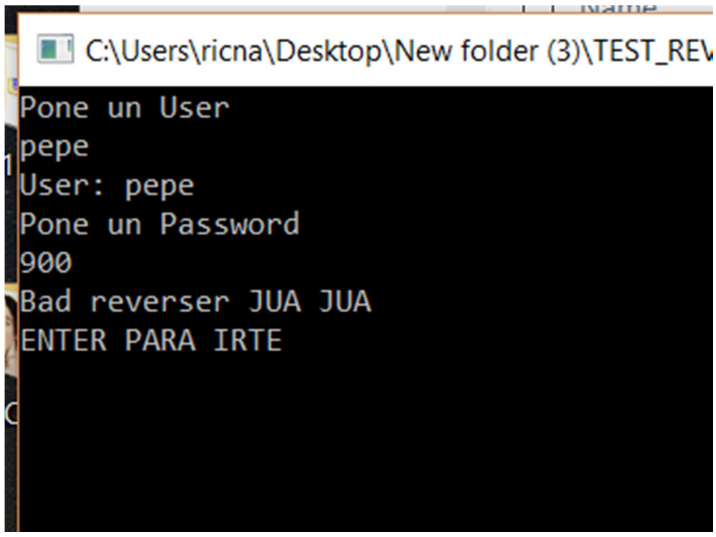
PART 12

To avoid boredom, we will include some exercises, in this case, I compiled another program called TEST_REVERSER.exe that is very simple, but it will help us see some new things in static reversing and debugging.

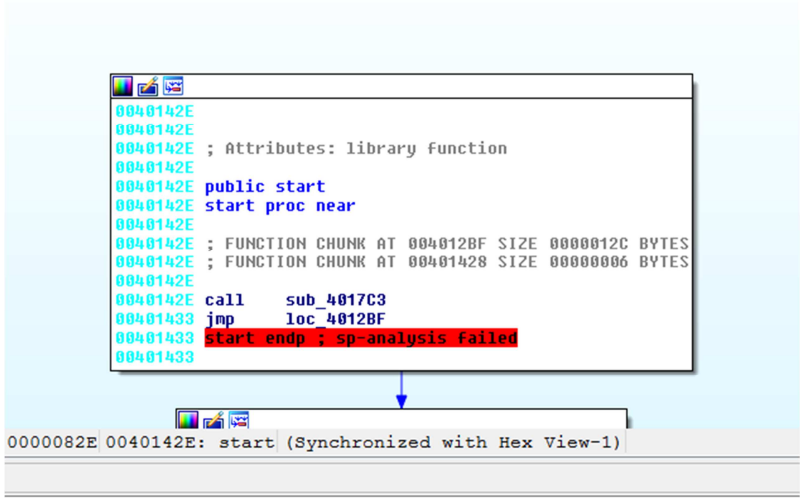
Let's run it outside IDA.



It needs a User and Password and then, it says we suck and it laughs at us.

Let's load it in IDA to see statically.

As I didn't use symbols. The situation is a little ugly.



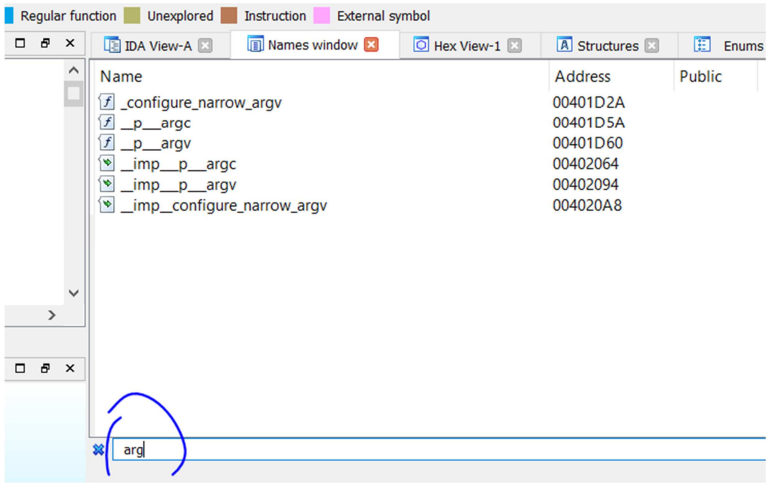
It doesn't open in the main, of course, but in the Entry Point. This is almost always like this and we have to manage to solve this little problem.

One of the ways to go to the hot part is searching the strings. We already know how to do that. Also, in these C++ programs, there is a way to find the **main** that sometimes works is the following.

We know that the main function receives the arguments: **argc**, **argv**, etc. The console arguments.

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

In the previous example, we saw that even if the arguments are not used, the PUSHes are they anyways. It's something by default. So, we can search in the NAMES tab to see if they are there.



When I say, "Open x tab." You know that you have to click on **VIEW-OPEN SUBVIEW-XXX**.

By pressing CTRL + F, we filter, for example, **arg** and let's double click on **_p_argc**.

```
00401D5A
00401D5A
00401D5A ; Attributes: thunk
00401D5A
00401D5A __p___argc proc near
00401D5A jmp     ds:__imp___p___argc
00401D5A __p___argc endp
00401D5A
```

We press **X** to search for the references.

```
0040139A pop     ecx

0040139B
0040139B loc_40139B:
0040139B call    __p___argv
004013A0 mov     edi, eax
004013A2 call    __p___argc
004013A7 mov     esi, eax
004013A9 call    __get_initial_narrow_environment
004013AE push    eax
004013AF push    dword ptr [edi]
004013B1 push    dword ptr [esi]
004013B3 call    sub_401070
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test    al, al
004013C4 jnz     short loc_4013CC

079A 0040139A: start-94 (Synchronized with Hex View-1)
```

There, we see how it calls the **_p_argv** and **_p_argc** functions and the return value goes to the **main** function content, here, it is 0x401070.

I see the same function in my IDA with symbols.

```
00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401070 main proc near
00401070
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr  4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
14,135) 00000470 00401070: main (Synchronized with Hex View-1)
```

And the reference.

```
0040139B
0040139B loc_40139B:
0040139B call    ___p___argv
004013A0 mov     edi, eax
004013A2 call    ___p___argc
004013A7 mov     esi, eax
004013A9 call    __get_initial_narrow_environment
004013AE push    eax             ; envp
004013AF push    dword ptr [edi] ; argv
004013B1 push    dword ptr [esi] ; argc
004013B3 call    main
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    __scrt_is_managed_app
004013C2 test    al, al
004013C4 jnz     short loc_4013CC
3C6 push    esi             ; Code
004013CC
```

Cheating is not the idea. I'm just verifying if the method to find the main works and that's true. Searching for the arguments references received by the console, we reach the **main**.

