PART 9

We are studying the use of the LOADER step by step. Later, we will see how the flags change with some instructions in a DEBUGGER.

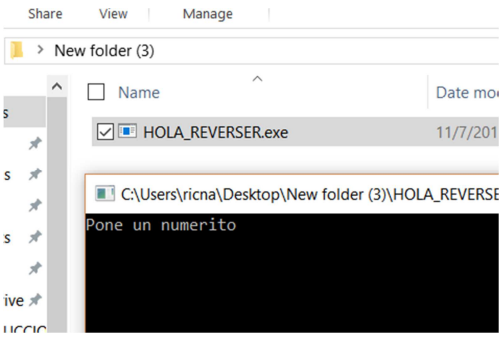
We'll see simple examples with a small crackme I compiled in VISUAL STUDIO 2015 to practice. We'll need the last version of VISUAL STUDIO 2015 C++ runtimes.

https://www.microsoft.com/es-ar/download/details.aspx?id=48145

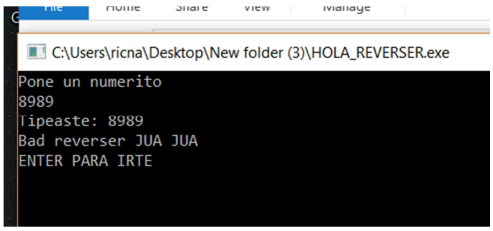
The Visual C++ Redistributable packages install the runtime components to execute compiled C++ Visual Studio 2015 programs.

Download your OS language and install it.

The HOLA_REVERSER.exe is included in the rar.

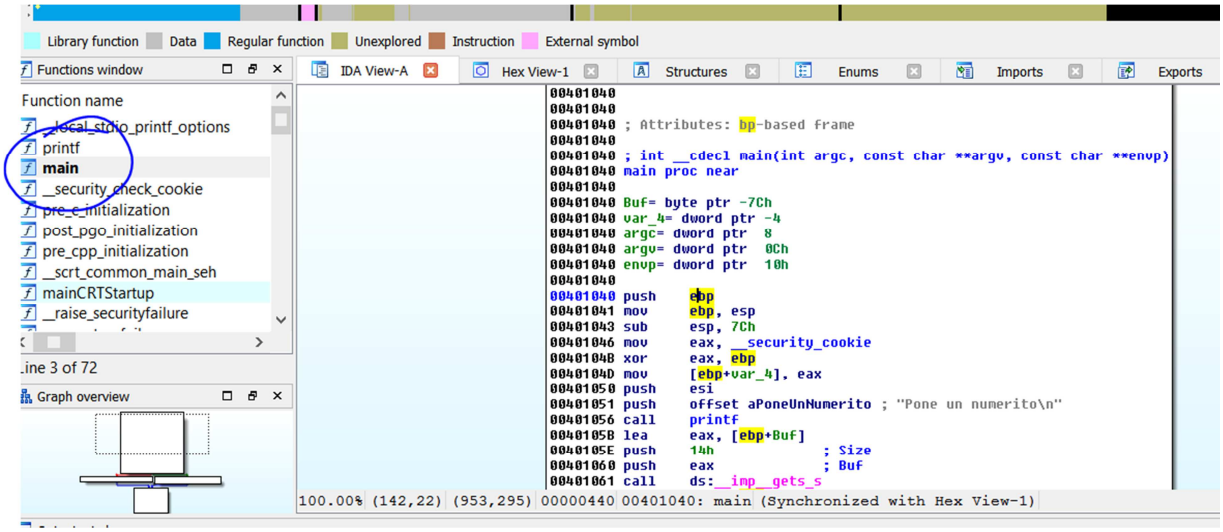


When entering a number it will say if I am a good or a bad reverser. ☺

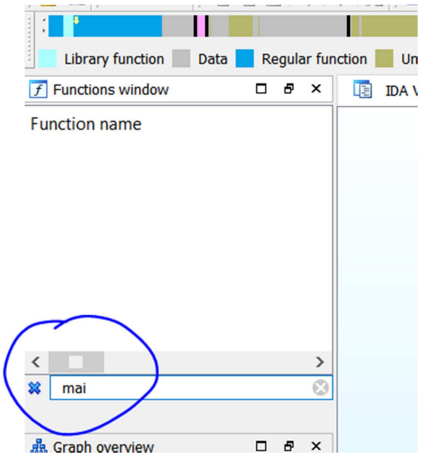


This is a super easy code. I will load it in IDA with just the LOADER.

