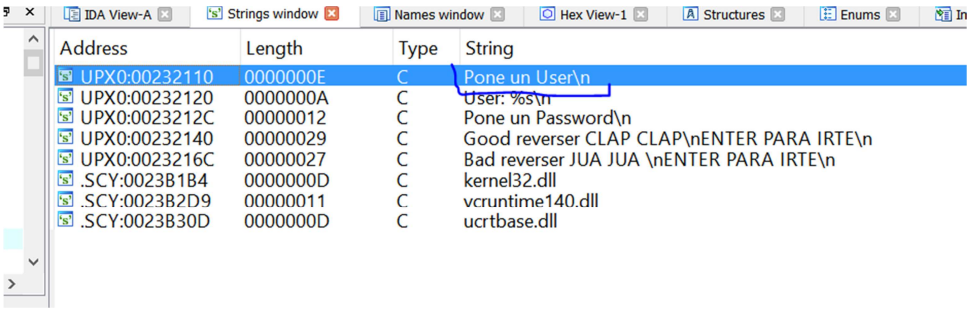
REVERSING WITH IDA PRO FROM SCRATCH
PART 18

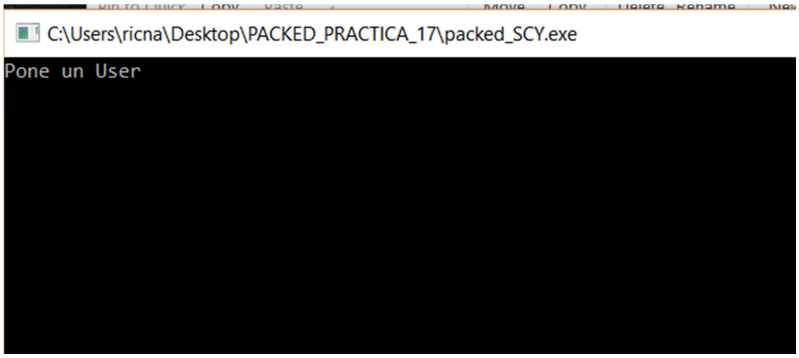
In the previous part, we unpacked the exercise executable and we made it run. Now, we will reverse it to see if we can make a keygen in Python.

It's not necessary to decompress to do a static analysis. We just need to get the OEP and do a TAKE MEMORY SNAPSHOT. Then, we copy the .idb file to other place and open it. This way, we could analyze it statically, but if we unpack it, we could debug it and that can help sometimes.

I load the unpacked file in IDA and I firstly see the strings.

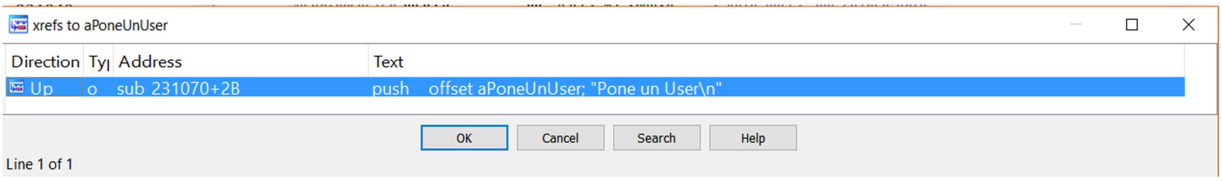


We know that the first thing the program does is printing: "Pone un user". In English, it means: "Enter a user".

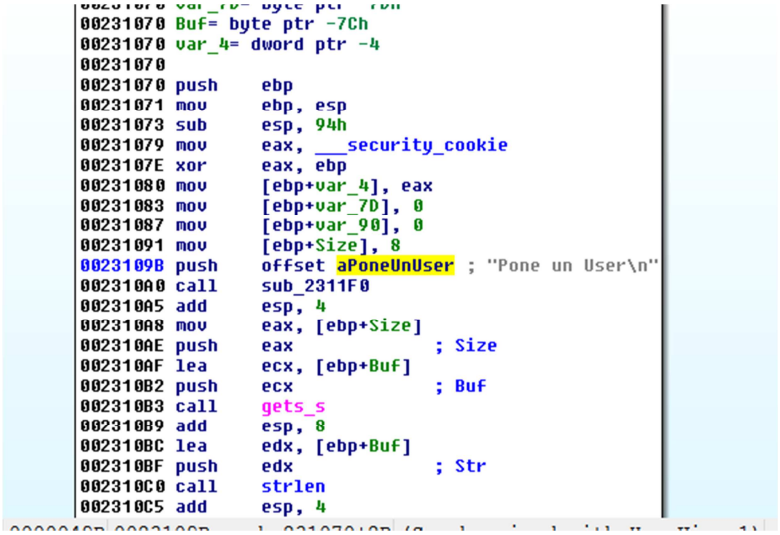


Double click on that string.

We find its reference with X.



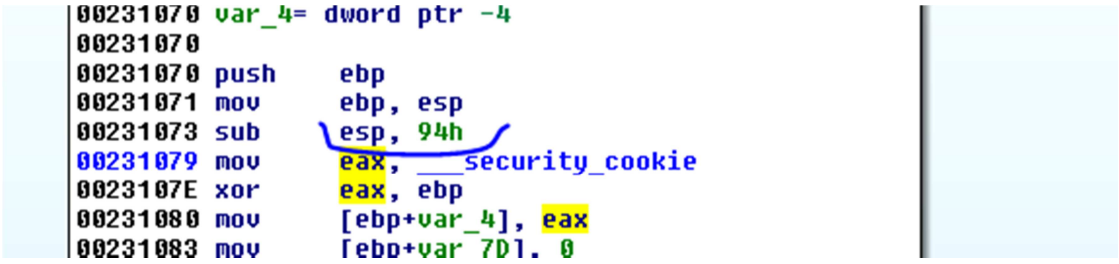
I go there.



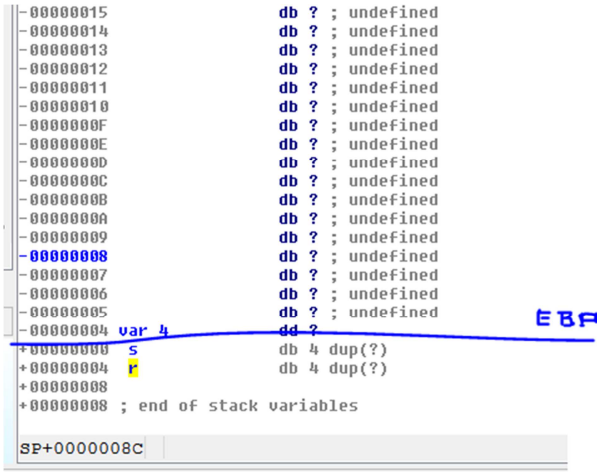
Let's reverse it statically from now on.

In the EBP-based functions, we said that it firstly saves the function EBP it called this with PUSH EBP in the stack and then it will use MOV EBP, ESP to set EBP with its reference value for this function from the place the variable argument positions and buffers are calculated.

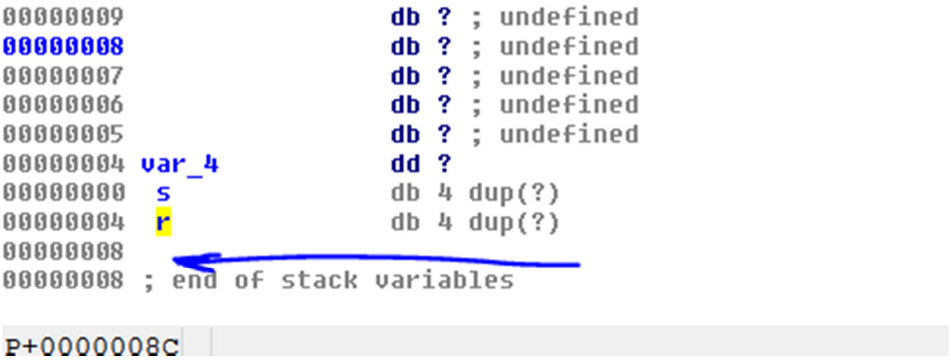
It reserves 0x94 bytes for the local variables and buffers from the base value of EBP.



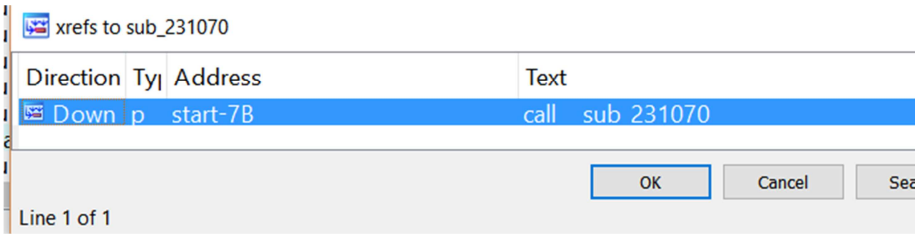
By double clicking on any variable or argument, the loader shows the stack static view.

