

REVERSING WITH IDA PRO FROM SCRATCH

PART 25

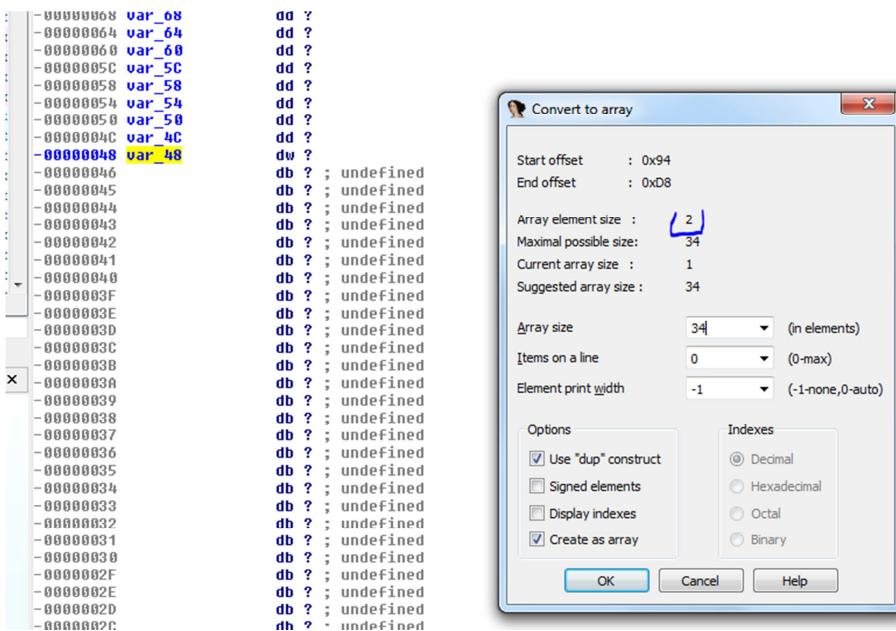
STRUCTURES

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 dwds, 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.

```

7 void * buf1;
8 FILE *f1;
9 int punt;
10 struct MyStruct
11 {
12     char * p_size; // largo 4
13     char size; // largo 1
14     int cookie; // largo 4
15     int leidos; // largo 4
16     int cookie2; // largo 4
17     int maxsize; // largo 4
18     int(*foo2)(char *); // largo 4
19     void * buf2; // largo 4
20     char buf[50]; // largo 50
21 };
22 void check(MyStruct * pvalores);
23 void check2(MyStruct * pvalores);
24
25 int main(int argc, char *argv[])
26 {
27     MyStruct valores;
28     MyStruct *pvalores;
29     valores.maxsize = 50;
30     valores.cookie = 0;
31     valores.cookie2 = 0;
32     valores.size = 0;
33 }

```

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 dwds 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`

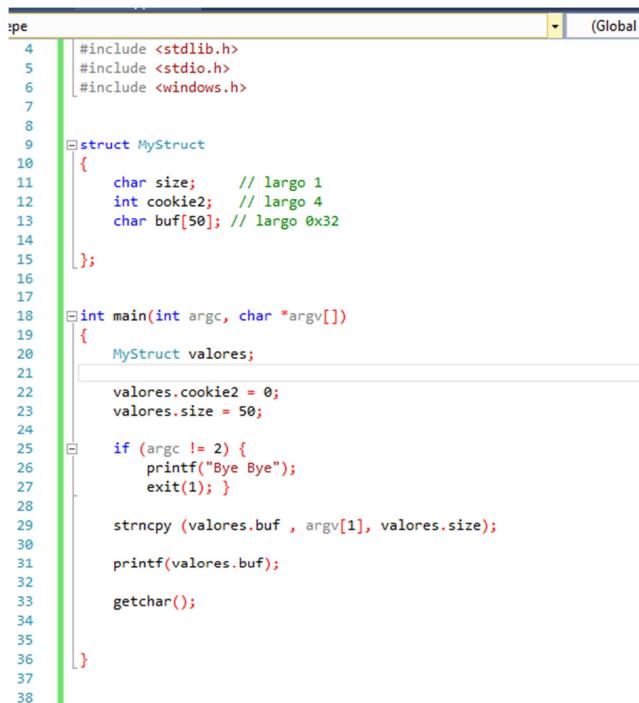
`void * buf2; // size 4 Pointer to a buffer`

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.



```
pe (Global)

4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <windows.h>
7
8
9 struct MyStruct
10 {
11     char size;      // largo 1
12     int cookie2;   // largo 4
13     char buf[50];  // largo 0x32
14 };
15
16
17
18 int main(int argc, char *argv[])
19 {
20     MyStruct valores;
21
22     valores.cookie2 = 0;
23     valores.size = 50;
24
25     if (argc != 2) {
26         printf("Bye Bye");
27         exit(1);
28     }
29
30     strncpy (valores.buf , argv[1], valores.size);
31
32     printf(valores.buf);
33
34     getchar();
35
36
37 }
38
```

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.



```
Copyright (c) 2000 Microsoft Corporation. Reservados todos los derechos.
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 **valores** 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 **valores**.

```
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.

The screenshot shows the IDA Pro interface with two windows. The top window is the assembly view, showing the `main` function. It includes annotations for variables: `valores` is annotated as a `MyStruct` pointer at `-40h`; its fields `var_4`, `argc`, `argv`, and `envp` are also annotated. The bottom window is the memory dump view, showing the memory starting at `00401034`. It displays the string "Bye Bye" followed by several null bytes. A red arrow points from the `valores` annotation in the assembly window to the `buf` field in the memory dump window, indicating the memory location where the copied data is stored.

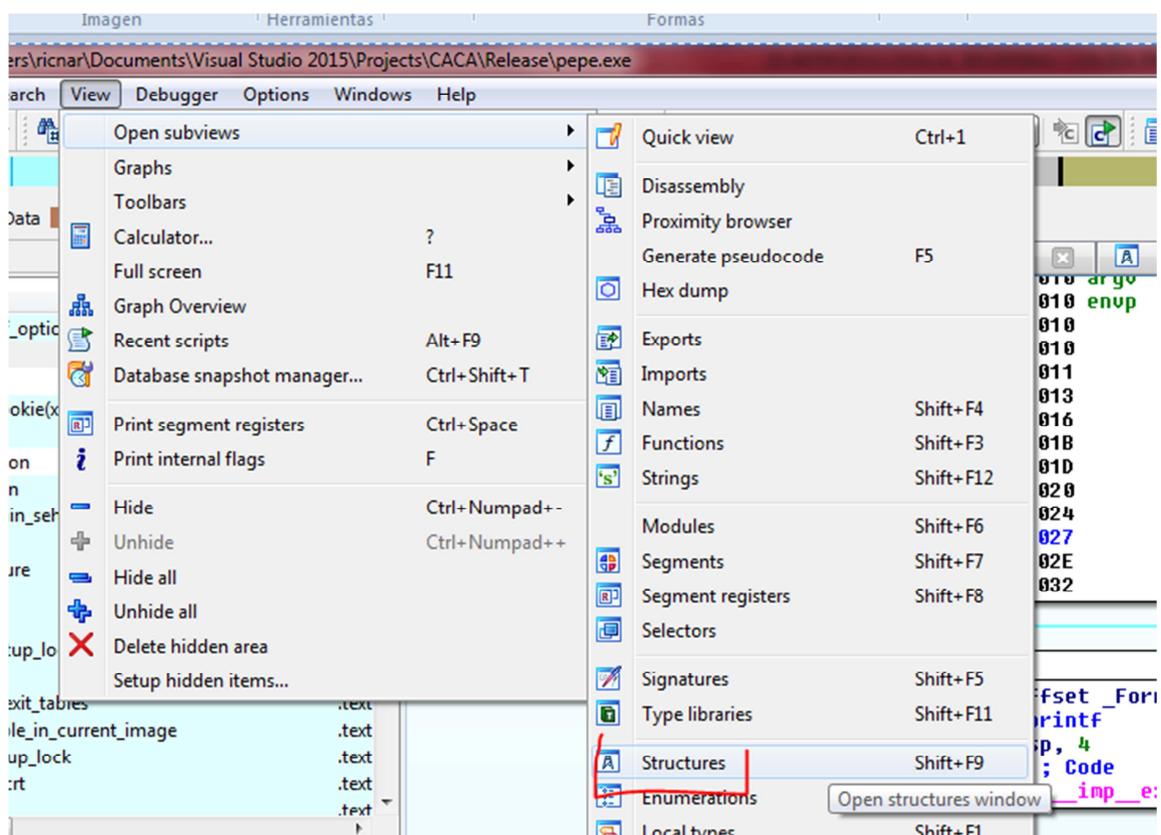
We see that when there are symbols everything is happiness. IDA detected values as a variable of type `MyStruct`, without problem. Even inside the code we see that it accesses the fields of the structure with its name perfectly.

```
00401024    mov    eax, [ebp+argv]
00401027    mov    [ebp+valores.cookie2], 0
0040102E    mov    [ebp+valores.size], 32h
00401032    jz     short $LN8
```