Let's rename the **main**.

```
0401393 push    dword ptr [esi]
0401395 call    _register_thread_local_exe_atexit_callback
040139A pop     ecx
0040139B
0040139B loc_40139B:
0040139B call    ___p___argv
004013A0 mov     edi, eax
004013A2 call    ___p___argc
004013A7 mov     esi, eax
004013A9 call    _get_initial_narrow_envi
004013AE push    eax
004013AF push    dword ptr [edi]
004013B1 push    dword ptr [esi]
004013B3 call    sub_401070
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test    al, al
004013C4 jnz     short loc_4013CC
07B3 004013B3: start-7B (Synchronized with
```

Rename address

Address: 0x401070

Name main

Maximum length of new names 15

Local name prefix @@

- [ ] Local name
- [x] Include in names list
- [ ] Public name
- [ ] Autogenerated name
- [ ] Weak name
- [ ] Create name anyway

OK Cancel Help

IDA renames the **args** automatically when we tell it that function is the **main**.

```
00401393 push    dword ptr [esi]
00401395 call    _register_thread_local_exe_atexit_callback
0040139A pop     ecx

0040139B
0040139B loc_40139B:
0040139B call    __p___argv
004013A0 mov     edi, eax
004013A2 call    __p___argc
004013A7 mov     esi, eax
004013A9 call    _get_initial_narrow_environment
004013AE push    eax             ; envp
004013AF push    dword ptr [edi] ; argv
004013B1 push    dword ptr [esi] ; argc
004013B3 call    main
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test    al, al
004013C4 jnz     short loc_4013CC

000007B3 004013B3: start-7B (Synchronized with Hex View-1)
```

Now, it looks like the symbol version. ☺

```
00401070
00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401070 main proc near
00401070
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0

3,247) 00000470 00401070: main (Synchronized with Hex View-1)
```

Here, the variables and the arguments are more numerous than the previous example.

Double click on any variable or argument to see the static stack representation.

```
IDA View-A    Stack of main    Names window    Hex
-00000013                 db ? ; undefined
-00000012                 db ? ; undefined
-00000011                 db ? ; undefined
-00000010                 db ? ; undefined
-0000000F                 db ? ; undefined
-0000000E                 db ? ; undefined
-0000000D                 db ? ; undefined
-0000000C                 db ? ; undefined
-0000000B                 db ? ; undefined
-0000000A                 db ? ; undefined
-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
```

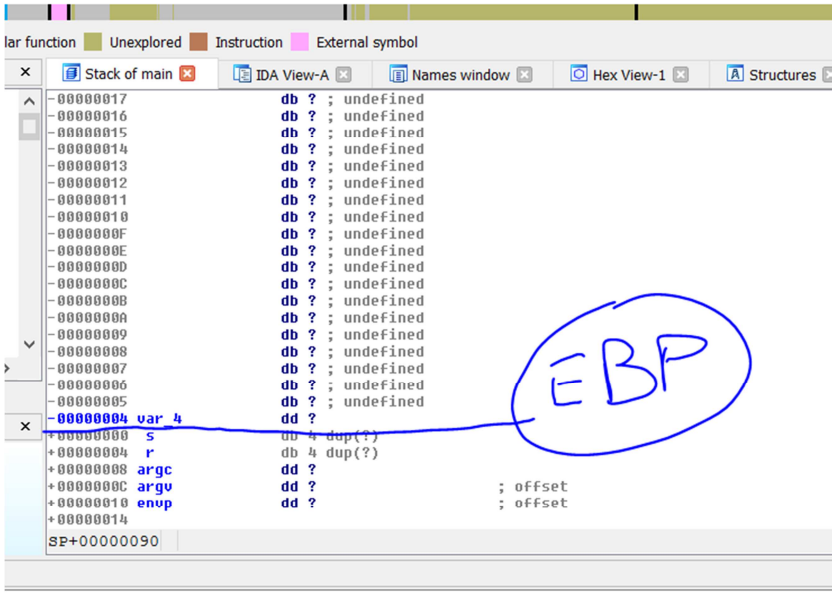
We see from down to up that the function args will be first and go always below the return address (**r)** because they are **PUSH**ed on the stack before **CALL**ing the function, and then it saves the return address on the stack.

We have the STORED EBP (**s**) that is the EBP of the function that called the **main**. It is saved on the stack when the function is executed by the PUSH EBP.

```
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0
```

Then, it moves ESP to EBP putting EBP in the value that it will have in this function for being the BASE where the arguments are referenced downward and the variables upward. And finally, **SUB ESP, 0x94** moves **ESP** making room for local variables and buffers. That's why they are above. In this case, it will be **0x94** because the compiler calculated that is what it needs to reserve the variable space according to how we program our function.

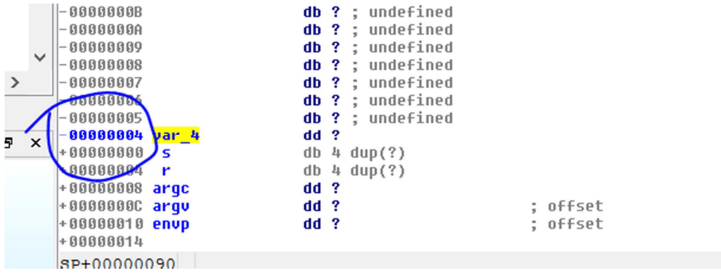
ESP has a value above that reserved space for local variables and EBP points to the BASE or HORIZONT that divides the above variables and the below STORED EBP, RETURN ADDRESS and ARGUMENTS.



