REVERSING WITH IDA PRO FROM SCRATCH
PART 20

Vulnerabilities.

In this part, we'll see a topic about vulnerabilities and how to analyze some of the easiest ones.

What is vulnerability?

In computer security, the word vulnerability refers to a system weakness allowing an attacker to violate confidentiality, integrity, availability, access control and system consistence or data and applications.

Vulnerabilities are the result of bugs or errors in a system design. Although, in a broader sense, they can also be the result of the same technological limitations because, firstly, there is not a 100% safe system. So, there are theoretical and real vulnerabilities known as exploits.

Definitions:

Bug: https://en.wikipedia.org/wiki/Software_bug
Exploit: https://en.wikipedia.org/wiki/Exploit_%28computer_security%29

A vulnerable program has bugs or programming errors and according to the kind of bugs, they can be exploited executing malicious code in it, but the authentication can also fail and allow the user do illegal actions, trigger crashes, allow privilege escalation, etc.

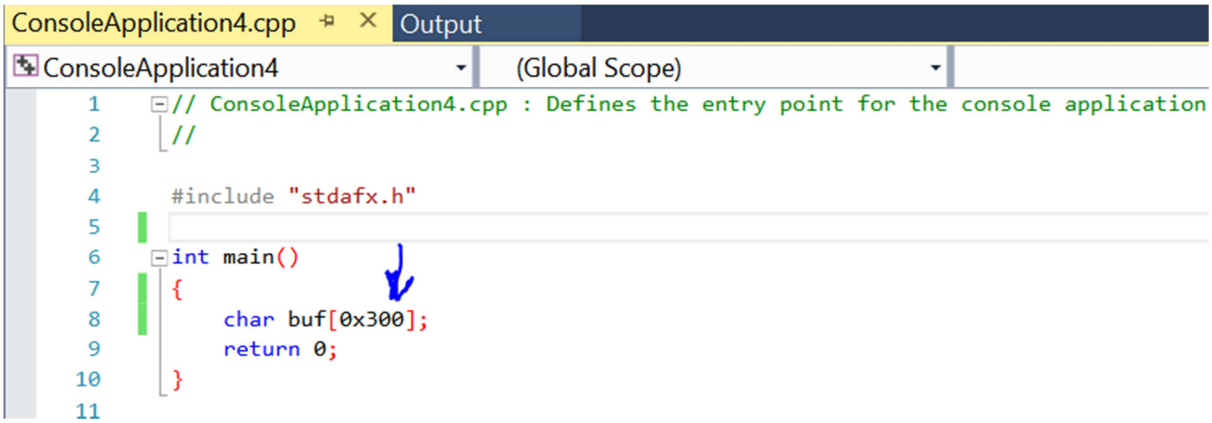
Of course, through memory corruption bugs. The easiest ones are the buffers overflows which are produced when a program reserves a memory zone or buffer to store data and, for some reason, the destination data size is not checked in an adequate way and the buffer overflows copying more than the reserved size being able to overwrite variables, arguments and pointers in the memory.

The easiest buffer overflow is the stack overflow that is when a buffer reserved in the stack overflows.

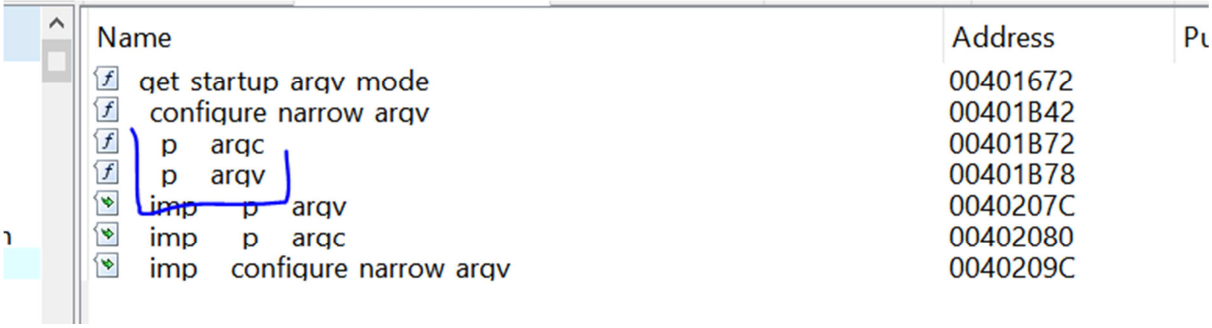
A buffer in a simple C program source code can be:

char buf[xxx];

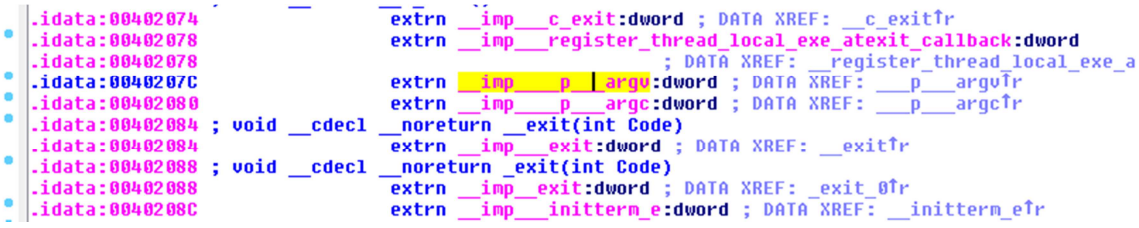
xxx is the buffer size, in this case, it is a 0x300 bytes stack buffer.



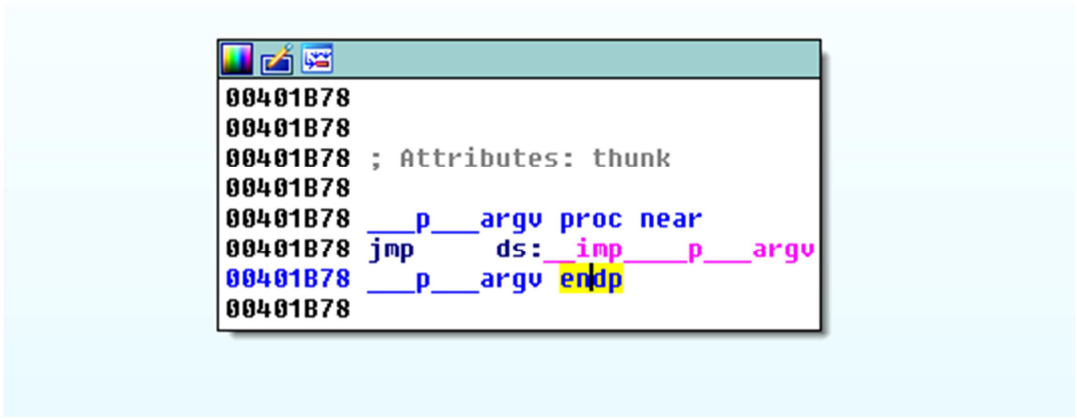
Obviously, the program does nothing, but we compile it and see it in IDA. It is attached in this tutorial as Compilado_1.exe.