Also, here.

```
00401049 $LN8:    ; Count
00401049    push   32h
0040104B    push   dword ptr [eax+4] ; Source
0040104E    lea    eax, [ebp+valores.buf]
00401051    push   eax ; Dest
00401052    call   ds:_imp_strncpy
00401058    lea    eax, [ebp+valores.buf]
0040105B    push   eax ; _Format
0040105C    call   _printf
00401061    add    esp, 10h
00401064    call   ds:_imp_getchar
0040106A    mov    ecx, [ebp+var_4]
0040106D    xor    eax, eax
0040106F    xor    ecx, ebp ; cookie
00401071    call   @_security_check_cookie@4 ; __security_check_cookie(x)
00401076    mov    esp, ebp
00401078    pop    ebp
00401079    retn
00401079 _main      endp
```

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.



```

IDA View-A | Hex View-1 | Structures | Enums
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/* : create structure member (data/ascii/array)
00000000 ; N : rename structure or structure member
00000000 ; U : delete structure member
00000000 ; [ 0000003C BYTES. COLLAPSED STRUCT MyStruct. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000050 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 000002CC BYTES. COLLAPSED STRUCT _CONTEXT. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000070 BYTES. COLLAPSED STRUCT _FLOATING_SAVE_AREA. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION FT. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _FILETIME. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _LARGE_INTEGER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT $FAF74743FBE1C8632047CFB668F7028A. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 0000000C BYTES. COLLAPSED STRUCT _oneexit_table_t. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000018 BYTES. COLLAPSED STRUCT CPHEN_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT _EH3_EXCEPTION_REGISTRATION. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000040 BYTES. COLLAPSED STRUCT _IMAGE_DOS_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 0000005C BYTES. COLLAPSED STRUCT _IMAGE_LOAD_CONFIG_DIRECTORY32. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT __type_info_node. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT $E77D007F04512BE58055836487C8CD8. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT GUID. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 00000010 BYTES. COLLAPSED STRUCT _EH4_SCOPETABLE. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [ 0000000C BYTES. COLLAPSED STRUCT _EH4_SCOPETABLE_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]

```

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

```

0000000 ; U      : delete structure member
0000000 ;
0000000
0000008 MyStruct      struc ; (sizeof=0x3C, align=0x4, copyof_192) ; XREF: _main/r
0000000 size          db ? ; undefined
0000001             ; XREF: _main+1E/w
0000002             db ? ; undefined
0000003             db ? ; undefined
0000004 cookie2       dd ? ; undefined
0000005             ; XREF: _main+17/w
0000006 buf           db 50 dup(?) ; undefined
0000007             ; XREF: _main+3E/o _main+48/o
0000008             db ? ; undefined
0000009             db ? ; undefined
000000A ends
000000B
000000C MyStruct      ends
000000D ; [00000050 BYTES. COLLAPSED STRUCT _EXCEPTION_RECORD. PRESS CTRL-NUMPAD+ TO EXPAND]
000000E ; [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.

```

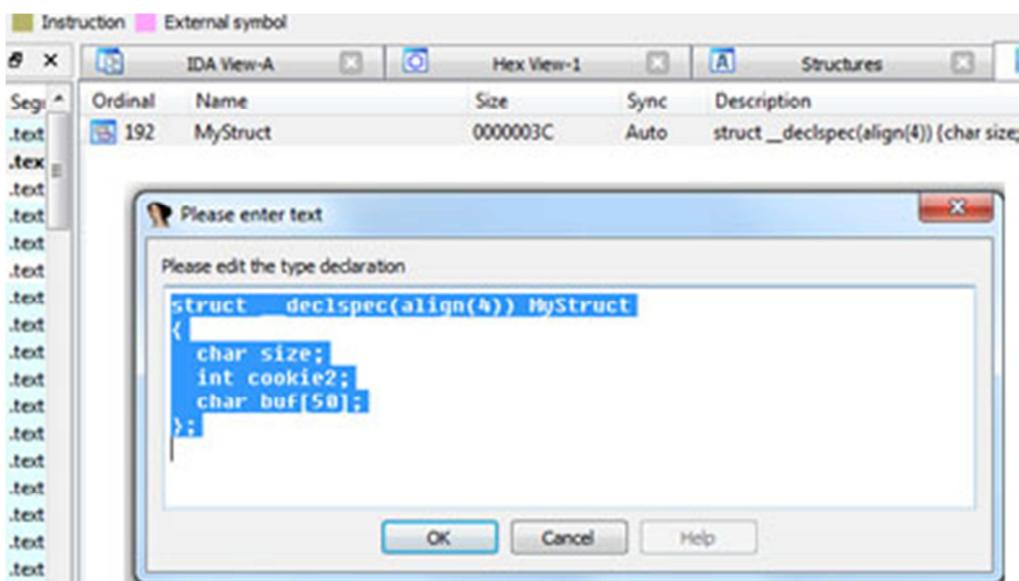
struct MyStruct
{
    char size; // size 1
    int cookie2; // size 4
    char buf[50]; // size 0x32
}

```

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.

Ordinal	Name	Size	Sync	Description
192	MyStruct	0000003C	Auto	struct __declspec(align(4)) {char size;int cookie2;char buf[50];}

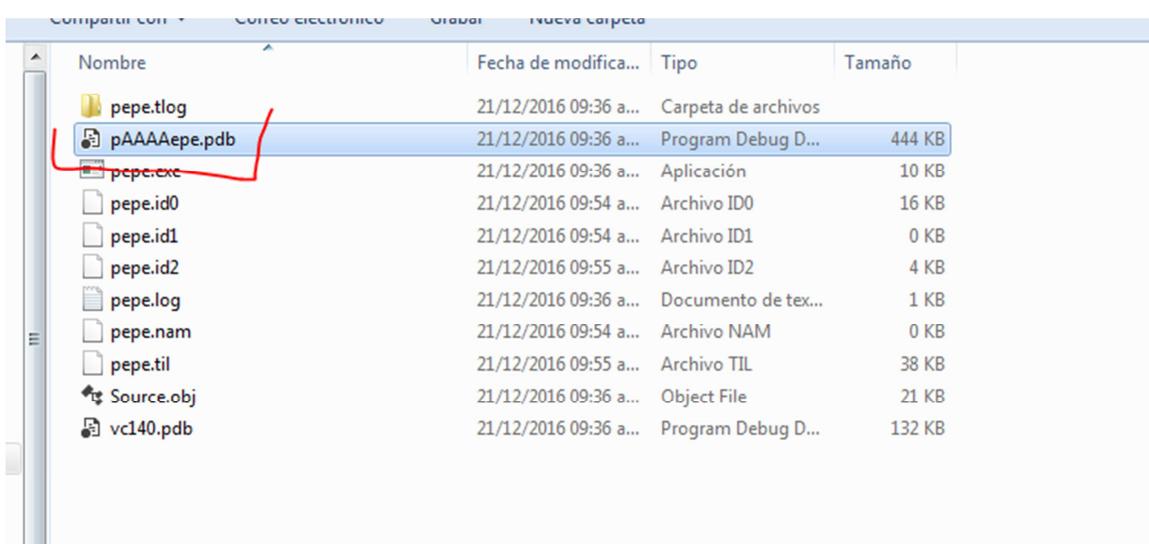
And I can see it by right clicking - EDIT.



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.