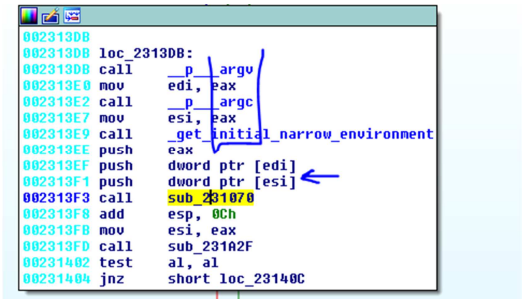


There is a function without arguments because those are pushed with the PUSHes before calling the function and they would be below the return address **r**, in this case, there is nothing below **r** so it it is a non-argument function.

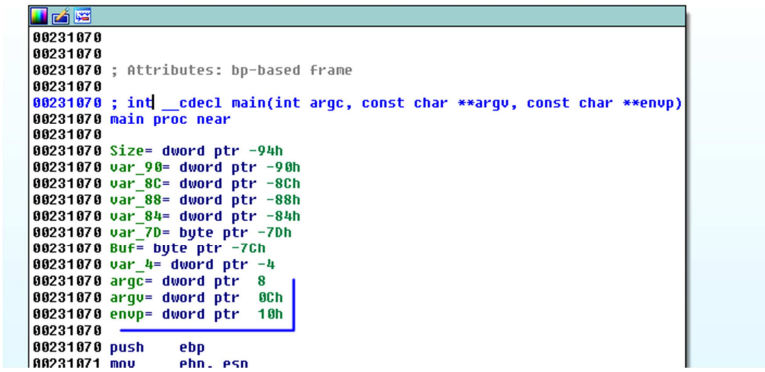


This is the same case as before. This is the main function and it has arguments that are: **argv** and **argc**, etc, but it doesn't use them inside the function, IDA doesn't consider them.

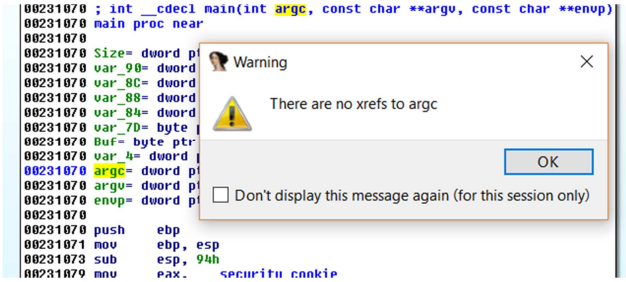




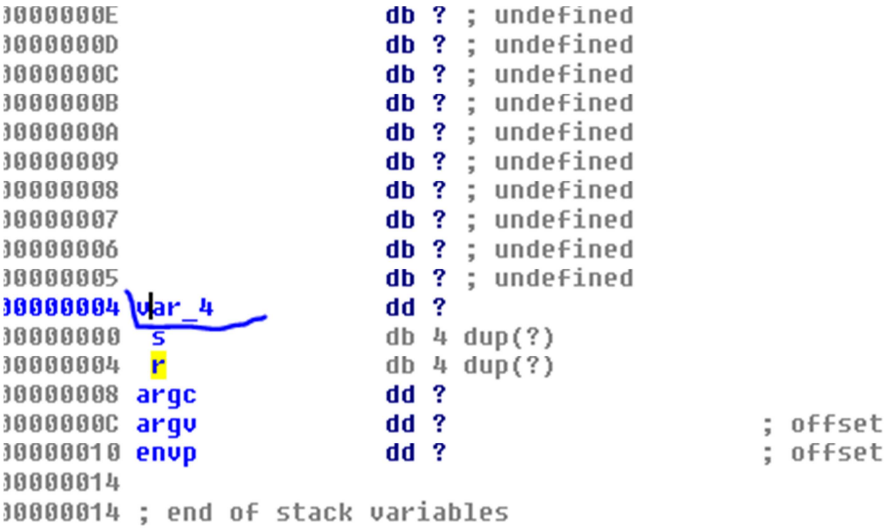
Let’s rename the function as main and IDA adds it the 3 arguments automatically.



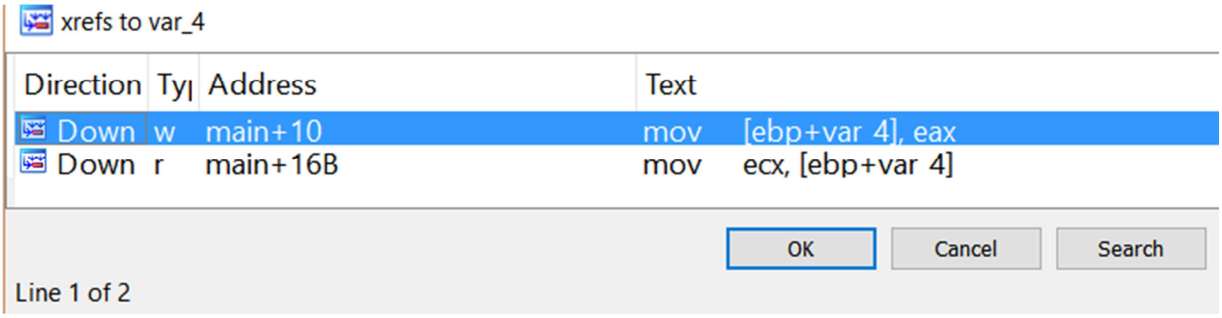
Also, if we press X on any of the three args.



They are not used, they’re not important.



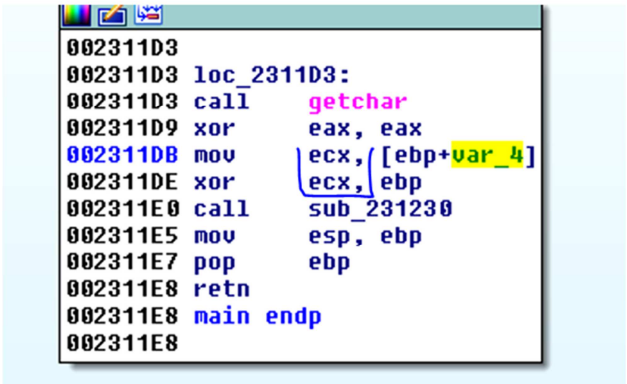
Coming back to the stack static view, we see the arguments below the return address as they should. Then, **s** means the STORED EBP. As we said it saves previous function EBP with PUSH EBP and above we have the variable space that it normally has that var_4 variable to protect the stack against buffer overflows.



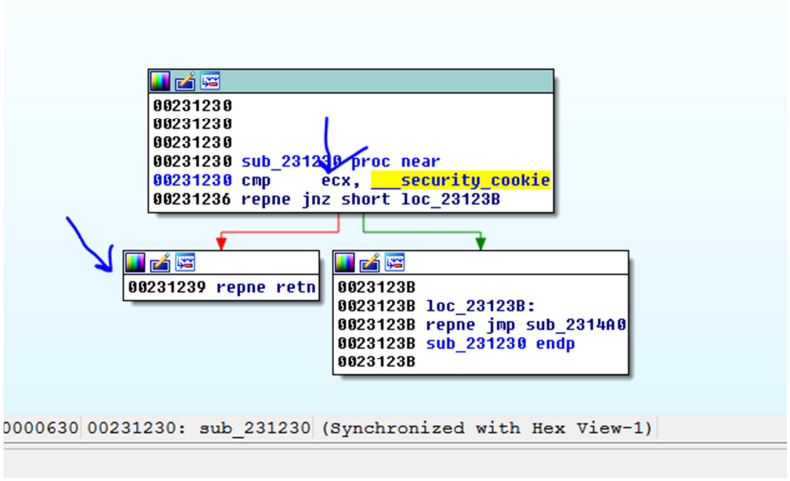
It has two references. One at the function start when it saves the security cookie value in the stack.

That is a random value XORed with EBP and saved there in var_4 when starting the function and the other reference is here.

cia es aqui.