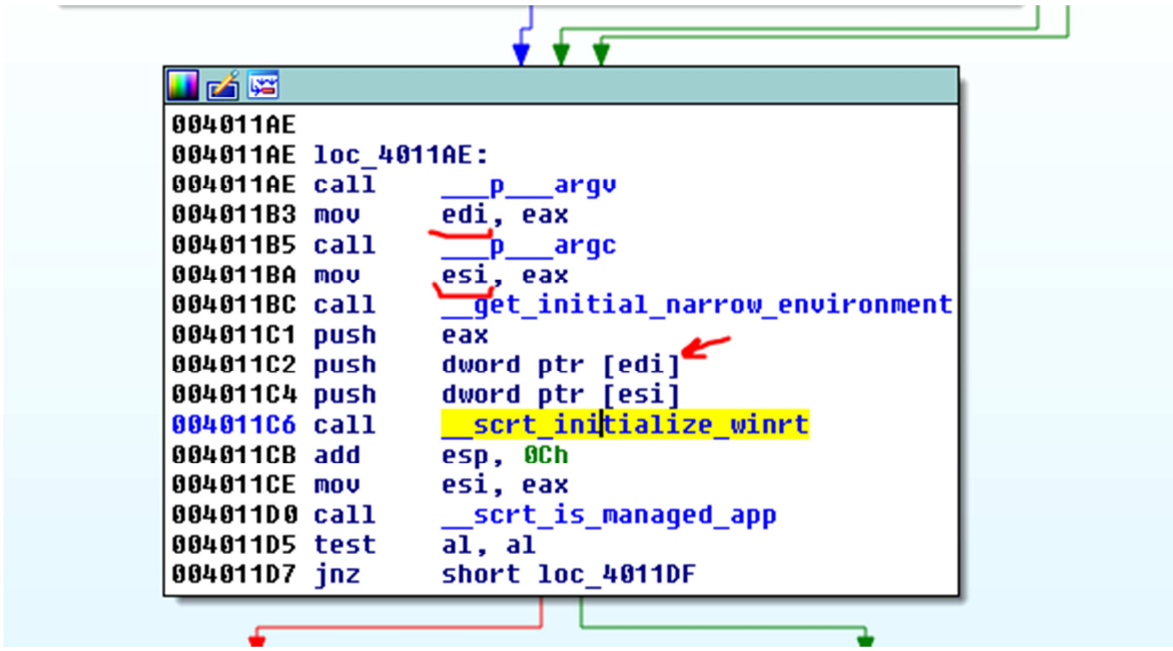
Searching the **argc** or **argv** references we get the main. Double click to get there.



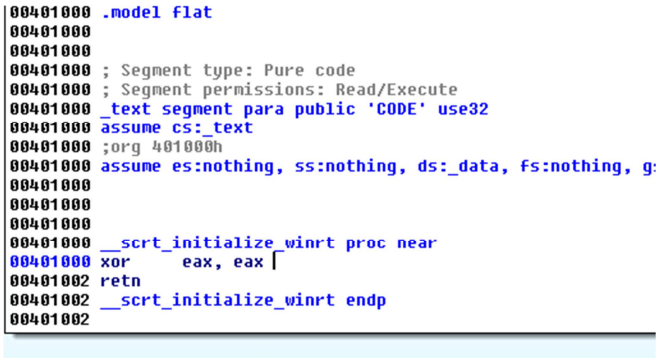
And press X.



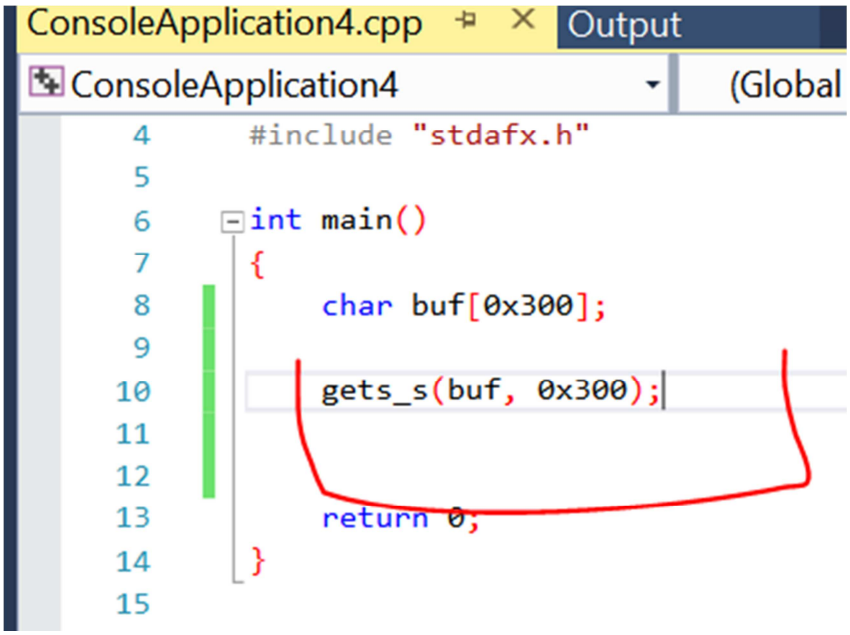
Searching references we get to the known block that calls the main, in this case, it renamed it. Maybe, because it does nothing.



If we enter the function, it reserves nothing because it doesn't use the buffer and it returns 0 in EAX.

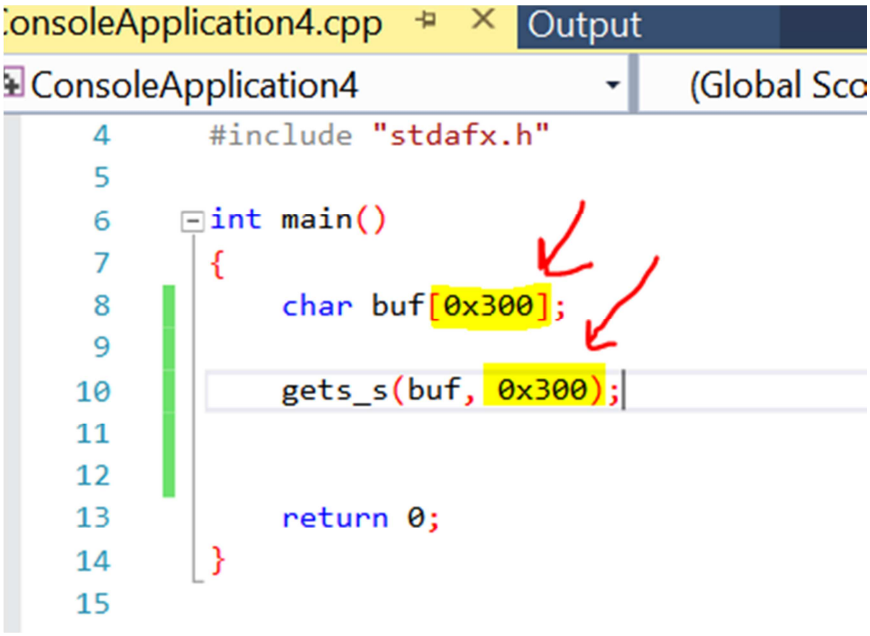


We have to use the buffer for it to finish reserving space.