```

00401010 ; Attributes: bp-based frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **environ)
00401010 _main proc near
00401010
00401010     var_40h = byte ptr -40h
00401010     var_3Ch = dword ptr -3Ch
00401010     best = byte ptr -38h
00401010     var_4 = dword ptr -4h
00401010     argc = dword ptr 8
00401010     argv = dword ptr 0Ch
00401010     enup = dword ptr 10h
00401010
00401010     push    ebp
00401011     mov     ebp, esp
00401013     sub     esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     [ebp+var_4], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [ebp+var_3C], 0
0040102E     mov     [ebp+var_40], 32h
00401032     jz      short loc_401049

```



```

00401034     push    offset aByeBye ; "Bye Bye"
00401039     call    sub_401080
0040103E     add    esp, 4
00401041     push    1 ; Code
00401043     call    ds:_imp_exit

```

```

00401049 loc_401049:         ; Count
00401049     push    32h
00401049     push    dword ptr [eax+4] ; Source
00401049     lea     eax, [ebp+best]
0040104B     push    eax ; Dest
0040104E     call    ds:strncpy
00401051     push    eax, [ebp+best]
00401052     call    ds:strlen
00401058     lea     eax, [ebp+var_4]
0040105B     push    eax
0040105C     call    sub_401080
00401061     add    esp, 10h
00401064     call    ds:getchar
00401069     mov     ecx, [ebp+var_4]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebp
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mov     esp, ebp
00401078     pop    ebp

```

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.

```

00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010 var_40 = byte ptr -40h
00401010 var_3C = dword ptr -3Ch
00401010 Dest = byte ptr -38h
00401010 CANARY = dword ptr -4
00401010 argc = dword ptr 8
00401010 argv = dword ptr 0Ch
00401010 envp = dword ptr 10h
00401010
00401010 push ebp
00401010 mov ebp, esp
00401013 sub esp, 40h
00401016 mov eax, __security_cookie
00401018 xor eax, ebp
0040101D mov [ebp+CANARY], eax
00401020 cmp [ebp+argc], 2
00401024 mov eax, [ebp+argv]
00401027 mov [ebp+var_3C], 0
0040102E mov [ebp+var_40], 32h
00401032 jz short loc_401049

```

```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds:_imp_exit

```

```

00401049
00401049 loc_401049: ; Count
00401049 push 32h
0040104B push dword ptr [eax+4] ; Source
0040104E lea eax, [ebp+Dest]
00401051 push eax ; Dest
00401052 call ds:strncpy
00401058 lea eax, [ebp+Dest]
0040105C push eax
00401061 call sub_401080
00401061 add esp, 10h
00401064 call ds:getchar
0040106A mov ecx, [ebp+CANARY]
0040106D xor eax, eax
0040106F xor ecx, ebp
00401071 call @_security_check_cookie@4 ; __security_check_cookie(x)
00401076 mov esp, ebp

```

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

```

00401010 var_3C = dword ptr -3Ch
00401010 Dest = byte ptr -38h
00401010 CANARY = dword ptr -4
00401010 argc = dword ptr 8
00401010 argv = dword ptr 0Ch
00401010 envp = dword ptr 10h
00401010
00401010 push ebp
00401010 mov ebp, esp
00401013 sub esp, 40h
00401016 mov eax, __security_cookie
00401018 xor eax, ebp
0040101D mov [ebp+CANARY], eax
00401020 cmp [ebp+argc], 2
00401024 mov eax, [ebp+argv]
00401027 mov [ebp+var_3C], 0
0040102E mov [ebp+var_40], 32h
00401032 jz short loc_401049

```

```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds:_imp_exit

```

```

00401049
00401049 loc_401049: ; Count
00401049 push 32h
0040104B push dword ptr [eax+4] ; Source
0040104E lea eax, [ebp+Dest]
00401051 push eax ; Dest
00401052 call ds:strncpy
00401058 lea eax, [ebp+Dest]
0040105B push eax
00401061 call sub_401080
00401061 add esp, 10h
00401064 retn

```

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.

```

00401010 envp          = oword ptr 100
00401010    push  ebp
00401011    mov   ebp, esp
00401013    sub   esp, 40h
00401016    mov   eax, __security_cookie
00401018    xor   eax, ebp
0040101D    mov   [ebp+CANARY], eax
00401020    cmp   [ebp+argc], 2
00401024    mov   eax, [ebp+argv]
00401027    mov   [ebp+var_3C], 0
0040102E    mov   [ebp+var_40], 32h
00401032    jz    short loc_401049

00401034    push  offset aByeBye ; "Bye Bye"
00401039    call   sub_401080
0040103E    add   esp, 4
00401041    push  1 ; Code
00401043    call   ds:_imp.exit

00401049 loc_401049:      ; Count
00401049    push  32h
0040104B    push  dword ptr [eax+4] ; Source
0040104B    lea   eax, [ebp+Dest]
0040104E    push  eax : Dest
00401051    push  eax
00401052    call   ds:strncpy
00401058    lea   eax, [ebp+Dest]
0040105B    push  eax
0040105C    call   sub_401080
00401061    add   esp, 10h
00401064    call   ds:getchar
0040106A    mov   ecx, [ebp+CANARY]
0040106D    xor   eax, eax
0040106F    xor   eax, ebp
00401071    call   @_security_check_cookie@4 ; __security_check_cookie(x)
00401076    mov   esp, ebp
00401078    pop   ebp
00401079    retn
00401079 _main           endp

```

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.

```

00401010
00401011    push  ebp
00401013    mov   ebp, esp
00401016    sub   esp, 40h
00401018    mov   eax, __security_cookie
0040101D    xor   eax, ebp
00401020    mov   [ebp+CANARY], eax
00401024    cmp   [ebp+argc], 2
00401027    mov   eax, [ebp+argv]
0040102E    mov   [ebp+var_3C], 0
00401032    mov   [ebp+var_40], 32h
00401032    jz    short loc_401049

00401049 loc_401049:      ; Count
00401049    push  32h
0040104B    push  dword ptr [eax+4] ; Source
0040104B    lea   eax, [ebp+Dest]
0040104E    push  eax : Dest
00401051    push  eax
00401052    call   ds:strncpy
00401058    lea   eax, [ebp+Dest]
0040105B    push  eax
0040105C    call   sub_401080
00401061    add   esp, 10h
00401064    call   ds:getchar
0040106A    mov   ecx, [ebp+CANARY]
0040106D    xor   eax, eax
0040106F    xor   eax, ebp
00401071    call   @_security_check_cookie@4 ; __security_check_cookie(x)
00401076    mov   esp, ebp
00401078    pop   ebp
00401079    retn
00401079 _main           endp

```

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

```

00401010      sub    esp, 40h
00401016      mov    eax, __security_cookie
0040101B      xor    eax, ebp
0040101D      mov    [ebp+CANARY], eax
00401020      cmp    [ebp+argc], 2
00401024      mov    eax, [ebp+argv]
00401027      mov    [ebp+var_3C], 0
0040102E      mov    [ebp+size], 32h
00401032      jz    short loc_401049

sh  offset aByeBye ; "Bye Bye"
11  sub_401080
d   esp, 4
sh  1 ; Code
11  ds:_imp_exit

```

Assembly pane (top):

```

00401049      mov    eax, size
00401049      loc_401049
00401048      db    ?
00401048      db    ?, undefined
0040104E      db    ?, undefined
0040104E      db    ?, undefined
00401051      ...
00401052      call   os::strcmp
00401058      lea    eax, [ebp+Dest]
00401058      push   eax

```

Memory dump pane (bottom):

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.