In my case (not yours), the main function appears among the functions. I can find it with **CTRL + F** in that tab or where I have the function tab open. That happens because I compiled the program with VISUAL STUDIO and it creates a symbol pdb file which IDA detects and loads the function names and variables I used. We see that in mine as in the source code it says **main** and below it says **printf**. Let's see what happens in yours.

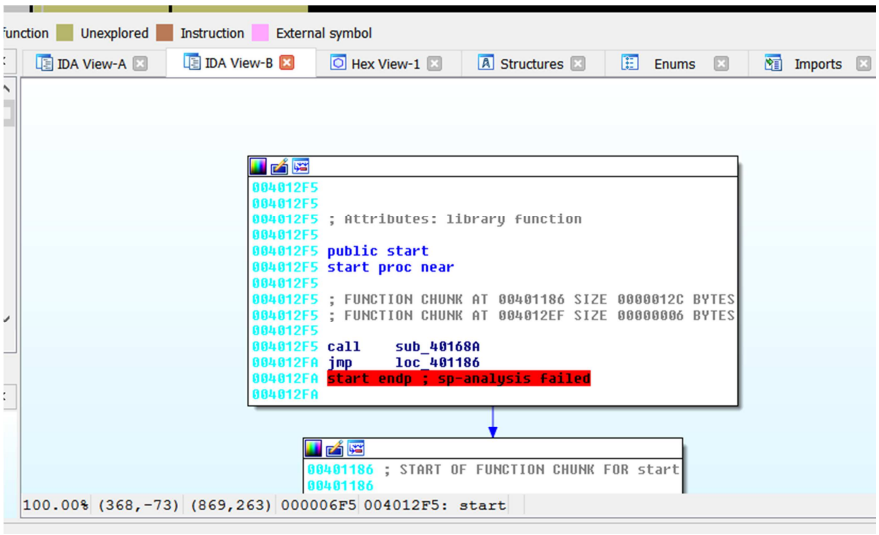


In yours, it doesn't say **main** because it doesn't have symbols. That's probable because nobody distributes a program with symbols.

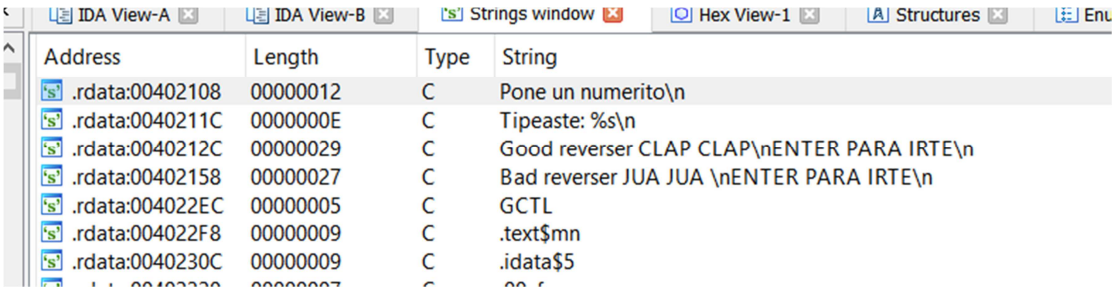


Normally, we will have symbols for system modules not for the owners of some program except in some cases. Here, I have the symbols because I am the owner. We will analyze it without them as a common program. ☺

We have it without symbols here.



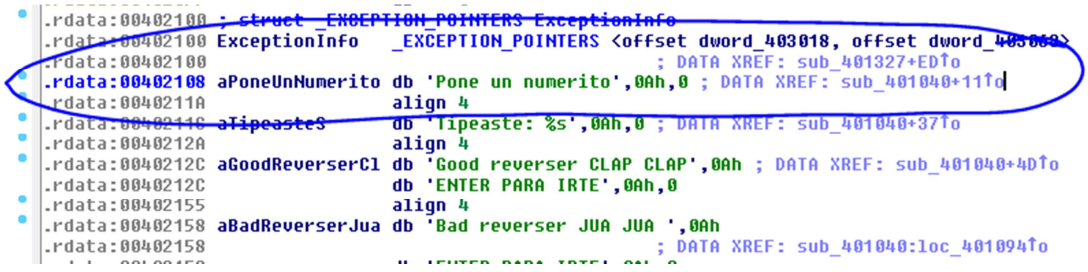
We don’t have much info, but we can see the strings.



