

REVERSING WITH IDA PRO FROM SCRATCH

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.

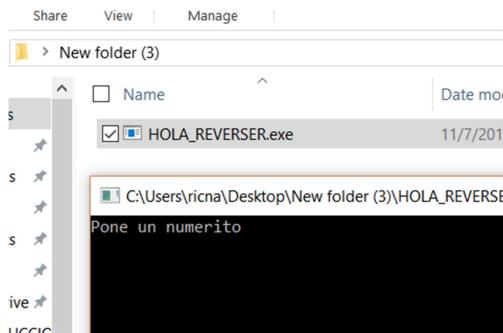
We'll see simple examples with a small crackme I compiled in VISUAL STUDIO 2015 to practice. We'll need the last version of VISUAL STUDIO 2015 C++ runtimes.

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

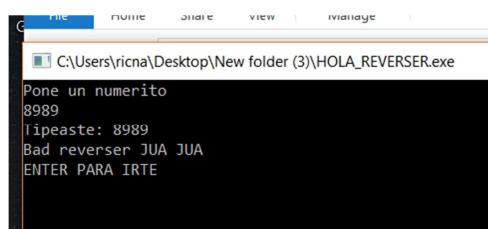
The Visual C++ Redistributable packages install the runtime components to execute compiled C++ Visual Studio 2015 programs.

Download your OS language and install it.

The HOLA_REVERSER.exe is included in the rar.

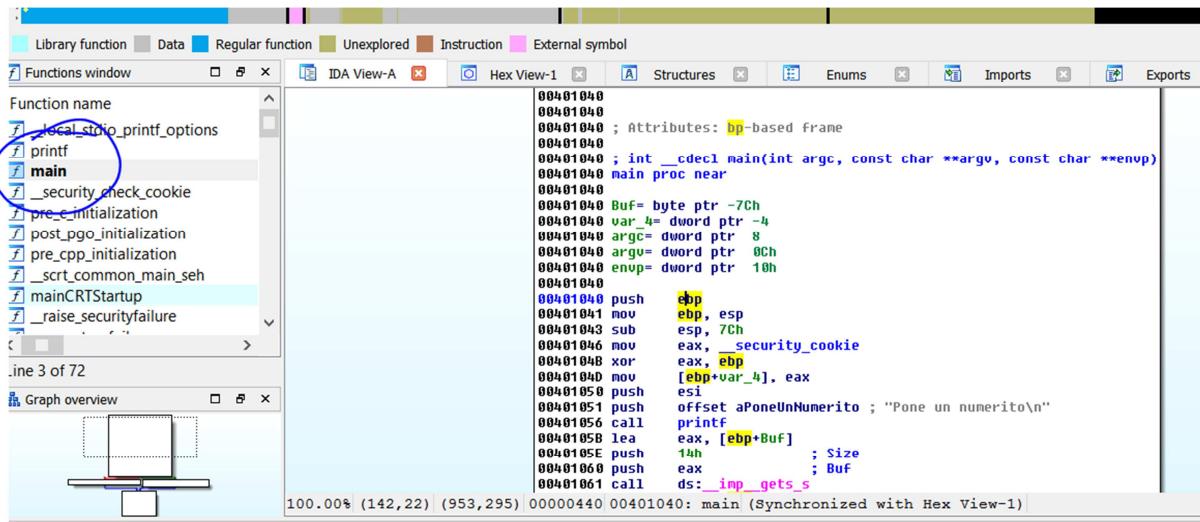


When entering a number it will say if I am a good or a bad reverser. ☺



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.