```

00401010      push   ebp
00401011      mov    ebp, esp
00401013      sub    esp, 40h
00401016      mov    eax, __security_cookie
0040101B      xor    eax, ebp
0040101D      mov    [ebp+CANARY], eax
00401020      cmp    [ebp+argc], 2
00401024      mov    eax, [ebp+argv]
00401027      mov    [ebp+var_3C], 0
0040102E      mov    [ebp+size], 32h
00401032      jz    short loc_401049

push  offset aByeBye ; "Bye Bye"
call  sub_401080
add   esp, 4
push  1 ; Code
call  ds:_imp_exit

```

Assembly pane (top):

Context menu for 'size' (bottom):

- Rename N
- Use standard symbolic constant
- byte ptr [ebp-40h]** Q (highlighted)
- byte ptr [ebp-64] H
- byte ptr [ebp-1000] B
- byte ptr [ebp-1000000b] R
- byte ptr [ebp-'@'] Alt+F1
- Manual...
- Undefine operand
- Edit function... Alt+P
- Hide Ctrl+Numpad+-
- Text view
- Proximity browser Numpad+-
- Undefine II

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

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010 size      = byte ptr -40h
00401010 var_3C      = dword ptr -3Ch
00401010 Dest      = byte ptr -38h
00401010 CANARY    = dword ptr -4
00401010 argc      = dword ptr 8
00401010 argv      = dword ptr 0Ch
00401010 envp      = dword ptr 10h
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 sub     esp, 40h
00401016 mov     eax, __security_cookie
00401018 xor     eax, ebp
0040101D mov     [ebp+CANARY], eax
00401020 cmp     [ebp+argc], 2
00401024 mov     eax, [ebp+argv]
00401027 mov     [ebp+var_3C], 0
0040102E mov     [ebp+size], 32h
00401032 jz      short loc_401049

```



```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds: _jmp_exit

```



```

00401049 loc_401049:          ; Count
00401049 push    32h
00401049 push    dword ptr [eax+4] ; Source
00401049 lea     eax, [ebp+Dest]
00401051 push    eax ; Dest
00401052 call    ds:strncpy
00401058 lea     eax, [ebp+Dest]
0040105B push    eax
0040105C call    sub_401080
00401061 add    esp, 10h
00401064 call    ds:getchar
0040106A mov     ecx, [ebp+CANARY]
0040106D xor     eax, eax
0040106F xor     ecx, ebp
00401071 call    @ security_check_cookie@4 ; security_check_cookie(x)

```

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.

```

00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010
00401010 size      = byte ptr -40h
00401010 cookie2   = dword ptr -3Ch
00401010 Dest      = byte ptr -38h
00401010 CANARY    = dword ptr -4
00401010 argc      = dword ptr 8
00401010 argv      = dword ptr 0Ch
00401010 envp      = dword ptr 10h
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 sub     esp, 40h
00401016 mov     eax, __security_cookie
00401018 xor     eax, ebp
0040101D mov     [ebp+CANARY], eax
00401020 cmp     [ebp+argc], 2
00401024 mov     eax, [ebp+argv]
00401027 mov     [ebp+cookie2], 0
0040102E mov     [ebp+size], 32h
00401032 jz      short loc_401049

```



```

push offset aByeBye ; "Bye Bye"
call sub_401080
add esp, 4
push 1 ; Code
call ds: _jmp_exit

```



```

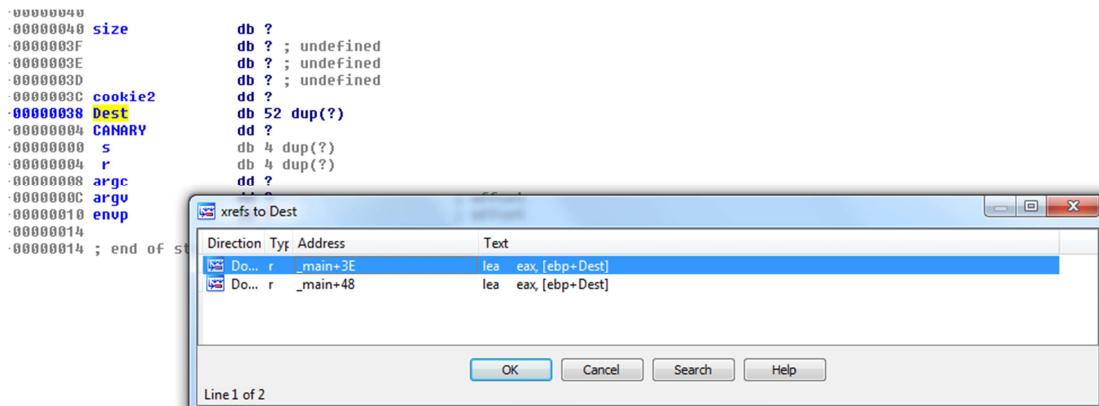
00401049 loc_401049:          ; Count
00401049 push    32h
00401049 push    dword ptr [eax+4] ; Source
00401049 lea     eax, [ebp+Dest]
00401051 push    eax ; Dest
00401052 call    ds:strncpy
00401058 lea     eax, [ebp+Dest]
0040105B push    eax
0040105C call    sub_401080
00401061 add    esp, 10h
00401064 call    ds:getchar
0040106A mov     ecx, [ebp+CANARY]
0040106D xor     eax, eax

```

Let's see the representation of the stack.

```
t -00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
t -00000040 ;
t -00000040
t -00000040 size db ?
t -0000003F db ? ; undefined
t -0000003E db ? ; undefined
t -0000003D db ? ; undefined
t -0000003C cookie2 dd ?
t -00000038 Dest db ?
t -00000037 db ? ; undefined
t -00000036 db ? ; undefined
t -00000035 db ? ; undefined
t -00000034 db ? ; undefined
t -00000033 db ? ; undefined
t -00000032 db ? ; undefined
t -00000031 db ? ; undefined
t -00000030 dh ? - 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, strcpy and the LEA obtains it.

The screenshot shows a debugger interface with assembly code in the background and a 'Convert to array' dialog box in the foreground.

Assembly Code (Background):

```

Seg1:
- 00000040 ; N      : rename
- 00000040 ; U      : undefine
.text = 00000040 ; Use data definition commands to create local variables and function arguments.
.text = 00000040 ; Two special fields "r" and "s" represent return address and saved registers.
.text = 00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
- 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
- 0000002F          db ? ; undefined
- 0000002E          db ? ; undefined
- 0000002D          db ? ; undefined
- 0000002C          db ? ; undefined
- 0000002B          db ? ; undefined
- 0000002A          db ? ; undefined
- 00000029          db ? ; undefined
- 00000028          db ? ; undefined
- 00000027          db ? ; undefined
- 00000026          db ? ; undefined
- 00000025          db ? ; undefined
- 00000024          db ? ; undefined
- 00000023          db ? ; undefined
- 00000022          db ? ; undefined
- 00000021          db ? ; undefined
- 00000020          db ? ; undefined
- 0000001F          db ? ; undefined
- 0000001E          db ? ; undefined
- 0000001D          db ? ; undefined
- 0000001C          db ? ; undefined
- 0000001B          db ? ; undefined
- 0000001A          db ? ; undefined
- 00000019          db ? ; undefined
- 00000018          db ? ; undefined
- 00000017          db ? ; undefined

```

Convert to array Dialog Box (Foreground):

Start offset	: 0x8
End offset	: 0x3C
Array element size	: 1
Maximal possible size	: 52
Current array size	: 1
Suggested array size	: 52
Array size	<input type="text" value="52"/> (in elements)
Items on a line	<input type="text" value="0"/> (0-max)
Element print width	<input type="text" value="-1"/> (-1-none,0-auto)
Options	<input checked="" type="checkbox"/> Use "dup" construct <input type="checkbox"/> Signed elements <input type="checkbox"/> Display indexes <input checked="" type="checkbox"/> Create as array
Indexes	<input checked="" type="radio"/> Decimal <input type="radio"/> Hexadecimal <input type="radio"/> Octal <input type="radio"/> Binary
Buttons:	
OK	Cancel
Help	

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

The screenshot shows a debugger interface with assembly code in the background.

Assembly Code (Background):

