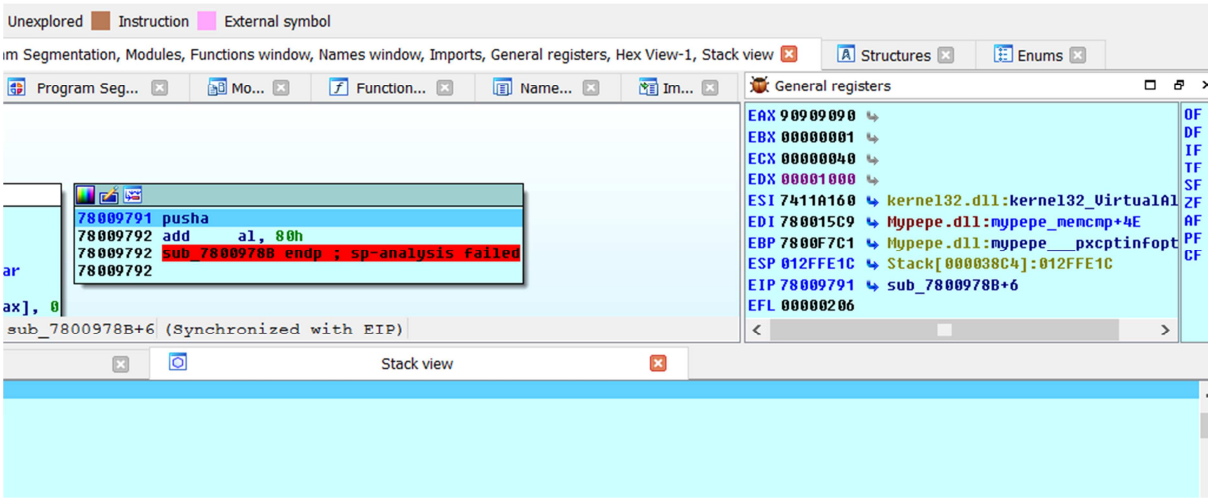


There, it is. The ADD AL, XX does nothing because the PUSHAD is done before. It will save the 90909090 in the stack. Let's add it.

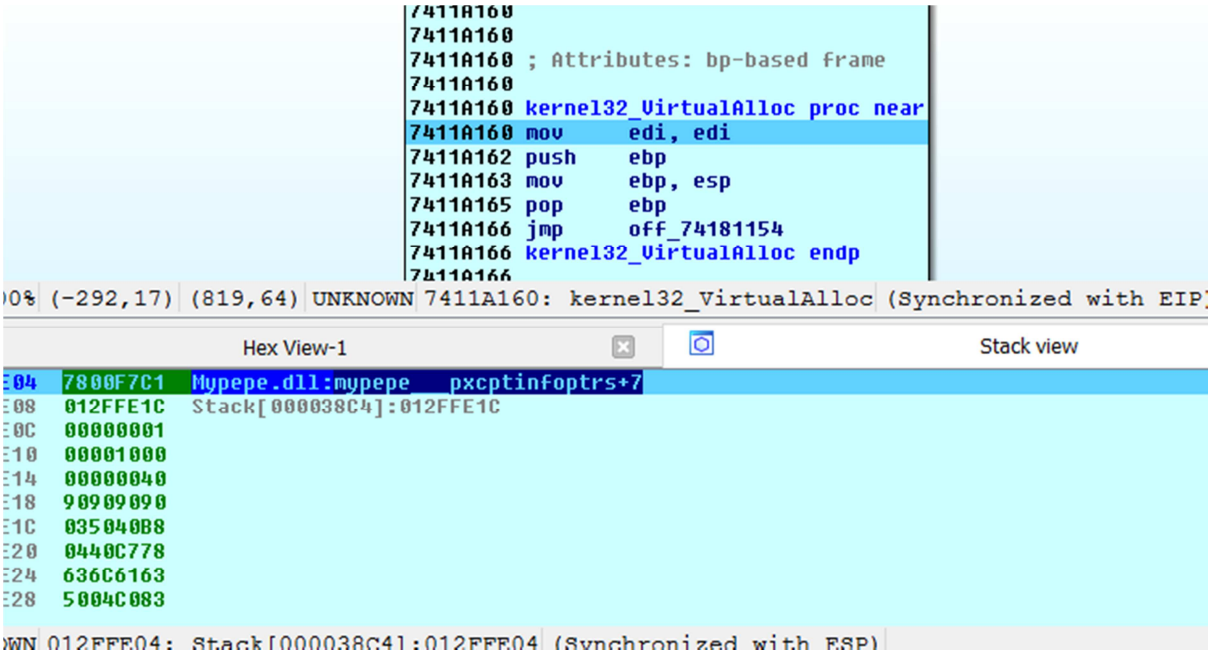


It should work with that. Let's trace it completely until VirtualAlloc.