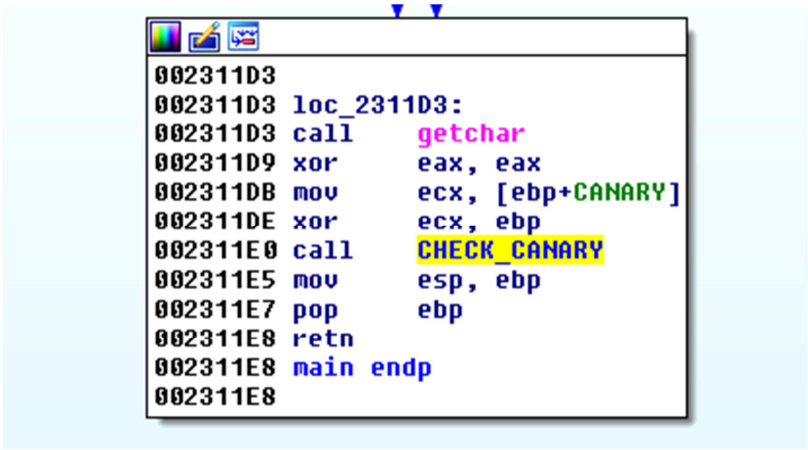


Where it recovers the original saved value again and it XORs it again with EBP to recover the original value in ECX and inside that CALL, it will check it.

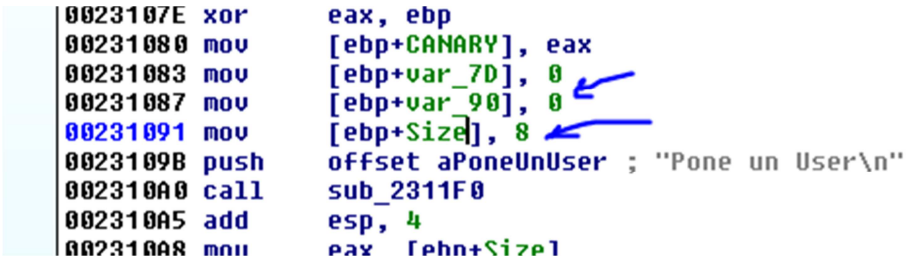


If everything's OK, it will return, but if ECX doesn't have the _security_cookie original value, it goes to the JMP towards EXIT and it avoids the function RET.

We'll see that it only can happen that it goes to EXIT if there was an OVERFLOW that overwritten the var_4 value inside the function. By now, rename var_4 as CANARY or COOKIE.

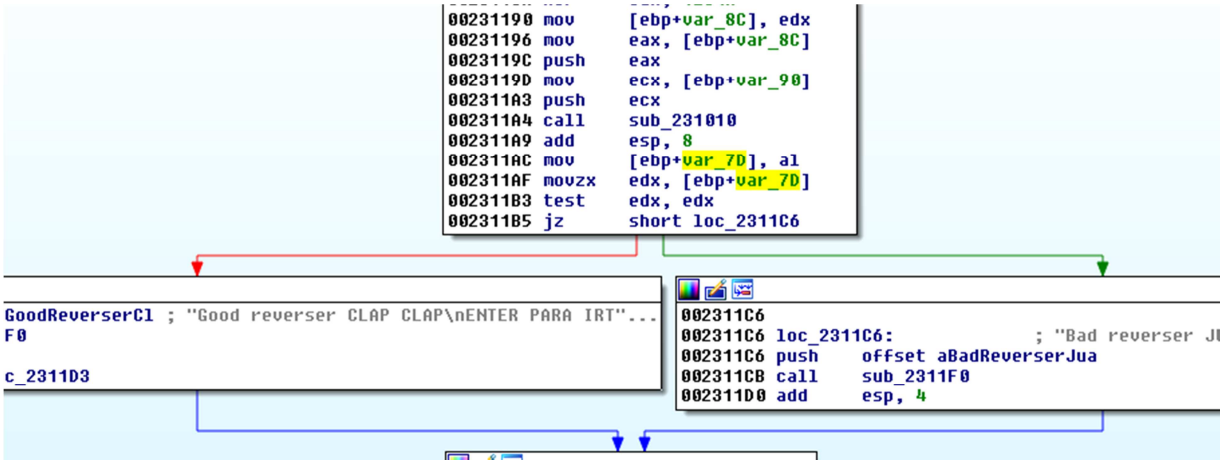


Now, it looks better.



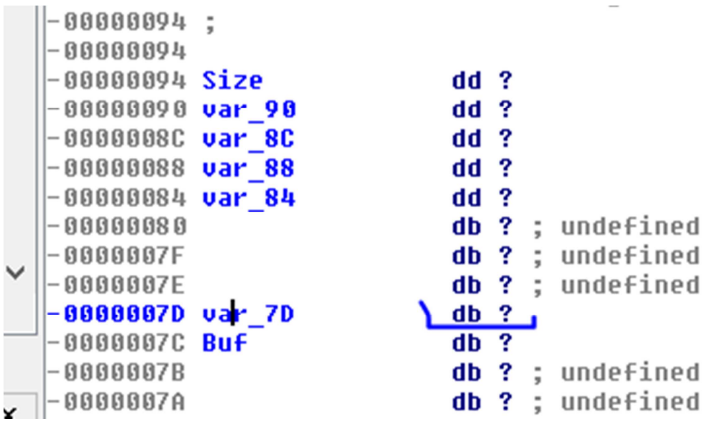
Then, we see two variables we don't know yet. They start from **0** and a variable called **size** and it starts from **8**.

If we see the **var_7d** references, they are used here.

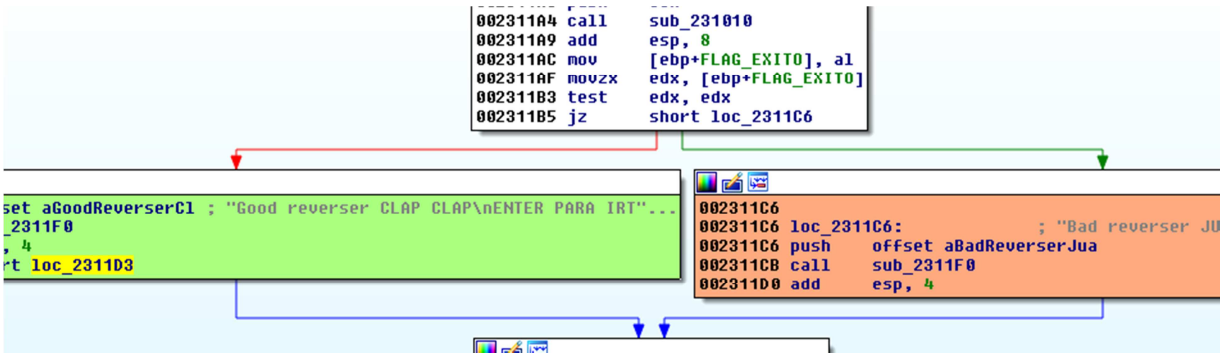


It saves the AL value in that variable when returning from a CALL and then, moves that byte to EDX to check if it is 0 or not to decide if we are good or bad reversers. So, it is a one-byte variable. Let's rename it as **FLAG_EXITO** that means **SUCCESS_FLAG** in English.

We verify in the static view that it is detected as byte.

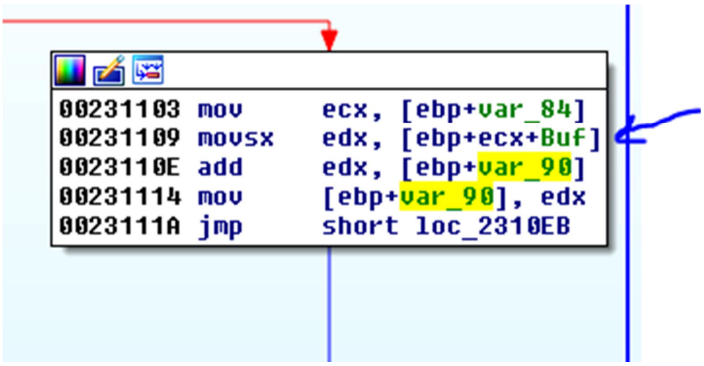


Rename it pressing N.