We have the strings that it uses to tell us we are wrong when entering an invalid serial.

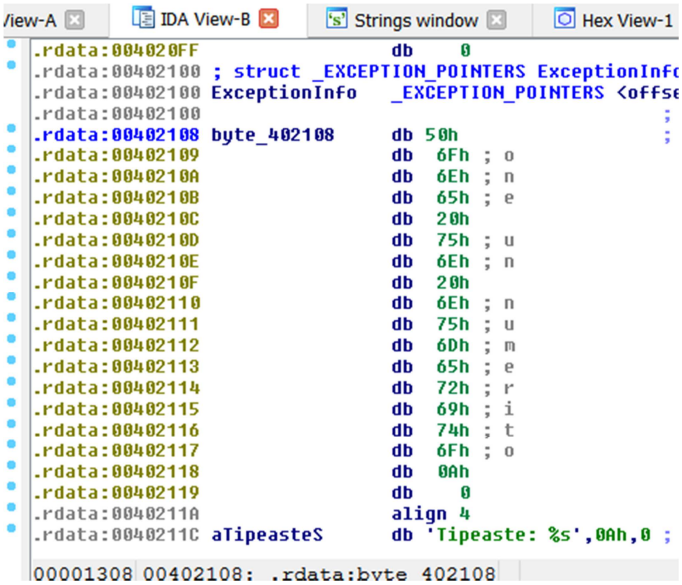
Double click on “Pone un numerito\n”

In English, it means “Enter a serial”.

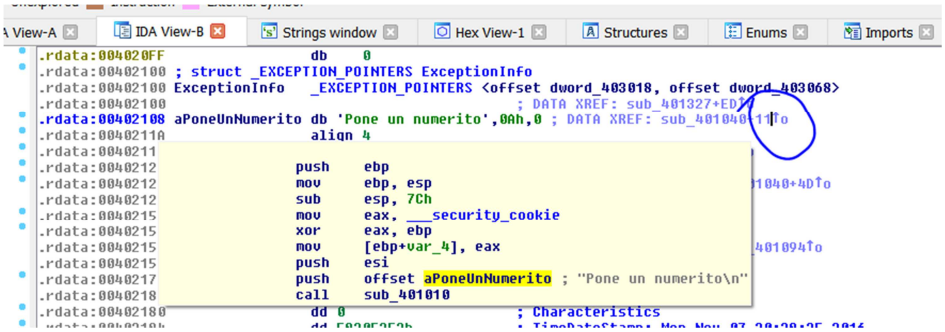


0x402108 is the pointer to the string. There is a tag next to the address that says “**a**” meaning it is an ASCII string and the rest belongs to the same string. We see **aPoneUnNumerito** then **db** because it is byte chain.

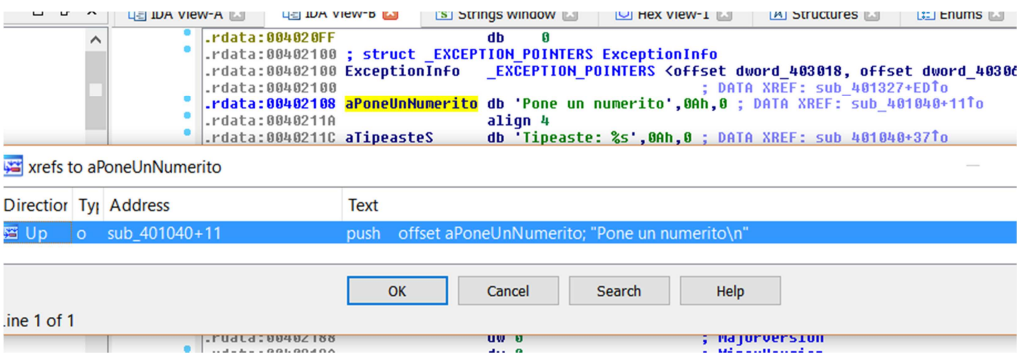
If we press **D**, we can see its bytes.