The screenshot shows the IDA Pro interface with the 'Functions' tab selected. In the left pane, a list of function names is shown, with 'main' highlighted and circled in blue. The right pane displays the assembly code for the 'main' function:

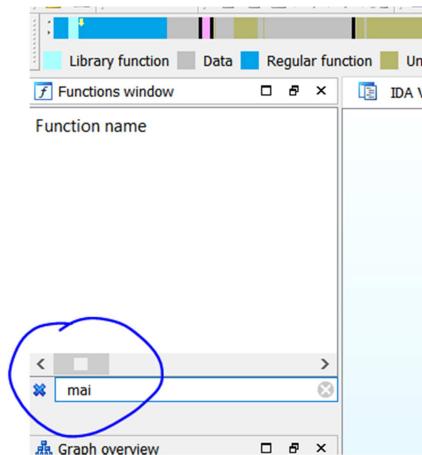
```

00401040
00401040 ; Attributes: bp-based Frame
00401040 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401040 main proc near
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 argc= dword ptr 8
00401040 argv= dword ptr 0Ch
00401040 envp= dword ptr 10h
00401040
00401040 push ebp
00401041 mov ebp, esp
00401043 sub esp, 7Ch
00401046 mov eax, _security_cookie
00401048 xor eax, ebp
0040104a mov [ebp+var_4], eax
00401050 push offset aPoneUnNumerito ; "Pone un numerito\n"
00401051 call printf
00401056 lea eax, [ebp+Buf]
00401058 push 14h ; size
00401060 push eax ; Buf
00401061 call ds:_imp__gets_s

```

At the bottom, the status bar indicates: 100.00% (142,22) (953,295) 00000440 00401040: main (Synchronized with Hex View-1)

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.

The screenshot shows the assembly view in IDA Pro. The assembly code for the 'start' function is displayed in a large window. The code includes instructions like 'call sub_40168A', 'jmp loc_401186', and 'start endp'. A red box highlights the instruction 'start endp ; sp-analysis Failed'. Below this window, a smaller window shows the start of the function chunk at address 00401186, labeled '00401186 ; START OF FUNCTION CHUNK FOR start'. The status bar at the bottom indicates 100.00% completion and file paths.

```
004012F5
004012F5
004012F5 ; Attributes: library function
004012F5
004012F5 public start
004012F5 start proc near
004012F5
004012F5 ; FUNCTION CHUNK AT 00401186 SIZE 0000012C BYTES
004012F5 ; FUNCTION CHUNK AT 004012EF SIZE 00000006 BYTES
004012F5
004012F5 call    sub_40168A
004012F8 jmp     loc_401186
004012F8 start endp ; sp-analysis Failed
004012F8
```

00401186 ; START OF FUNCTION CHUNK FOR start

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

The screenshot shows the 'Strings window' in IDA Pro. It lists several strings found in memory, each with its address, length, type, and string content. The strings include prompts for entering serials and error messages for invalid entries.

Address	Length	Type	String
.rdata:00402108	00000012	C	Pone un numerito\n
.rdata:0040211C	0000000E	C	Tipeaste: %s\n
.rdata:0040212C	00000029	C	Good reverser CLAP CLAP\nENTER PARA IRTE\n
.rdata:00402158	00000027	C	Bad reverser JUA JUA \nENTER PARA IRTE\n
.rdata:004022EC	00000005	C	GCTL
.rdata:004022F8	00000009	C	.text\$mn
.rdata:0040230C	00000009	C	.idata\$5
		C	...

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".

The screenshot shows the 'Registers window' in IDA Pro. A specific string entry at address 00402108 is highlighted with a blue oval. The string is 'Pone un numerito', followed by a carriage return and a newline character. This is likely the serial input field for the program.

```
.rdata:00402108 : struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402108 ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_405000>
.rdata:00402108
.rdata:00402108 aPoneUnNumerito db 'Pone un numerito',0Ah,0 ; DATA XREF: sub_401327+EDt0
.rdata:00402110 align 4
.rdata:00402110
.rdata:00402110 aTipeaste$ db 'Tipeaste: %s',0Ah,0 ; DATA XREF: sub_401040+37t0
.rdata:00402110 align 4
.rdata:00402120 aGoodReverserCl db 'Good reverser CLAP CLAP',0Ah ; DATA XREF: sub_401040+4Dt0
.rdata:00402120 db 'ENTER PARA IRTE',0Ah,0
.rdata:00402155 align 4
.rdata:00402158 aBadReverserJua db 'Bad reverser JUA JUA ',0Ah
.rdata:00402158 ; DATA XREF: sub_401040:loc_401094t0
        .text$mn
        .idata$5
```

