PART 25

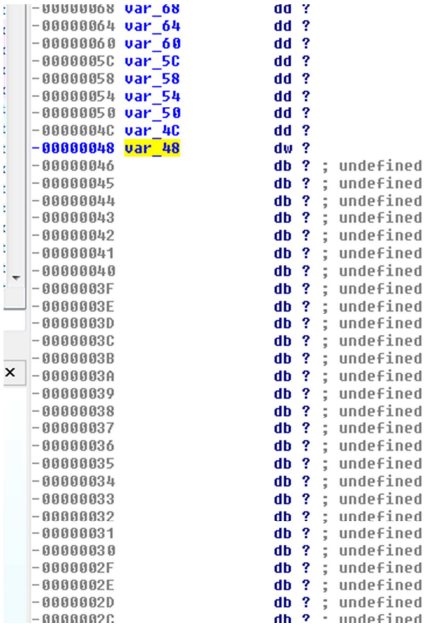
STRUCTURES

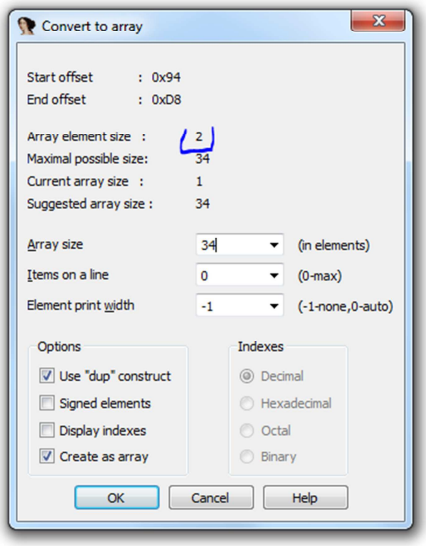
In this part, we will begin the study of how IDA PRO helps to reverse when the program uses structures.

At the end of this part, the solutions of IDA3 and IDA4 will be briefly as it is already a known topic so the solution will be very brief.

What is a structure?

We do not need a very technical definition but we saw that the ARRAYS were container data types, which reserved a space in memory for their fields which were all of the same type, so there may be ARRAYS of bytes, of words, of dwords, the subject Is that in the same array there can be no fields of different type.

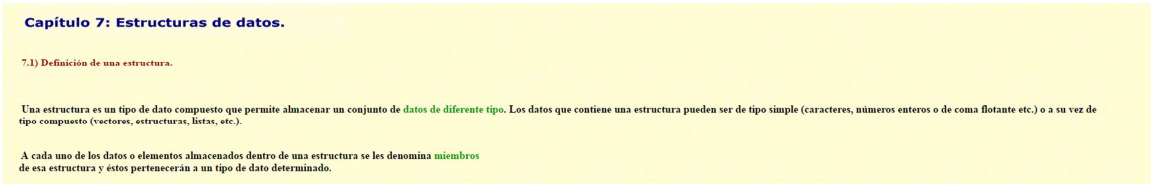




There we see an example of an ARRAY that its size is 34, and each element has size 2 or each element is a Word, so the total length of it will be 34 * 2 or 68 decimal.

In this example Array, each element is a word or it occupies 2 bytes, if I want an array that has elements of 1 single byte I must build another array, since I cannot mix data of different size or type in it.

The Structure on the other hand allows to mix different types of data of different sizes within the same.



Capítulo 7: Estructuras de datos.

7.1) Definición de una estructura.

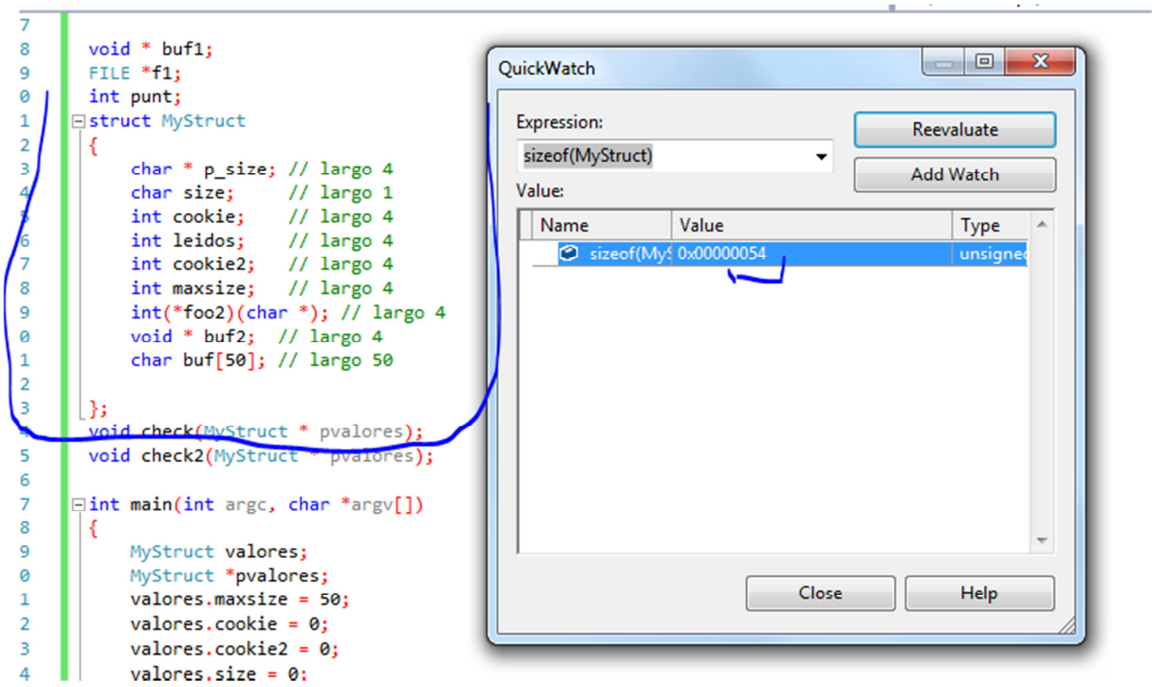
Una estructura es un tipo de dato compuesto que permite almacenar un conjunto de datos de diferente tipo. Los datos que contiene una estructura pueden ser de tipo simple (caracteres, números enteros o de coma flotante etc.) o a su vez de tipo compuesto (vectores, estructuras, listas, etc.).

A cada uno de los datos o elementos almacenados dentro de una estructura se les denomina miembros de esa estructura y éstos pertenecerán a un tipo de dato determinado.

There, the definition is more elegant but that is, we can have a container of different types of data and here the length will be the sum of the length of all members or fields.

```
struct MyStruct
{
	char * p_size;
	char size;
	int cookie;
	int leidos;
	int cookie2;
	int maxsize;
	int(*foo2)(char *);
	void * buf2;
	char buf[50];

};
```

In C ++, we could define a structure that way, in this example we see the structure called MyStruct that has several fields inside, it is not necessary to be great geniuses of the programming to realize that it is not the same an INT variable than a Char variable or a 50-byte buffer. If I have Visual Studio, I can see the size of the entire structure that is 0x54.



I do this to later verify what we do in IDA. It does not matter much if you do not have much idea of how Visual Studio is handled. They take it as information.

We see that the sum total of the fields gives me a little less than the amount that it assigns in compiling but that usually happens to allocate a little more.

The pointer to a character variable is 4 bytes long because it is a pointer that is a dword and its contents point to a variable character.

char * p_size; // size 4

The same thing. The other pointers to function and to buffer are pointers that are dwords or that its length is 4 bytes and that point to different types of data of different length, but in the structure only the pointer is saved or that each only adds 4 bytes .

int(*foo2)(char *); // size 4 Pointer to a function

void * buf2; // size 4 Pointer to a buffer

However the last buffer is not a pointer (it does not have the asterisk (*), so it is a buffer that is directly inside the structure occupying 50 bytes of it).

```
char buf[50]; // size 50 decimal
```

The important thing about all this, rather than the long ones, is to realize that the structures are containers of different types of data and that it is very difficult that when disassembling, IDA can recognize each field and its type automatically, especially if we do not have the symbols.

Let's make a simple example to get accustomed to detect and manage the structure in IDA.

```
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>


struct MyStruct
{
    char size;     // largo 1
    int cookie2;   // largo 4
    char buf[50]; // largo 0x32

};


int main(int argc, char *argv[])
{
    MyStruct valores;

    valores.cookie2 = 0;
    valores.size = 50;

    if (argc != 2) {
        printf("Bye Bye");
        exit(1); }

    strncpy (valores.buf , argv[1], valores.size);

    printf(valores.buf);

    getchar();


}
```

We see a simple code that receives an argument through the console if we do not execute it with some argument it will say "Bye Bye"

We could start it like this.

```
C:\Users\ricnar\Documents\Visual Studio 2015\Projects\CACA\DEBUG>pepe.exe aaaaaaaaaa
aaaaaaaaaa

C:\Users\ricnar\Documents\Visual Studio 2015\Projects\CACA\DEBUG>
```

If we just double clicked or we would not put some argument it would check that **argc** which is the number of arguments is different from **2** and throws us out (the number of arguments includes the name of the executable so in this case two arguments would be the first pepe.exe and the second aaaaaaaaaaa)

```
if (argc != 2) {
        printf("Bye Bye");
        exit(1); }
```

We see that it skips that check since **argc** is **2**. In addition to the definition of this structure as data type, we can make that there are variables that in addition to being of type int, char, float or whatever type and also we can do it like a MyStruct type. In the same way we declare an integer variable, for example putting the data type in front.

```
int pepe;
```

It is similar in the case of the structure-type variable.

```
MyStruct valores ;
```

So the variable values will be of MyStruct type will have the same definition, the same length and the same fields. We could create several different MyStruct variables.

```
MyStruct pinguyo;
```

And to refer to the fields of some of them, it is used:

```
valores.size
```

```
pinguyo.size
```

```
valores.cookie2
```

In this way, the values are assigned in the program to some fields of the variable values.

```
valores.cookie2 = 0;
valores.size = 50;
```

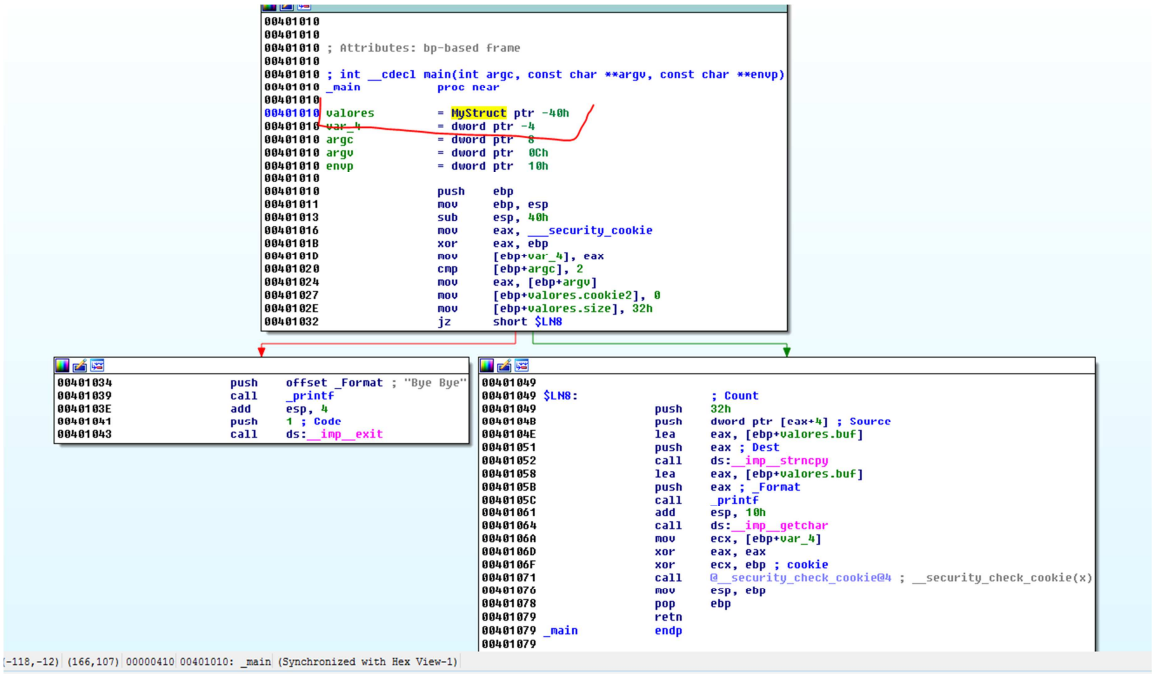
And then, it does a strncpy of the Aes that are in the **argv[1]** variable to the buffer of 50 bytes decimal **valores.buf**, taking **size** as max, the value of **valores.size** that is 50.

strncpy (valores.buf , argv[1], valores.size);

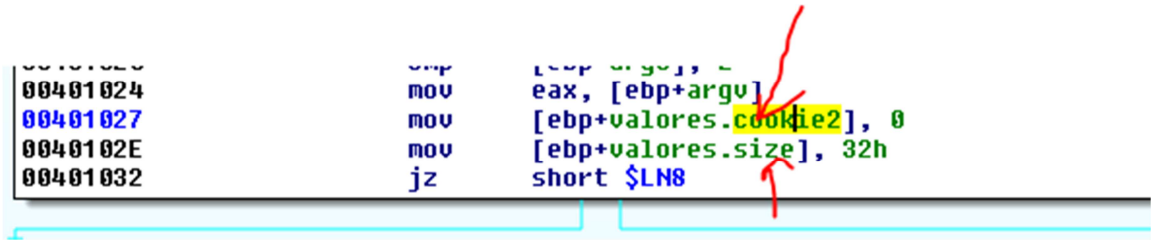
So, there will be no overflow because it copies 50 bytes to a buffer of 50 bytes long, it does not overflow. Then it prints what we typed that is now stored in valores.buf to display it.