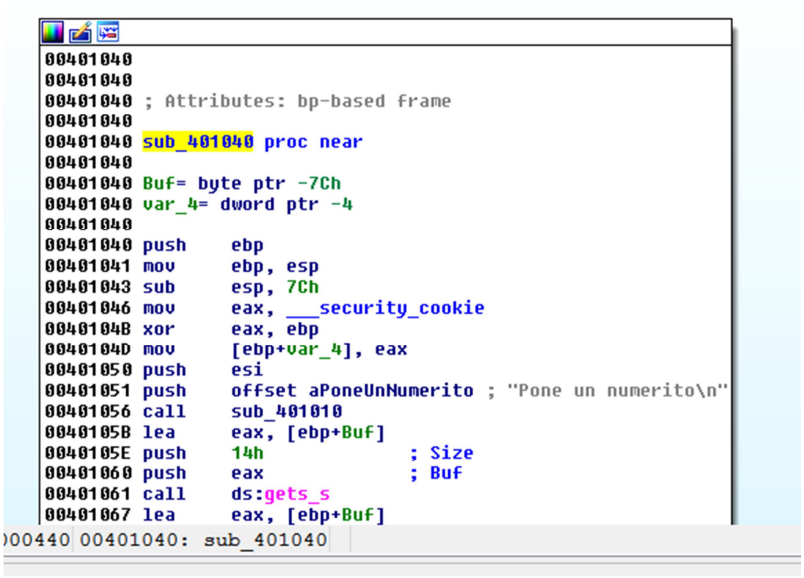
OK. Let’s press **A** to create the string again.



Placing the mouse on the reference little arrow, we see where it is called, but it’s better if we press **X** to see the reference list.



We are in the main function. Here, it is not called **main**, although the buffer name, that is generic, tagged it as **Buf**.

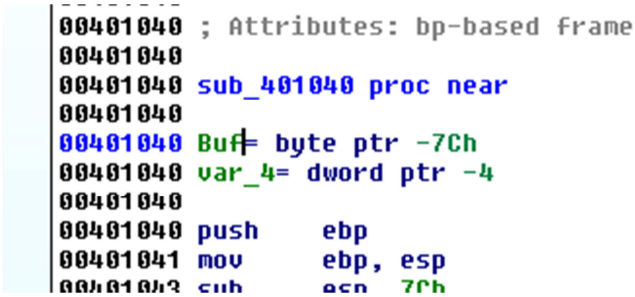


We are supposed not to know the source code, but if I see it…

```
int cookie;
char buf[120];
int max = 20;
printf("Pone un numerito\n");
```

I realize that it optimized some variables I created, for example, **cookie** and **max** that were replaced by constants and left just the buffer that in my code it was 120 bytes long in decimal.

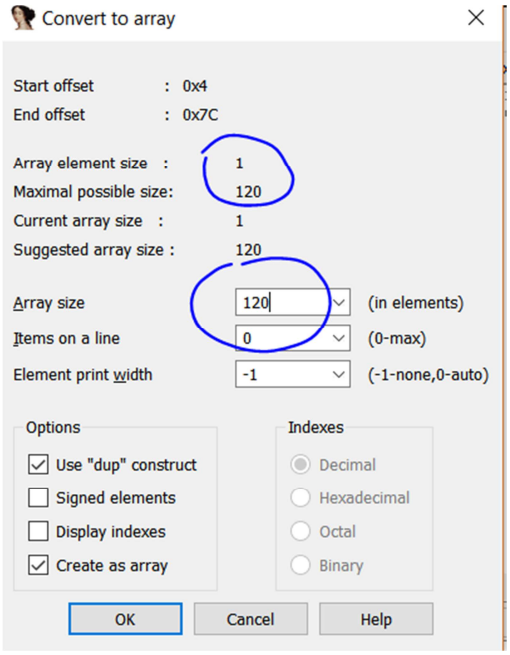
How can we know, in IDA, the size of a stack buffer without having the source code?



In the function top part, we see the variable and argument list. If we click on any of them, it will take us to the stack static view where we have the variable positions, buffers, arguments, etc. in a static way and the distance among them.