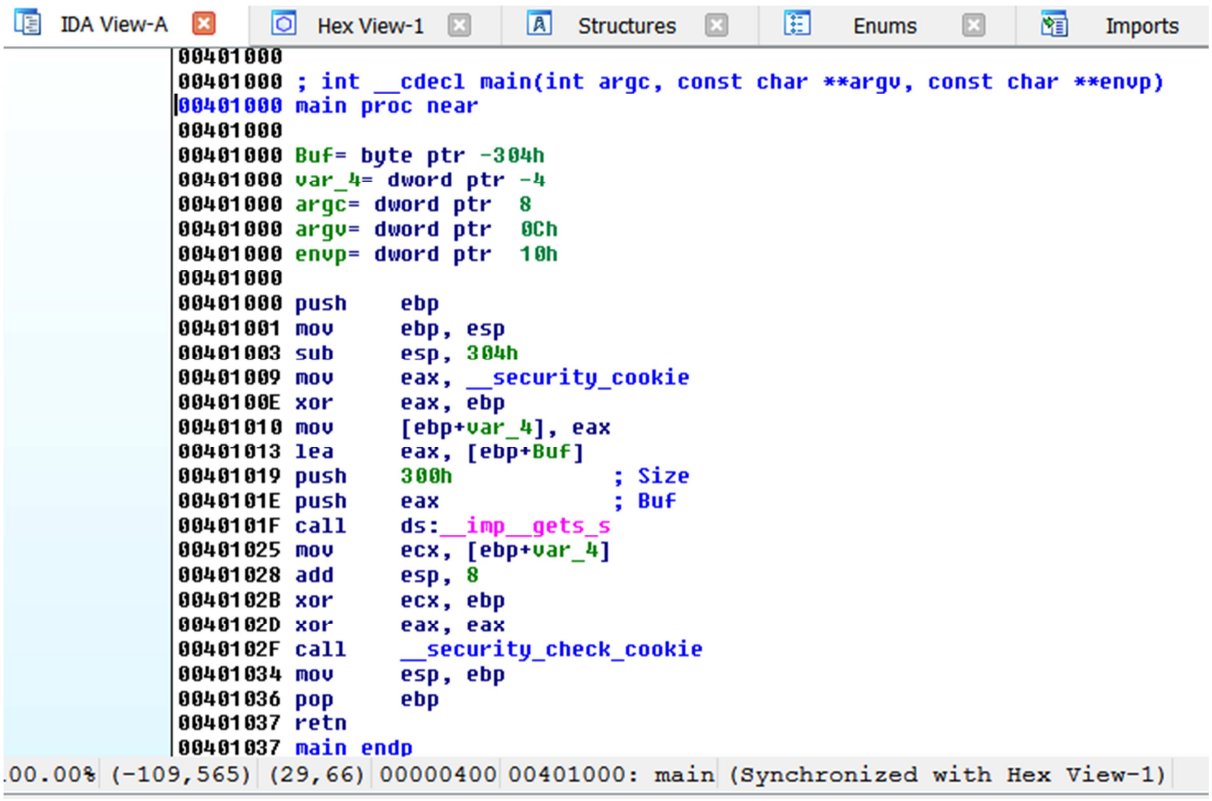
Now, we use the gets_s function for the user to type something in the console and it saves it in the buffer.

That function has two arguments: the buffer and the character max length to avoid overflows. In the example, there is no overflow because the copied size is not greater than the created buffer of 0x300 bytes.



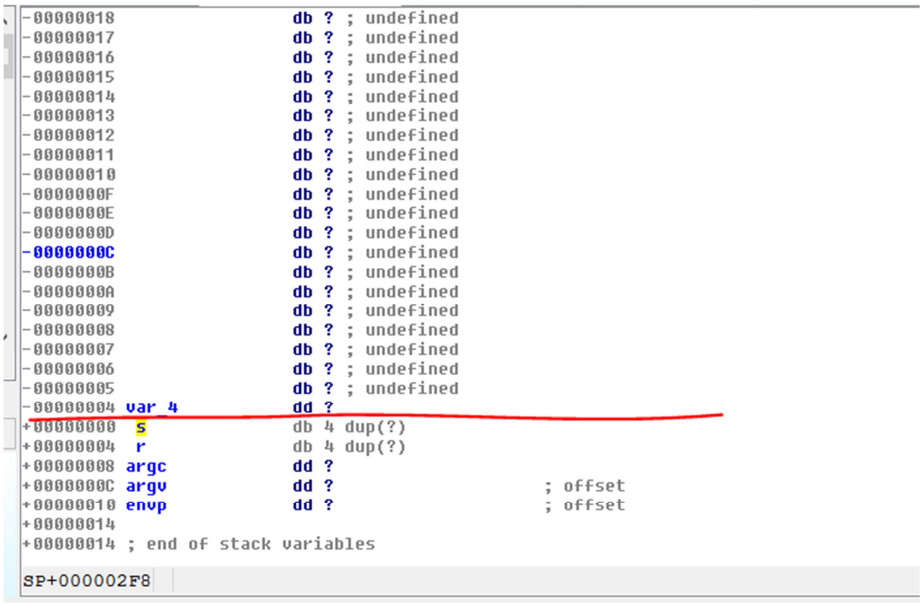
There is no possibility of overflow while what we copy is less or equal to 0x300 bytes.

Let’s see it in IDA. This is attached as Compilado_2.exe.



I compiled it with symbols and IDA also found them in my PC.
It looks better. We see the main with its args and variables.

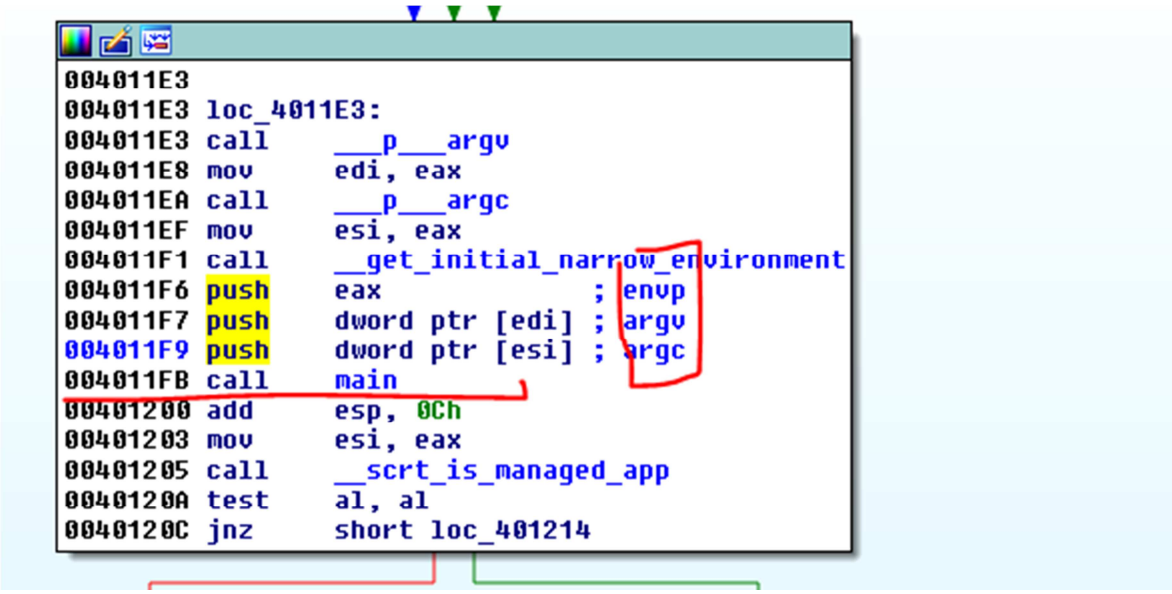
Let’s analyze it with IDA, reversing it.



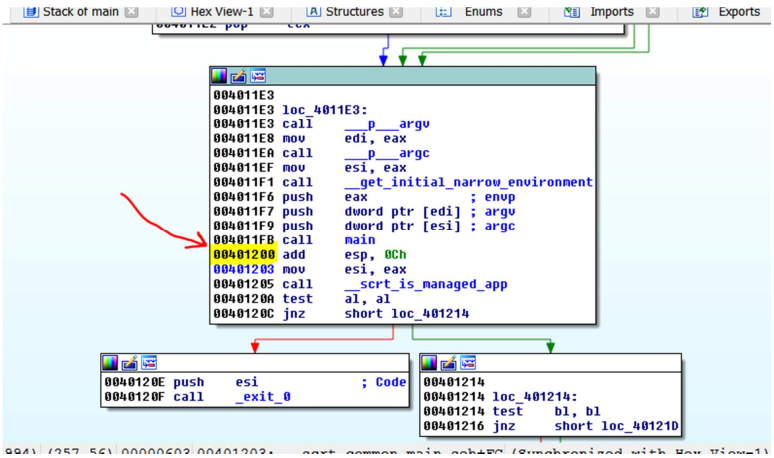
If we double click on any variable or arg, we go to the stack static view.

We see the 3 args: **envp**, **argv** and **argc** that we don't use in the main.

They are pushed into the stack before calling the main.



It saves the RETURN ADDRESS that is the address it saves to know where to return after leaving the CALL. In this case, the return address will be 0x401200 if there is no randomization.