printf(valores.buf);

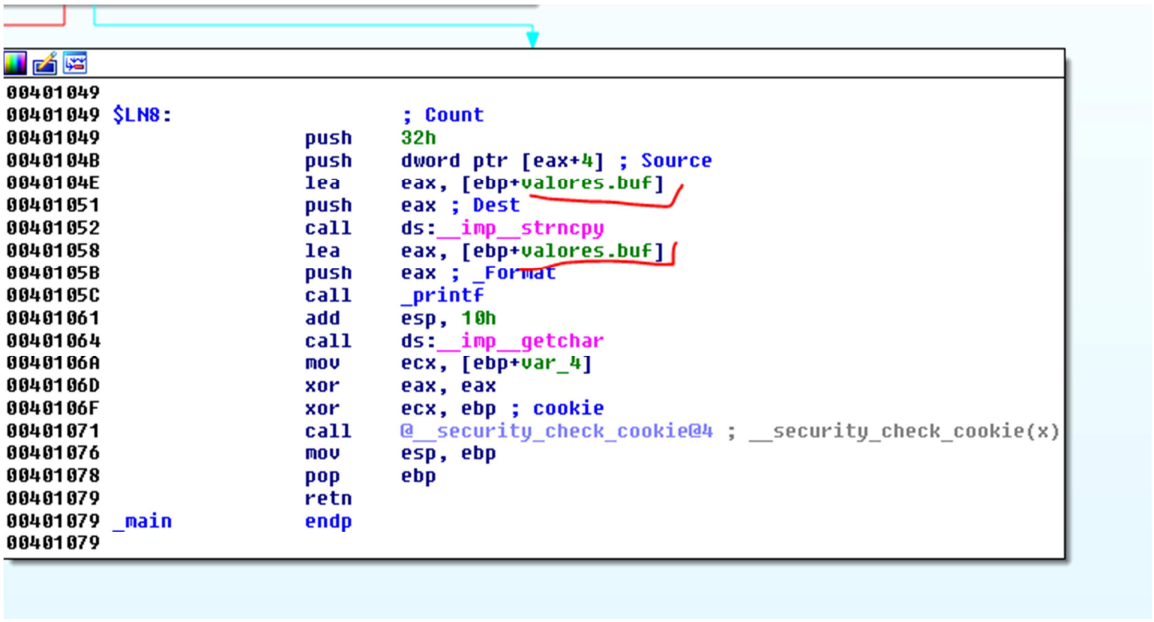
And finally, there is a call to getchar() so it will not close until we press a key and we can see the Aes. Now, let's open the executable in IDA and we'll see how we can interpret this.



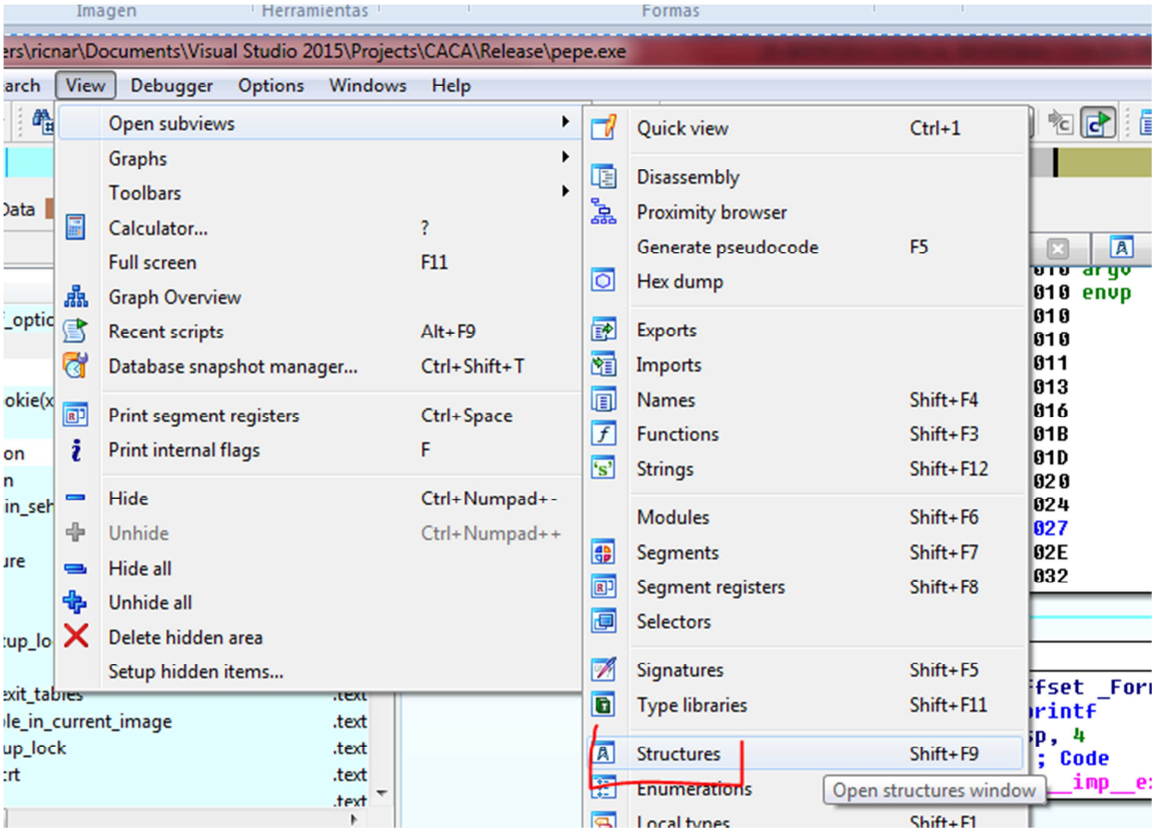
We see that when there are symbols everything is happiness. IDA detected values as a variable of type MyStruct, without problem. Even inside the code we see that it accesses the fields of the structure with its name perfectly.

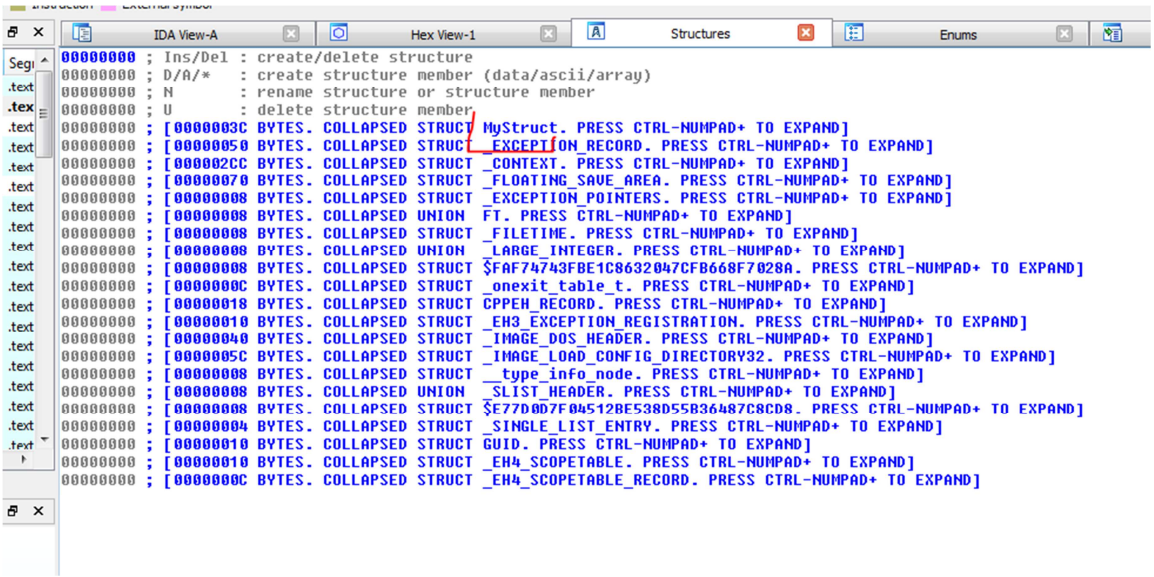


Also, here.



We see that it detects the 50 byte buffer. It already has it renamed as valores.buf in the strncpy and in the final printf. Even if we go to the structures tab.





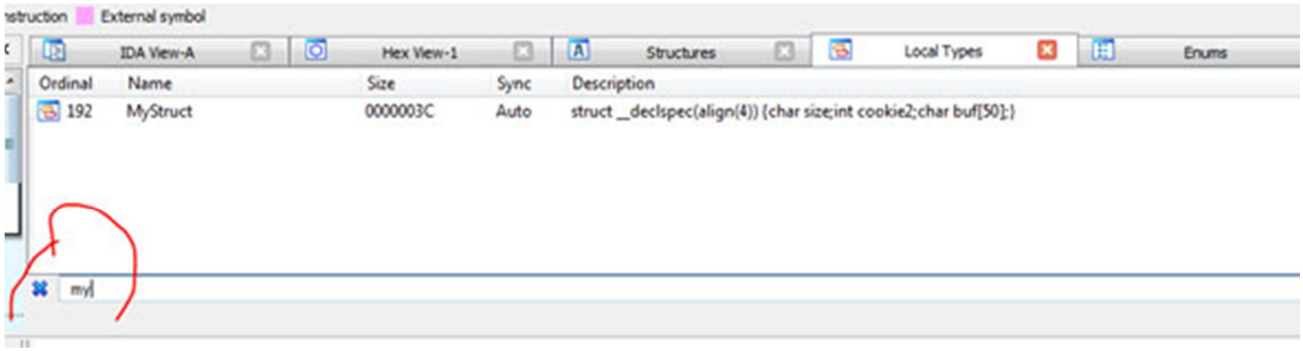
We see that the structure is defined if we press CTRL plus + there.

```
0000000 ; U       : delete structure member
0000000 ; -------------------------------------------------------------------------
0000000
0000000 MyStruct        struc ; (sizeof=0x3C, align=0x4, copyof_192) ; XREF: _main/r
0000000 size            db ?                    ; XREF: _main+1E/w
0000001                 db ? ; undefined
0000002                 db ? ; undefined
0000003                 db ? ; undefined
0000004 cookie2         dd ?                    ; XREF: _main+17/w
0000008 buf             db 50 dup(?)            ; XREF: _main+3E/o _main+48/o
000003A                 db ? ; undefined
000003B                 db ? ; undefined
000003C MyStruct        ends
000003C
0000000 ; [00000050 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]
0000000 ; [000002CC BYTES. COLLAPSED STRUCT _CONTEXT. PRESS CTRL-NUMPAD+ TO EXPAND]
```

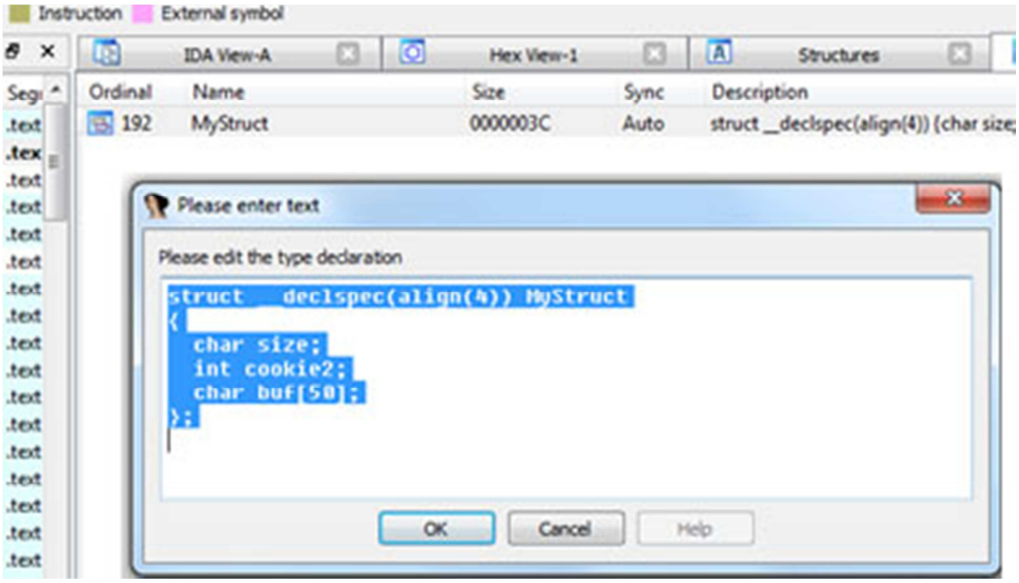
We see that lengths and names are according to what the variable size of 1 byte or db, the cookie2 variable of 4 bytes or dd had defined and buf of 50 bytes decimal.

```
struct MyStruct
{
        char size;     // size 1
        int cookie2;   // size 4
        char buf[50]; // size 0x32

};
```

Even in IDA, there is a LOCAL TYPES tab for editing and entering structures in C++ format. I can see if it is there. There are many, but as I have the filter with CTRL + F I enter **My** and it shows.