```

.text = 00000040 ; use data definition commands to create local variables ar
- 00000040 ; Two special fields "r" and "s" represent return address
- 00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
- 00000040 ;
- 00000040 size      db ?
- 0000003F          db ? ; undefined
- 0000003E          db ? ; undefined
- 0000003D          db ? ; undefined
- 0000003C cookie2   dd ?
- 00000038 Dest      db 52 dup(?)
- 00000038 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

```

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`.

The screenshot shows the assembly view of the OllyDbg debugger. The assembly code is:

```

00401049  loc_401049:    push    ; Count
00401049                 32h
00401049                 push    dword ptr [eax+4] ; Source
0040104B                 push    eax, [ebp+Dest]
0040104E                 lea     eax, [ebp+Dest]
00401051                 push    eax ; Dest
00401052                 call    ds:strncpy

```

A red circle highlights the instruction `push 32h`. Below the assembly, the debugger's output window shows:

```

Output window
Python> { 0x32
50
Python

```

A red arrow points from the value `50` in the output window back to the highlighted `push 32h` instruction.

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.

The screenshot shows the stack variable dump in the OllyDbg debugger. The stack layout is as follows:

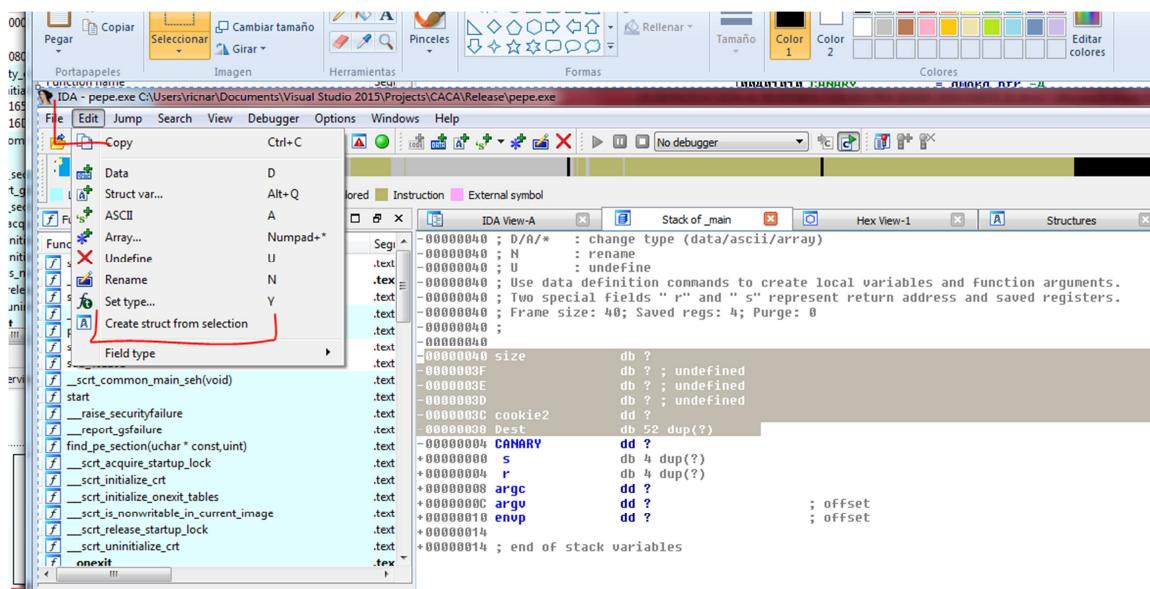
Address	Variable	Type	Value
-00000040		dw	?
-00000040	size	db	?
-0000003F		db	?
-0000003E		db	?
-0000003D		db	?
-0000003C	cookie2	dd	?
-00000038 Dest		db	52 dup(?)
-00000034	CANARY	dd	?
+00000000	s	db	4 dup(?)
+00000004	r	db	4 dup(?)
+00000008	argc	dd	?
+0000000C	argv	dd	?
+00000010	envp	dd	?
+00000014			; offset
+00000014			; offset
+00000014 ; end of stack variables			

Red boxes highlight the `size` variable at address `-00000038`, the `Dest` variable at address `-00000038`, and the `CANARY` variable at address `-00000034`.

That will possibly cover that.

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

```
.text 00000040 , 0 ; .STRUCTURE
.text -00000040 ; Use data definition commands to create local variables and function arguments
.text -00000040 ; Two special fields "r" and "s" represent return address and saved registers
.text -00000040 ; Frame size: 40; Saved regs: 4; Purge: 0
.text -00000040 ;
.text -00000040 size db ?
.text -0000003F db ? ; undefined
.text -0000003E db ? ; undefined
.text -0000003D db ? ; undefined
.text -0000003C cookie2 dd ?
.text -00000038 Dest db 52 dup(?)
.text -00000034 CANARY dd ?
.text +00000000 s db 4 dup(?)
.text +00000004 r db 4 dup(?)
.text +00000008 argc dd ?
.text +0000000C argv dd ? ; offset
.text +00000010 envp dd ? ; offset
.text +00000014
.text +00000014 ; end of stack variables
.text
```



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 struct_0
00000000 size
00000001 db ?
00000002 db ? ; undefined
00000003 db ? ; undefined
00000004 cookie2
00000008 Dest
0000003C struct_0
0000003C ends

```

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

```

text 00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ TO EXP
text 00000000 ; [ 00000008 BYTES. COLLAPSED UNION _SLIST_HEADER. PRESS CTRL-NUMPAD+ TO EXPAND]
text 00000000 ; [ 00000008 BYTES. COLLAPSED STRUCT _SLIST_HEADER:$83AF6D9DC8E3B10431D79B3849571
text 00000000 ; [ 00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO EXP
text 00000000 ;
text 00000000 ;
text 00000000 MyStruct
text 00000000 size
text 00000001 db ?
text 00000002 db ? ; undefined
text 00000003 db ? ; undefined
text 00000004 cookie2
text 00000008 Dest
text 0000003C MyStruct
text 0000003C ends

```

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

```

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

```

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

```

00401010 ; Attributes: bp-based Frame
00401010 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401010 _main proc near
00401010     .values = MyStruct ptr -40h
00401010     CANARY = dword ptr -4
00401010     argc = dword ptr 8
00401010     argv = dword ptr 0Ch
00401010     envp = dword ptr 10h
00401010
00401010     push    ebp
00401010     mov     ebp, esp
00401013     sub    esp, 40h
00401016     mov     eax, __security_cookie
00401018     xor     eax, ebp
0040101D     mov     eax, [ebp+CANARY], eax
00401020     cmp     [ebp+argc], 2
00401024     mov     eax, [ebp+argv]
00401027     mov     [eax+__security_cookie2], 0
0040102E     mov     [ebp+valores.size], 32h
00401032     jz      short loc_401049

01034     push    offset aByeBye ; "Bye Bye"
01039     call    sub_401080
0103E     add    esp, 4
01041     push    1 ; Code
01043     call    ds:_imp_exit

00401049 loc_401049: ; Count
00401049     push    32h
00401049     push    dword ptr [eax+4] ; Source
00401049     lea     eax, [ebp+valores.Dest]
00401049     push    eax ; Dest
00401051     call    ds:_strncpy
00401052     lea     eax, [ebp+valores.Dest]
00401053     push    eax
0040105C     call    sub_401080
00401061     add    esp, 10h
00401064     call    ds:_getchar
0040106A     mov     ecx, [ebp+CANARY]
0040106D     xor     eax, eax
0040106F     xor     ecx, ebp
00401071     call    @_security_check_cookie@4 ; __security_check_cookie(x)
00401076     mnu    esp, ebp
00401078     pop    ebp
00401079     retn
00401079 _main endp
00401079