It will return there after executing the main. So, it must save that 0x401200 value in the stack just above the 3 args.

```
-00000009                db ? ; undefined
-00000008                db ? ; undefined
-00000007                db ? ; undefined
-00000006                db ? ; undefined
-00000005                db ? ; undefined
-00000004 var_4          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
```

Then, it starts executing the main. The first thing is the PUSH EBP.

```
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr  8
00401000 argv= dword ptr  0Ch
00401000 envp= dword ptr  10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub     esp, 304h
00401009 mov     eax, __security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h            ; Size
0040101E push    eax             ; Buf
0040101F call    ds:__imp__gets_s
00401025 mov     ecx, [ebp+var_4]
00401028 add     esp, 8
0040102B xor     ecx, ebp
0040102D xor     eax, eax
0040102F call    __security_check_cookie
00401034 mov     esp, ebp
```

That saves, in the stack, the EBP value used by the function that called the main. Just above the 0x401200 return address, we don’t know what value it will have because it changes with the execution, but it is the stored EBP of the parent function of this.

IDA View-A | Stack of main | Hex View-1 | Structures

```
-00000008                db ? ; undefined
-00000007                db ? ; undefined
-00000006                db ? ; undefined
-00000005                db ? ; undefined
-00000004 var_4          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
```

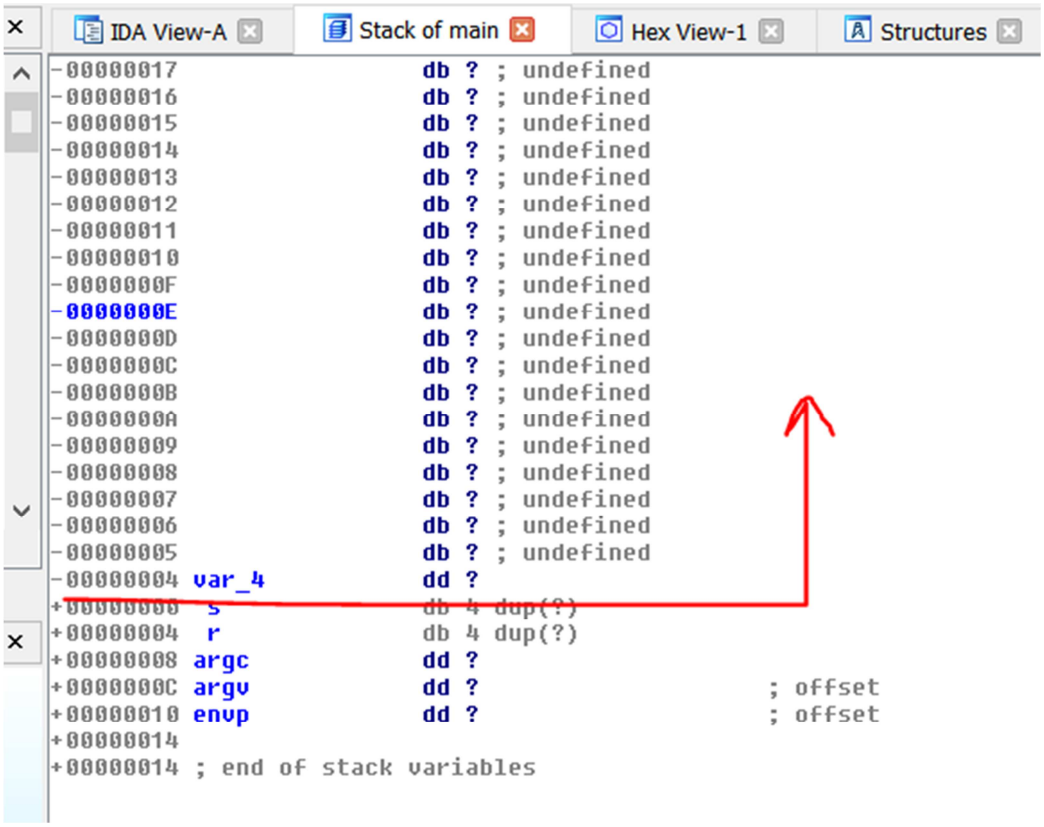
It will be saved in the stack above the return address.

The next thing executed is:

```
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr  8
00401000 argv= dword ptr  0Ch
00401000 envp= dword ptr  10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub     esp, 304h
00401009 mov     eax, __security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
```

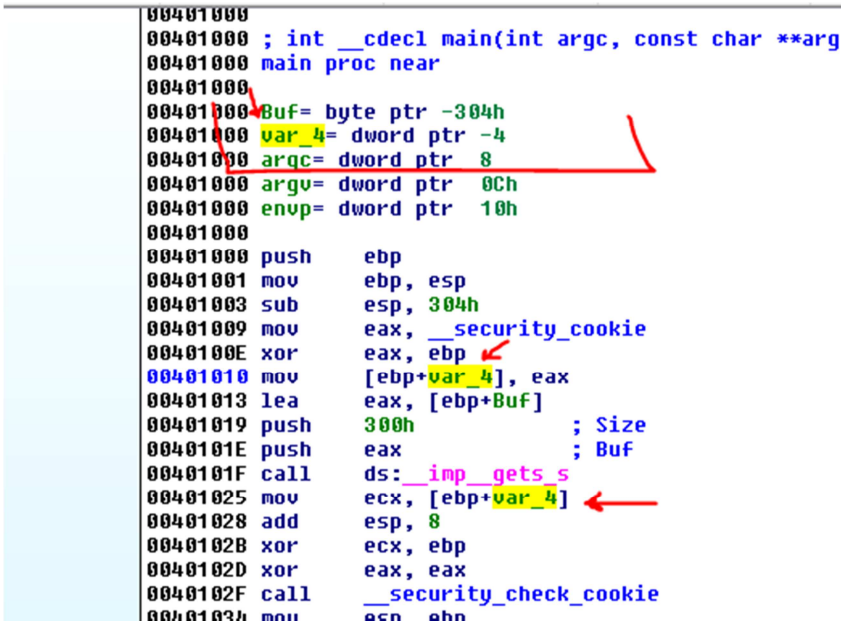
It establishes EBP as the BASE in this function equaling it with ESP. This is a MOV. It just changes the EBP value not the stack.

Then, sub esp, 0x304 moves ESP upwards reserving space for local variables and stack buffers located above the stored EBP and ESP will be working in an EBP-based function. Just above the reserved space.

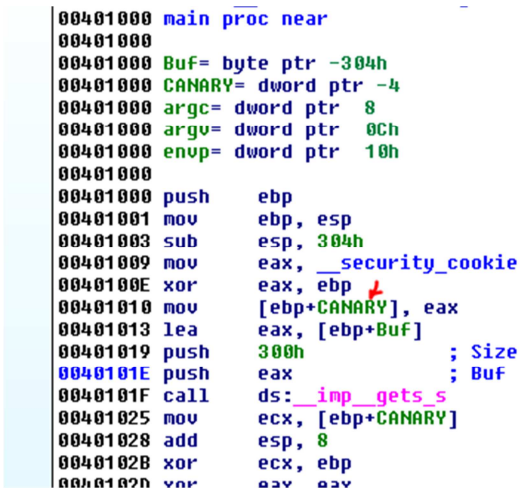


There, we see the reserved space for variables and buffers just above the s (STORED EBP).

The first variable almost always found is the stack protection CANARY. In this case, it is called var_4.