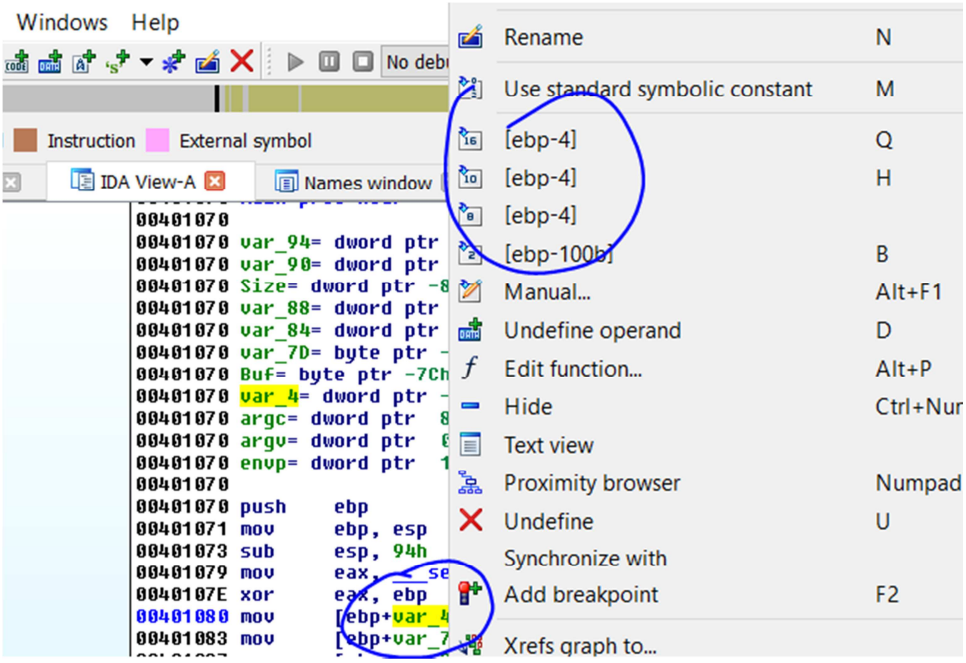
That's why in EBP based functions, once the called EBP function value is saved with PUSH EBP and then ESP is moved to EBP, 000000000 is shown as a horizon and above we see the minus signs (-) and downward plus sings (+)

var_4 has the -00000004 before because it is taking EBP as a BASE or zero. The math address would be EBP-4.

Downward, **argc** would be **EBP+8** seeing the left column.

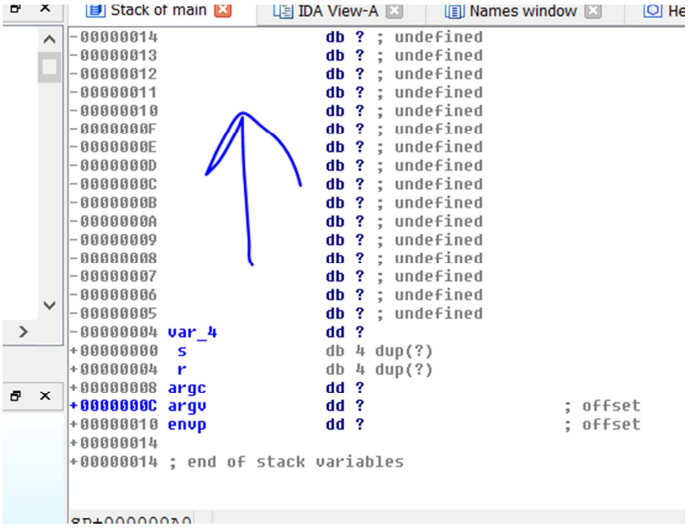


We can verify that in the disassembly inside the main function where it uses var_4. Right click there.



Going back to the static stack distribution.

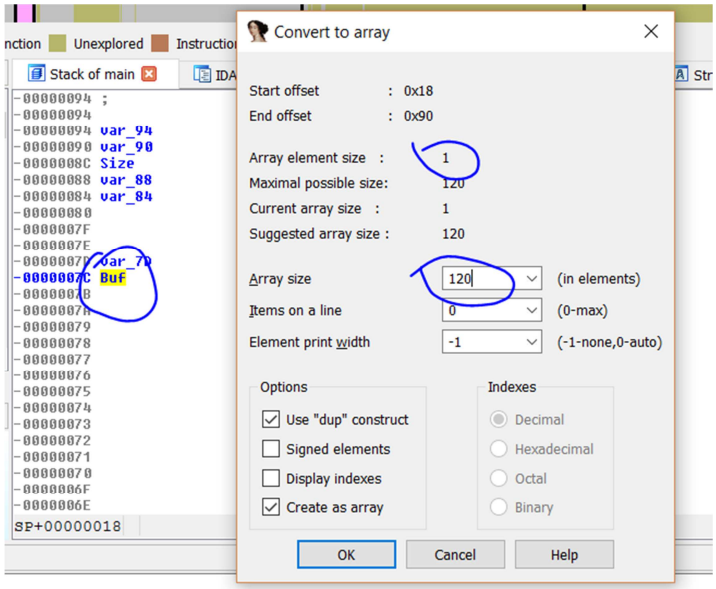
When we see there an empty space where there are no contiguous variables is possibly because above there is a buffer (more ahead, we'll see cases where an empty space is a structure). Let's scroll up a bit.



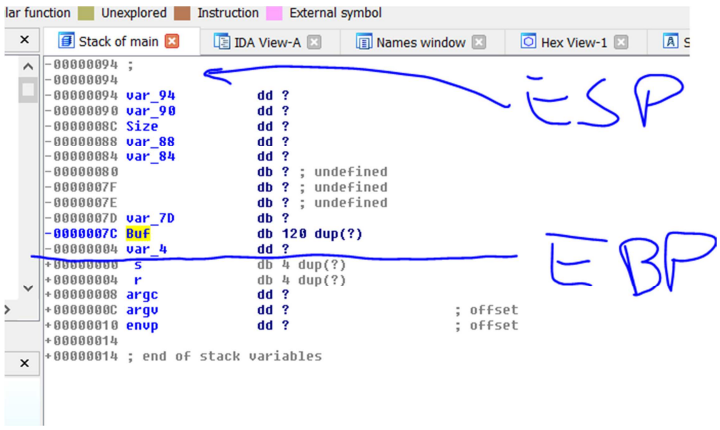
There, we see Buf that is the first variable above the empty zone. Right click - ARRAY.

```
-00000094 ;
-00000094
-00000094 var_94          dd ?
-00000090 var_90          dd ?
-0000008C Size            dd ?
-00000088 var_88          dd ?
-00000084 var_84          dd ?
-00000080                 db ? ; undefined
-0000007F                 db ? ; undefined
-0000007E                 db ? ; undefined
-0000007D var_7D          db ?
-0000007C Buf             db ?
-0000007B                 db ? ; undefined
-0000007A                 db ? ; undefined
-00000079                 db ? ; undefined
-00000078                 db ? ; undefined
-00000077                 db ? ; undefined
-00000076                 db ? ; undefined
-00000075                 db ? ; undefined
-00000074                 db ? ; undefined
```