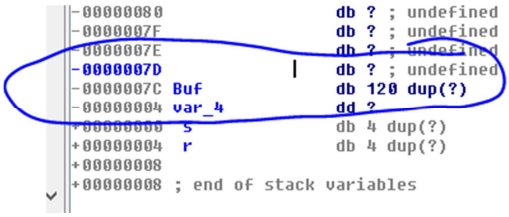
```
-00000080 ; Use data definition commands to create local variables and function arguments.
-00000080 ; Two special fields " r" and " s" represent return address and saved registers.
-00000080 ; Frame size: 80; Saved regs: 4; Purge: 0
-00000080 ;
-00000080
-00000080                 db ? ; undefined
-0000007F                 db ? ; undefined
-0000007E                 db ? ; undefined
-0000007D                 db ? ; undefined
-0000007C Buf             db ?
-0000007B                 db ? ; undefined
-0000007A                 db ? ; undefined
-00000079                 db ? ; undefined
-00000078                 db ? ; undefined
-00000077                 db ? ; undefined
-00000076                 db ? ; undefined
-00000075                 db ? ; undefined
-00000074                 db ? ; undefined
-00000073                 db ? ; undefined
-00000072                 db ? ; undefined
-00000071                 db ? ; undefined
-00000070                 db ? ; undefined
```

There, we have **Buf**, but it is defined as byte **db.** To change it into a character array or buffer, right click on the word **Buf** and select the **Array** option.



There, we see that until the next variable or whatever it is below on the stack it detects a size of 120 decimal and the array element is the size of each field as it is 1 it is a character array or bytes and its size is 120*1 or just 120.

If I accept…



I see that the 120-byte buffer matches my source code, although the compiler could do it bigger while it is 120 as minimum it is OK, in this case, they are the same. DUP means duplicate (it should be multiply) 120 times the character? As the value is not defined yet, it is a static empty buffer.

```
int cookie;
char buf[120];
int max = 20;
printf("Pone un numerito\n");
```

I will clarify the static stack view later, but below **Buf** there is a dword (dw) variable called **var_4**.

S and **R** are the saved EBP of the parent function which called it and the RETURN ADRESS as we saw when entering a function, the arguments were pushed first, then it used a CALL to enter the function that saved the RETURN ADRESS and the variables were above **S**.

VARIABLES

…

…

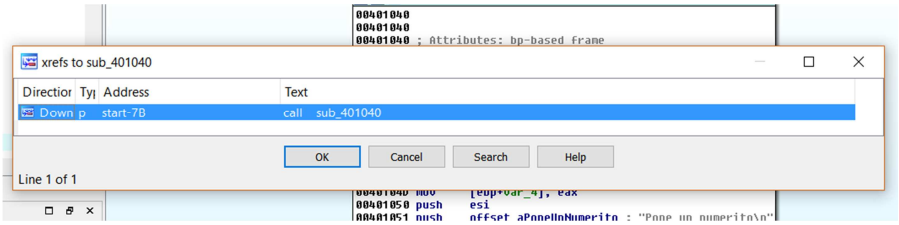
S (stored ebp -it generally comes from PUSH EBP that is the first instruction of the function)

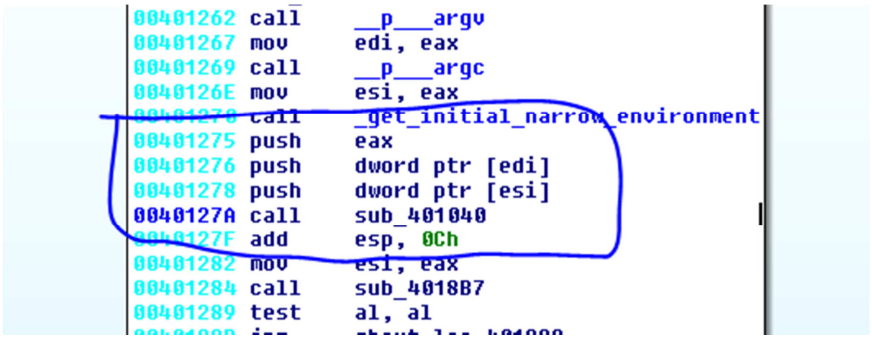
R (return address)

ARGUMENTS

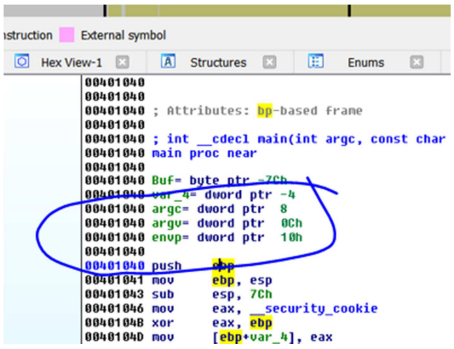
As the arguments are pushed on the stack before placing the return address, they will be below it. Then, above the return address comes the STORED EBP generated by the PUSH EBP that is generally the first function instruction and there is space above for the local variables. We will see that later.