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.

```

View-A IDA View-B Strings window Hex View-1
.rdata:004020FF db 0
.rdata:00402100 ; struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402100 ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_403068>
.rdata:00402100 ;
.rdata:00402108 byte_402108 db 50h ;
.rdata:00402109 db 6Fh ; o
.rdata:0040210A db 6Eh ; n
.rdata:0040210B db 65h ; e
.rdata:0040210C db 20h
.rdata:0040210D db 75h ; u
.rdata:0040210E db 6Eh ; n
.rdata:0040210F db 20h
.rdata:00402110 db 6Eh ; n
.rdata:00402111 db 75h ; u
.rdata:00402112 db 60h ; m
.rdata:00402113 db 65h ; e
.rdata:00402114 db 72h ; r
.rdata:00402115 db 69h ; i
.rdata:00402116 db 74h ; t
.rdata:00402117 db 6Fh ; o
.rdata:00402118 db 0Ah
.rdata:00402119 db 0
.rdata:0040211A align 4
.rdata:0040211C aTipeasteS db 'Tipeaste: %s',0Ah,0 ;
00001308 00402108: .rdata:bvte 402108

```

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

```

View-A IDA View-B Strings window Hex View-1 Structures Enums Imports
.rdata:004020FF db 0
.rdata:00402100 ; struct _EXCEPTION_POINTERS ExceptionInfo
.rdata:00402100 ExceptionInfo _EXCEPTION_POINTERS <offset dword_403018, offset dword_403068>
.rdata:00402100 ;
.rdata:00402108 aPoneUnNumerito db 'Pone un numerito',0Ah,0 ; DATA XREF: sub_401327+EDt
.rdata:00402108 align 4
.rdata:00402111
.rdata:00402112 push ebp
.rdata:00402112 mov ebp, esp
.rdata:00402112 sub esp, 7Ch
.rdata:00402115 mov eax, __security_cookie
.rdata:00402115 xor eax, ebp
.rdata:00402115 mov [ebp+var_4], eax
.rdata:00402115 push esi
.rdata:00402117 push offset aPoneUnNumerito ; "Pone un numerito\n"
.rdata:00402117 call sub_401010
.rdata:00402118 dd 0 ; Characteristics
.rdata:00402118 dd 0 ; Characteristics

```

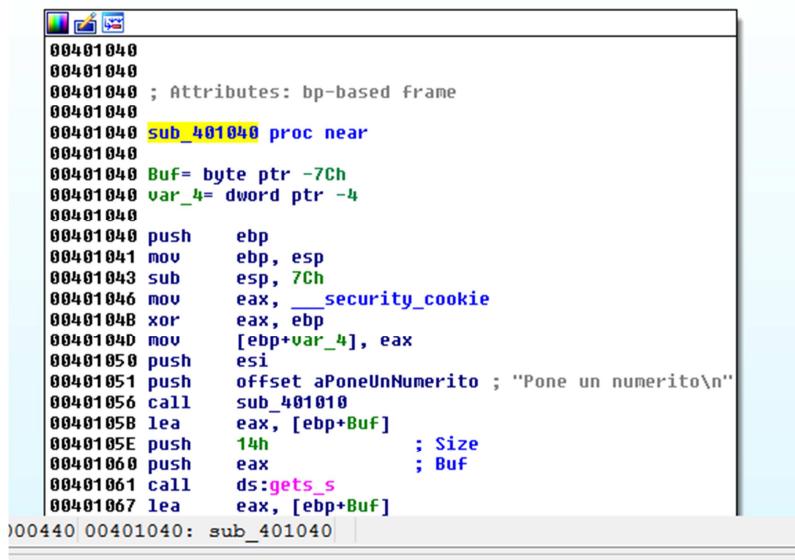
Placing the mouse on the reference little arrow, we see where it is called, but it's better if we press **X** to see the reference list.