We come to the PUSHAD.



It look good. Press F7.



We came to VirtualAlloc. We see the arguments in the stack. The first one will be the place where it will return from the API. If I see, it will be the PUSH ESP-RET. Then, we have the API arguments. Let's check.

VirtualAlloc function

Reserves, commits, or changes the state of a region of pages in the virtual address space of t
Memory allocated by this function is automatically initialized to zero.

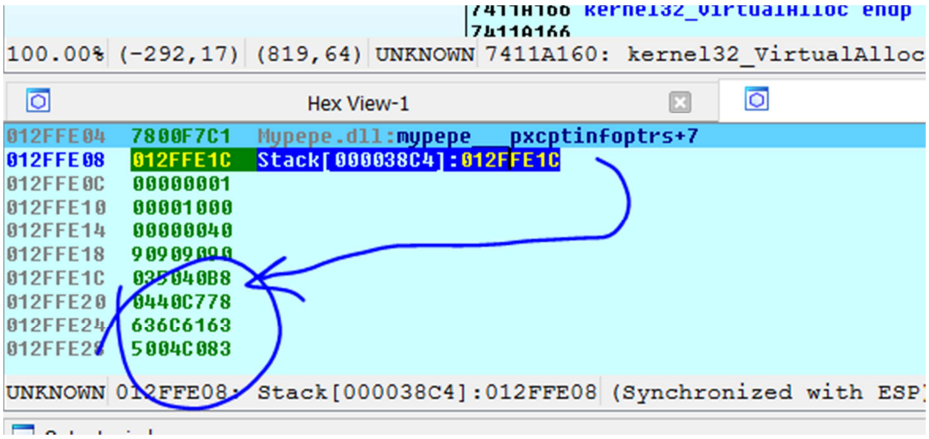
To allocate memory in the address space of another process, use the VirtualAllocEx function