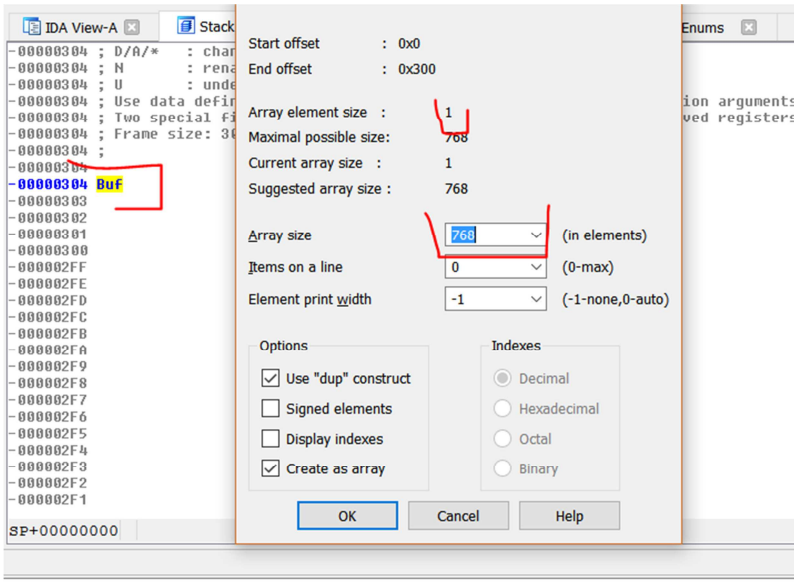
There, we see that it reads the _security cookie value that is a random value that changes in each execution. It is XORed with EBP and saved in **var_4**. Let's rename it as CANARY.



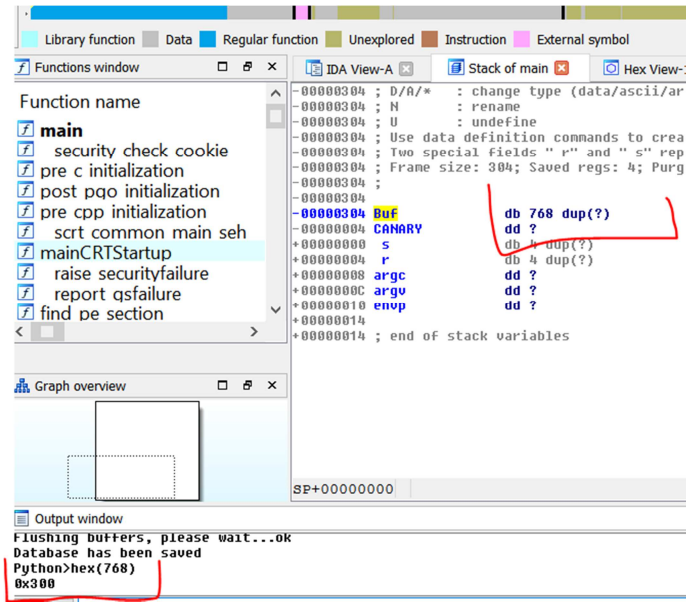
Above the CANARY is Buf. Let's see it in the stack static view.

```
-00000304 ; Two special fields " r" and " s" represent return address an
-00000304 ; Frame size: 304; Saved regs: 4; Purge: 0
-00000304 ;
-00000304
-00000304 Buf             db ?
-00000303                 db ? ; undefined
-00000302                 db ? ; undefined
-00000301                 db ? ; undefined
-00000300                 db ? ; undefined
-000002FF                 db ? ; undefined
-000002FE                 db ? ; undefined
-000002FD                 db ? ; undefined
-000002FC                 db ? ; undefined
-000002FB                 db ? ; undefined
-000002FA                 db ? ; undefined
-000002F9                 db ? ; undefined
-000002F8                 db ? ; undefined
-000002F7                 db ? ; undefined
-000002F6                 db ? ; undefined
-000002F5                 db ? ; undefined
-000002F4                 db ? ; undefined
-000002F3                 db ? ; undefined
```

When I see an empty space in the static view, there is a buffer possibly. Right click on Buf and select ARRAY.



Its size is 768 decimal by 1 that is the size of each element. So, 768 in hex is 0x300.



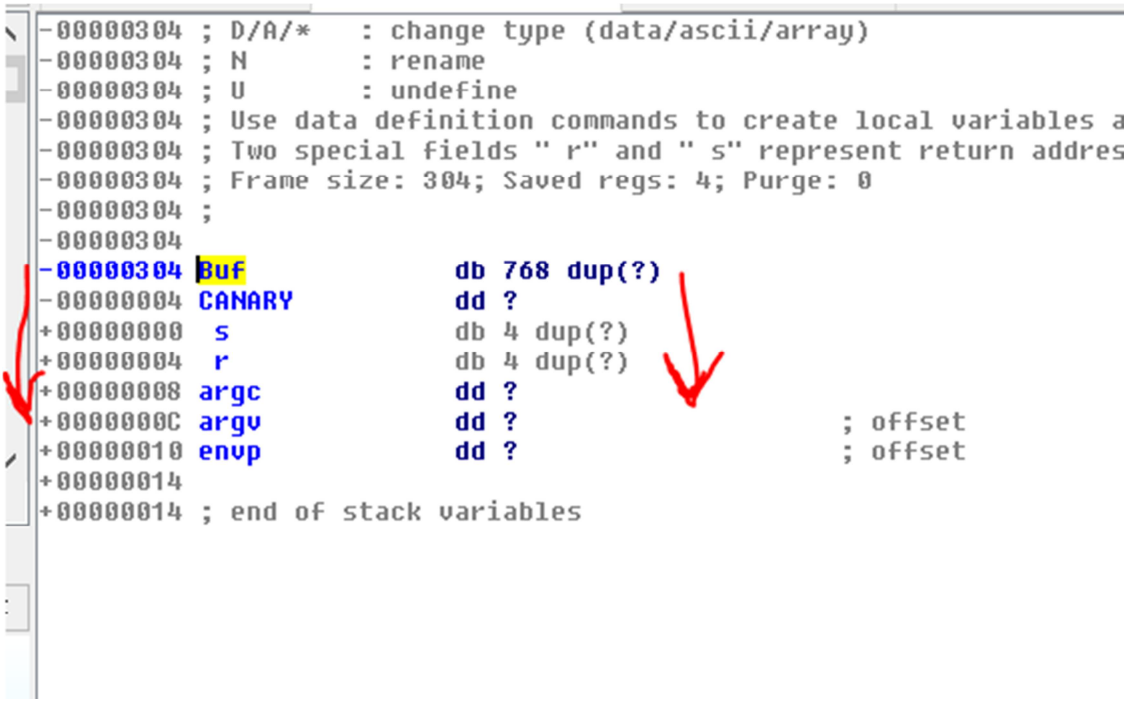
We accept it and Buf is defined as 0x300 bytes.

There, we see the gets_s CALL, 0x300 max size and the other arg is the buffer address got through the LEA.