Director	Ty	Address	Text
Up	o	sub_401040+11	push offset aPoneUnNumerito; "Pone un numerito\n"

OK Cancel Search Help

Line 1 of 1

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



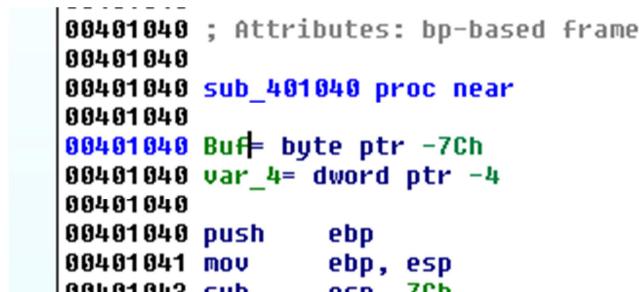
```
00401040 ; Attributes: bp-based frame
00401040 sub_401040 proc near
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, __security_cookie
00401048 xor     eax, ebp
0040104D mov     [ebp+var_4], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un numerito\n"
00401056 call    sub_401010
0040105B lea     eax, [ebp+Buf]
0040105E push    14h          ; Size
00401060 push    eax          ; Buf
00401061 call    ds:gets_s
00401067 lea     eax, [ebp+Buf]
00440 00401040: sub_401040 |
```

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");
```

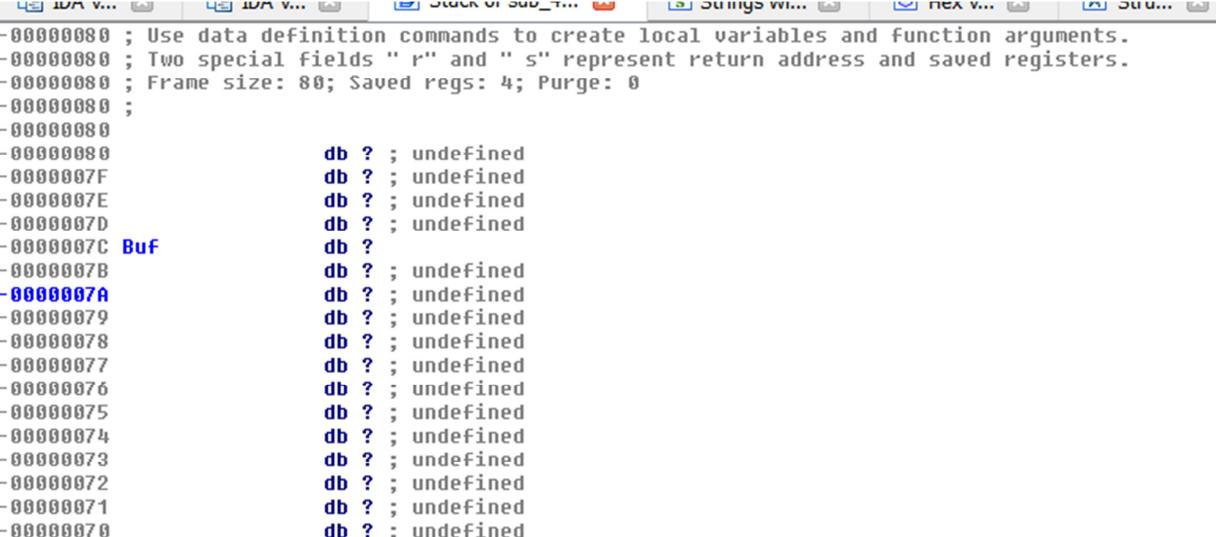
I realize that it optimized some variables I created, for example, **cookie** and **max** that were replaced by constants and left just the buffer that in my code it was 120 bytes long in decimal.

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