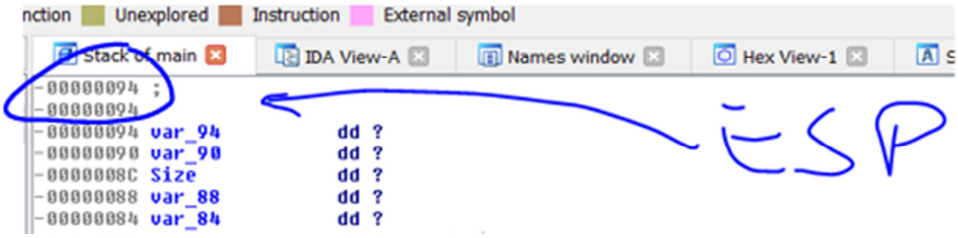
There, it is the array size = 120 because it is compound with 120 elements of 1 byte.



Now, the stack representation looks better.



We see the EBP base. Remember that once EBP and ESP are equal in MOV EBP, ESP then, it subtracts 0x94 to ESP and ESP stays working above the variable zone.



We see the zone where ESP is after that SUB ESP, 0x94.

On the left, we see -00000094 that will be ESP=EBP-094. Then, it will continue going up while working among other subfunctions, etc. But while it is inside the main function and until it quits it will work from 0x94 up because it will respect the reserved part for variables.

Once, we reviewed the static stack view we reverse the variable because the arguments are known (**argc**, **argv**, etc.)

```
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0
```

var_4 is the COOKIE_SEGURIDAD variable. It reads the value XORed with EBP and saves it on the stack to protect it fromoverflows. So, let’s rename it.

```
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 COOKIE_SEGURIDAD= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    sub_4011B0
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size]
004010AE push    eax             ; Size
004010AF lea     ecx, [ebp+Buf]
004010B2 push    ecx             ; Buf
153) (499,242) 00000480 00401080: main+10
```

The printf API when it doesn't have symbols it isn't shown, but seeing the strings it prints in a console and seeing the 0x4011B0 function.

```
Names window   Hex View-1   Structures   Enums
004011B0 arg_0= dword ptr  8
004011B0 arg_4= byte ptr  0Ch
004011B0
004011B0 push    ebp
004011B1 mov     ebp, esp
004011B3 sub     esp, 8
004011B6 call    sub_401000
004011BB lea     eax, [ebp+arg_4]
004011BE mov     [ebp+var_4], eax
004011C1 mov     ecx, [ebp+var_4]
004011C4 push    ecx
004011C5 push    0
004011C7 mov     edx, [ebp+arg_0]
004011CA push    edx
004011CB push    1
004011CD call    ds:__acrt_iob_func
004011D3 add     esp, 4
004011D6 push    eax
004011D7 call    sub_401040
004011DC add     esp, 10h
004011DF mov     [ebp+var_8], eax
004011E2 mov     [ebp+var_4], 0
004011E9 mov     eax, [ebp+var_8]
004011EC mov     esp, ebp
004011EE pop     ebp
004011EF retn
5D7 004011D7: sub_4011B0+27
```

And there, in 0x401040.

```
Names window   Hex View-1   Structures   Enums
00401040 arg_0= dword ptr  8
00401040 arg_4= dword ptr  0Ch
00401040 arg_8= dword ptr  10h
00401040 arg_C= dword ptr  14h
00401040
00401040 push    ebp
00401041 mov     ebp, esp
00401043 mov     eax, [ebp+arg_C]
00401046 push    eax
00401047 mov     ecx, [ebp+arg_8]
0040104A push    ecx
0040104B mov     edx, [ebp+arg_4]
0040104E push    edx
0040104F mov     eax, [ebp+arg_0]
00401052 push    eax
00401053 call    sub_401030
00401058 mov     ecx, [eax+4]
0040105B push    ecx
0040105C mov     edx, [eax]
0040105E push    edx
0040105F call    ds:__stdio_common_vfprintf
00401065 add     esp, 18h
00401068 pop     ebp
00401069 retn
00401069 sub_401040 endp
00401069
)440 00401040: sub_401040
```

Let's rename 0x4011B0 as **printf**.

```
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    printf
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size]
004010AE push    eax             ; Size
```

Let’s go ahead.

```
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
```

xrefs to Size

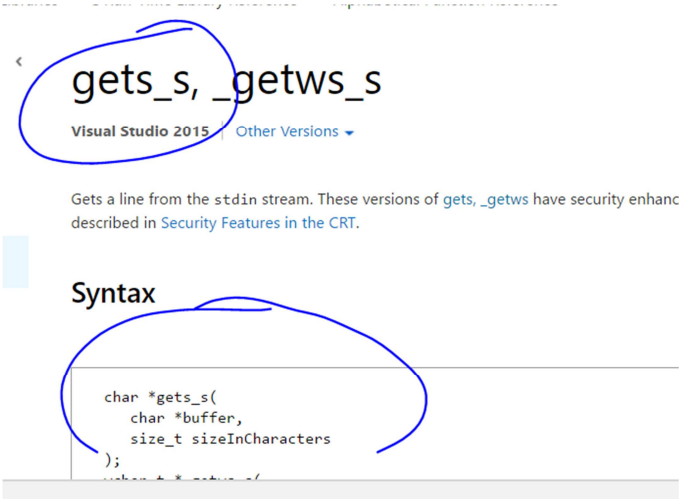
Directior	Ty	Address	Text
	w	main+21	mov [ebp+Size], 8
Down	r	main+38	mov eax, [ebp+Size]
Down	r	main+BE	mov edx, [ebp+Size]

OK Cancel Search Help

The size variable starts with 8 and it never changes its value. There are just two readings among the next references. So, let’s rename that size variable as **size_CONST_8**.

```
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size_CONST_8], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    printf
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size_CONST_8]
004010AE push    eax             ; Size
004010AF lea     ecx, [ebp+Buf]
004010B2 push    ecx             ; Buf
004010B3 call    ds:gets_s
004010B9 add     esp, 8
```

Then, the call to get_s is an evolution of the gets function, but with a character limit that you can type. In this case, the max value would be 8 that is pushed with the PUSH EAX and then, the LEA gets the variable address **Buf** that is the BUFFER.