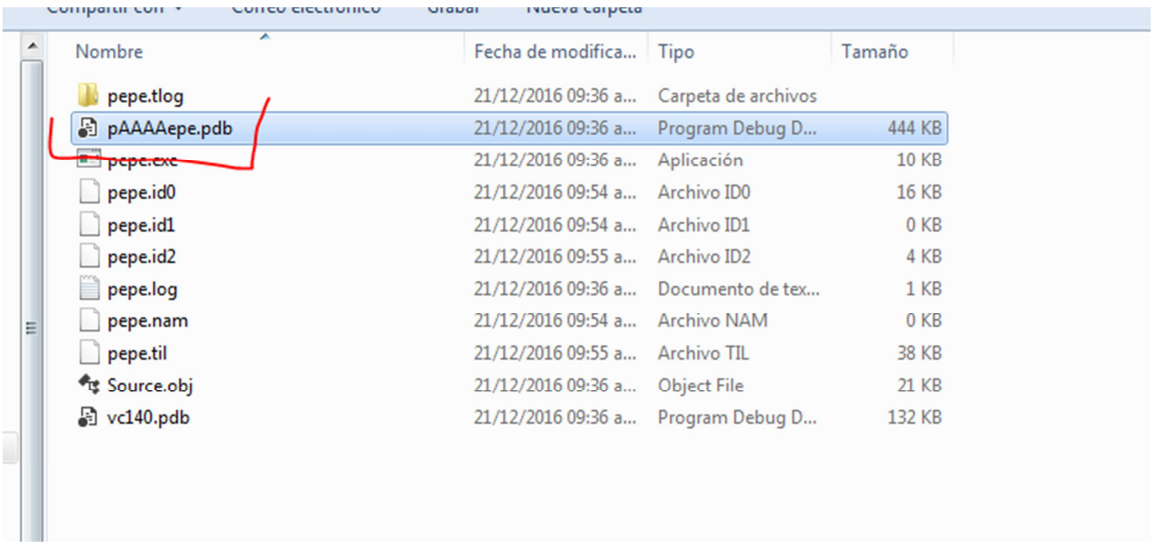
And I can see it by right clicking - EDIT.



That is correct.

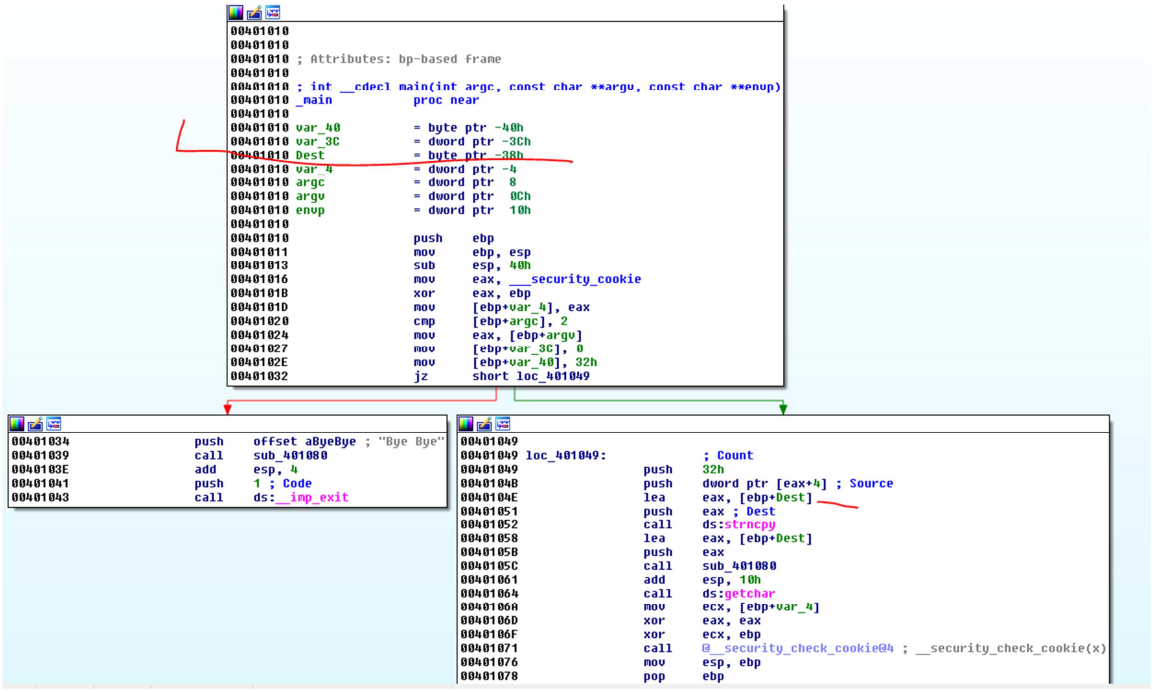
But since life is not all joy and almost never we will have symbols, we will have to perspire a little since IDA cannot detect without them the fields or the structure, although it gives us the interactive tools to do so.

If I compile and rename pepe.pdb.



It will not find the symbols and the thing will be more tricky.

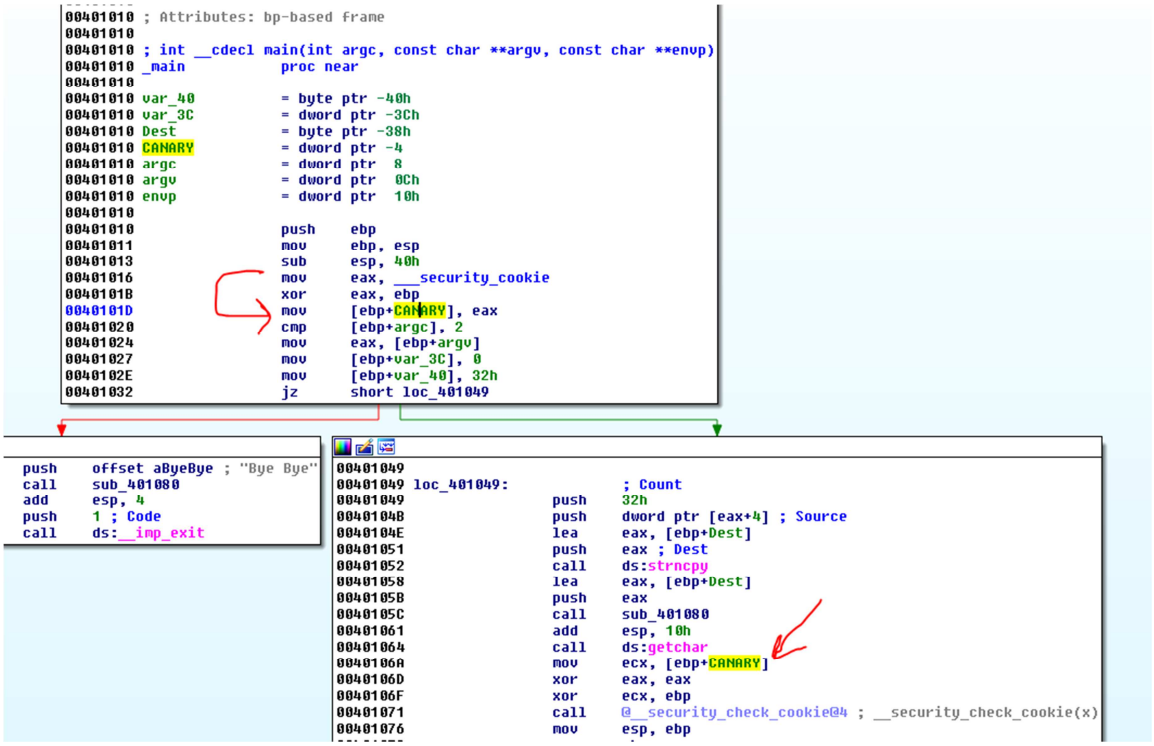
I reopen Pepe.exe and re-disassemble it.



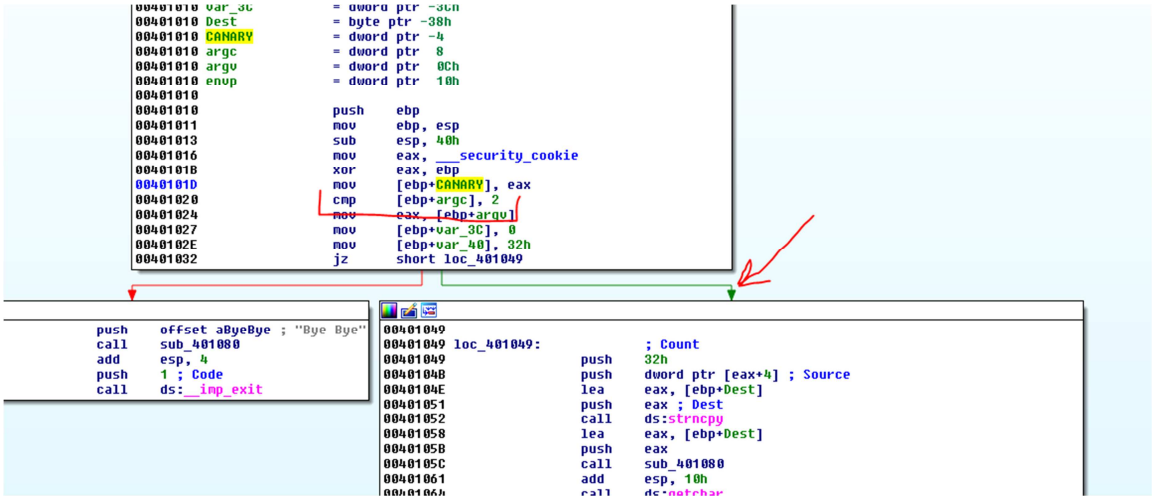
We see that the structure is no longer detected and the fields of it are there, but as individual variables. Someone might object that it could be reversed like this.

The point is that this is a program as a single function, and a simple structure, but we will see later in programs with many functions and complex structures where they pass the same from one to another function by means of the start address of the Structure, which is very difficult to know in the middle of the program but you are setting it up as a structure, to which each field corresponds.

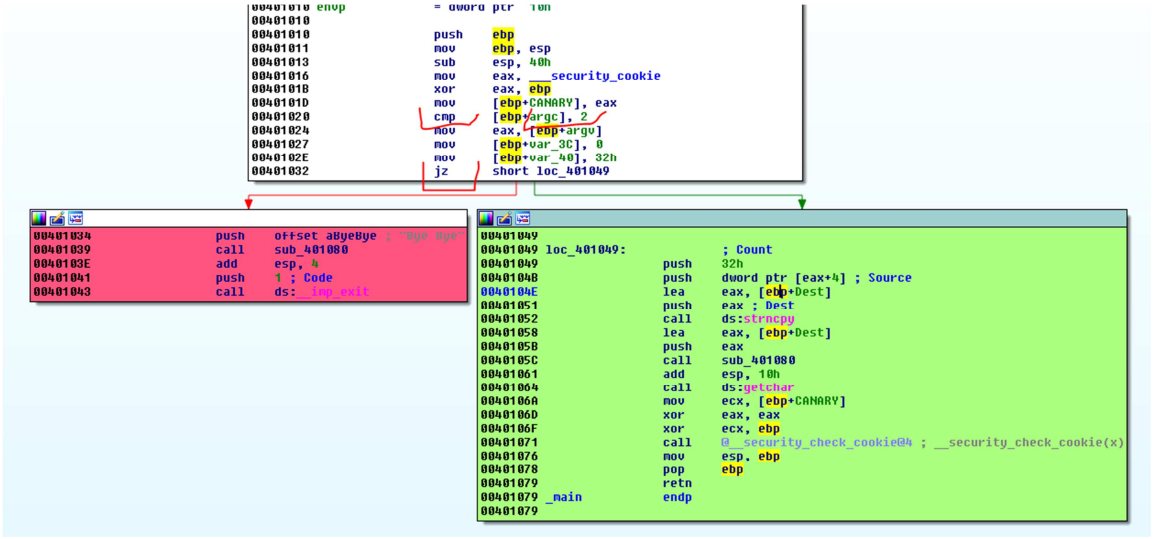
We will see it alike, but for now believe me, in reversing we must know how to work as structures, which is a structure. For now, in this case, reversing as individual variables will work, but let's make it and we do as if we already know that it is a structure, later we will see how to detect when it is a structure and when they are loose local variables.



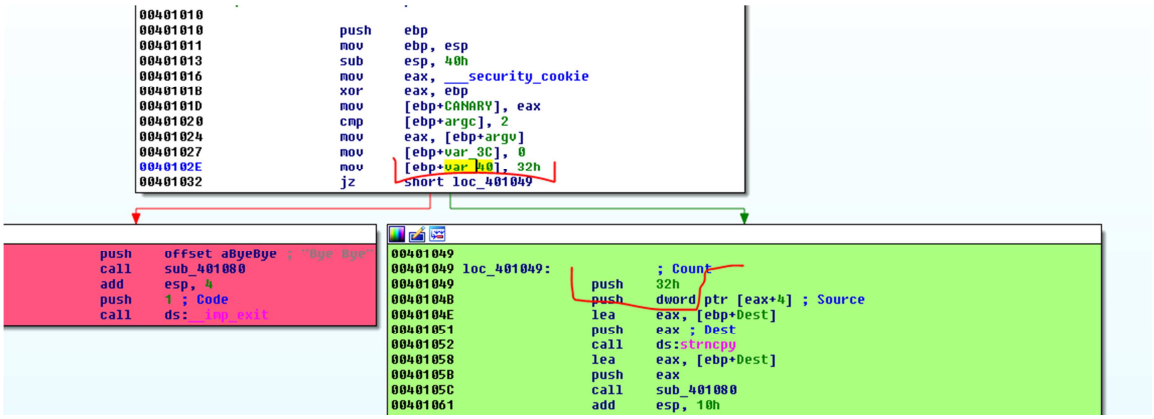
We already know the CANARY of previous reversing, so we rename it.