If I press X to see where that function is called from…





And I go there, I see that there are some **PUSH**es that send arguments to the function that I saw when it worked with symbols and here, they're not. 3 arguments are missing below. Here is the image using symbols.

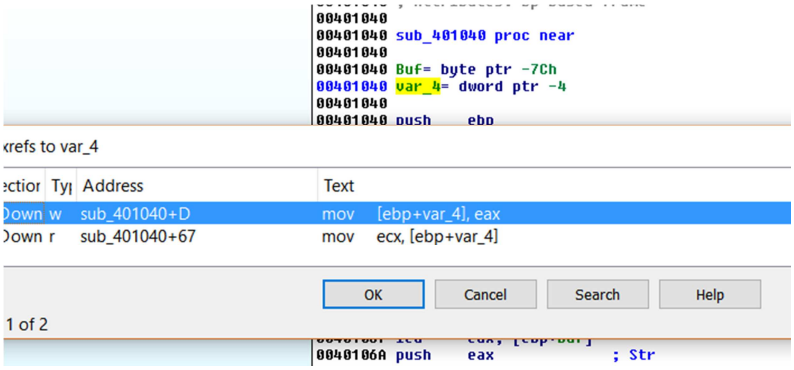


IDA detected that they are never used. Those arguments are: **argc**, **argv** and **envp**. They are by default in the main, but as there was never any reference nor use in the function, IDA deleted them to clear the panorama.

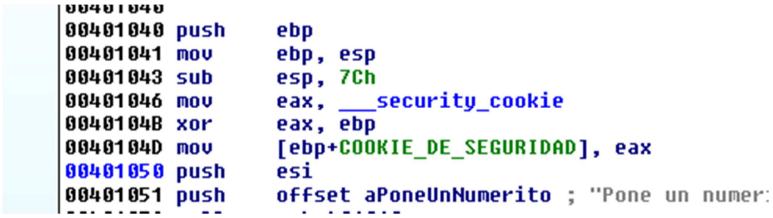
```
int main()
{

        int cookie;
        char buf[120];
        int max = 20;
        printf("Pone un numerito\n");
```

Besides, in my code they weren't taken into account. So that, everything's OK. It is a non-argument function when it has symbols and it sees that's the main. It uses them because the default PUSHes need them, but for a common function and when it doesn't know that's the main, it clears them because it never uses them.

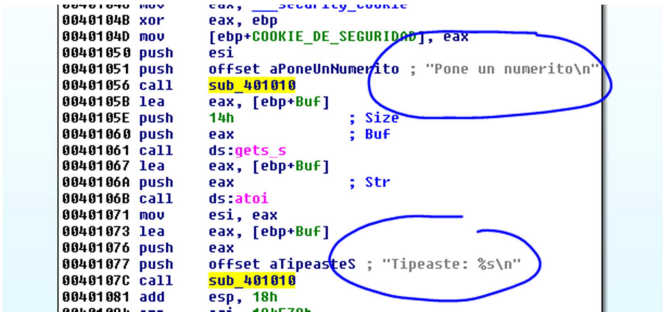
When we want to see where a variable is called from we mark it and press X.



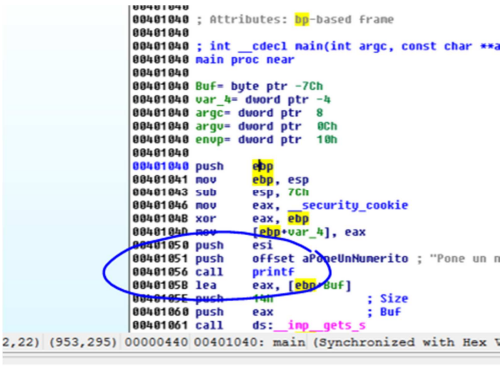
We see that **var_4** is used in two places. For those who don't know, that is cookie I didn't program. It's for stack overflow protection. It saves it in the function start and checks it before quitting the function. By now, we rename it as COOKIE_DE_SEGURIDAD or SECURITY_COOKIE.