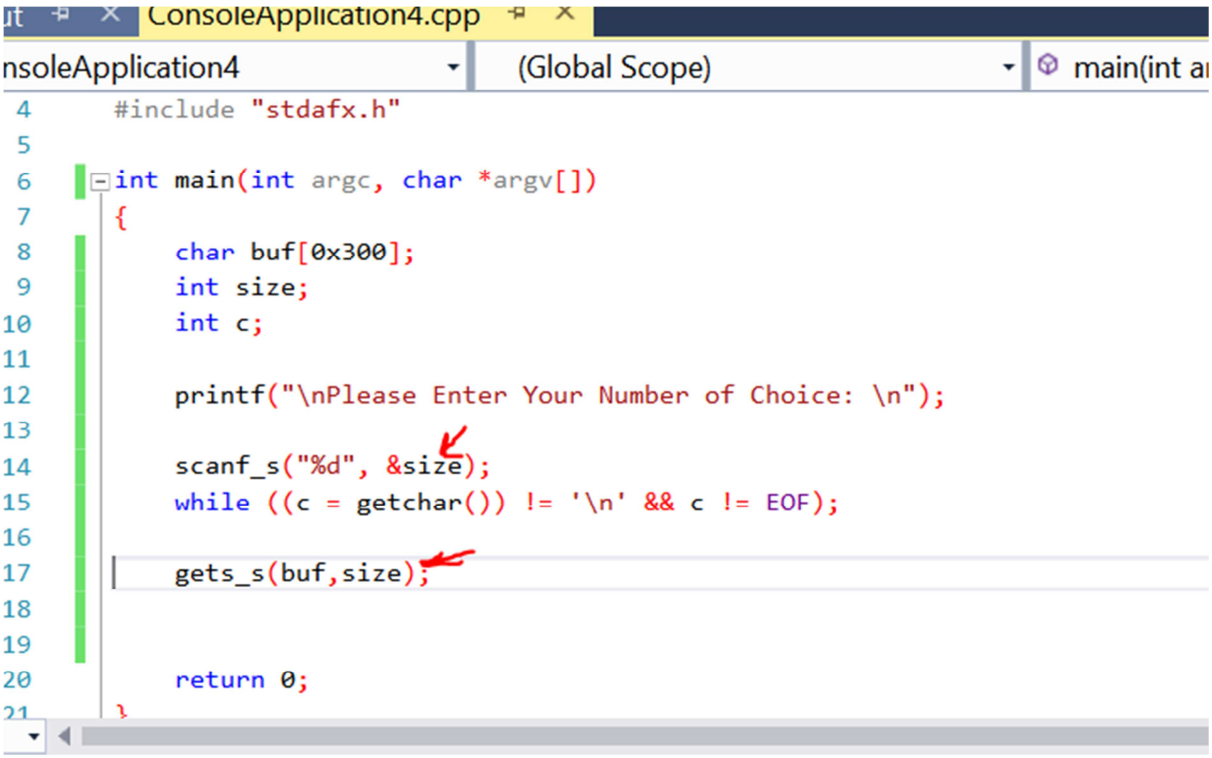
```
0040100E xor     eax, ebp
00401010 mov     [ebp+CANARY], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h            ; Size
0040101E push    eax             ; Buf
0040101F call    ds:__imp__gets_s
00401025 mov     ecx, [ebp+CANARY]
```

We verified that the buf size is 0x300 and it copies 0x300 max through the gets_s.

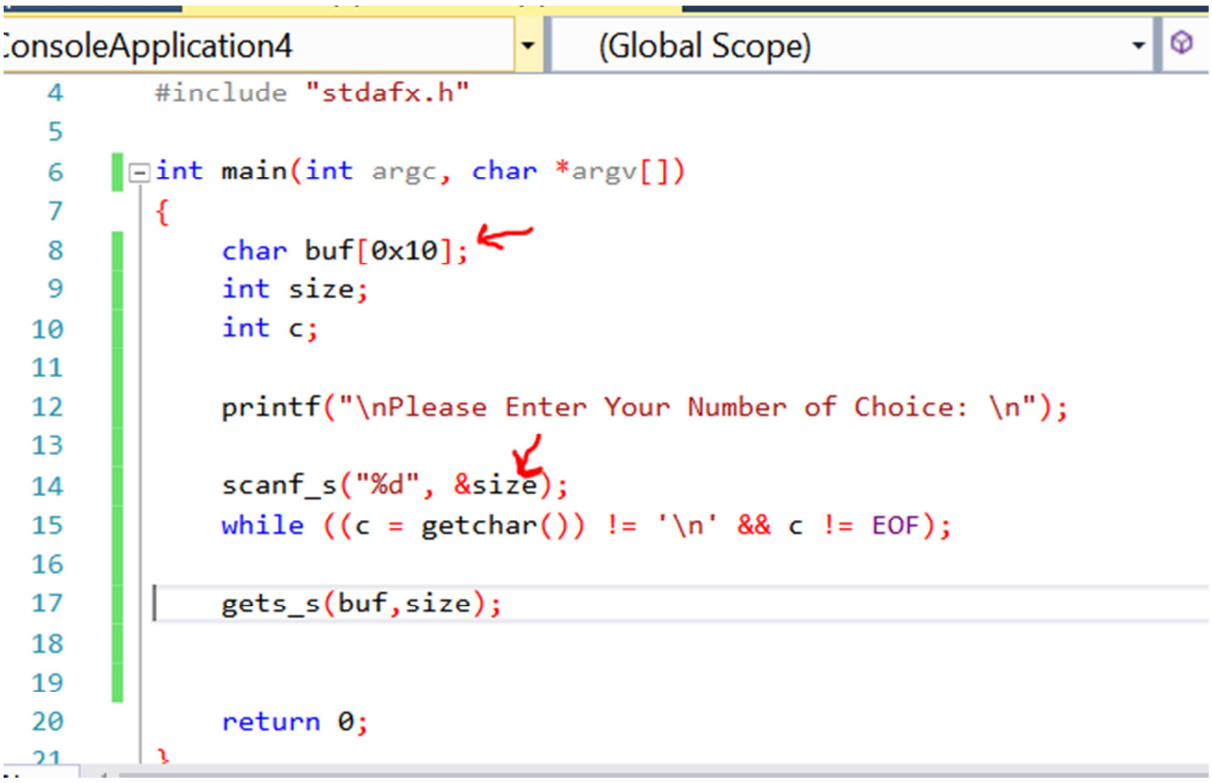
Obviously, if we could have overflowed the buffer copying more than 0x300, we had overwritten the CANARY, STORED EBP and thel RETURN ADDRESS that are below the BUFFER.



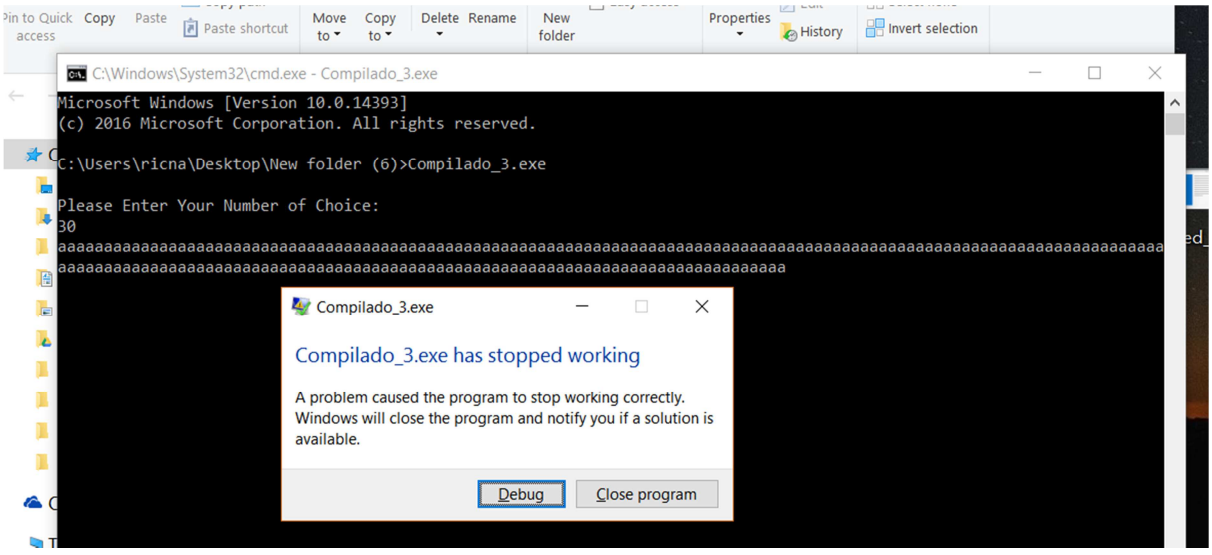
But this is not the case. This is a good example of a buffer written in a correct way.



Many times, the user knows the data size to be entered. If so, that size should be checked well to avoid overflows.