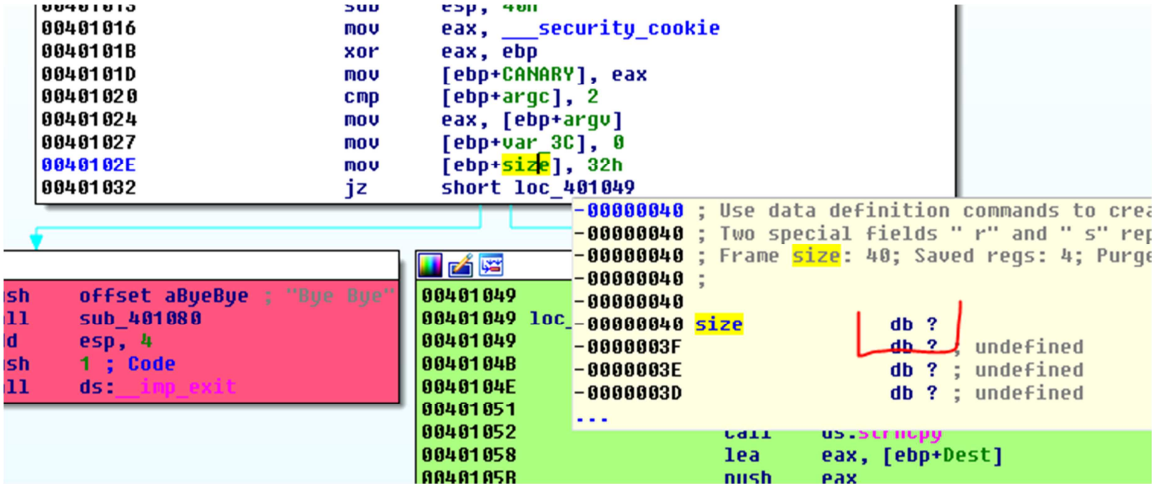
This compares argc with 2. If it is the same, it goes on and if not, it will print Bye Bye and Exit, we paint that in red and the good part in green.



Then, we see that it saves 0x32 in a variable and that later, it uses 0x32 as size of strncpy, in our code, it used the variable as long, but here to gain space it puts it directly with PUSH 0x32.

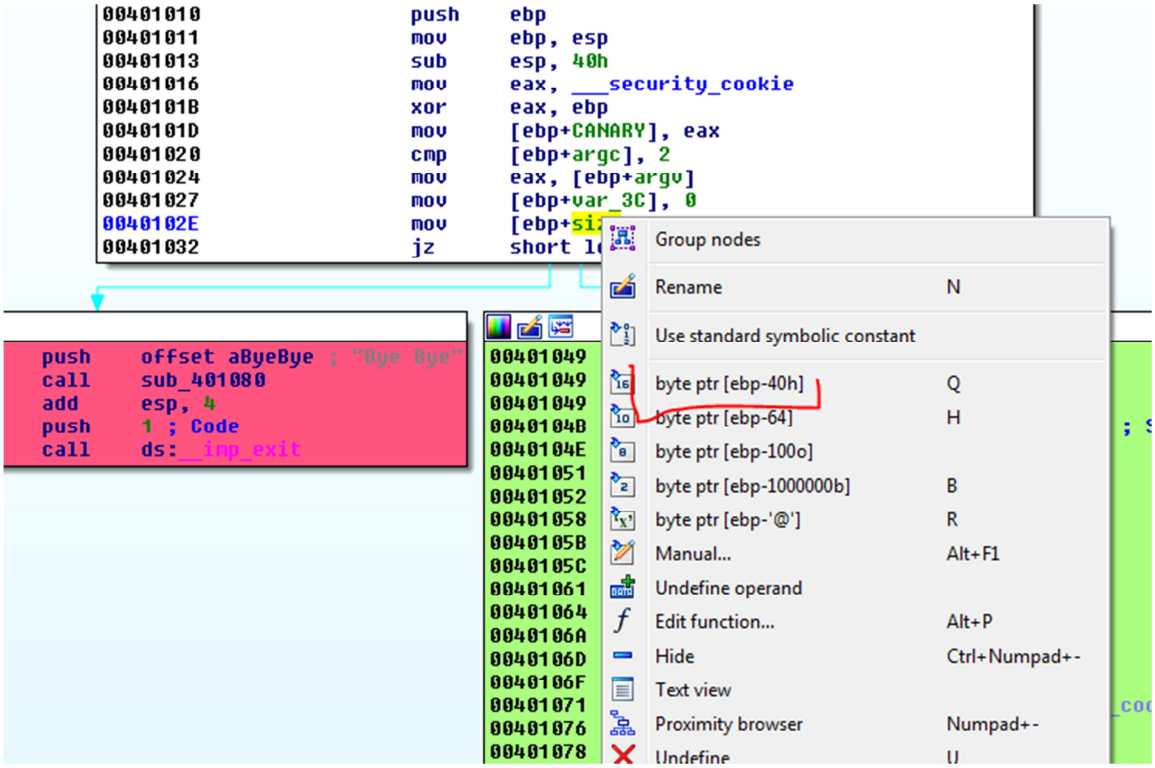


It does not use the var_40 variable anymore. Anyway, I will rename it to size.

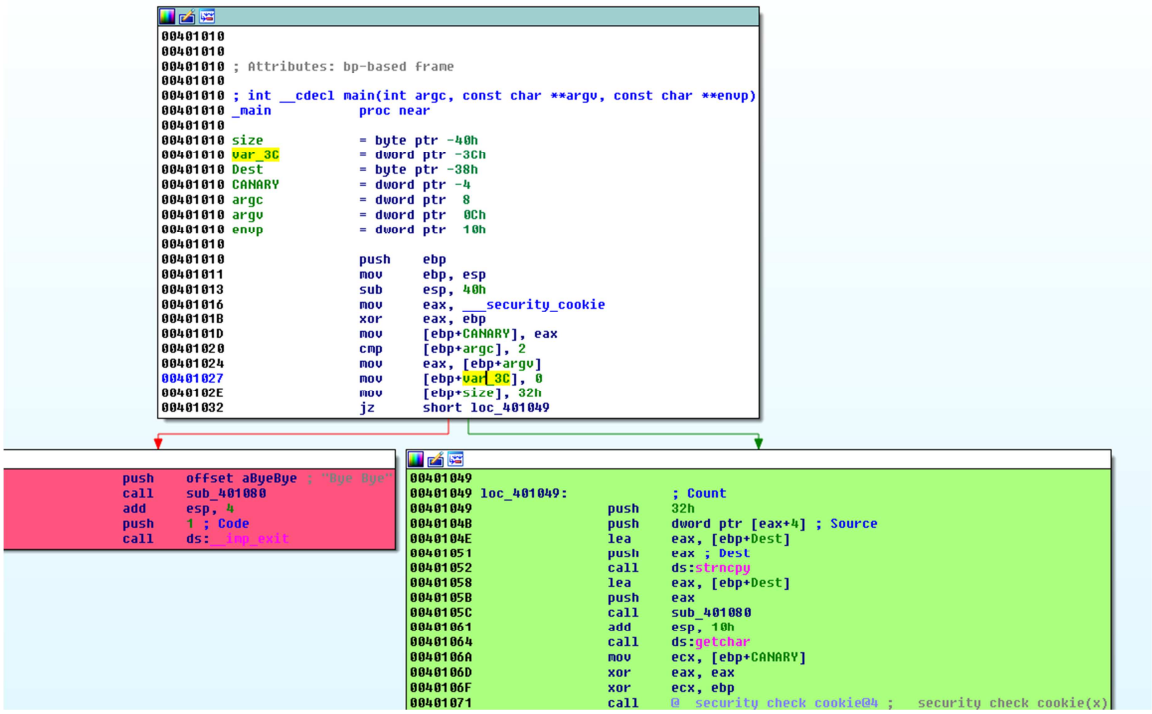


Passing the mouse over it, we see that it detects it as a byte variable (db).

We also see by right clicking that the other representations show us that it is a variable of a byte since the instruction says it.

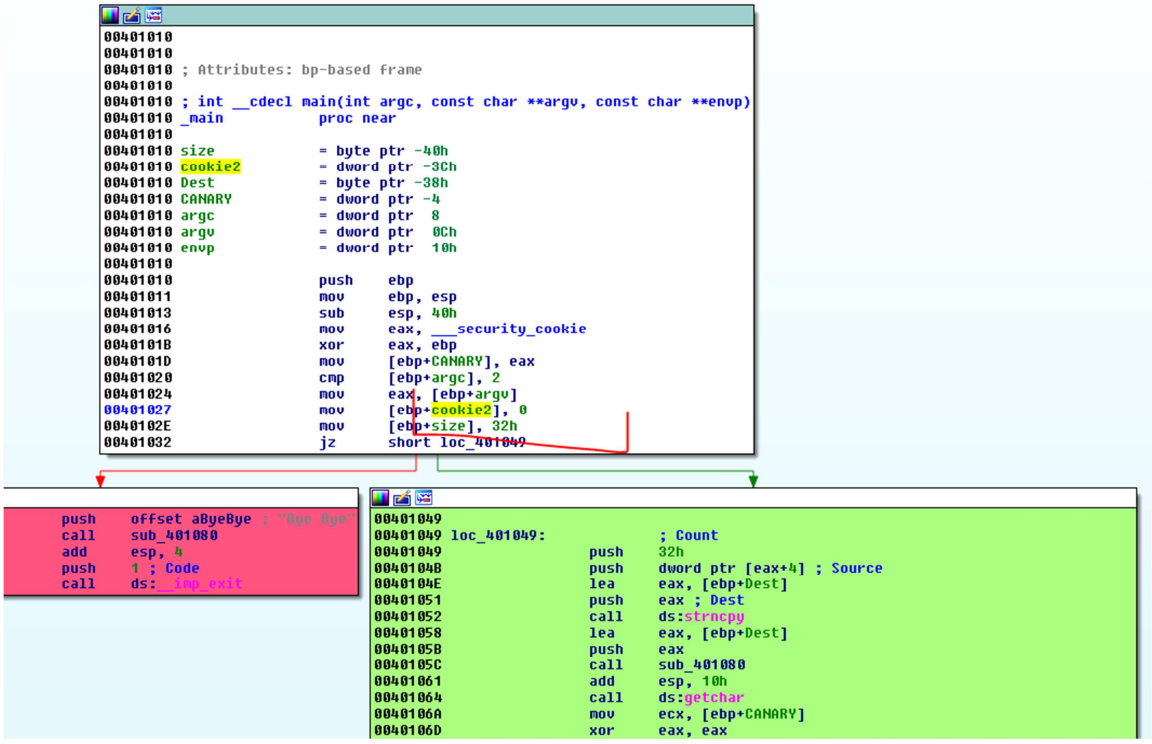


We know that in the original code the cookie2 variable was not used and set it to zero.



valores.cookie2 = 0;

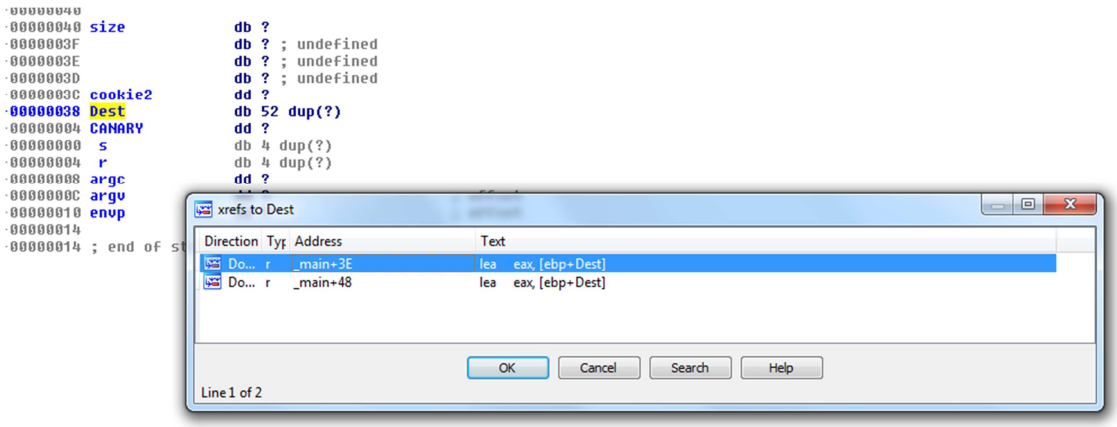
We will rename it as a cookie2 but if we did not know, we would name it. Anyway, it does not use it anymore and it does not affect anything.