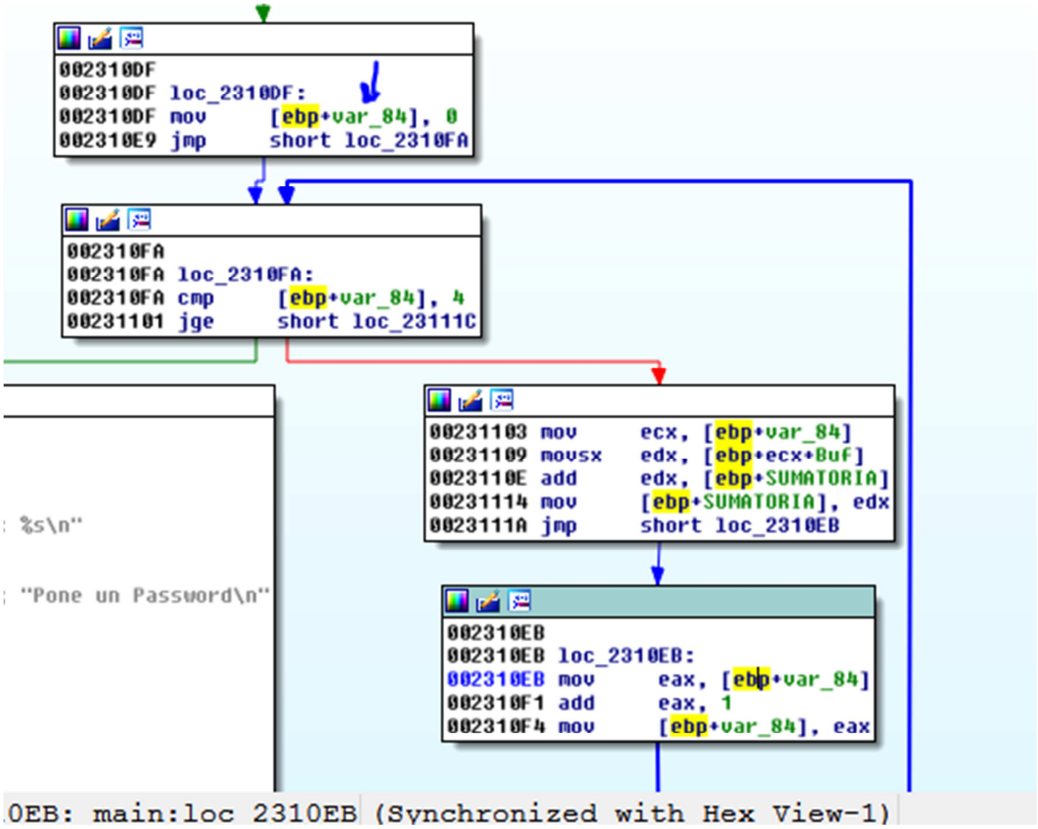


It looks better. It painted the good boy block in green and in red or orange to see the bad boy content.

Obviously, if it were only patching that JZ, that would be the exact point, but we'll try to reach the final of this.

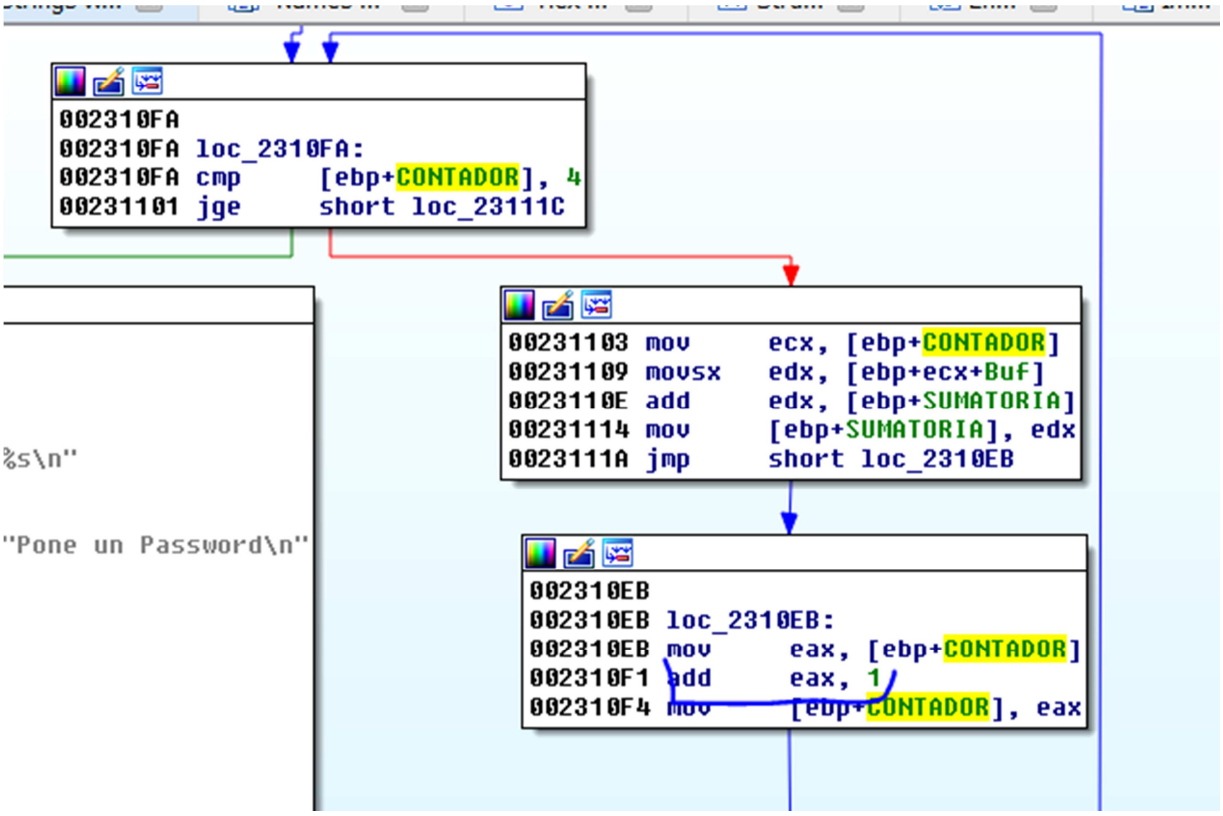


We see that the other var_90 variable that was 0 at the start, it adds the byte read from the Buf reading one by one and it moves them to EDX at 0x231109 and then, it adds 0 in the first loop and EDX always saves the addition of all bytes. We'll see what there is in the Buf content it is reading, but by now rename it as **SUMATORIA** or **ADDITION** in English.

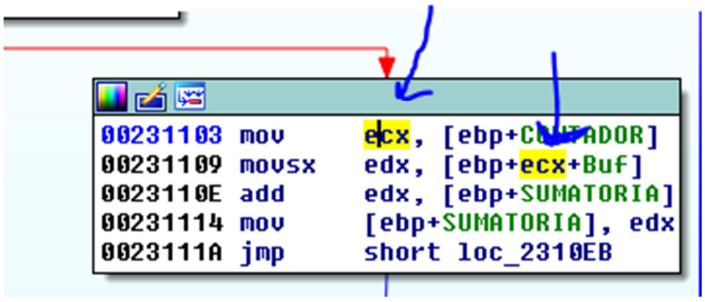


var_84 is the counter of that LOOP where it adds, it adds just the first four bytes because it exits when the value is greater or equals 4.

There, we see the counter and how it increases.



Obviously, it also adds at 0x231109 at the Buf start to read or add the next bytes.



That loop reads the Buf bytes and adds them saving the result in SUMATORIA.

Let's see the Buf content.

```
00231083 mov     [ebp+FLAG_EXITO], 0
00231087 mov     [ebp+SUMATORIA], 0
00231091 mov     [ebp+Size], 8
0023109B push    offset aPoneUnUser ; "Pone un User\n"
002310A0 call    sub_2311F0
002310A5 add     esp, 4
002310A8 mov     eax, [ebp+Size]
002310AE push    eax             ; Size
002310AF lea     ecx, [ebp+Buf]
002310B2 push    ecx             ; Buf
002310B3 call    gets_s
002310B9 add     esp, 8
002310BC lea     edx, [ebp+Buf]
002310BF push    edx             ; Str
002310C0 call    strlen
```

Buf equals 8 at the beginning as máximum with the name User using **gets_s** to enter by the keyboard.

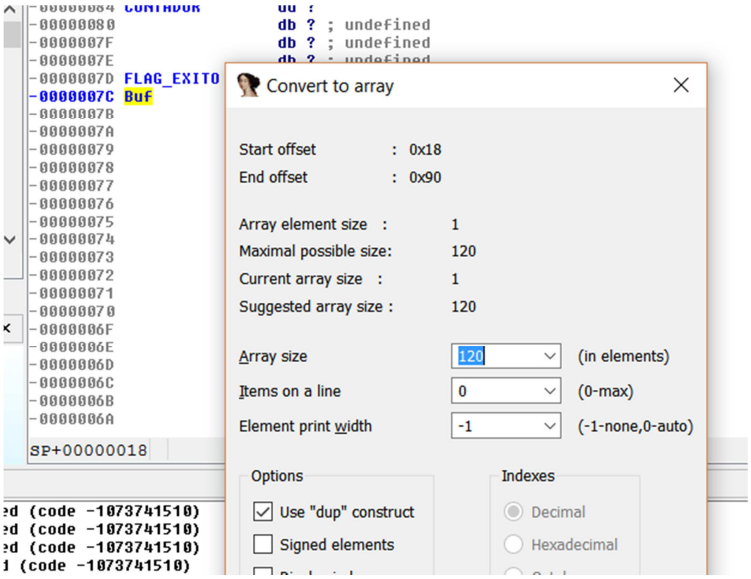
Let's rename that function as printf

```
00231087 mov     [ebp+SUMATORIA], 0
00231091 mov     [ebp+Size], 8
0023109B push    offset aPoneUnUser ; "Pone un User\n"
002310A0 call    sub_2311F0
002310A5 add     esp, 4
```

Ready.