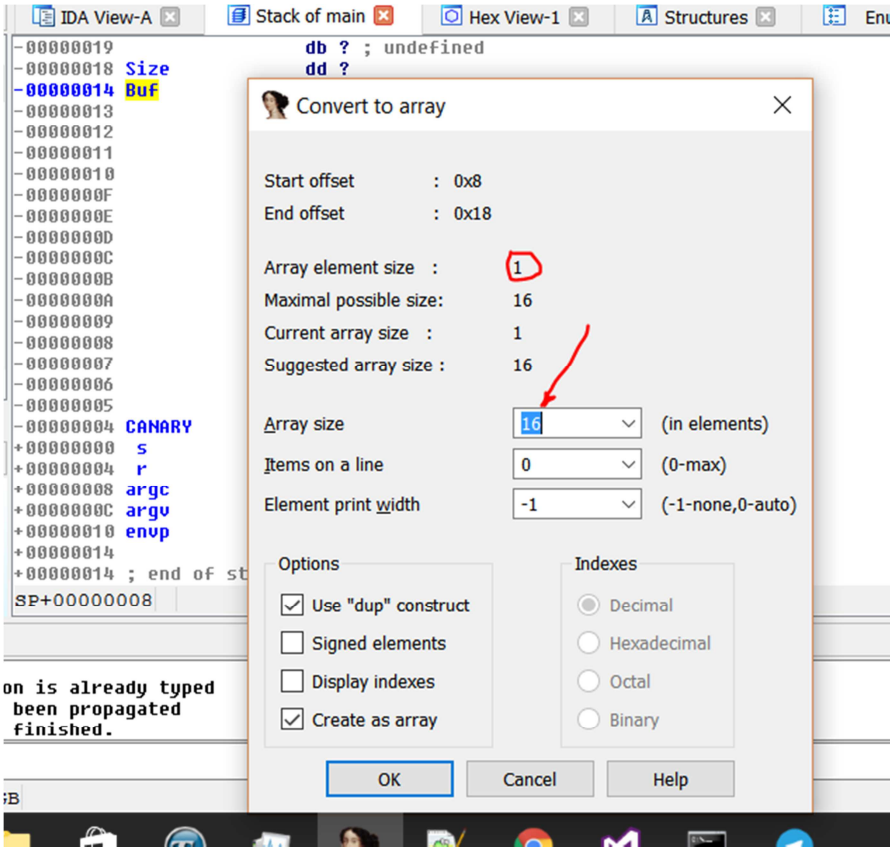
There, we see a 0x10-byte buffer or 16 decimal. The user has the possibility of typing the size of the gets_s buffer. There are no max value checks. If I compile it and execute it (Compilado_3.exe).



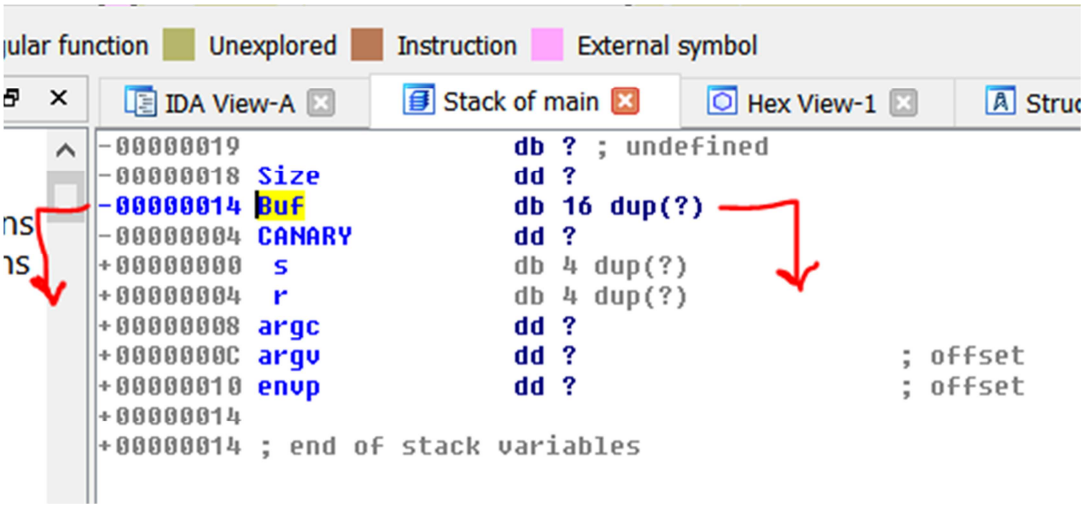
There is a BUG and the program is vulnerable. If we loaded in IDA even if we didn't have the source code.

```
00401090 ; int __cdecl main(int argc, const char **argv, const
00401090 main proc near
00401090
00401090 Size= dword ptr -18h
00401090 Buf= byte ptr -14h
00401090 CANARY= dword ptr -4
00401090 argc= dword ptr  8
00401090 argv= dword ptr  0Ch
00401090 envp= dword ptr  10h
00401090
00401090 push    ebp
00401091 mov     ebp, esp
00401093 sub     esp, 18h
00401096 mov     eax, __security_cookie
0040109B xor     eax, ebp
0040109D mov     [ebp+CANARY], eax
004010A0 push    esi
004010A1 push    offset aPleaseEnterYou ; "\nPlease Enter Your
004010A6 call    printf
004010AB lea     eax, [ebp+Size]
004010AE push    eax
004010AF push    offset aD       ; "%d"
004010B4 call    scanf_s
004010B9 mov     esi, ds:__imp__getchar
004010BF add     esp, 0Ch
```

The CANARY is there as before. Just above, it is the Buf. Let’s see its size in the stack view.

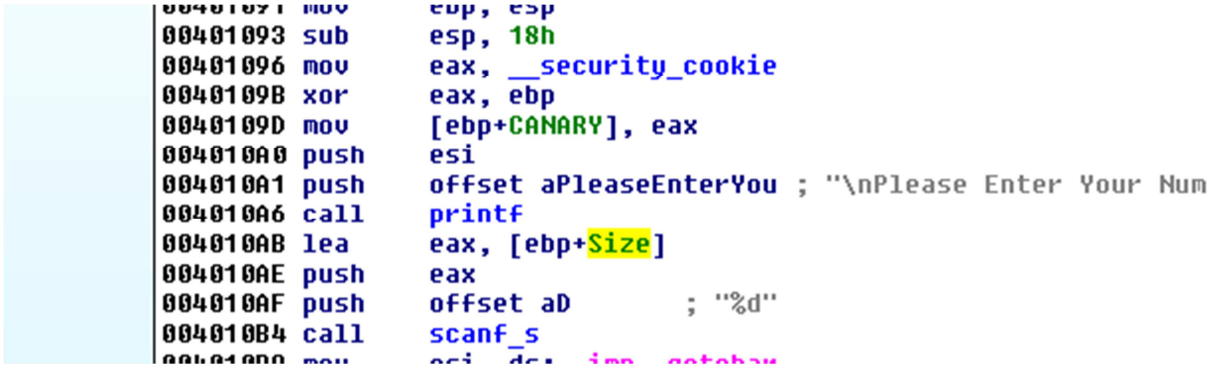


The buffer length is 16 bytes by 1 that is the length of each element. So, it is 0x10 hex.