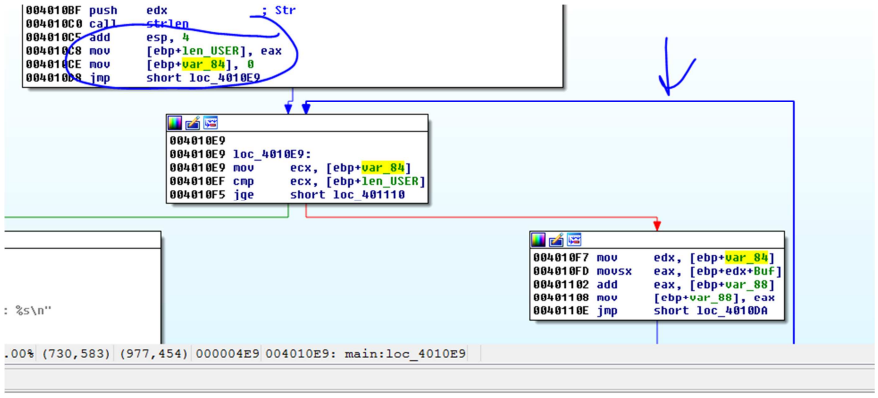
If we enter less than 8 characters and press ENTER, it will cut the input and it will return.

We know that in **Buf**, the User we typed will be and have 8 chars max.

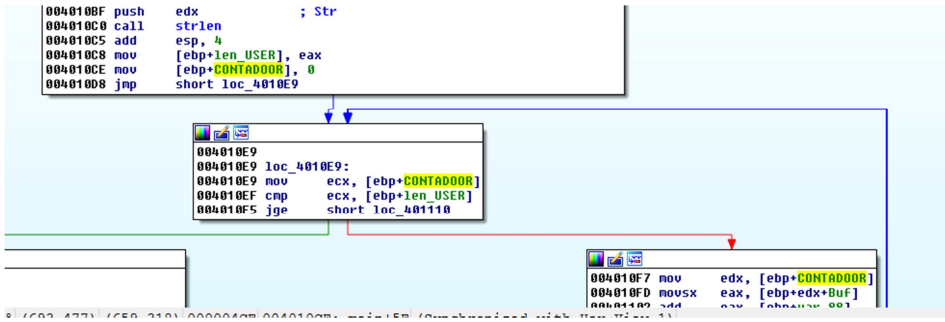
```
004010B3 call    ds:gets_s
004010B9 add     esp, 8
004010BC lea     edx, [ebp+Buf]
004010BF push    edx             ; Str
004010C0 call    strlen
004010C5 add     esp, 4
004010C8 mov     [ebp+var_90], eax
004010CE mov     [ebp+var_84], 0
004010D8 jmp     short loc_4010E9
```

Then, it pushes the buffer address again with PUSH EDX as the **strlen** API argument to get the string length that is in **Buf** and corresponds to **User**. It saves the size in that **var_90**. Let's rename **var_90** as **len_USER**.

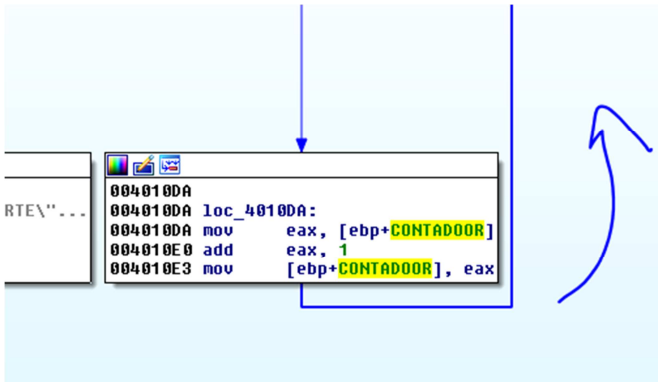
```
004010AE push    eax             ; Size
004010AF lea     ecx, [ebp+Buf]
004010B2 push    ecx             ; Buf
004010B3 call    ds:gets_s
004010B9 add     esp, 8
004010BC lea     edx, [ebp+Buf]
004010BF push    edx             ; Str
004010C0 call    strlen
004010C5 add     esp, 4
004010C8 mov     [ebp+len_USER], eax
004010CE mov     [ebp+var_84], 0
004010D8 jmp     short loc_4010E9
```



The blue arrow indicates a backward jump that can be a LOOP and there, it starts the LOOP counter, var_84, we also see that in 0x4010F5 there is a conditional jump that evaluates its exit condition. The counter starts in 0 and it will exit when being greater or equal to what we typed in **len_USER**.



The counter increases at the end of the LOOP here.



There, it moves the counter to EAX, increases it and then it saves it again.

```
004010F7 mov     edx, [ebp+CONTADOOR]
004010FD movsx   eax, [ebp+edx+Buf]
00401102 add     eax, [ebp+var_88]
00401108 mov     [ebp+var_88], eax
0040110E jmp     short loc_4010DA
```

.

It moves the first BUFFER byte from EBP+EDX+BUF because EBP+BUF is added to the counter that now is 0, but it will increase while the LOOP works. It will add all the character values I typed. var_88 starts at 0 and it will add the hex values of each BUFFER string char in each loop.

We see a new instruction: **MOVSX**.

MOVSX and MOVZX

Both take a byte and move it to a register. MOVZX fills the high bytes with 0. MOVSX takes into account the sign. If it is positive, less or equal to 0x7F, it fills with 0 and if it is negative, 0x80 or greater, it fills con 0xFF.

MOVZX EAX,[XXXX]

If the content of XXXX is 0x40, EAX will be 0x00000040.

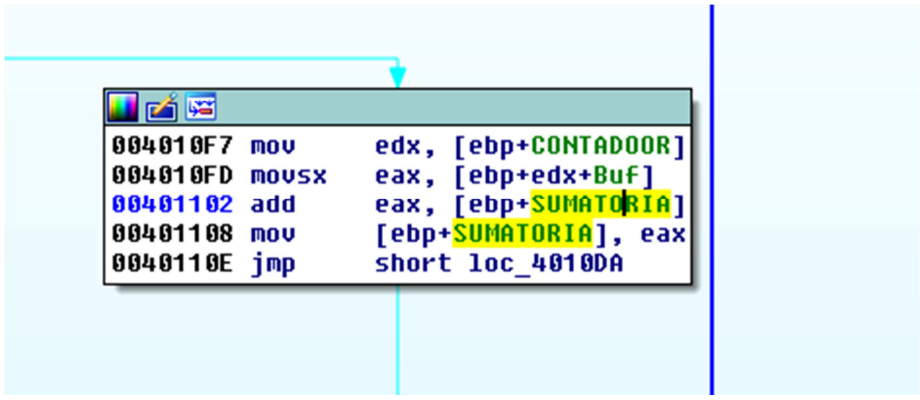
We also have MOVZX EAX,CL.