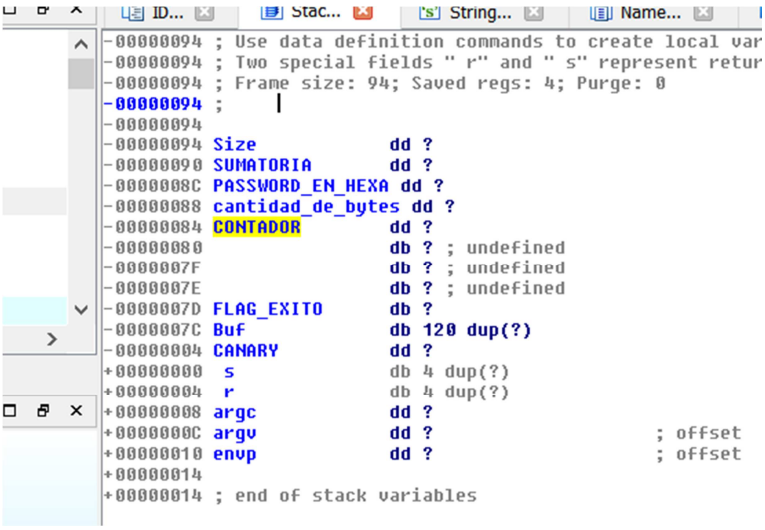
```
00231087 mov     [ebp+SUMATORIA], 0
00231091 mov     [ebp+Size], 8
0023109B push    offset aPoneUnUser ; "Pone un User\n"
002310A0 call    printf
002310A5 add     esp, 4
```

Also, in the static view, we see the Buf size with Right click- ARRAY.



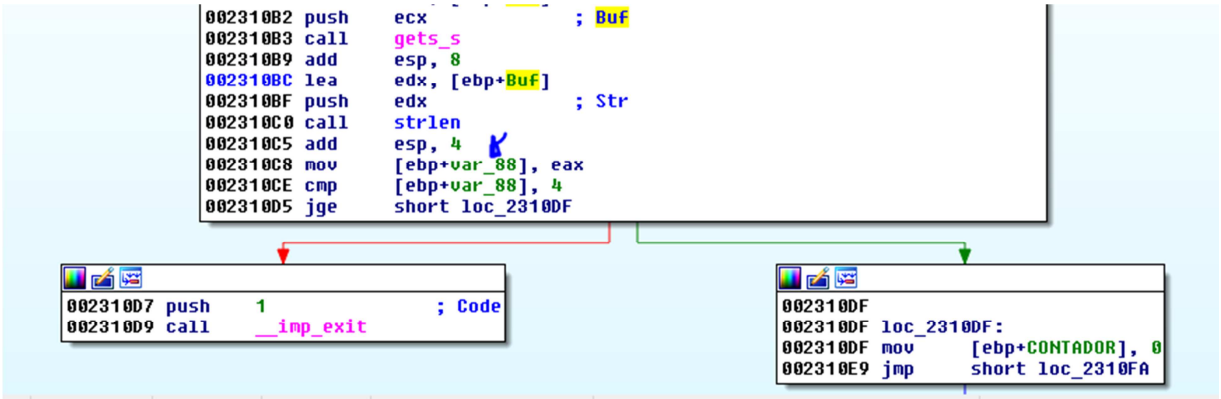
It matches the source code.

```
int main(int argc, char *argv[])
{
        bool resul = false;
        int pass;
        //int user;
        int sum=0;
        char buf[120];
        int max = 8;
        printf("Pone un User\n");
```

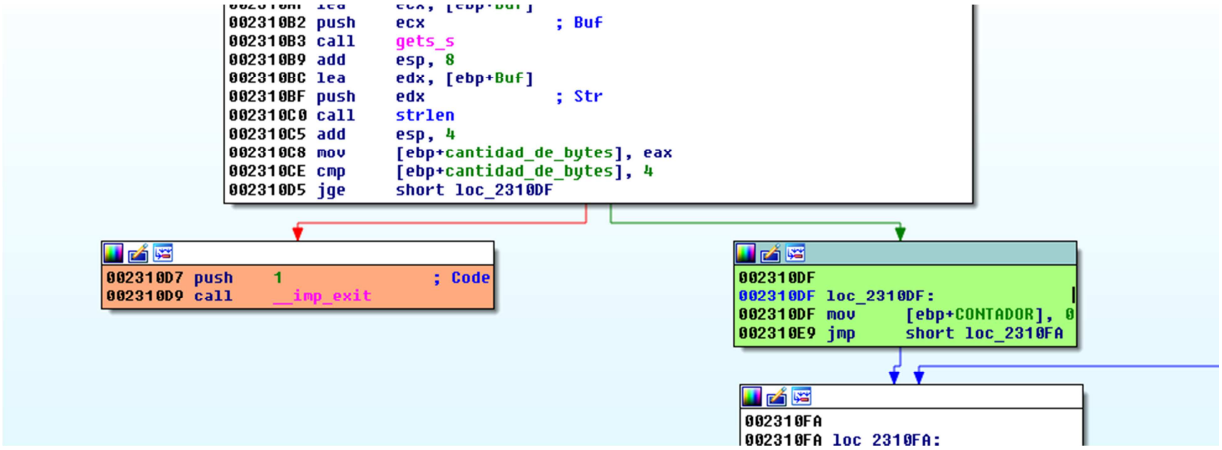


The stack representation is clearer.

After receiving in the Buf, it calls **strlen** to know the size of what you typed.

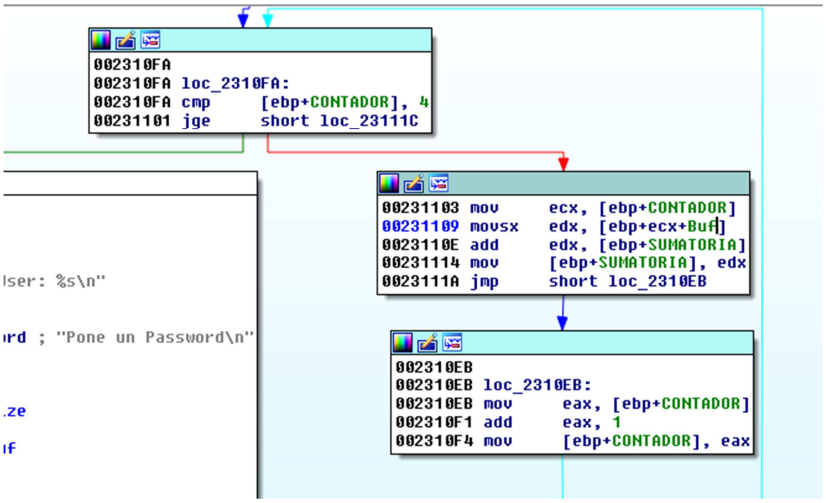


var_88 is the number of bytes you typed.

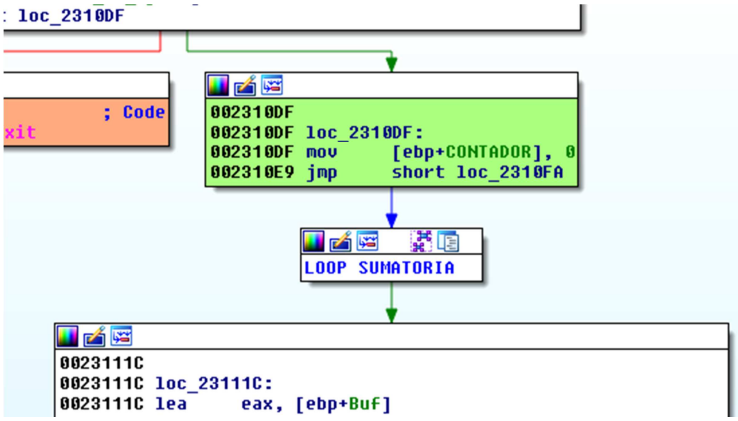


If it is less than 4, it goes to EXIT.