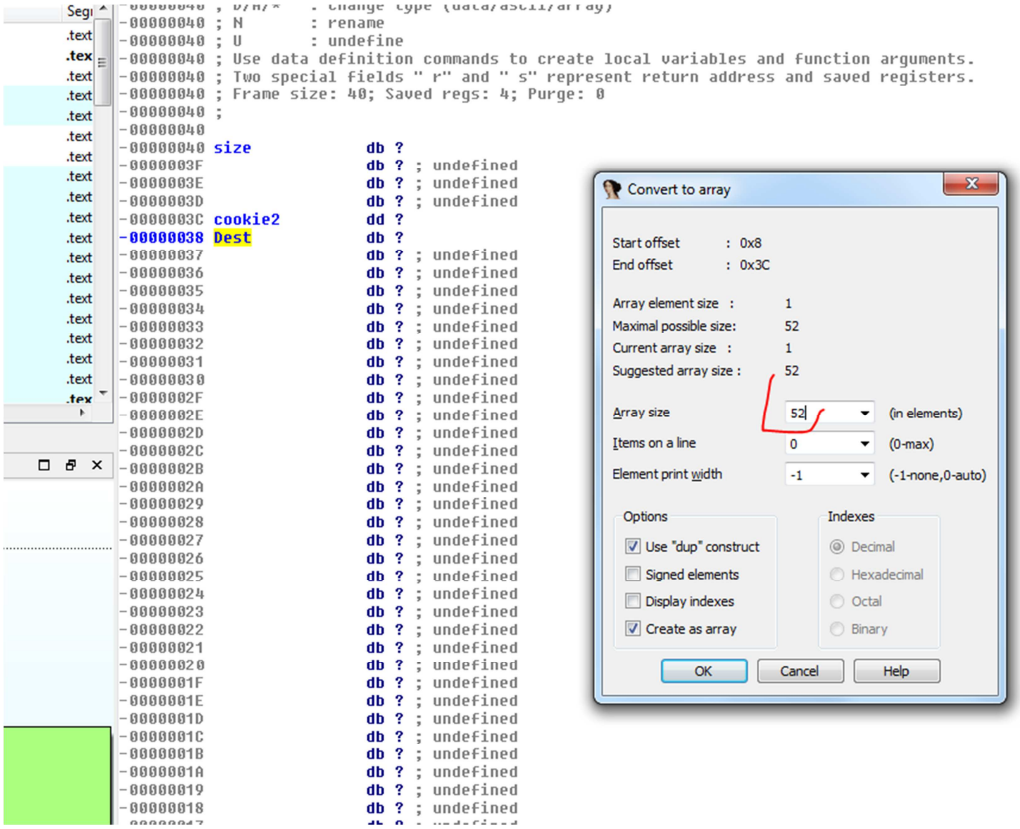
Let's see the representation of the stack.

```
-00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
-00000040 ;
-00000040
-00000040 size            db ?
-0000003F                 db ? ; undefined
-0000003E                 db ? ; undefined
-0000003D                 db ? ; undefined
-0000003C cookie2         dd ?
-00000038 Dest            db ?
-00000037                 db ? ; undefined
-00000036                 db ? ; undefined
-00000035                 db ? ; undefined
-00000034                 db ? ; undefined
-00000033                 db ? ; undefined
-00000032                 db ? ; undefined
-00000031                 db ? ; undefined
-00000030                 db ? ; undefined
```

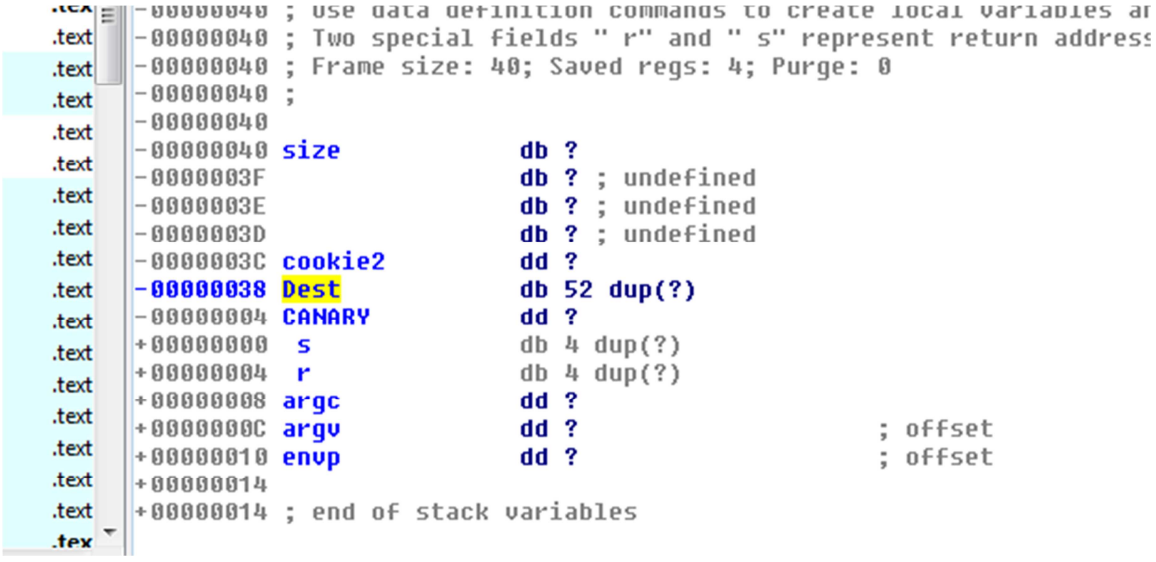
We are missing Dest. We see empty space below, which tells us that it can be a buffer, and also when I look for references with X.



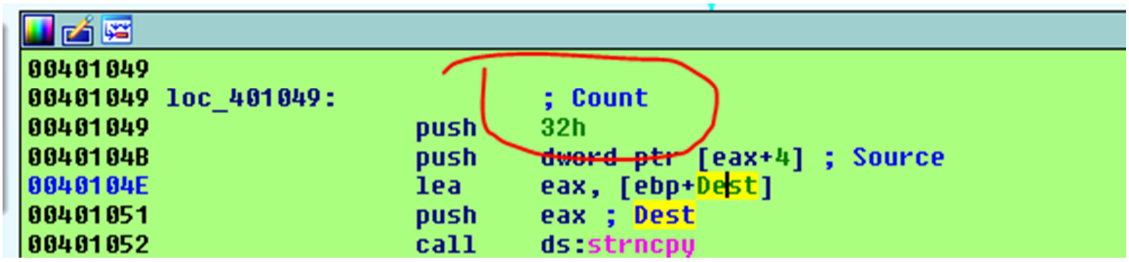
Almost always the buffers will have some reference that is LEA, since to be filled, we will have to pass the address of it to some function as, in this case, strncpy and the LEA obtains it.

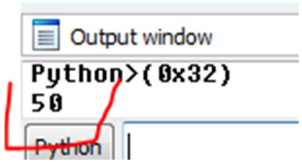


In this case, it reserved 52 instead of 50 which is usually the case.



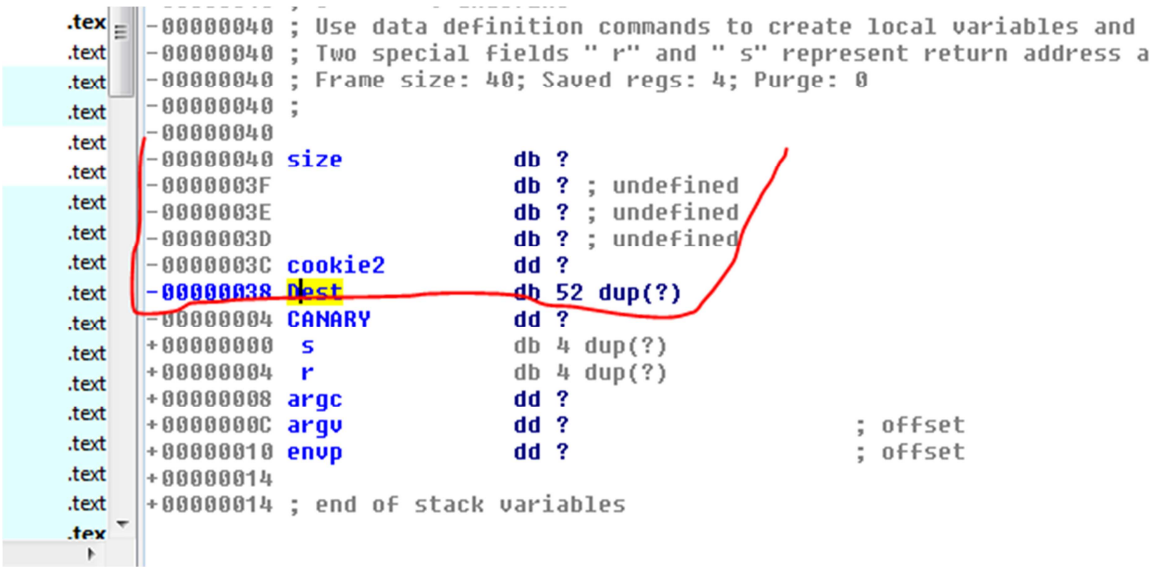
We already see the representation of the complete stack. With this, we can know that there is no overflow because we know that the Dest buffer has as long 52 and it copies 0x32 bytes hex in it with the strncpy.





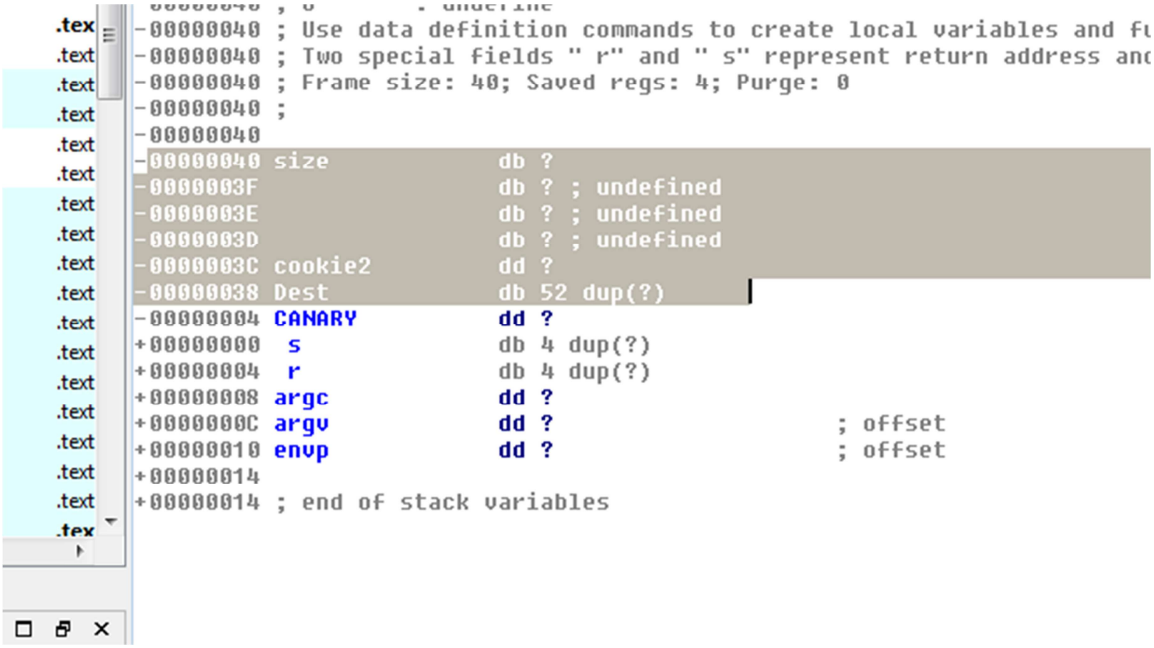
This copies 50 decimal bytes to a buffer of 52 which makes it not vulnerable and there is no overflow. With this, it would be good but we must start slowly with the subject structures and although this example is not necessary, we will use it.

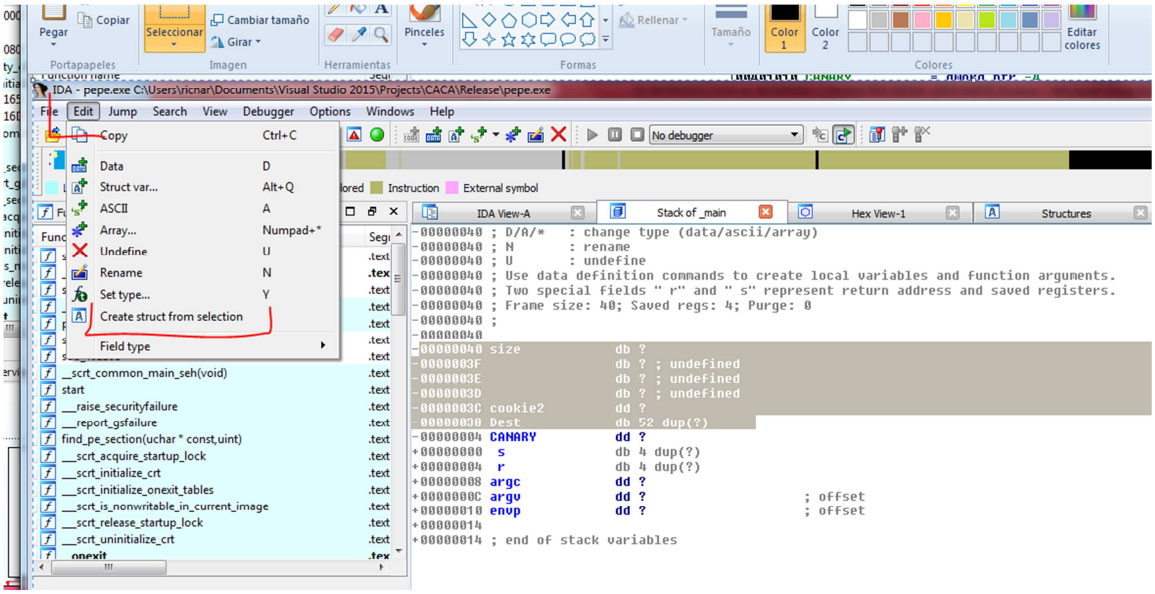
Let's go to the representation of the stack. If there is a structure it will not include the CANARY that is added by the compiler.



That will possibly cover that.

So, I mark that area and I go to the menu to EDIT-CREATE STRUCT FROM SELECTION.