```
00401040 ; Attributes: bp-based frame
00401040 sub_401040 proc near
00401040 Buf= byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
```

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.

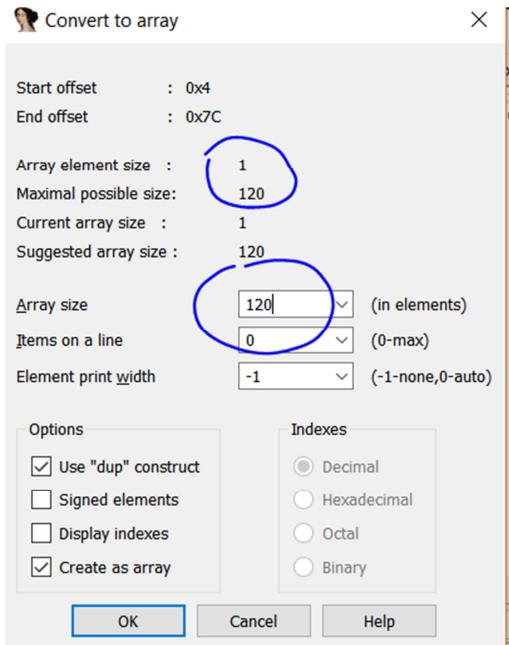


```

-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
-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...

```
-00000080          db ? ; undefined
-0000007F          db ? ; undefined
-0000007E          db ? ; undefined
-0000007D          db ? ; undefined
-0000007C Buf      db 120 dup(?)
-00000084 var_4    dd ?
+00000080 S        db 4 dup(?)
+00000084 R        db 4 dup(?)
+00000088
+00000088 ; end of stack variables
```

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 ADDRESS 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 ADDRESS and the variables were above **S**.

VARIABLES

...

...

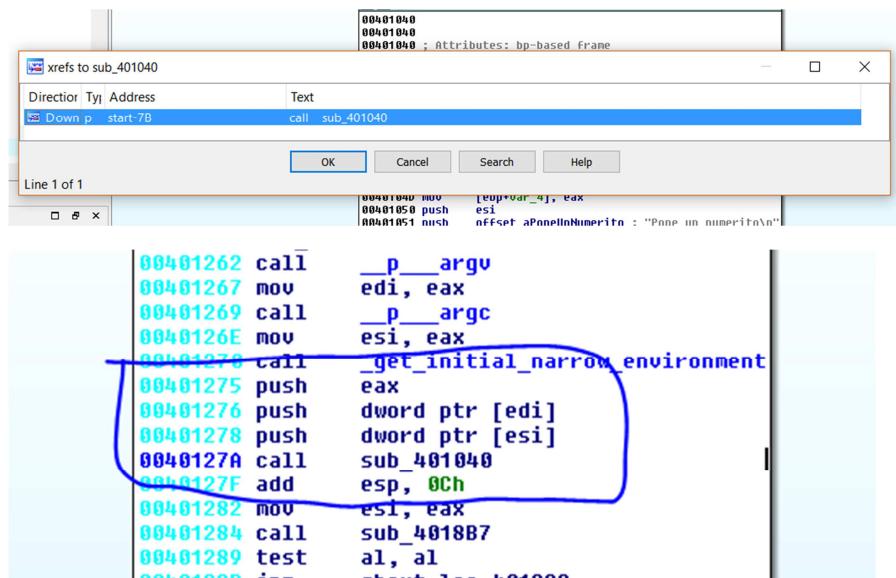
S (stored ebp -it generally comes from PUSH EBP that is the first instruction of the function)

R (return address)

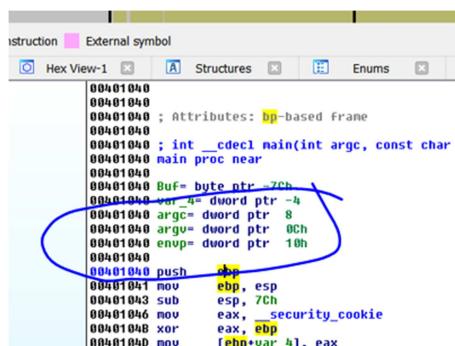
ARGUMENTS

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 **PUSHes** 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.