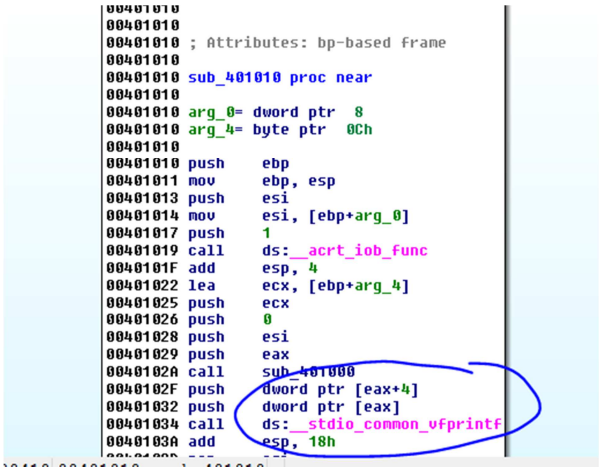
When it prints the strings, it invokes a CALL that I didn't program, but maybe it ends up going to **printf** to print the strings.



In the program with symbols, we see that it detected it directly as **printf**.



Anyways, if we see inside the CALL and we know that the arguments are strings that are printed in a console we deduce that is **printf**.



Inside that function, we see that it ends in **vfprintf**, so that, we rename it.

```
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, ___security_cookie
0040104B xor     eax, ebp
0040104D mov     [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un
00401056 call    printf_0
0040105B lea     eax, [ebp+Buf]
0040105E push    14h             ; Size
00401060 push    eax             ; Buf
00401061 call    ds:gets_s
00401067 lea     eax, [ebp+Buf]
0040106A push    eax             ; Str
0040106B call    ds:atoi
00401071 mov     esi, eax
00401073 lea     eax, [ebp+Buf]
00401076 push    eax
00401077 push    offset aTipeasteS ; "Tipeaste: %s\
0040107C call    printf_0
```

Just the 120-byte buffer is left. Let's analyze what it does with it.

Let's see what are the arguments for **get_s** that is function that receives what we type in a console.

gets_s, _getws_s

Visual Studio 2015 | Other Versions

Gets a line from the stdin stream. These versions of gets, _getws have security described in Security Features in the CRT.

Syntax

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

It has two arguments. A pointer to the buffer and the max size that lets us type.

```
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "P
00401056 call    printf_0
0040105B lea     eax, [ebp+Buf]
0040105E push    14h             ; Size
00401060 push    eax             ; Buf
00401061 call    ds:gets_s
00401067 lea     eax, [ebp+Buf]
0040106A push    eax             ; Str
0040106B call    ds:atoi
```

LEA finds a variable address, in this case, it is the pointer to the buffer called **Buf** pushed by **PUSH EAX** and then, with **PUSH 0x14** it indicates the max characters to be typed.

```
int cookie;
char buf[120];
int max = 20;
printf("Pone un numerito\n");

gets_s(buf,max);
```

In my source code, I have the same **get_s** with two arguments: the **Buf** and the maximum that I had used a variable called **max** = 20, but the compiler, to save space, used 20 decimal that is 0x14 in hex in that argument because it won't use it anymore.

Without executing it, I know that in the buffer, I'll have the characters I type in a console.

Then, the same pointer is pushed as an argument to the **atoi** function.

```
00401067 lea     eax, [ebp+Buf]
0040106A push    eax             ; Str
0040106B call    ds:atoi
00401071 mov     esi, eax
```

What does it do?

atoi, _atoi_l, _wtoi, _wtoi_l

Visual Studio 2015 | Otras versiones

Publicada: julio de 2016

Convierte una cadena en un entero.