And that will remain so, if we see in the tab structures.

```
00000000 ; [00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXP
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struct_0        struc ; (sizeof=0x3C, mappedto_35) ; XREF: _main/r
00000000 size            db ?
00000001                 db ? ; undefined
00000002                 db ? ; undefined
00000003                 db ? ; undefined
00000004 cookie2         dd ?
00000008 Dest            db 52 dup(?)
0000003C struct_0        ends
0000003C
```

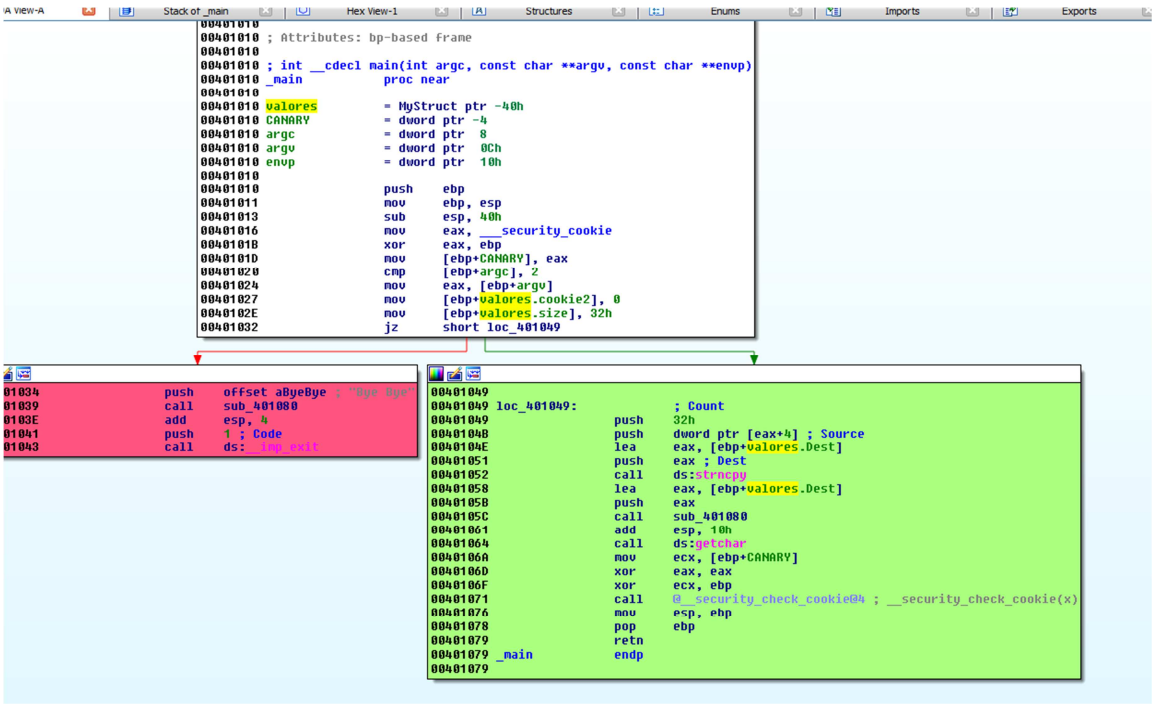
A struct_0 variable of type struct could look like this but we will rename it to match the code.

```
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO E
00000000 ; [00000008 BYTES. COLLAPSED UNION  _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER::$83AF6D9DC8E3B10431D79B304957
00000000 ; [00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXP
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 MyStruct        struc ; (sizeof=0x3C, mappedto_35) ; XREF: _main/r
00000000 size            db ?
00000001                 db ? ; undefined
00000002                 db ? ; undefined
00000003                 db ? ; undefined
00000004 cookie2         dd ?
00000008 Dest            db 52 dup(?)
0000003C MyStruct        ends
0000003C
```

And in the representation of the stack to the variable of type MyStruct, we rename it to **valores**.

```
-00000040 ; U       : undefine
-00000040 ; Use data definition commands to create local va
-00000040 ; Two special fields " r" and " s" represent retu
-00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
-00000040 ;
-00000040
-00000040 valores        MyStruct ?
-00000004 CANARY         dd ?
+00000000  s            db 4 dup(?)
+00000004  r            db 4 dup(?)
+00000008 argc           dd ?
+0000000C argv           dd ?                    ; offset
+00000010 envp           dd ?                    ; offset
+00000014
+00000014 ; end of stack variables
```

So, we're getting a bit like when we had the symbols.

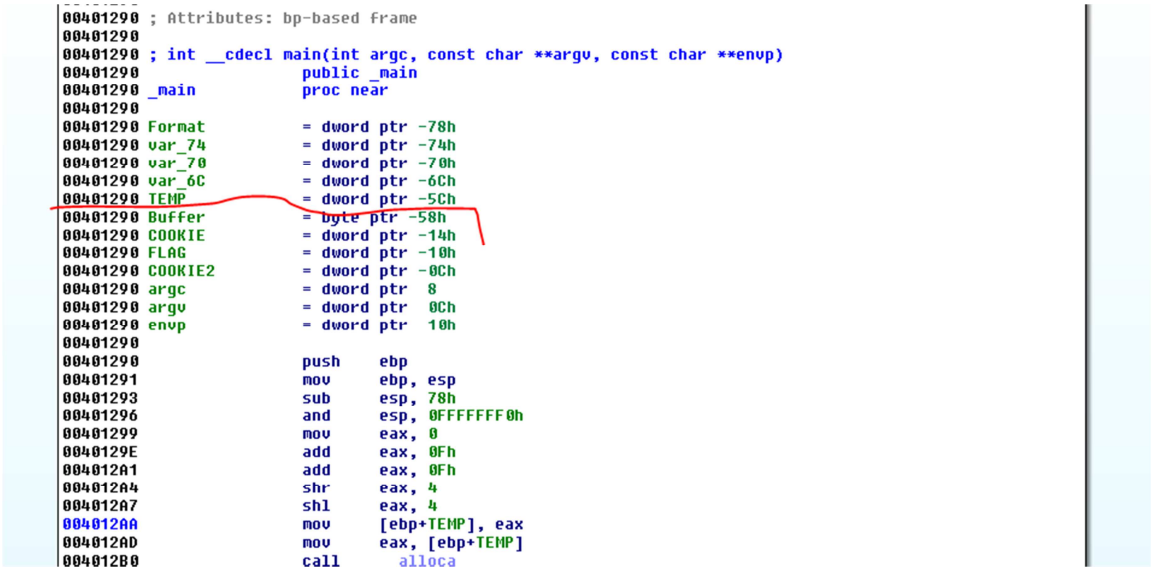


We see that at least in this function that is where the **valores** variable is defined it renamed the fields automatically to valores.xxxx

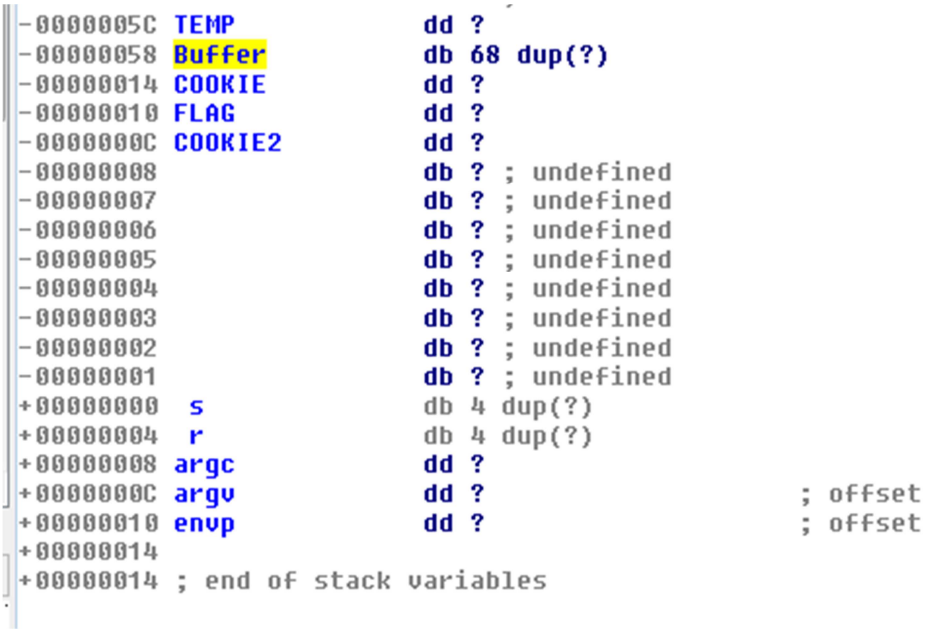
Obviously, this is the simplest way. We have to know that many times in complex structures will have to reverse field to field and fight to have it as complete as possible.

In the next part, we will continue with more complicated examples of structures.

Let's look at the IDA3 and IDA4 solutions.



The TEMP variables above are added by the compiler.



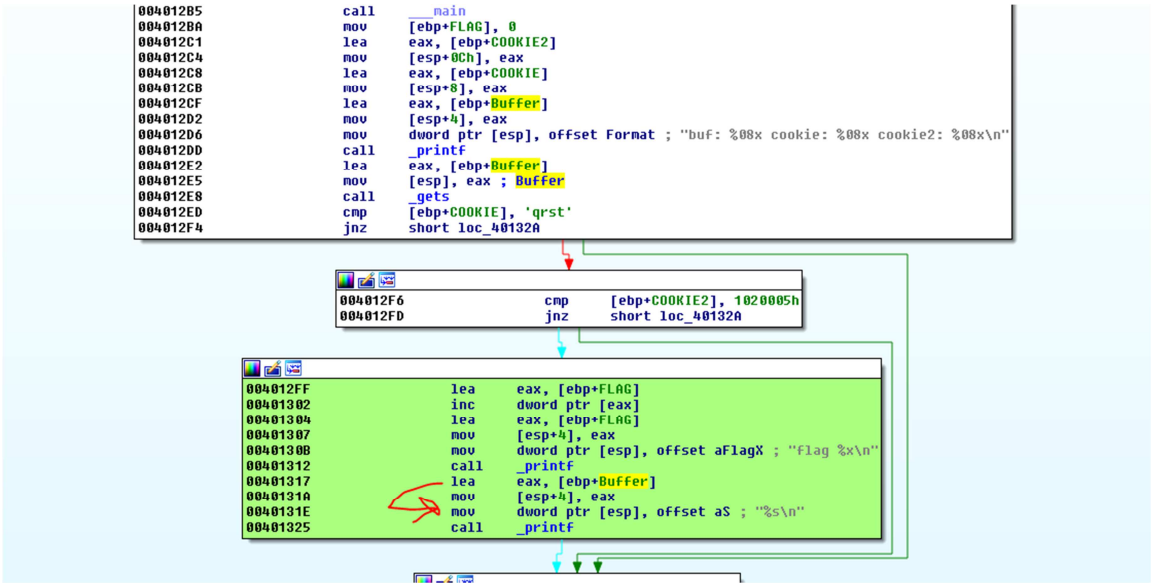
There, I see the representation of the stack and the variables that I have to overwrite COOKIE and COOKIE2, we know that it gets into gets, so it is vulnerable. There is no limit to the number of bytes you input.

Let's see how many bytes I need to get to overflow to COOKIE.

As COOKIE is just below BUFFER that measures 68 bytes decimal, then passing:

68 * 'A' + 'ABCD'

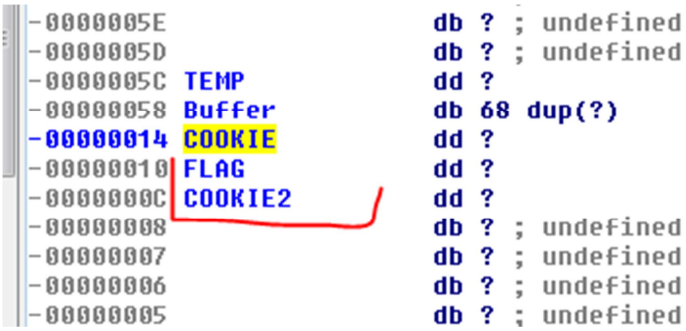
I would overflow the buffer and overwrite COOKIE, let's see what value I need in that variable to get to good boy.

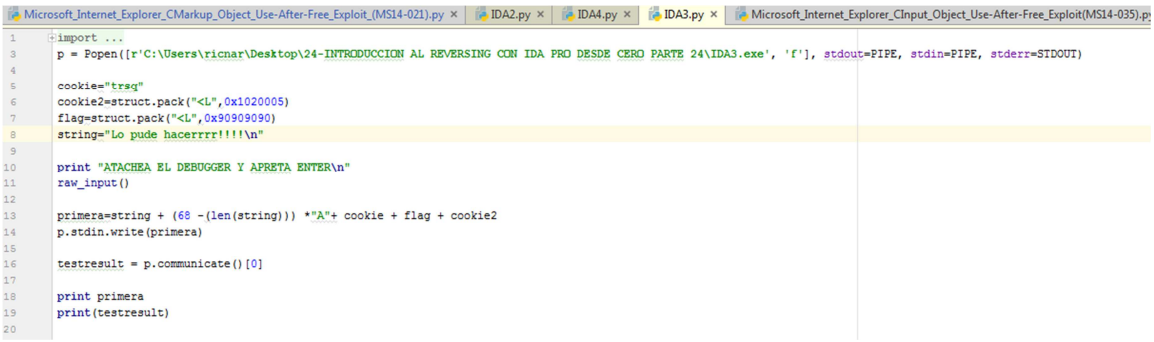


We see that even though there is no good guy defined, it lets me print what I want, since it prints what I keep in the Buffer if I pass the check of both cookies, I can put in the Buffer instead of Aes.

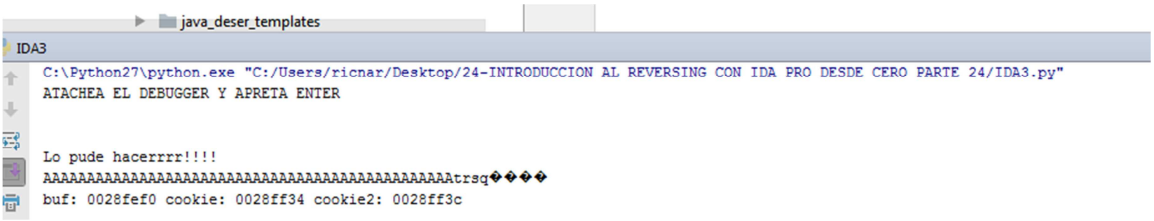
“Lo pude hacerrrr!!!!” In English, it means “I could do it.”

We'll be putting the script together. Also, we see that after COOKIE it comes 4 bytes for Flag and the next 4 will be the COOKIE2.