```

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.

```

00401290 ; Attributes: bp-based Frame
00401290 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401290     public _main
00401290 _main proc near
00401290
00401290     Format = dword ptr -78h
00401290     var_74 = dword ptr -74h
00401290     var_70 = dword ptr -70h
00401290     var_6C = dword ptr -6Ch
00401290     TEMP = dword ptr -5Ch
00401290     Buffer = byte ptr -58h
00401290     COOKIE = dword ptr -14h
00401290     FLAG = dword ptr -10h
00401290     COOKIE2 = dword ptr -8h
00401290     argc = dword ptr 8
00401290     argv = dword ptr 0Ch
00401290     envp = dword ptr 10h
00401290
00401290     push    ebp
00401291     mov     ebp, esp
00401293     sub    esp, 78h
00401296     and    esp, 0FFFFFFF0h
00401299     mov     eax, 0
0040129E     add    eax, 0Fh
004012A1     add    eax, 0Fh
004012A4     shr    eax, 4
004012A7     shl    eax, 4
004012AA     mov     [ebp+TEMP], eax
004012AD     mov     eax, [ebp+TEMP]
004012B0     call    alloca

```

The TEMP variables above are added by the compiler.

```
-00000005C TEMP          dd ?
-000000058 Buffer        db 68 dup(?)
-000000014 COOKIE         dd ?
-000000010 FLAG           dd ?
-00000000C COOKIE2        dd ?
-000000008
-000000007
-000000006
-000000005
-000000004
-000000003
-000000002
-000000001
+000000000 s             db 4 dup(?)
+000000004 r             db 4 dup(?)
+000000008 argc          dd ?
+00000000C argv          dd ?                      ; offset
+000000010 envp          dd ?                      ; offset
+000000014 ; end of stack variables
```

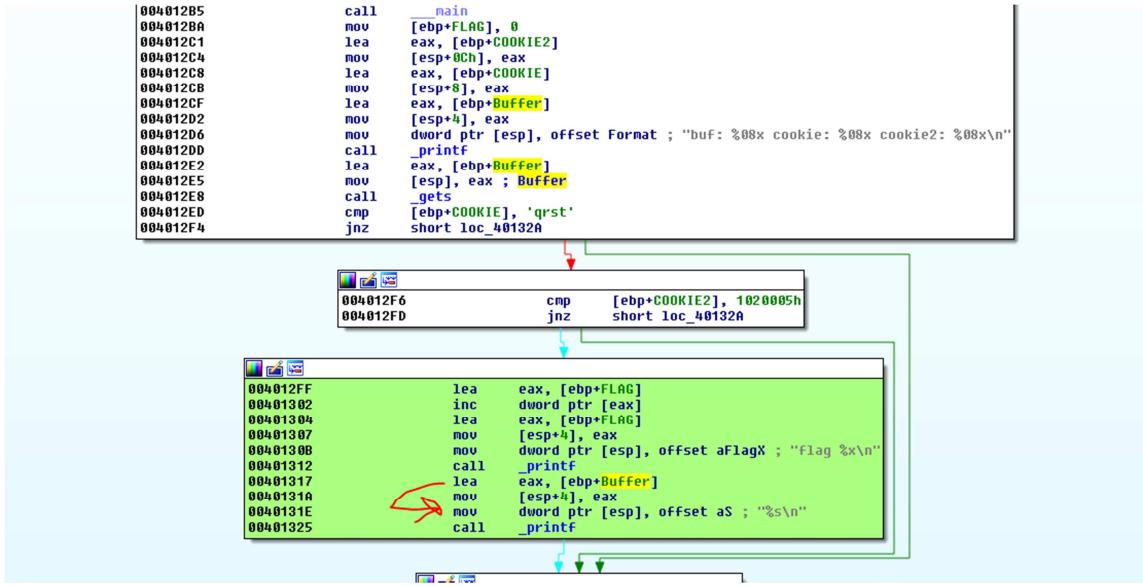
There, I see the representation of the stack and the variables that I have to overwrite COOKIE and COOKIE2, we know that it gets into gets, so it is vulnerable. There is no limit to the number of bytes you input.

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.

```

-00000005E      db ? ; undefined
-00000005D      db ? ; undefined
-00000005C      TEMP
-000000058      Buffer
-000000014      COOKIE
-000000019      FLAG
-00000000C      COOKIE2
-000000008      db ? ; undefined
-000000007      db ? ; undefined
-000000006      db ? ; undefined
-000000005      db ? ; undefined

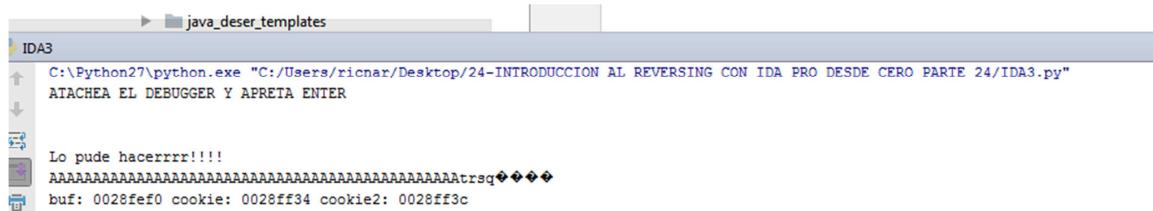
```

```

1 import ...
2 p = Popen(['C:\Users\ricmar\Desktop\124-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24\IDA3.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)
3
4
5 cookie="trsq"
6 cookie2=struct.pack("<L",0x1020005)
7 flag=struct.pack("<L",0x90909090)
8 string="Lo pude hacerrrr!!!!\n"
9
10 print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
11 raw_input()
12
13 primera=string + (60 -(len(string))) *"A"+ cookie + flag + cookie2
14 p.stdin.write(primer)
15
16 testresult = p.communicate()[0]
17
18 print primera
19 print(testresult)
20

```

If I run it.



The screenshot shows the IDA Pro interface with a file named "java_deser_templates". The assembly code window displays the following command:

```
C:\Python27\python.exe "C:/Users/ricnar/Desktop/24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24/IDA3.py"
```

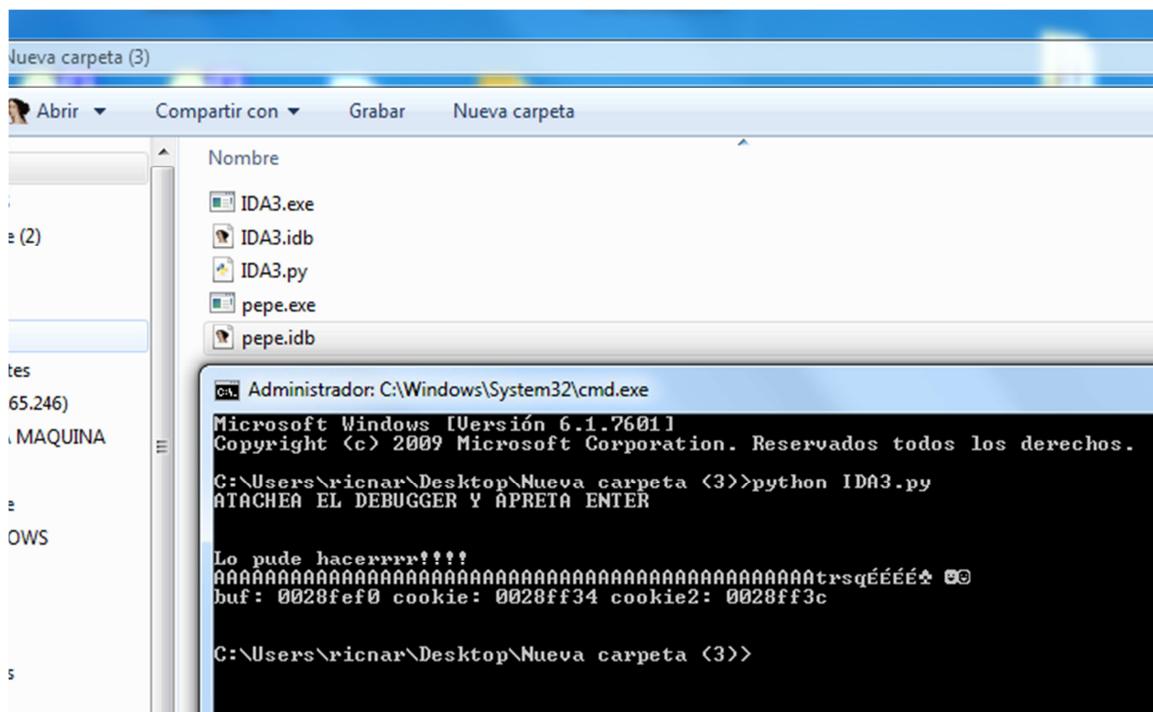
Below the assembly code, the debugger window shows the output of the program:

```
ATACHEA EL DEBUGGER Y APRETA ENTER
```

```
Lo pude hacerrrr!!!!
AAAAAAAAAAAAAAAAAAAAAAAAtrsq♦♦♦♦
buf: 0028fef0 cookie: 0028ff34 cookie2: 0028ff3c
```