This is similar. It will take byte value and it will complete the high bytes with 0.

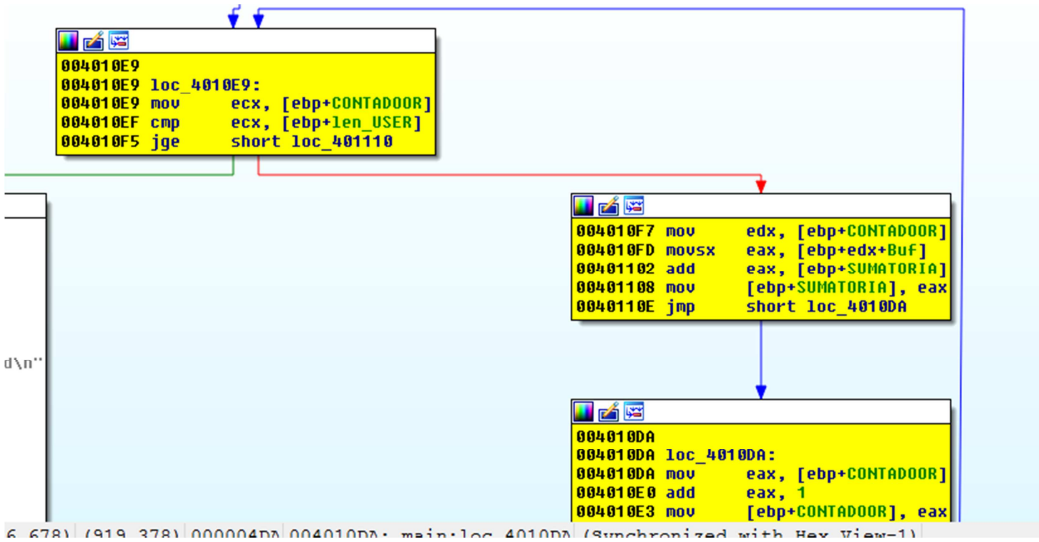
MOVSX EAX,CL

It takes into account the sign. If CL is 0x40, EAX will be 0x00000040 and if it were 0x85 as it is negative, EAX will be 0xFFFFFF85.

As we typed letters and number in the console and they are positive hex values, we won’t have troubles. It will add the values and save one by one.

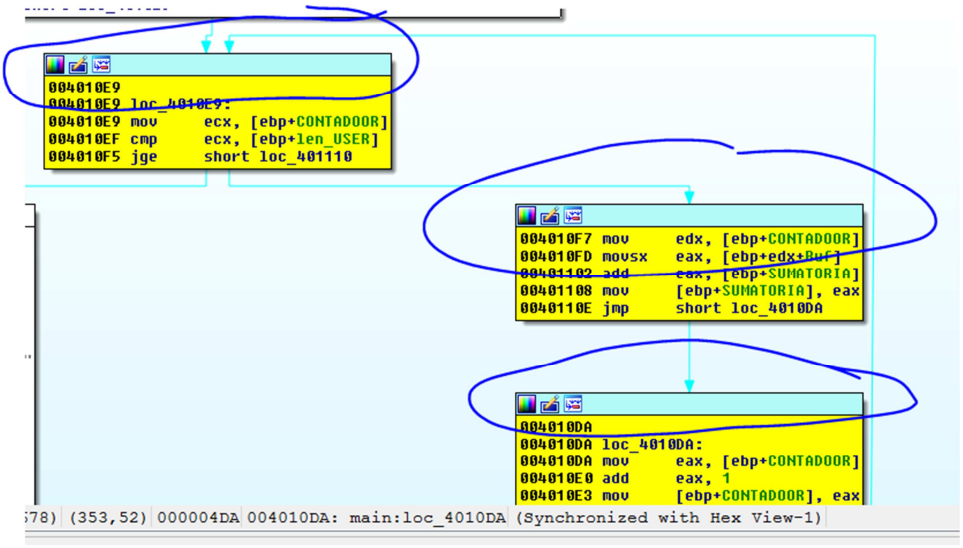


The LOOP is a char addition. Let's paint it the same color.

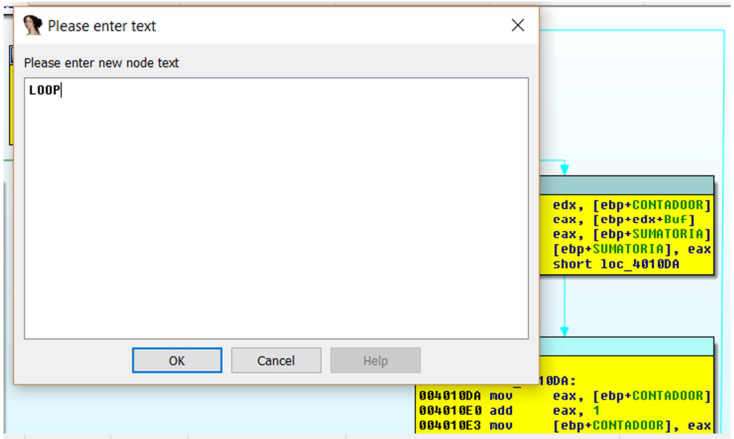


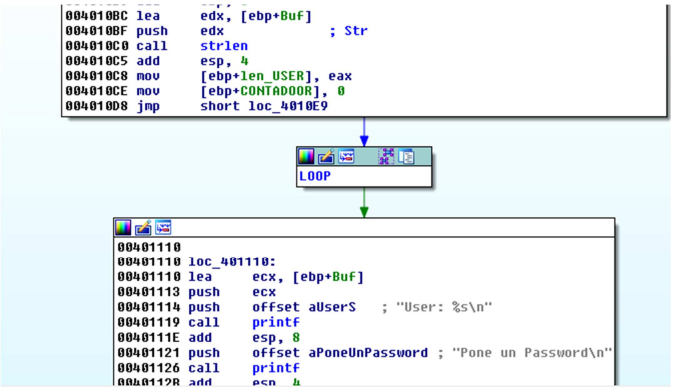
I zoomed them in a little dragging and releasing the below block a little up.