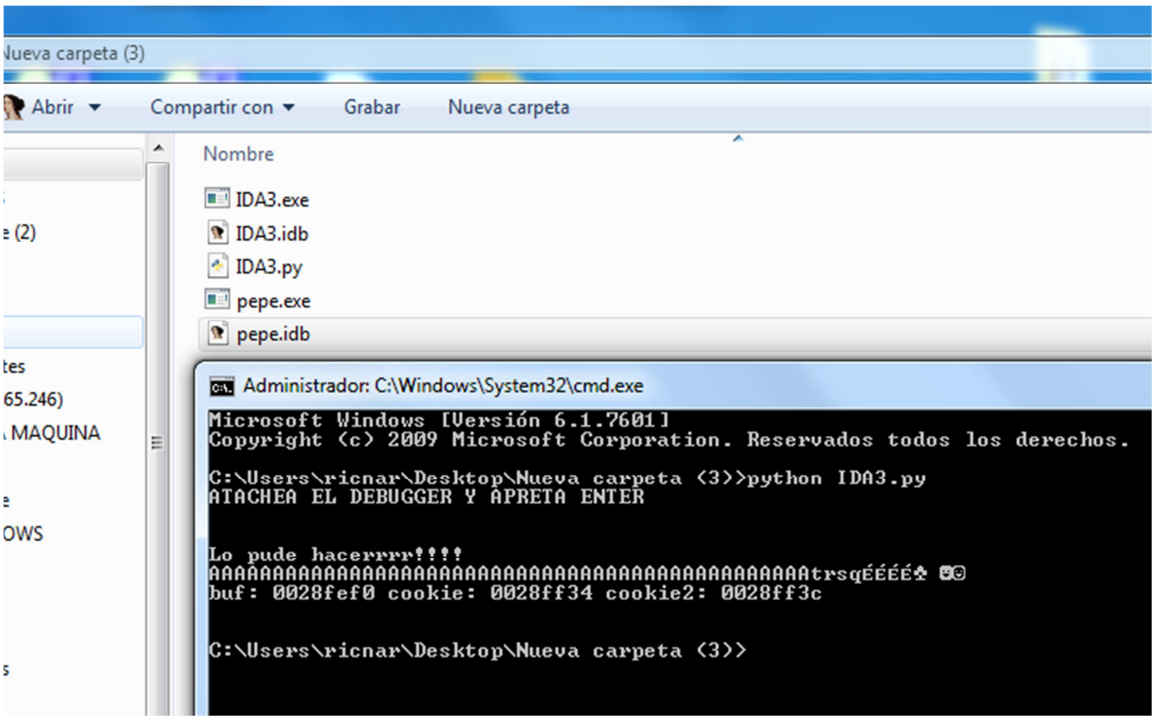


If I run it.



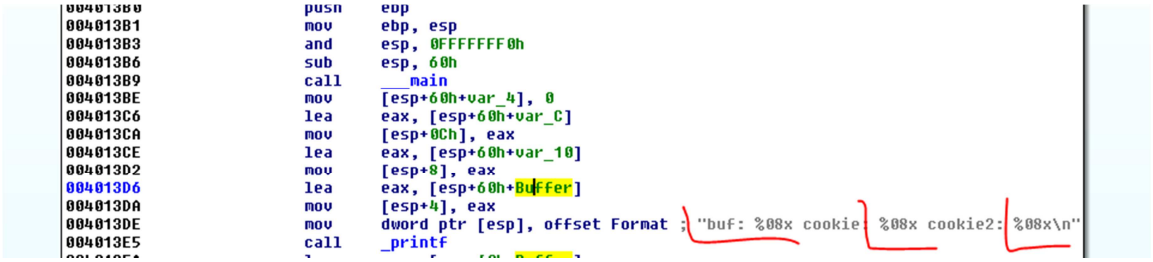
We see that I set the string in front and subtracted 68 bytes from the length of the same string so that it stays being 68, and it does not move what was overwritten in cookie and cookie2.

string + (68 -(len(string))) ***"A"**

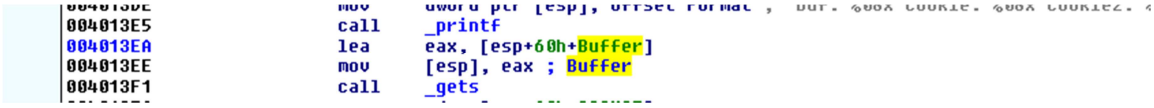


In the ida4 we see that there is a check function.

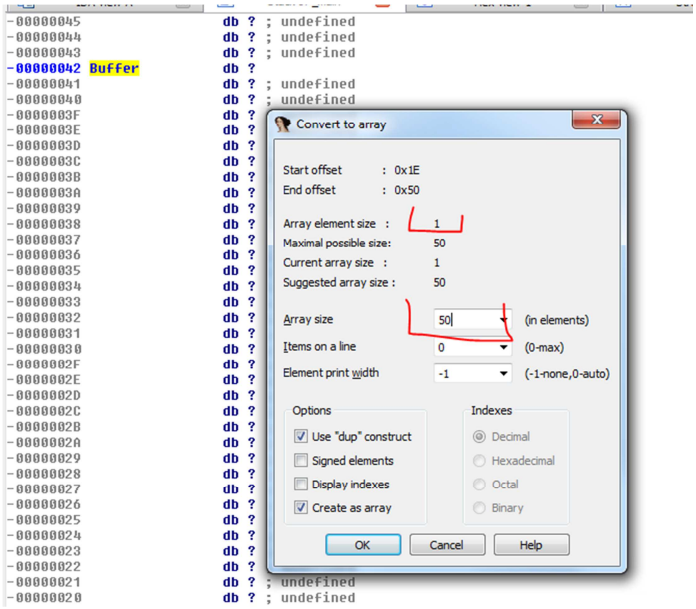
Let's reverse the main.



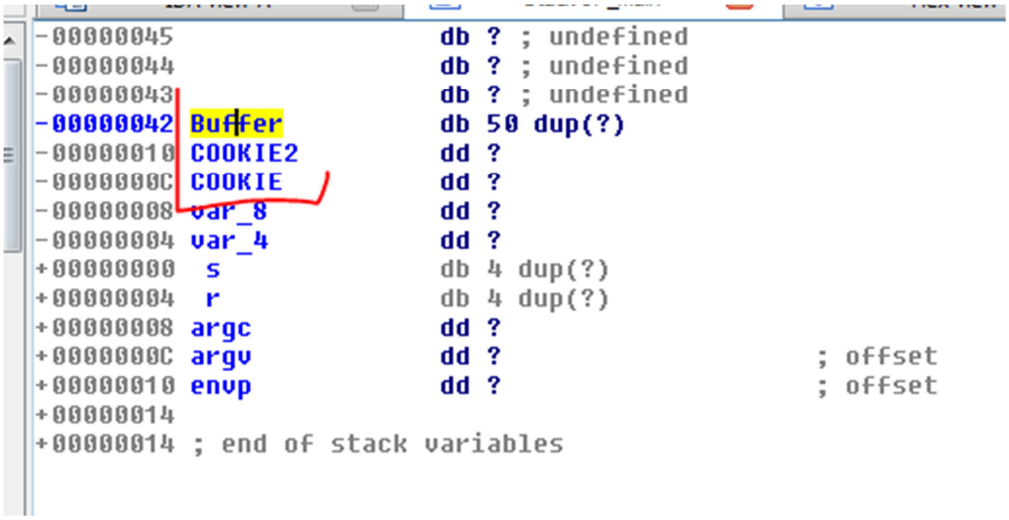
We see that in printf, it prints three addresses the one of cookie, the one of cookie2 and the one of Buffer. So, let's change that accordingly.



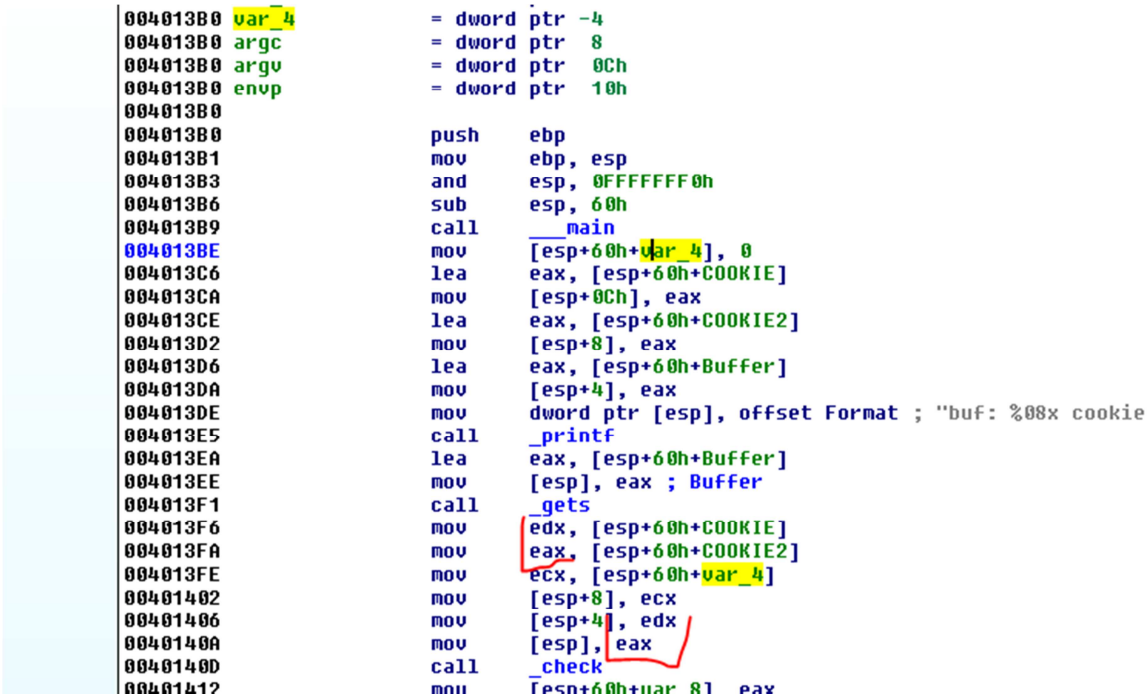
Then, it passes the Buffer to **gets**, so we know it will be vulnerable, just look at the length of the buffer.



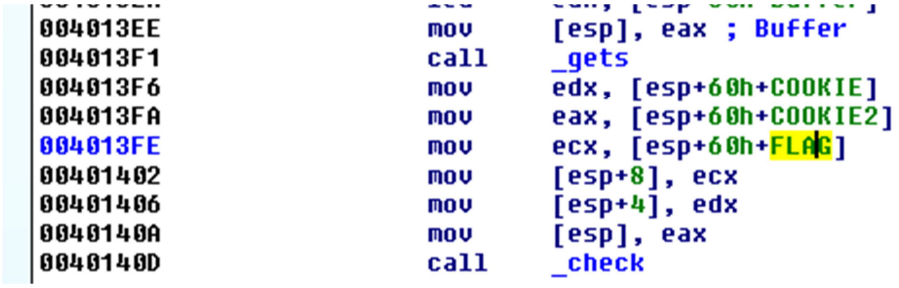
We see that it is 50 decimal.



And then there are two cookies of 4 bytes each.

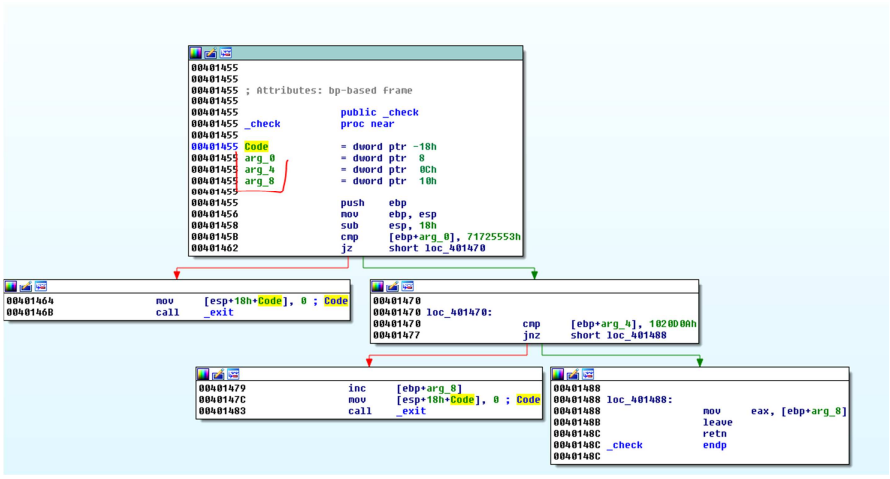


We see that the two cookies are passed as arguments to the check function, plus a var_4 variable that we do not yet know what it is. We will call it FLAG. Later, we will change the name.

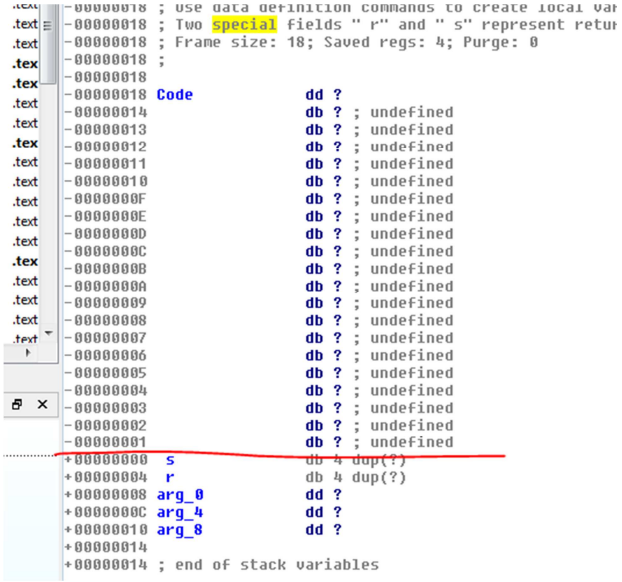


We see that the most distant argument is FLAG, then COOKIE and finally COOKIE2. Let's enter the function.