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"



The screenshot shows a Windows File Explorer window titled "Nueva carpeta (3)" containing files: IDA3.exe, IDA3.idb, IDA3.py, pepe.exe, and pepe.idb.

Below the File Explorer is a terminal window titled "Administrador: C:\Windows\System32\cmd.exe". The terminal shows the following command and its output:

```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\ricnar\Desktop\Nueva carpeta <3>>python IDA3.py
ATACHEA EL DEBUGGER Y APRETA ENTER

Lo pude hacerrrr!!!!
AAAAAAAAAAAAAAAAAAAAAAAAtrsqÉÉÉÉ@ @@
buf: 0028fef0 cookie: 0028ff34 cookie2: 0028ff3c

C:\Users\ricnar\Desktop\Nueva carpeta <3>>
```

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

```

00401380    push  ebp
00401381    mov   ebp, esp
00401383    and   esp, 0FFFFFFF0h
00401386    sub   esp, 60h
00401389    call  main
0040138E    mov   [esp+60h+var_4], 0
004013C6    lea   eax, [esp+60h+var_C]
004013CA    mov   [esp+60h+var_54], eax
004013CE    lea   eax, [esp+60h+var_10]
004013D2    mov   [esp+60h+var_58], eax
004013D6    lea   eax, [esp+60h+Buffer]
004013D9    mov   [esp+60h+var_5C], eax
004013DE    mov   [esp+60h+Format], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004013E5    call  _printf
004013EA    lea   eax, [esp+60h+Buffer]
004013EE    mov   [esp+60h+Format], eax ; Buffer
004013F1    call  _gets
004013F6    mov   edx, [esp+60h+var_C]
004013FA    mov   eax, [esp+60h+var_10]
004013FE    mov   ecx, [esp+60h+var_4]
00401402    mov   [esp+60h+var_58], ecx
00401406    mov   [esp+60h+var_5C], edx
0040140B    mov   [esp+60h+Format], eax
0040140D    call  _check
00401412    mov   [esp+60h+var_8], eax
00401416    mov   eax, [esp+60h+var_8]
0040141A    mov   [esp+60h+var_50], eax
0040141E    mov   [esp+60h+Format], offset aFlagX ; "Flag %x\n"
00401425    call  _printf
00401428    cmp   [esp+60h+var_8], 35224158h
00401432    jnz   short loc_40144E

```

Memory Dump Windows:

- 00401434: mov [esp+60h+Format], offset Str ; "You win man"
- 0040143B: call _puts
- 00401440: jmp short loc_40144E
- 00401442: 00401442 loc_40144E: ; "You lose man"
- 00401443: mov [esp+60h+Format], offset aYouLoseMan
- 00401444: call _puts

Let's reverse the main.

```

00401380    push  ebp
00401381    mov   ebp, esp
00401383    and   esp, 0FFFFFFF0h
00401386    sub   esp, 60h
00401389    call  main
0040138E    mov   [esp+60h+var_4], 0
004013C6    lea   eax, [esp+60h+var_C]
004013CA    mov   [esp+60h+var_54], eax
004013CE    lea   eax, [esp+60h+var_10]
004013D2    mov   [esp+60h+var_58], eax
004013D6    lea   eax, [esp+60h+Buffer]
004013D9    mov   [esp+60h+var_5C], eax
004013DE    mov   dword ptr [esp], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004013E5    call  _printf

```

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.

```

00401380    mov   uwuwo_pcr [esp], 0FF5E FORMAT , 0000 0000 COOKIE, 0000 COOKIE2, 0
004013E5    call  _printf
004013EA    lea   eax, [esp+60h+Buffer]
004013EE    mov   [esp], eax ; Buffer
004013F1    call  _gets

```

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

The screenshot shows a debugger interface with a memory dump window on the left and a 'Convert to array' dialog box on the right. The memory dump shows various memory locations, many of which are marked as undefined ('db ?'). A specific location at address 0x00000042 is highlighted and labeled 'Buffer'. The 'Convert to array' dialog has the following settings:

- Start offset: 0x1E
- End offset: 0x50
- Array element size: 1
- Maximal possible size: 50
- Current array size: 1
- Suggested array size: 50
- Array size: 50 (in elements)
- Items on a line: 0 (0-max)
- Element print width: -1 (-1-none,0-auto)
- Use "dup" construct
- Signed elements
- Display indexes
- Create as array
- Indexes: Decimal selected

We see that it is 50 decimal.

The screenshot shows a debugger interface with a variable dump window. Several variables are listed, each with its name and type. Annotations with red boxes highlight specific entries:

- 'Buffer' at address 0x00000042 is annotated with a red box.
- 'COOKIE' at address 0x00000010 is annotated with a red box.
- 'COOKIE' at address 0x0000000C is annotated with a red box.
- 'var_8' at address 0x00000008 is annotated with a red box.
- 'var_4' at address 0x00000004 is annotated with a red box.

And then there are two cookies of 4 bytes each.

```

004013B0 var_4          = dword ptr -4
004013B0 argc           = dword ptr 8
004013B0 argv           = dword ptr 0Ch
004013B0 envp           = dword ptr 10h
004013B0
004013B0     push    ebp
004013B1     mov     ebp, esp
004013B3     and     esp, 0FFFFFFF0h
004013B6     sub     esp, 60h
004013B9     call    __main
004013BE     mov     [esp+60h+var_4], 0
004013C6     lea     eax, [esp+60h+COOKIE]
004013CA     mov     [esp+0Ch], eax
004013CE     lea     eax, [esp+60h+COOKIE2]
004013D2     mov     [esp+8], eax
004013D6     lea     eax, [esp+60h+Buffer]
004013DA     mov     [esp+4], eax
004013DE     mov     dword ptr [esp], offset Format ; "buf: %08x cookie"
004013E5     call    _printf
004013EA     lea     eax, [esp+60h+Buffer]
004013EE     mov     [esp], eax ; Buffer
004013F1     call    _gets
004013F6     mov     edx, [esp+60h+COOKIE]
004013FA     mov     eax, [esp+60h+COOKIE2]
004013FE     mov     ecx, [esp+60h+var_4]
00401402     mov     [esp+8], ecx
00401406     mov     [esp+4], edx
0040140A     mov     [esp], eax
0040140D     call    _check
00401412     mull   Fesn+60h+var_41    pay

```

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.