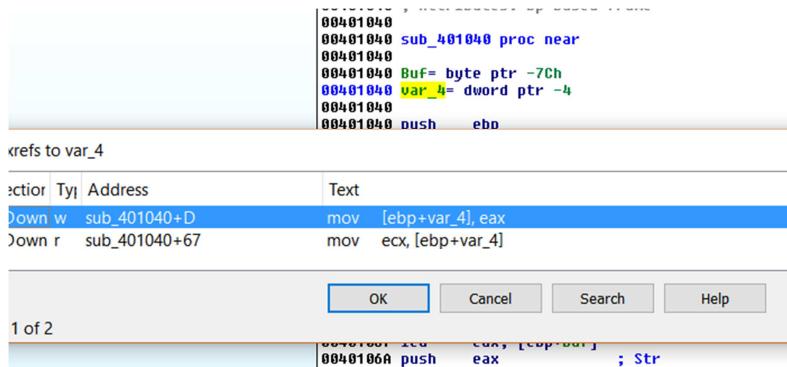
```

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**.

```

00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 sub     esp, 7Ch
00401046 mov     eax, __security_cookie
00401048 xor     eax, ebp
0040104D mov     [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push    esi
00401051 push    offset aPoneUnNumerito ; "Pone un numerito"

```

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.

```

00401040 mov    eax, __security_cookie
00401040 xor    eax, ebp
00401040 mov    [ebp+COOKIE_DE_SEGURIDAD], eax
00401050 push   esi
00401051 push   offset aPoneUnNumerito ; "Pone un numerito\n"
00401056 call   sub_401010
00401058 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\n"
0040107C call   sub_401010
00401081 add   esp, 18h
00401081 add   esp, 18h

```

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

```

00401040 ; Attributes: bp-based frame
00401040 int _cdecl main(int argc, const char **argv)
00401040 main proc near
00401040
00401040 Buf byte ptr -7Ch
00401040 var_4= dword ptr -4
00401040 argc= dword ptr 8
00401040 argv= dword ptr 0Ch
00401040 envp= dword ptr 10h
00401040
00401040 push  ebp
00401041 mov  ebp, esp
00401043 sub  esp, 7Ch
00401046 mov  eax, security_cookie
00401048 xor  eax, ebp
0040104D mov  [ebp+var_4], eax
00401050 push  esi
00401051 push  offset aPoneUnNumerito ; "Pone un n
00401056 call  printf
00401058 lea  eax, [ebp+Buf]
0040105E push  14h          ; Size
00401060 push  eax          ; Buf
00401061 call  ds:_imp_gets_s

```

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**.

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 sub_401010 proc near
00401010
00401010 arg_0= dword ptr 8
00401010 arg_4= byte ptr 0Ch
00401010
00401010 push  ebp
00401011 mov  ebp, esp
00401013 push  esi
00401014 mov  esi, [ebp+arg_0]
00401017 push  1
00401019 call  ds:_acrt_iob_func
0040101F add  esp, 4
00401022 lea  ecx, [ebp+arg_4]
00401025 push  ecx
00401026 push  0
00401028 push  esi
00401029 push  eax
0040102A call  sub_401000
0040102F push  dword ptr [eax+4]
00401032 push  dword ptr [eax]
00401034 call  ds:_stdio_common_vfprintf
0040103A add  esp, 18h
0040103A add  esp, 18h

```

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

```

00401041 mov    esp, 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 i
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\n"
0040107C call   printf_0
00401081 ...