Some people group them to be more comfortable. By holding CTRL and clicking on the upper bar of each block.



Right click - GROUP NODES and we name it, for example, LOOP.





If we want to see something, we select UNGROUP NODES.

It prints the USER and asks you to type a PASSWORD.

```
LOOP

00401110
00401110 loc_401110:
00401110 lea     ecx, [ebp+Buf]
00401113 push    ecx
00401114 push    offset aUserS   ; "User: %s\n"
00401119 call    printf
0040111E add     esp, 8
00401121 push    offset aPoneUnPassword ; "Pone un Password\n"
00401126 call    printf
0040112B add     esp, 4
0040112E mov     edx, [ebp+Size_CONST_8]
```

Then, it calls **get_s** again using the same buffer and max.

```
00401126 call    printf
0040112B add     esp, 4
0040112E mov     edx, [ebp+Size_CONST_8]
00401134 push    edx             ; Size
00401135 lea     eax, [ebp+Buf]
00401138 push    eax             ; Buf
00401139 call    ds:gets_s
```

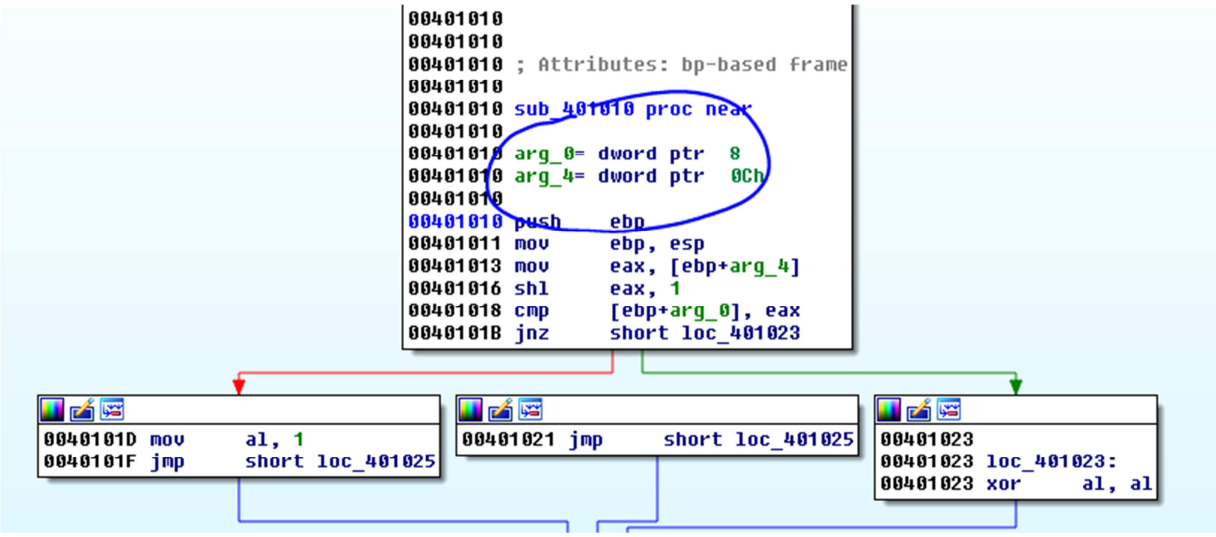
It can reuse the same BUFFER for the PASSWORD. Anyways, it already calculated the chars hex values addition and it won't use the USER string anymore.

```
0040113F add     esp, 8
00401142 lea     ecx, [ebp+Buf]
00401145 push    ecx             ; Str
00401146 call    ds:atoi
0040114C add     esp, 4
0040114F mov     [ebp+var_94], eax
```

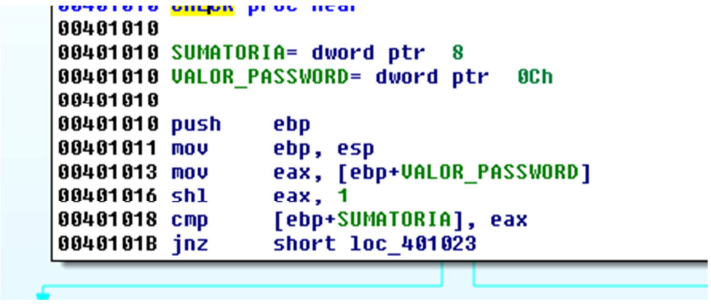
Now, it will take the PASSWORD and convert it into hex as in the previous example using ATOI.

```
00401142 lea     ecx, [ebp+Buf]
00401145 push    ecx             ; Str
00401146 call    ds:atoi
0040114C add     esp, 4
0040114F mov     [ebp+VALOR_PASSWORD], eax
00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax
00401163 call    sub_401010
00401168 add     esp, 8
```

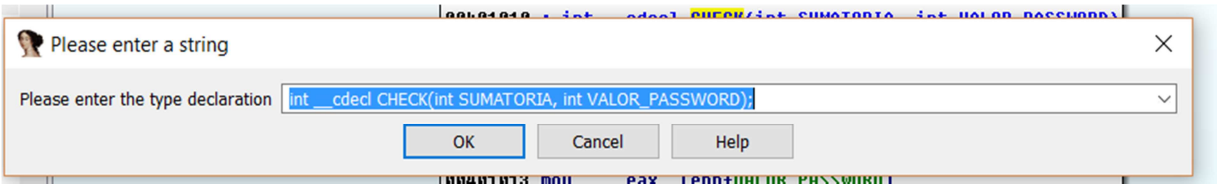
There, it will push the VALOR_PASSWORD with the PUSH EDX and the ADDITION with the PUSH EAX. Those will be the two arguments pushed in the 0x401010 function. Let's enter it.



We see the two arguments. VALOR_PASSWORD is below because it was pushed on the stack and the second one, ADDITION, is above.



I rename them according to that. Then, to verify if it is OK, right click on sub_0x401010 and select SET_TYPE.