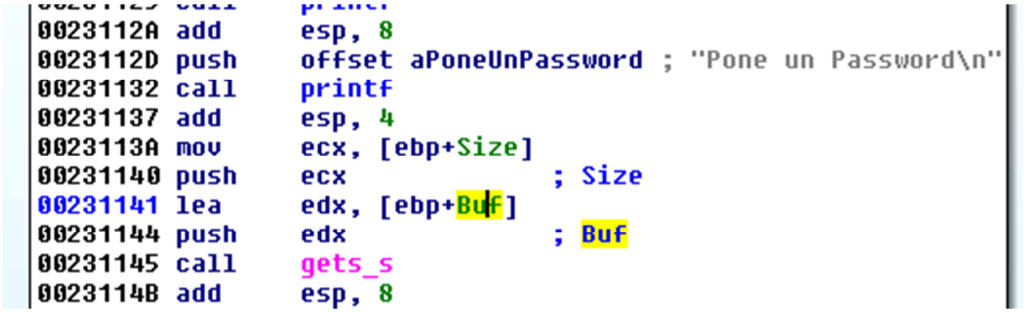
That LOOP adds the first four bytes of the User we type. So, let's regroup it to see it better. Click on each block bar pressing CTRL.



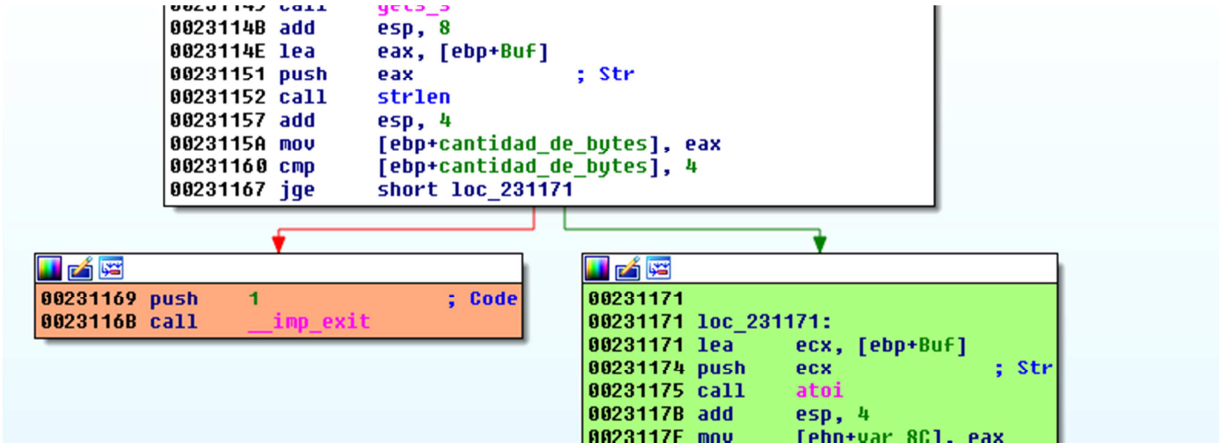
It looks better. With right click - GROUP NODES you can ungroup them just selecting UNGROUP.



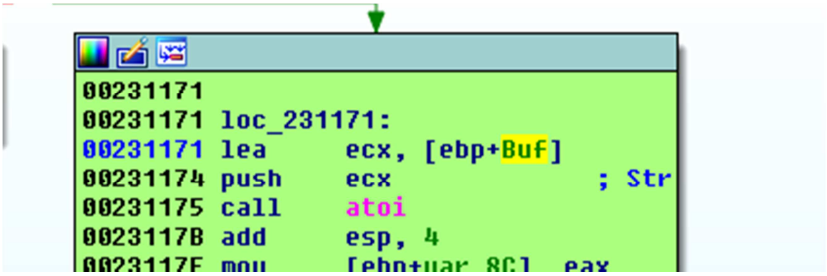
It uses Buf again to enter the password because it already saved the addition of the user's first 4 bytes.



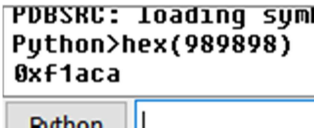
It uses **strlen** to know its size and if it is less than 4, it goes to EXIT.



If it is 4 or greater, it continues for the green block.



Then, it takes the password and converts it into hexadecimal with ATOI. In Python, it would be similar to the hex() function.



```
00231171
00231171 loc_231171:
00231171 lea     ecx, [ebp+Buf]
00231174 push    ecx             ; Str
00231175 call    atoi
0023117B add     esp, 4
0023117E mov     [ebp+PASSWORD_EN_HEXA], eax
00231184 mov     edx, [ebp+PASSWORD_EN_HEXA]
0023118A xor     edx, 1234h
00231190 mov     [ebp+PASSWORD_EN_HEXA], edx
00231196 mov     eax, [ebp+PASSWORD_EN_HEXA]
0023119C push    eax
0023119D mov     ecx, [ebp+SUMATORIA]
```

It XORs the password in hex with 0x1234 and it saves it again in the same variable.

```
00231184 mov     edx, [ebp+PASSWORD_EN_HEXA]
0023118A xor     edx, 1234h
00231190 mov     [ebp+PASSWORD_EN_HEXA], edx
00231196 mov     eax, [ebp+PASSWORD_EN_HEXA]
0023119C push    eax
0023119D mov     ecx, [ebp+SUMATORIA]
002311A3 push    ecx
002311A4 call    CHEQUEO_EXITO
002311A9 add     esp, 8
```

It will compare the user's first 4 bytes and the XORed hex value with 0x1234 of the password with CHEQUEO_EXITO. Its result makes it jump to good or bad boy.

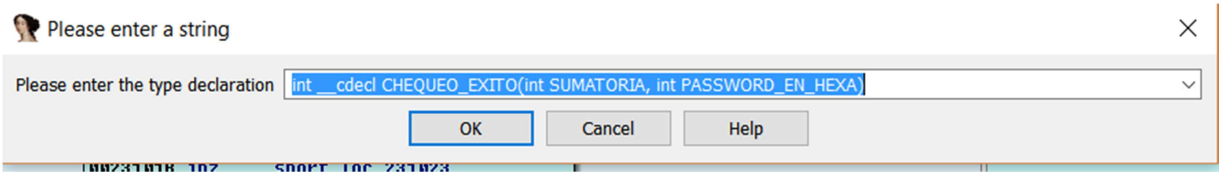
There, we see the two arguments, arg_4 will be PUSHed first.

```
00231190 mov     [ebp+PASSWORD_EN_HEXA], edx
00231196 mov     eax, [ebp+PASSWORD_EN_HEXA]
0023119C push    eax
0023119D mov     ecx, [ebp+SUMATORIA]
002311A3 push    ecx
002311A4 call    CHEQUEO_EXITO
002311A9 add     esp, 8
002311AC mov     [ebp+FLAG_EXITO], al
```

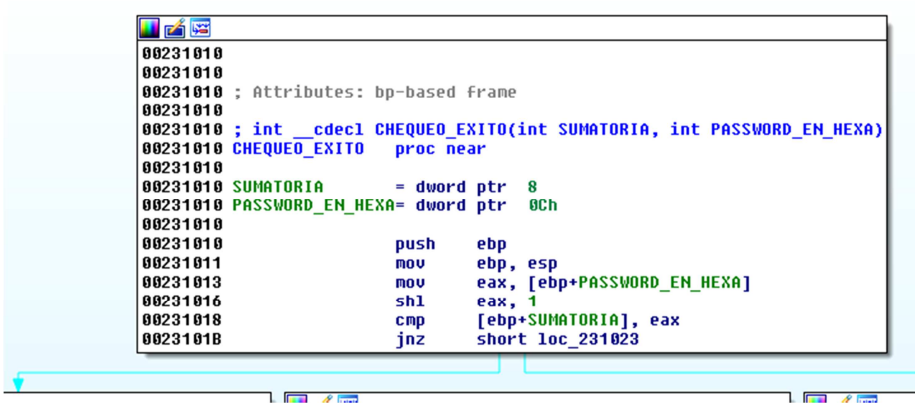
Let's rename both arguments inside the function.

```
00231010 CHEQUEO_EXITO   proc near
00231010
00231010 SUMATORIA       = dword ptr  8
00231010 PASSWORD_EN_HEXA= dword ptr  0Ch
00231010
00231010                 push    ebp
00231011                 mov     ebp, esp
00231013                 mov     eax, [ebp+PASSWORD_EN_HEXA
00231016                 shl     eax, 1
00231018                 cmp     [ebp+SUMATORIA], eax
0023101B                 jnz     short loc_231023
```

Let’s propagate the arguments with right click SET TYPE.