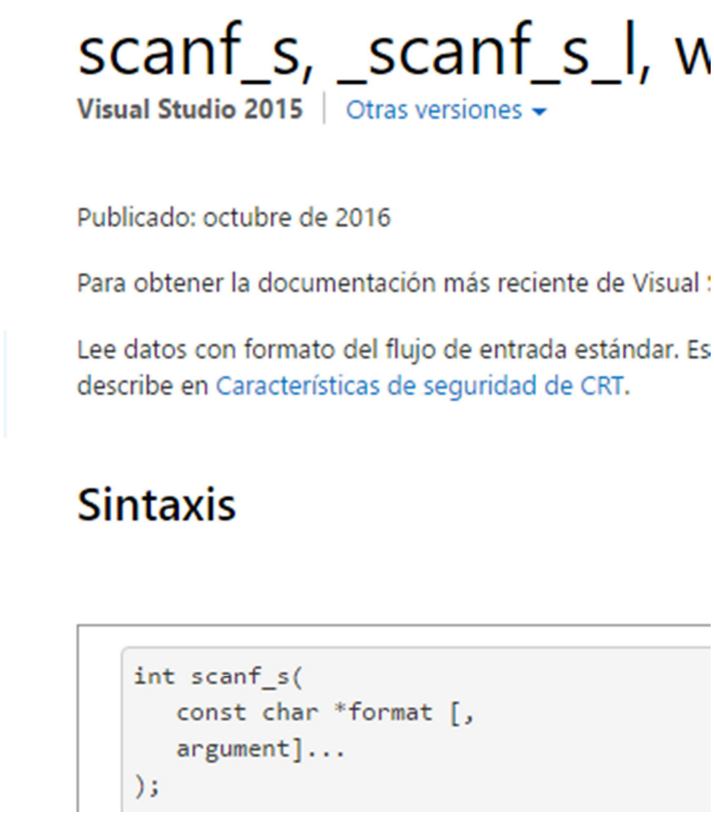
If we could copy more than 16 bytes in the buffer, it would overflow and overwrite the CANARY, STORED EBP and the RETURN ADDRESS.

Let's see the size variable.



After printing the "**Please enter Your Number**" message with **printf**, it calls scanf_s to enter a number with the keyboard that is saved in the size variable which is a dword and it receives the address with LEA.

Let's see the scanf_s function.



scanf_s, _scanf_s_l, w

Visual Studio 2015 | Otras versiones

Publicado: octubre de 2016

Para obtener la documentación más reciente de Visual

Lee datos con formato del flujo de entrada estándar. Es
describe en Características de seguridad de CRT.

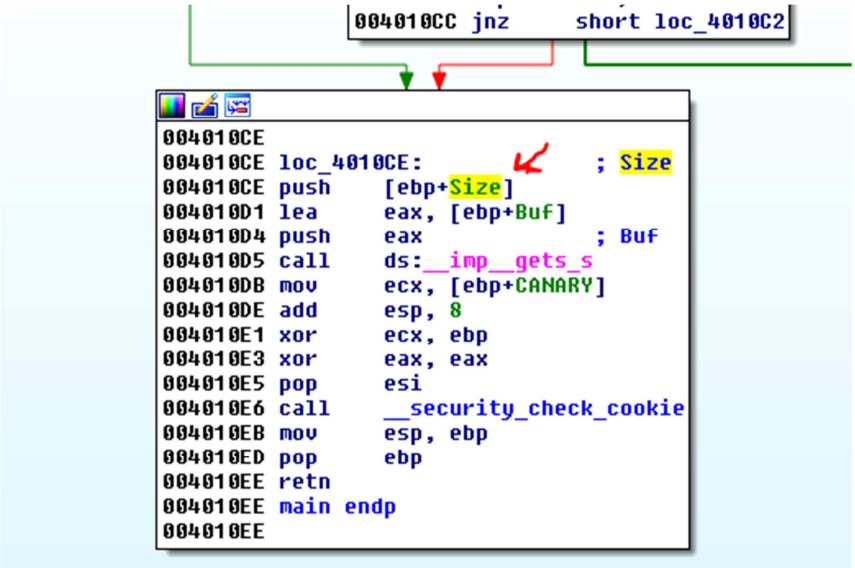
Sintaxis

```
int scanf_s(
   const char *format [,
   argument]...
);
```

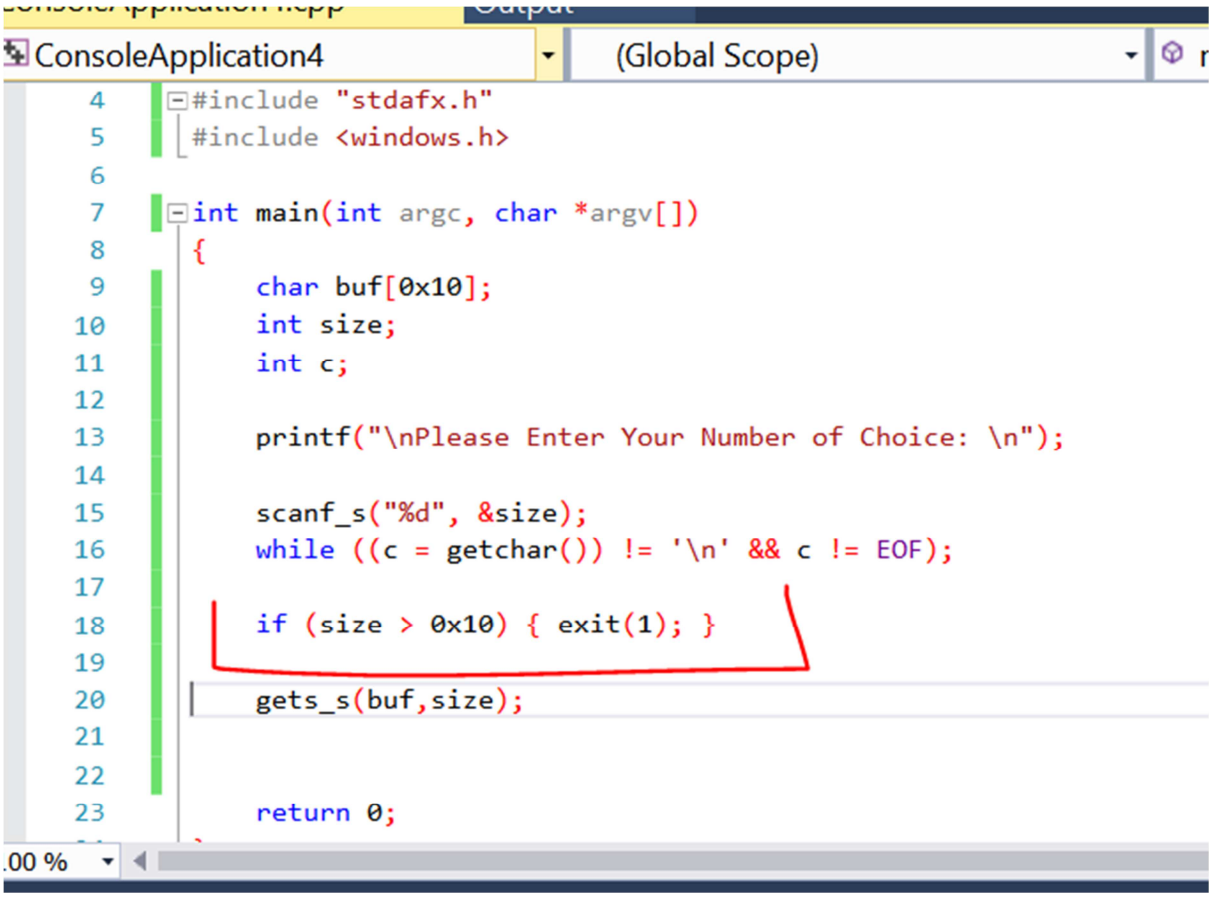
The scanf_s function reads data from the stdin standard input stream and writes the data to the location provided in argument. Each argument must be a pointer to a variable of a type that corresponds to a type specifier in format. If the copying takes place between overlapping strings, the behavior is undefined.

So it's like the opposite of **printf**. Instead of printing with a format, it enters from console with a format to a Buffer, in this case the format is "%d" by which it is interpreted like a decimal number.

This way, when it calls gets_s using that size to be typed, it will copy that number of bytes and if it is greater than 0x10 it will overflow.



A possible solution would be to check the length of the size before copying.



It would be good to analyze it to see if this solution makes it not vulnerable or it still is, analyze it. It is called VULNERABLE_o_NO.exe and we will discuss it in the next part.

Ricardo Narvaja

Translated by: @IvinsonCLS