```

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 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.

```
00401060  call    ds:_atoi
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.

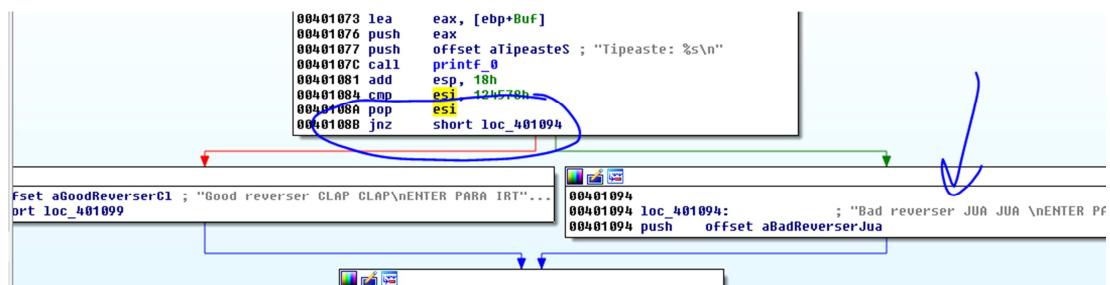
```

0040106B call    ds:atoi
00401071 mov     esi, eax
00401073 lea     eax, [ebp+Buf]
00401076 push    eax
00401077 push    offset aTipeasteS ; "Tipeaste: %s\n"
0040107C call    printf_0
00401081 add    esp, 18h
00401084 cmp    esi, 124578h
0040108A pop    esi
0040108B jnz    short loc_401094

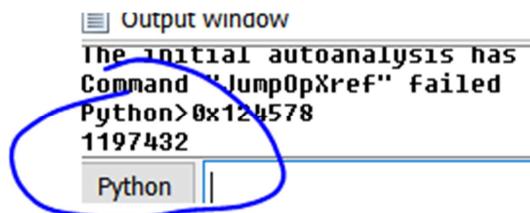
```

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...

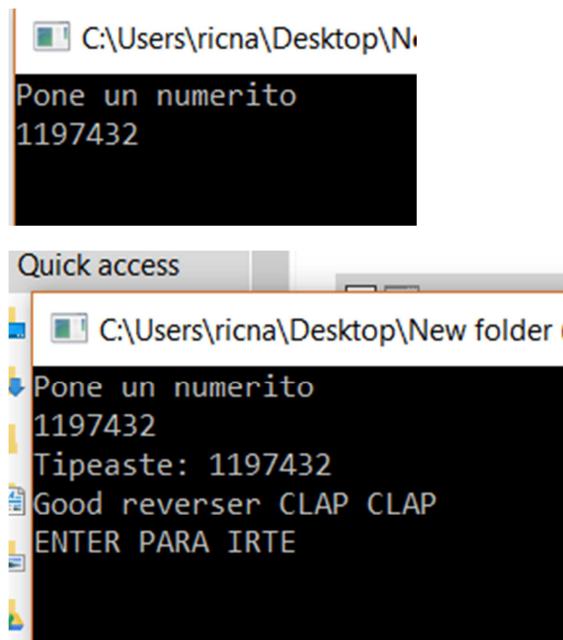


If that comparison is not the equal or zero (JNZ), it takes me to BAD REVERSER, but if they equal, we go to GOOD REVERSER. Let's try the Python console to see the decimal value of 0x124578.



```
Output window
The initial autoanalysis has
Command "JumpOpXref" failed
Python> 0x124578
1197432
```

Let's try that serial in the crackme.



```
C:\Users\ricna\Desktop\New folder
Pone un numerito
1197432
```

Quick access

```
C:\Users\ricna\Desktop\New folder
Pone un numerito
1197432
Tipeaste: 1197432
Good reverser CLAP CLAP
ENTER PARA IRTE
```

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

Ricardo Narvaja

Translated by: @lvinsonCLS