With that, IDA will try to declare the function with its arguments to show them in the reference and we rename the function too as CHECK.

```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl CHECK(int VALOR_PASSWORD, int SUMATORIA)
00401010 CHECK proc near
00401010
00401010 VALOR_PASSWORD= dword ptr  8
00401010 SUMATORIA= dword ptr  0Ch
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+SUMATORIA]
00401016 shl     eax, 1
00401018 cmp     [ebp+VALOR_PASSWORD], eax
0040101B jnz     short loc_401023
```

```
0040101D mov     al, 1
0040101F jmp     short loc_401025
```

```
00401021 jmp     short loc_401025
```

```
00401023
00401023 loc_401023:
00401023 xor     al, al
```

And if we go to the reference.

```
0040113F add     esp, 8
00401142 lea     ecx, [ebp+Buf]
00401145 push    ecx             ; Str
00401146 call    ds:atoi
0040114C add     esp, 4
0040114F mov     [ebp+VALOR_PASSWORD], eax
00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx             ; VALOR_PASSWORD
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax             ; SUMATORIA
00401163 call    CHECK
00401168 add     esp, 8
0040116B mov     [ebp+var_7D], al
0040116E movzx   ecx, [ebp+var_7D]
```

We see that IDA spreads that and tells me, there, EAX has the ADDITION and EDX the VALOR_PASSWORD.

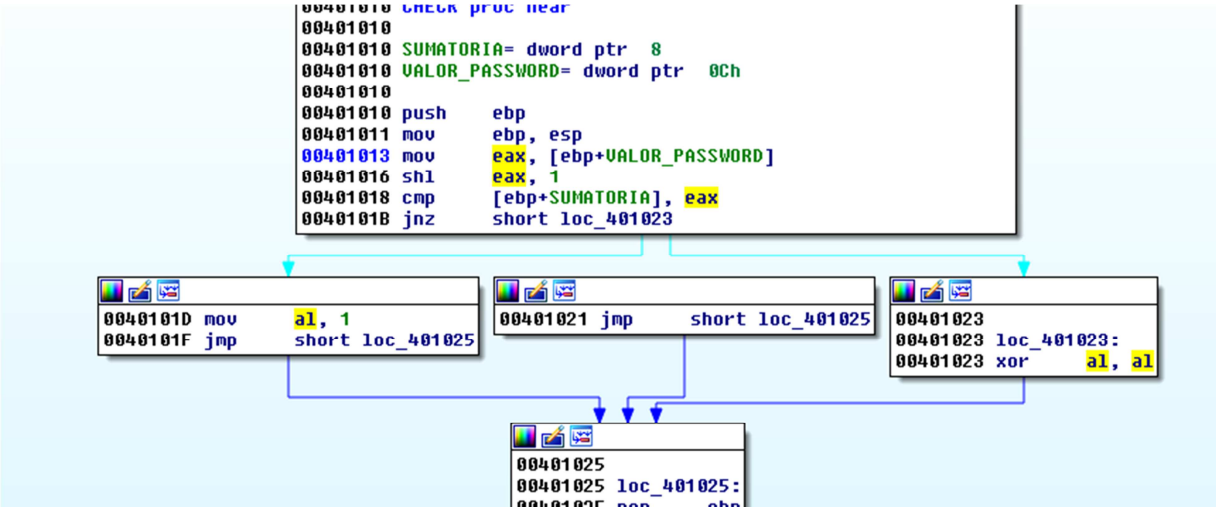
And what does the CHECK function do with both?

Let's see what it compares, but before it takes the PASSWORD value and does SHL EAX, 1.

Examples

`shl eax, 1` — Multiply the value of EAX by 2 (if the most significant bit is 0)

SHL shifts the bytes left filling with 0 the bytes that fall by a side, but specifically, SHL REG, 1 is the same that multiplying by 2.



It takes the password value and multipiles it by 2 and then, it compares it to the char addition of the USER.

```
Python>ord("p")
112
```

With that, we get the numerical value of character. We can make a formula that adds all the string chars of **pepe** as USER.

```
Python>hex(ord("p")+ord("e")+ord("p")+ord("e"))
0x1aa
```

The addition is 0x1AA. The value we typed will be multiplied by 2 before comparing this to 0x1AA. So, the right password should be a value that multiplied by 2 the result is 0x1AA.

X*2=0x1AA

Let's clear.

X=0x1AA/2 in decimal with ATOI was converted into HEXA.

```
Python>0x1aa/2
213
```

If in USER, I type **pepe** and in pass: **213** what will happen?

```
C:\Users\ricna\Desktop\New folder (3)\TEST_
Pone un User
pepe
User: pepe
Pone un Password
213
Good reverser CLAP CLAP
ENTER PARA IRTE
```

We didn't see when the comparison is equal inside the check function.

```
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+12]
00401016 shl     eax, 1
00401018 cmp     [ebp+SUMATORIA], eax
0040101B jnz     short loc_401023

0040101D mov     al, 1
0040101F jmp     short loc_401025

00401021 jmp     short loc_401025

00401023
00401023 loc_401023:
00401023 xor     al, al

00401025
00401025 loc_401025:
00401025 pop     ebp
```

If they aren't equal, it goes to the red block and returns 0. If they are equal, it goes to the green block and returns 1. Let's see what happens with that return value.

```
0040114F mov     [ebp+VALOR_PASSWORD], eax
00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx             ; VALOR_PASSWORD
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax             ; SUMATORIA
00401163 call    CHECK
00401168 add     esp, 8
0040116B mov     [ebp+var_7D], al
0040116E movzx   ecx, [ebp+var_7D]
00401172 test    ecx, ecx
00401174 jz      short loc_401185
```

It saves it there. Let's rename it as FLAG_EXITO that means SUCCESS_FLAG.

```
00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx             ; VALOR_PASSWORD
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax             ; SUMATORIA
00401163 call    CHECK
00401168 add     esp, 8
0040116B mov     [ebp+FLAG_EXITO], al
0040116E movzx   ecx, [ebp+FLAG_EXITO]
00401172 test    ecx, ecx
00401174 jz      short loc_401185

ffset aGoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
rintf
sp, 4
hort loc_401192

00401185
00401185 loc_401185:             ; "Bad reverser JUA
00401185 push    offset aBadReverserJua
0040118A call    printf
0040118F add     esp, 4

100.00% (149,1115) (960,202) 0000058A 0040118A: main+11A (Synchronized with Hex View-1)
```

If it is 0, it goes to BAD REVERSER, but if it is 1, it goes to GOOD BOY as it happened here.

I would like you to debug it and check all we reversed setting breakpoints and seeing the values in each case until the final check.

Ricardo Narvaja

Translated by: @IvinsonCLS