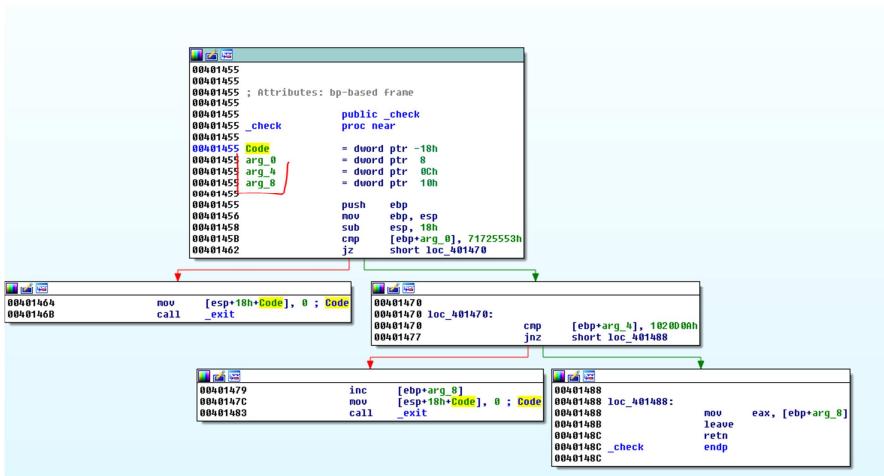
```

004013EE     mov     [esp], eax ; Buffer
004013F1     call    _gets
004013F6     mov     edx, [esp+60h+COOKIE]
004013FA     mov     eax, [esp+60h+COOKIE2]
004013FE     mov     ecx, [esp+60h+FLAG]
00401402     mov     [esp+8], ecx
00401406     mov     [esp+4], edx
0040140A     mov     [esp], eax
0040140D     call    _check

```

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.

```
00401455 ; USE DATA DEFINITION COMMANDS TO CREATE LOCAL VAR
.text = 00401455 ; Two special fields "r" and "s" represent return
.text = 00401455 ; Frame size: 18; Saved regs: 4; Purge: 0
.text = 00401455 ;
.text = 00401455
.text = 00401455 Code dd ?
.text = 00401455 db ?, undefined
.text = 00401455 s db 4 dup(?)
+ 00401455 r db 4 dup(?)
+ 00401455 arg_0 dd ?
+ 00401455 arg_4 dd ?
+ 00401455 arg_8 dd ?
+ 00401455 db ?, undefined
+ 00401455 end of stack variables
```

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

```

00401455
00401455
00401455 ; Attributes: bp-based frame
00401455
00401455 _check          public _check
00401455     proc near
00401455
00401455     Code           = dword ptr -18h
00401455     COOKIE2        = dword ptr  8
00401455     COOKIE         = dword ptr  0Ch
00401455     FLAG           = dword ptr  10h
00401455
00401455     push    ebp
00401456     mov     ebp, esp
00401458     sub     esp, 18h
00401458     cmp     [ebp+COOKIE2], 71725553h
00401462     jz      short loc_401470

```

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.

```

00401455
00401455
00401455 ; Attributes: bp-based frame
00401455
00401455     public _check
00401455     proc near
00401455
00401455     Code           = dword ptr -18h
00401455     COOKIE2        = dword ptr  8
00401455     COOKIE         = dword ptr  0Ch
00401455     FLAG           = dword ptr  10h
00401455
00401455     push    ebp
00401456     mov     ebp, esp
00401458     sub     esp, 18h
00401458     cmp     [ebp+COOKIE2], 71725553h
00401462     jz      short loc_401470

```

Please enter a string

Please enter the type declaration: `int _cdecl check(int COOKIE2, int COOKIE, int FLAG)`

OK Cancel Help

```

00401458     sub    esp, 18h
00401458     cmp    [ebp+COOKIE2], 71725553h
00401462     jz     short loc_401470

```

```

mov    [esp+18h+Code], 0 ; Code
call   _exit

```

```

00401470
00401470 loc_401470:

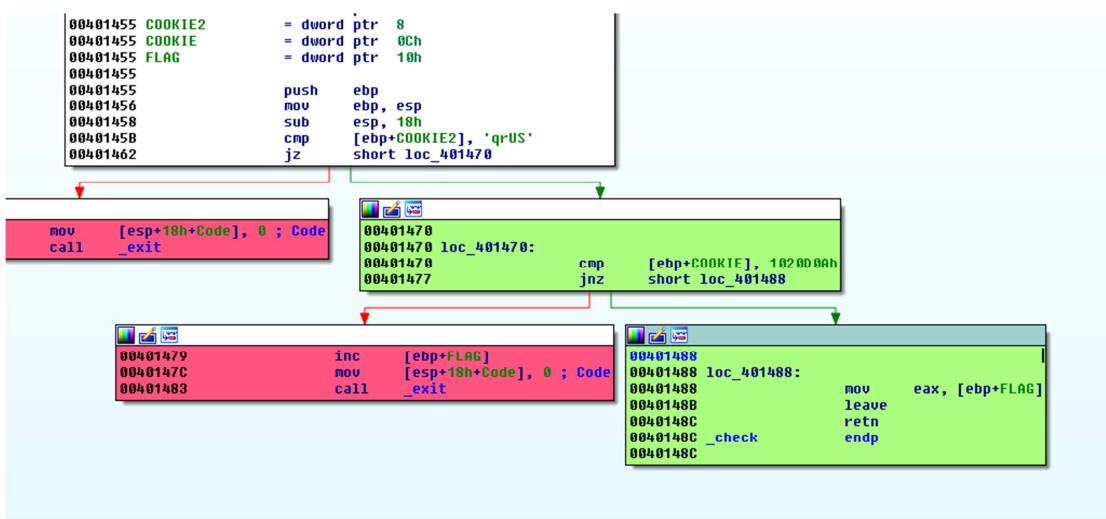
```

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
mnll  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.

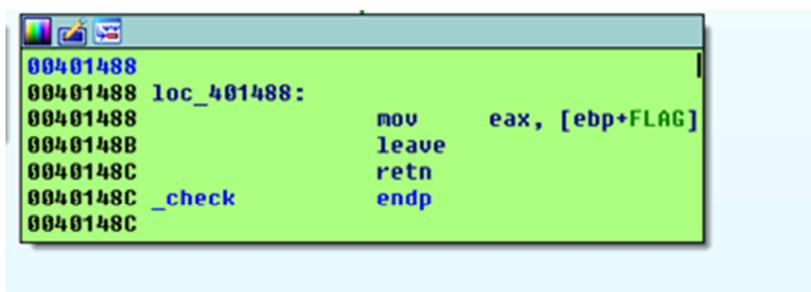
```

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.



```

00401406    mov    [esp+4], edx ; COOKIE
0040140A    mov    [esp], eax ; COOKIE2
0040140D    call   _check
00401412    mov    [esp+60h+var_8], eax
00401416    mov    eax, [esp+60h+var_8]
0040141A    mov    [esp+4], eax
0040141E    mov    dword ptr [esp], offset aFlagX ; "flag %x\n"
00401425    call   _printf
0040142A    cmp    [esp+60h+var_8], 35224158h
00401432    jnz   short loc_401442

```

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

```

call   _printf
cmp   [esp+60h+var_8], 35224158h
jnz   short loc_401442

```

```

dptr [esp], offset Str ; "You win man"
ts
rt loc_40144E

```

```

00401442
00401442 loc_401442:
00401442             ; "You "
00401442             mov    dword pi
00401449             call   _puts

```

```

import ...
p = Popen([r'C:\Users\ricnar\Desktop\24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24\IDA4.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)

cookie2="SURq"
cookie=struct.pack("<L",0x41424344)
flag=struct.pack("<L",0x35224158)
fruta=struct.pack("<L",0x90909090)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera= 50 *"A"+ cookie2 + cookie + fruta + _flag

p.stdin.write(primer)
testresult = p.communicate()[0]

print primera
print(testresult)

```

Let's try it.

```

C:\Python27\python.exe "C:/Users/ricnar/Desktop/24-INTRODUCCION AL REVERSING CON IDA PRO DESDE CERO PARTE 24/IDA4.py"
ATACHEA EL DEBUGGER Y APRETA ENTER

AAAAAAA...ASURqDCBA♦♦♦♦XA"5
buf: 0028fedc cookie: 0028ff10 cookie2: 0028ff14
flag 35224158
You win man

```

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

Ricardo Narvaja

Translated by: @IvinsonCLS