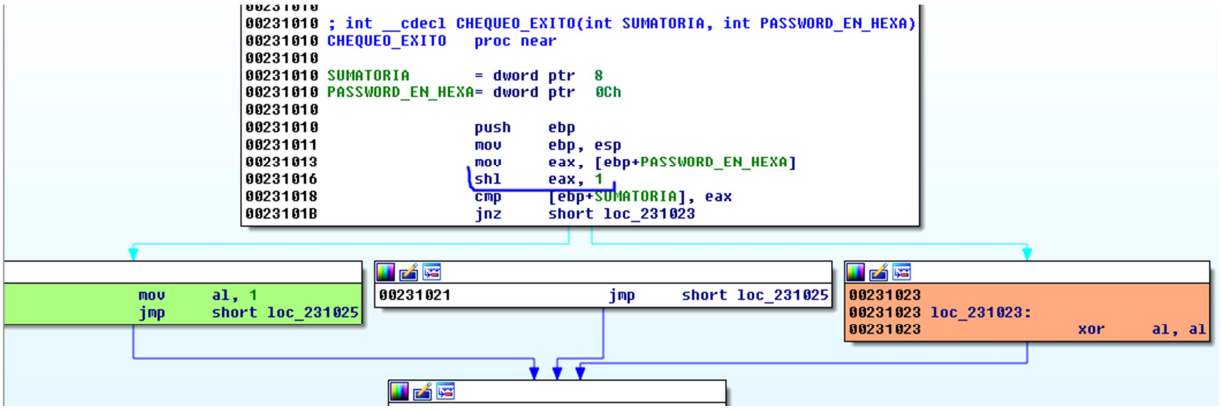


So, the function is declared.



We’ll see if the references match.

The blue messages that appear when we propagate match the names in the reference. So, everything’s OK.



Before comparing both, it does SHL EAX, 1 that is similar to multiply by 2.

So, if they are equal, it goes to the green block where it moves 1 to AL and it will return it as the FLAG_EXITO that decides if we are good or bad reversers.

Summarizing

It takes the USER's first 4 bytes and adds them.

It converts the PASSWORD into hex and XORs it with 0x1234, then it multiplies by 2.

We'll do a formula supposing we know the USER because the keygen is based on that. With some USER it finds the correspondent password.

X = PASSWORD converted into HEXA
(X ^ 0x1234)* 2 = SUMATORIA
If we clear X
X ^ 0x1234= (SUMATORIA/2)

X= (SUMATORIA/2) ^ 0x1234

The XOR function is invertible.

A ^ B = C

A = B ^ C

The thing is that the X value we need to find is equal to:

X= (SUMATORIA/2) ^ 0x1234

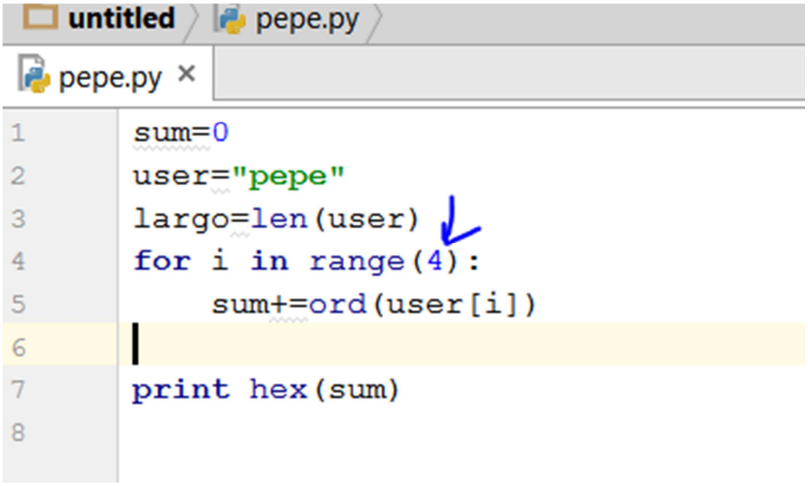
If my name were pepe which is valid because is less than 8 bytes, the byte addition would be.

pepe.py

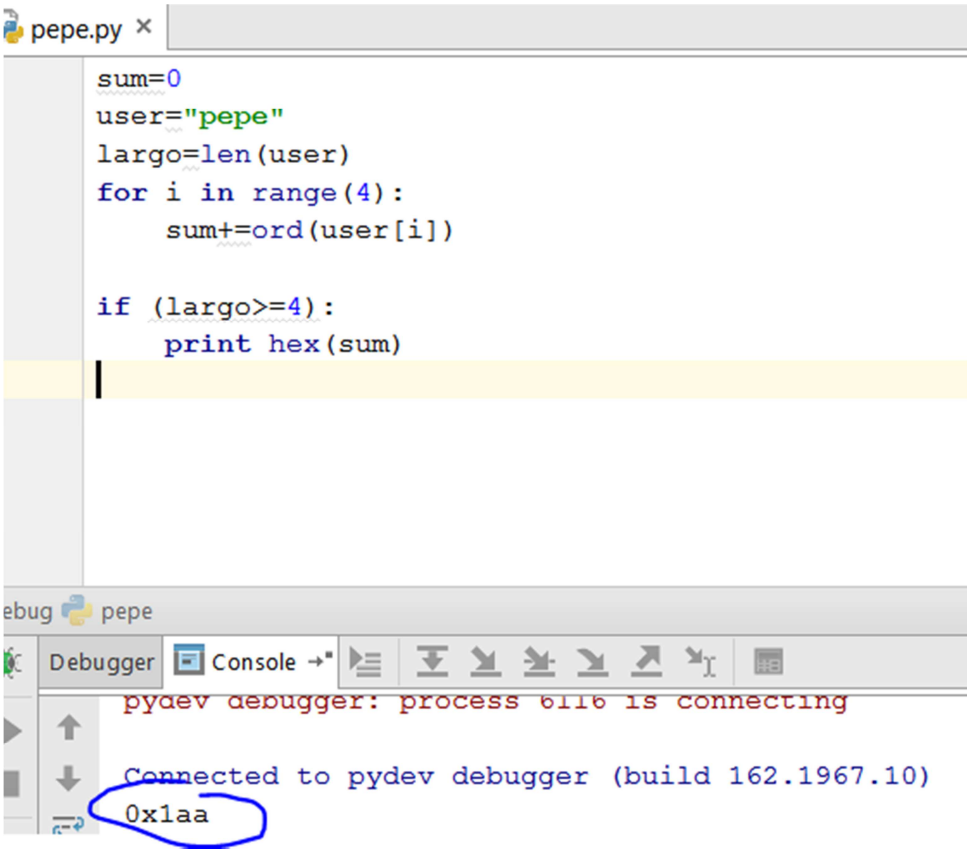
```
sum=0
user="pepe"
largo=len(user)
for i in range(largo):
    sum+=ord(user[i])

print hex(sum)
```

There, I got the addition for my user pepe.

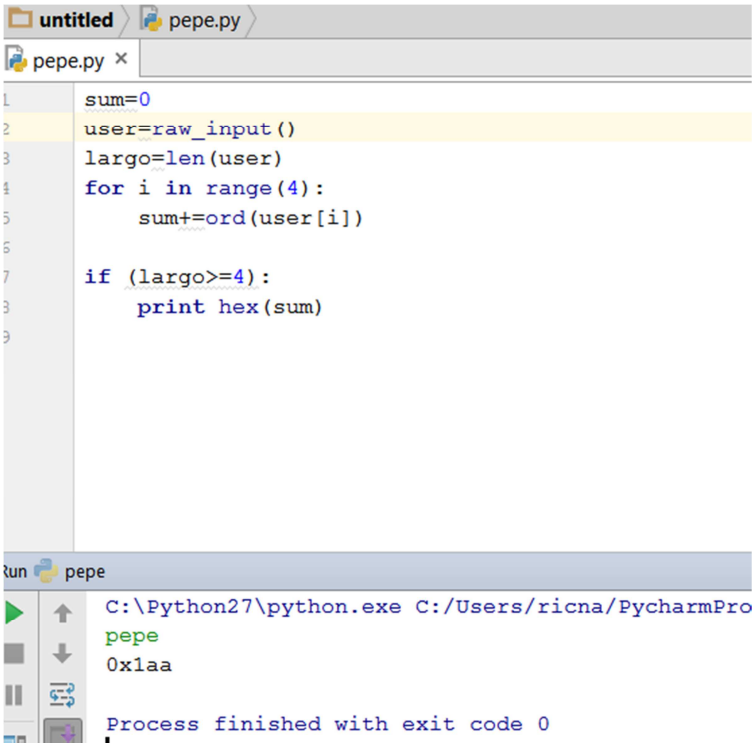


Remember it doesn't add all bytes but the first four bytes.