Syntax

C++

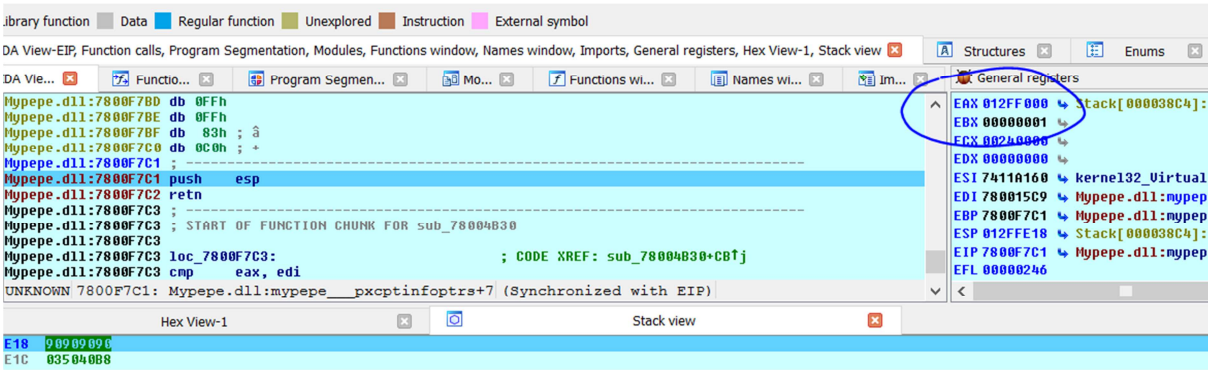
```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

The address to unprotect belongs to the stack and point just where my shellcode is.

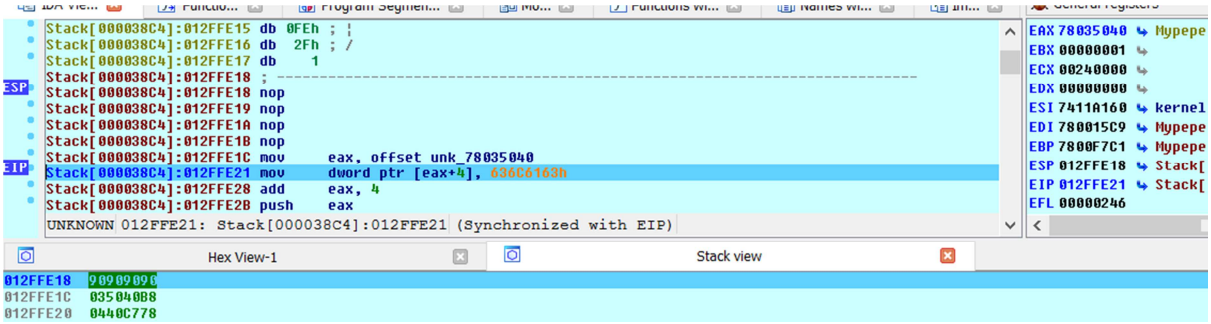


Then, we have the 1 size that will unprotect 0x1000 because this is the minimum block to unprotect. Independently of what you enter. Then, 0x1000 that is the allocation type constant and 0x40 that is the other **flprotect** constant.

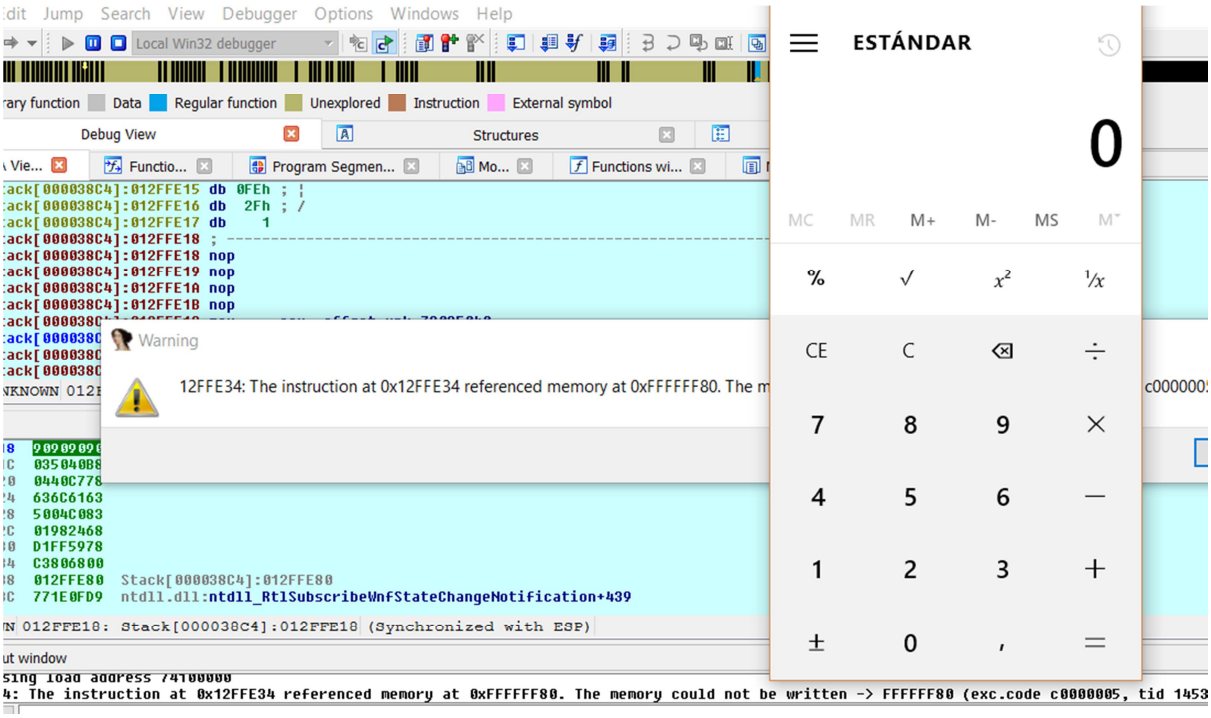
If I execute the API RET pressing CTRL + F7, I see that it returns to the PUSH ESP-RET.



If it returns an address in EAX, it is all correct and unprotected. I can execute my code now. I continue tracing with F7.



It came to my shellcode without problems and executed. (Alleluia)



And it ends up executing the calculator. This can be automatized with **mona**, but we'll see that in the next part.

WE GOT IT. ☺

Ricardo Narvaja

Translated by: @IvinsonCLS