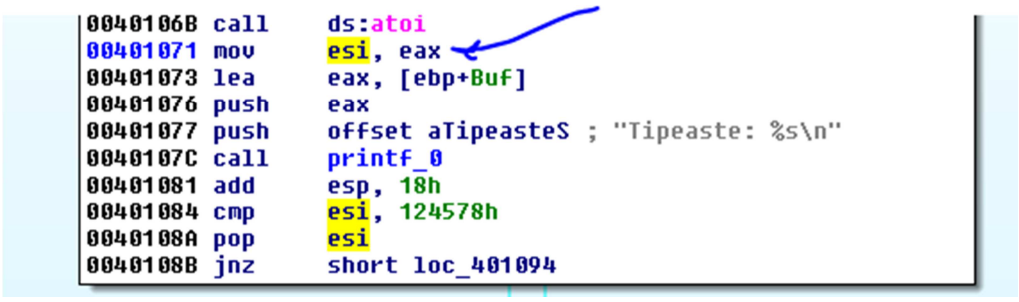
Sintaxis

```
int atoi(
   const char *str
);
```

It converts the string into an integer and if it can't because of an overflow, it produces an error returning 0. The same thing happens when it exceeds the maximum negative. The idea is that all you type will be converted into a number. If you type 41424344, it will convert it into hex in EAX.

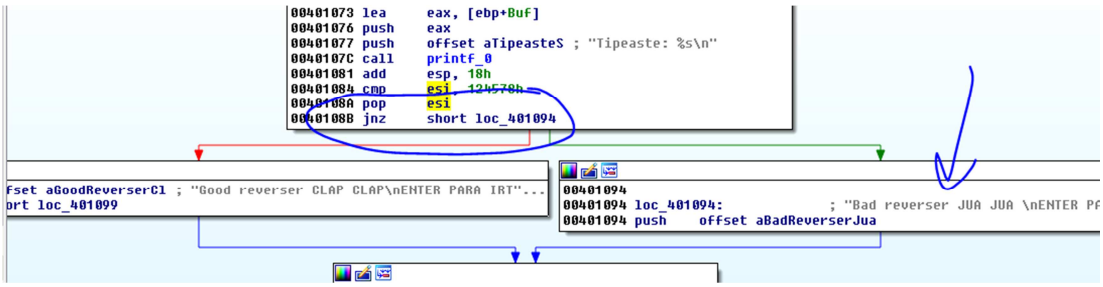
Return Value

Each function returns the int value generated. It interprets the input characters as a number. It returns **0** for **atoi** and **_wtoi** if the input cannot be converted into a value of that type.

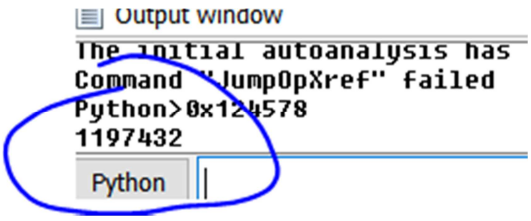


The return value in EAX is moved to ESI and after printing the original string that was typed, it compares the value to 0x124578.

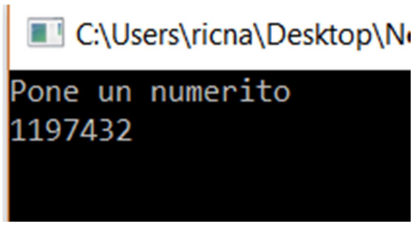
As what we type is interpreted as a decimal number string returned as hex and it is compared to that constant, giving it that constant decimal value it should work because that comparison is valid…

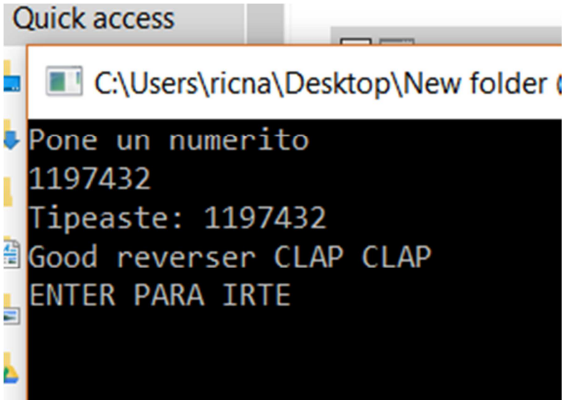


If that comparison is not the equal or zero (JNZ), it takes me to BAD REVERSER, but if they equal, we go to GOOD REVERSER. Let's try the Python console to see the decimal value of 0x124578.



Let's try that serial in the crackme.





This was a very simple example of static reversing to familiarize with the LOADER.

Ricardo Narvaja

Translated by: @IvinsonCLS