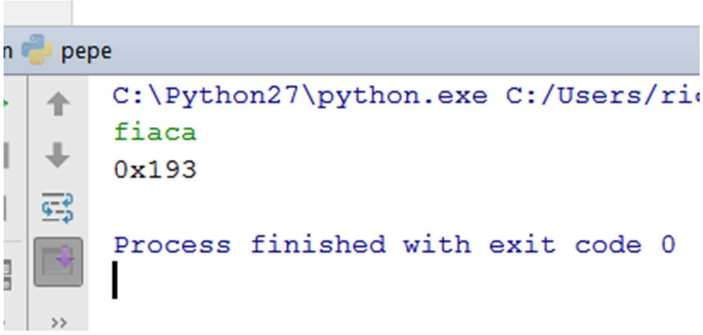
There, it looks better because it checks if it is greater or equal to 4 as the program does.

We can make a generic keygen for any user.



Using raw_input we get all we type by console.

The result for pepe is similar. The addition is 0x1aa, but I can get it for any user.



We had the formula.

X= (SUMATORIA/2) ^ 0x1234

It should divide it by 2 and XOR it with 0x1234 to find the hex password.

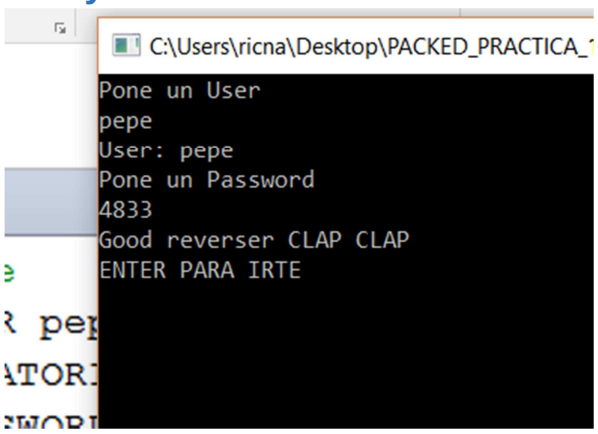
pepe.py

```
1   sum=0
2   user=raw_input()
3   largo=len(user)
4   for i in range(4):
5       sum+=ord(user[i])
6
7   print "USER",user
8
9   if (largo>=4):
10      print "SUMATORIA",hex(sum)
11      password= (sum/2) ^ 0x1234
12      print "PASSWORD",password
13
```

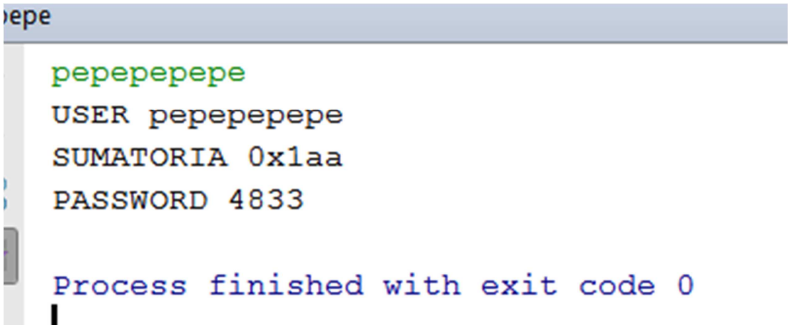
Run pepe

```
pepe
USER pepe
SUMATORIA 0x1aa
PASSWORD 4833
```

If I try it.

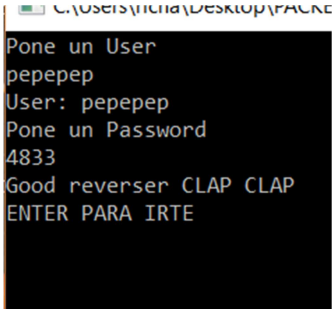


We already have the keygen we need to make the conversion of the password from hex to decimal because Python always prints in decimal by default.



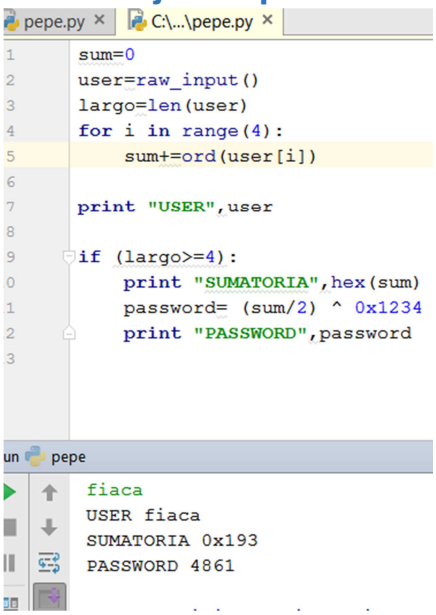
We see it adds just the first 4 characters of user's name. It doesn't matter if the password is greater, but the initial 4 characters are similar.

The exercise crashes when typing 8 characters because it should be 8 characters in total including the terminating null.

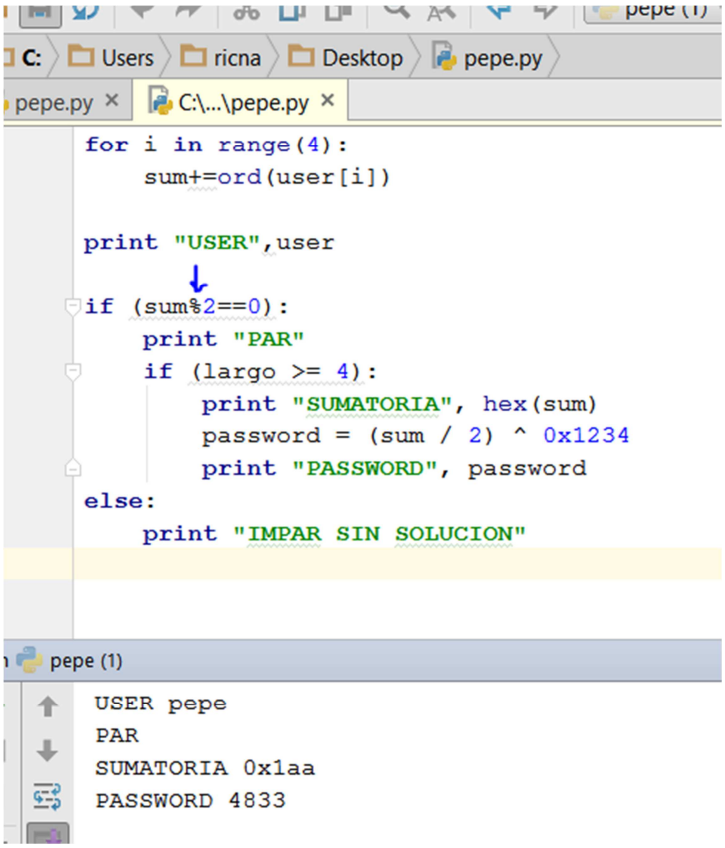


Until 7 it works well.

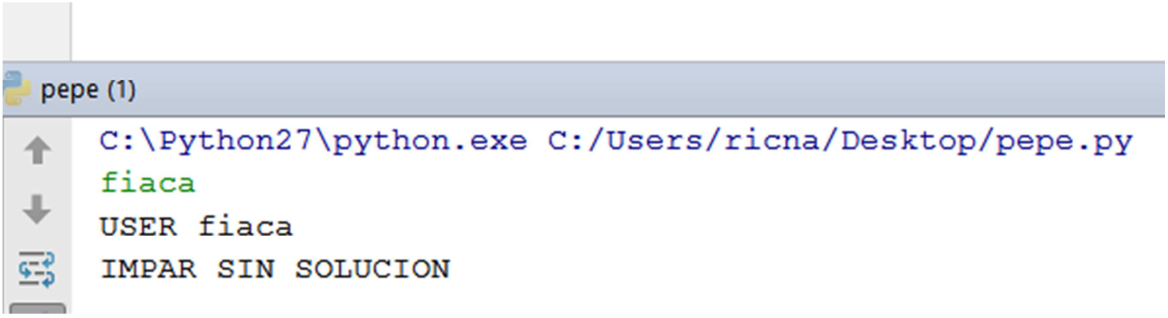
There is just a problema when the addition is negative.



It wouldn’t have any solution because the password is multiplied by 2 and being an integer multiplication it will never be negative. So, we add it that check.



There, we verify the rest of dividing by two. If it is 0, it is impair it has no solution.



I think that our keygen is pretty good. So, we can finish this part.

Ricardo Narvaja
Translated by: @IvinsonCLS