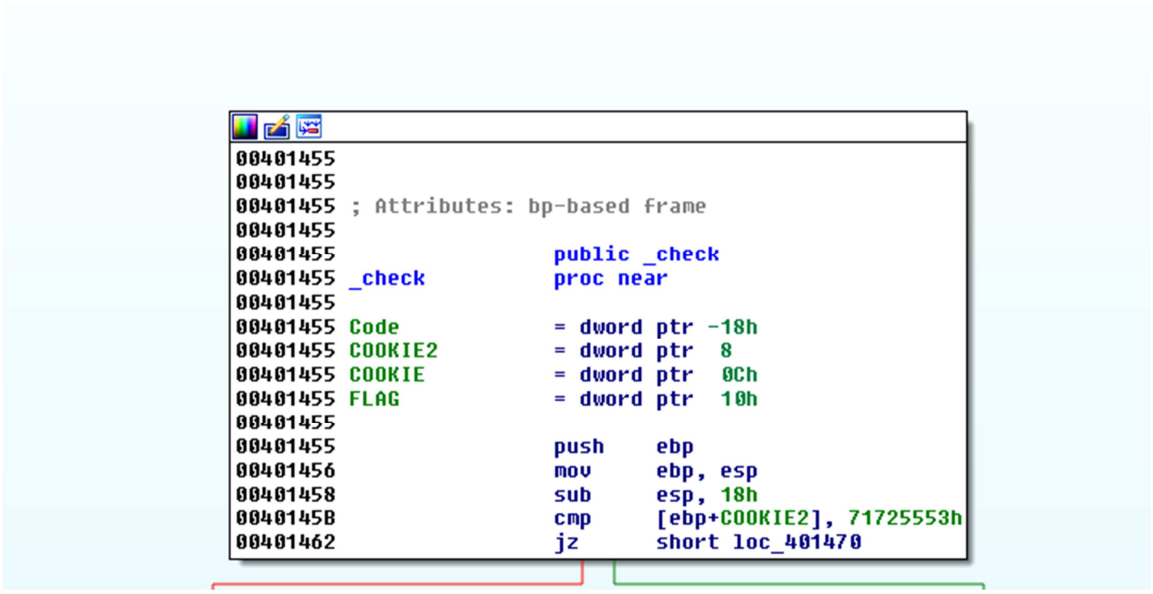
We see three arguments and a local variable.



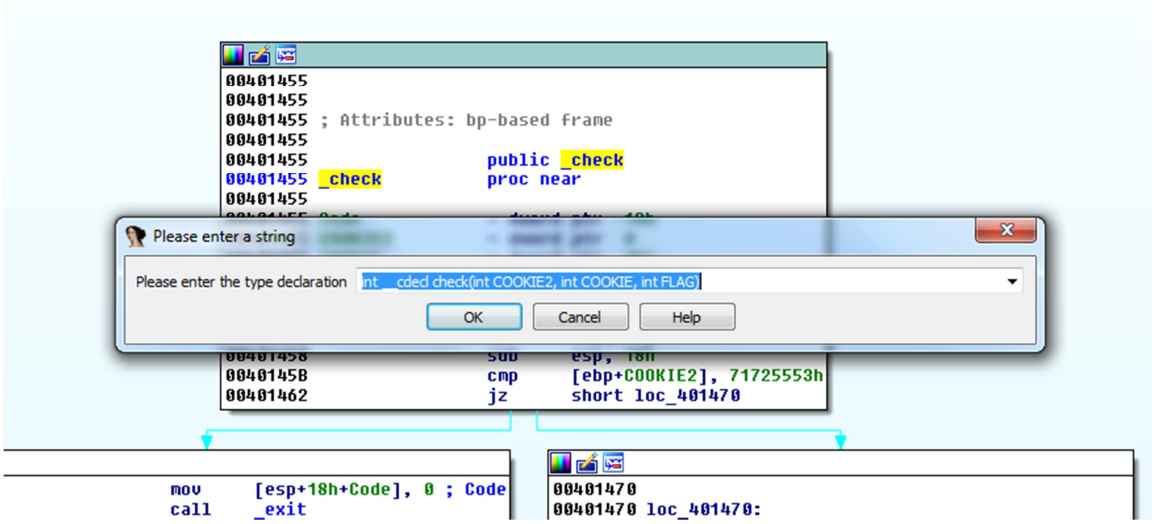
In the stack representation, this looks fine.



What is under the line will be arguments and above variables.

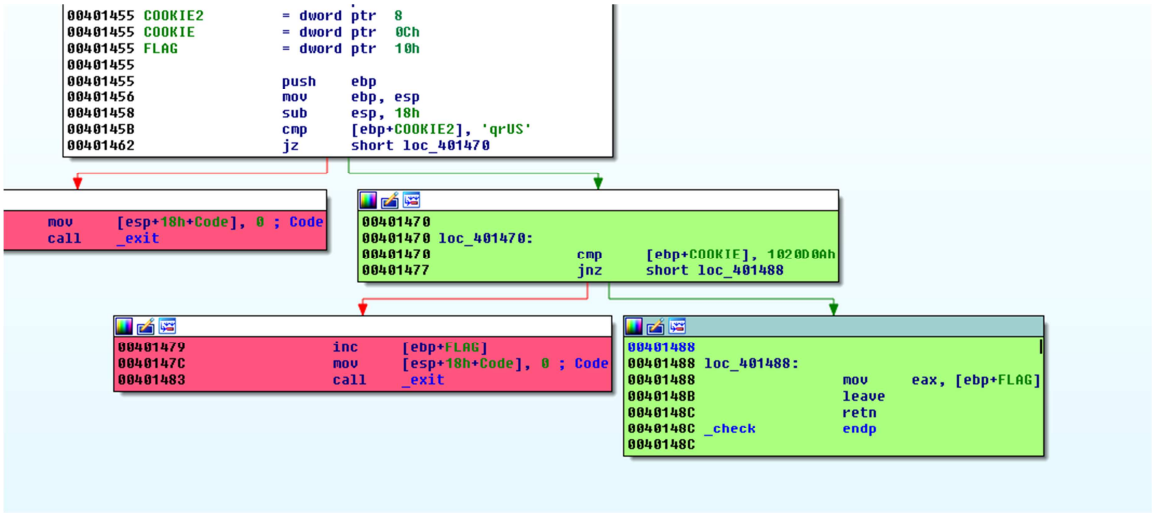


I rename them in the order they are passed and I **right click - set type** to propagate them to the main and see that everything is fine.



We see that it matches.

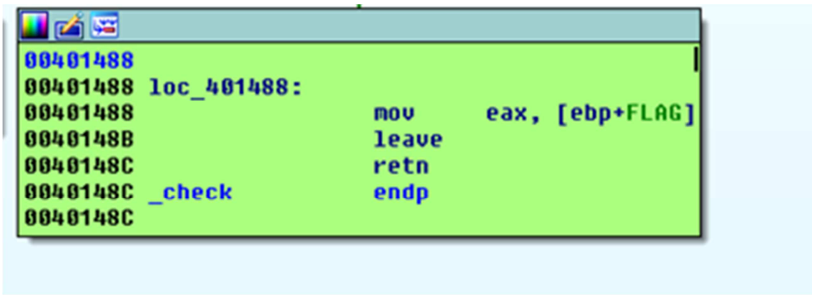
```
lea     eax, [esp+60h+Buffer]
mov     [esp], eax ; Buffer
call    _gets
mov     edx, [esp+60h+COOKIE]
mov     eax, [esp+60h+COOKIE2]
mov     ecx, [esp+60h+FLAG]
mov     [esp+8], ecx ; FLAG
mov     [esp+4], edx ; COOKIE
mov     [esp], eax ; COOKIE2
call    _check
mov     [esp+60h+var_8], eax
mov     eax, [esp+60h+var_8]
```

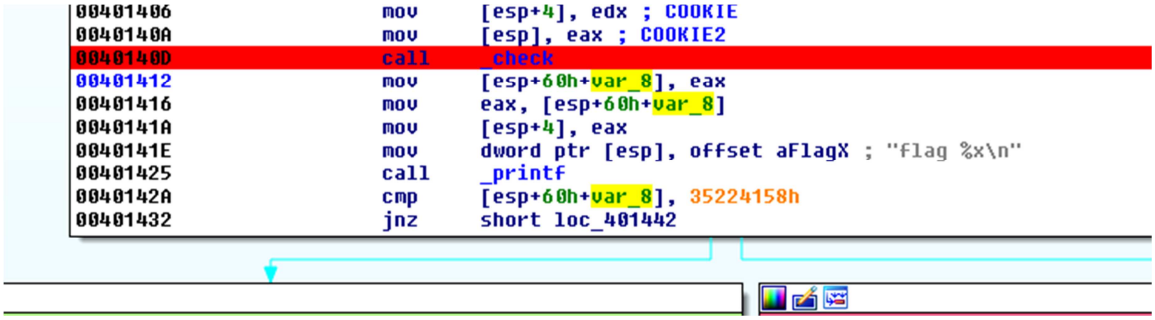


We see that the only way to get to the **ret** and not go to exit is to follow the green blocks and for this, cookie1 must be compared to **qrUS** and cookie2 to **0x10200d0a** and should not be equal to that value to exit the green block. Let's add those values In the script.

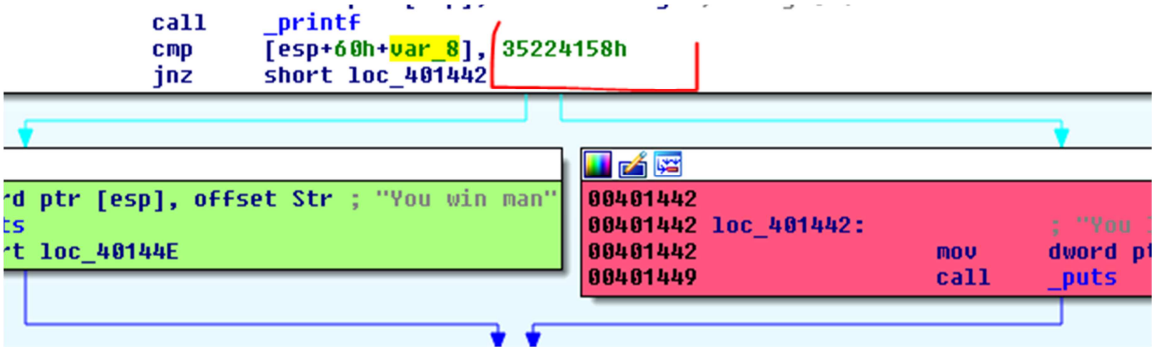
```
import ...
p = Popen([r'C:\Users\ricnar\Desktop\24-INTRODUCCION AL REVERSING CON ID

cookie2="SUrq"
cookie=struct.pack("<L",0x41424344)
```

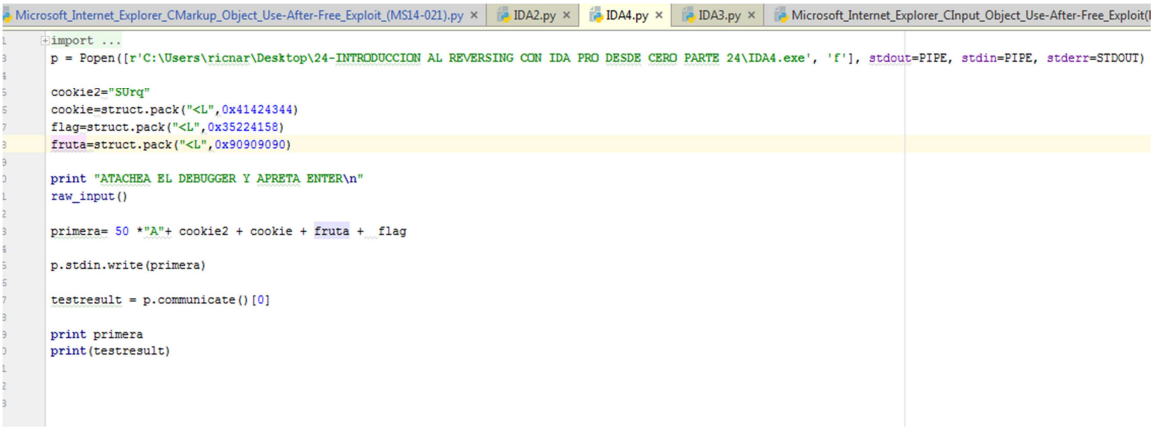
With that, we left the check function but there is still something missing, we see that the value returned by the check function is the flag value.



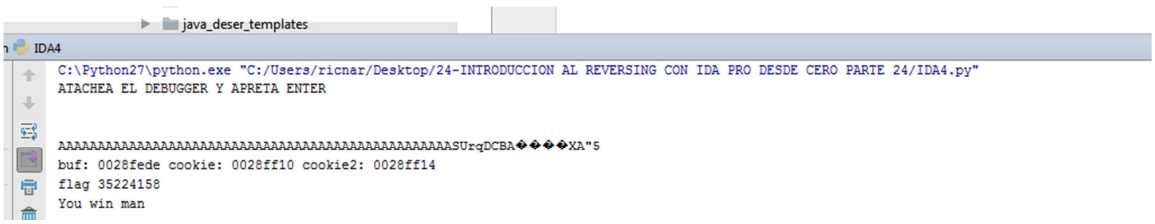


That is the value that it saves in var_8 and in the end it compares, so that flag must be that value 35224158.





Let’s try it.



So, it works. See you in the next part with more structures.

Ricardo Narvaja **Translated by: @IvinsonCLS**