

REVERSING WITH IDA PRO FROM SCRATCH

PART 18

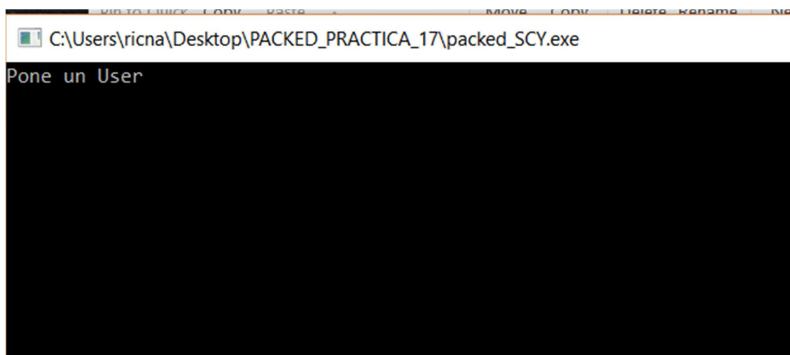
In the previous part, we unpacked the exercise executable and we made it run. Now, we will reverse it to see if we can make a keygen in Python.

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.

Address	Length	Type	String
UPX0:00232110	0000000E	C	Pone un User\n
UPX0:00232120	0000000A	C	User: %s\n
UPX0:0023212C	00000012	C	Pone un Password\n
UPX0:00232140	00000029	C	Good reverser CLAP CLAP\nENTER PARA IRTE\n
UPX0:0023216C	00000027	C	Bad reverser JUA JUA \nENTER PARA IRTE\n
.SCY:0023B1B4	0000000D	C	kernel32.dll
.SCY:0023B2D9	00000011	C	vcruntime140.dll
.SCY:0023B30D	0000000D	C	ucrtbase.dll

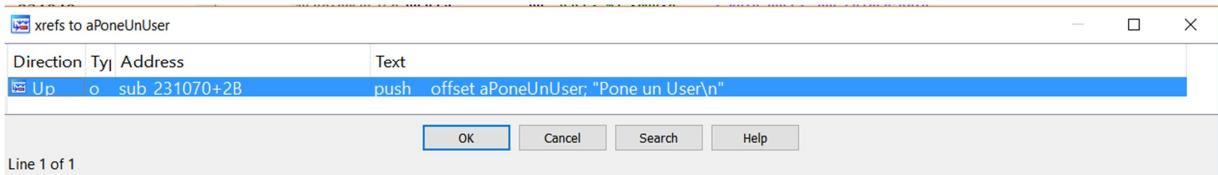
We know that the first thing the program does is printing: "Pone un user". In English, it means: "Enter a user".



Double click on that string.

```
UPX0:002320E4 dword_2320E4          dd 0          ; DATA XREF: start-124To
UPX0:002320E8
UPX0:002320F0 dword_2320F0          dd offset sub_231241, offset loc_2312E5
UPX0:00232110 aPoneUnUser          db 'Pone un User',0Ah,0 ; DATA XREF: sub_231070+2BTo
UPX0:00232114 align 10h
UPX0:00232120 aUsers              db 'User: %s',0Ah,0      ; DATA XREF: sub_231070+80To
UPX0:00232124 align 4
UPX0:0023212C aPoneUnPassword    db 'Pone un Password',0Ah,0 ; DATA XREF: sub_231070+BDTo
UPX0:0023213E
UPX0:00232140 aGoodReverserCl    db 'Good reverser CLAP CLAP',0Ah ; DATA XREF: sub_231070+147To
UPX0:00232140 db 'ENTER PARA IRTE',0Ah,0
UPX0:00232169 align 4
```

We find its reference with X.



I go there.

```

00231070 var_4= byte ptr -7Ch
00231070 var_4= dword ptr -4
00231070
00231070 push    ebp
00231071 mov     ebp, esp
00231073 sub     esp, 94h
00231079 mov     eax, __security_cookie
0023107E xor     eax, ebp
00231080 mov     [ebp+var_4], eax
00231083 mov     [ebp+var_7D], 0
00231087 mov     [ebp+var_90], 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]
002310A9 push    eax
002310AF lea     ecx, [ebp+Buf]
002310B2 push    edx
002310B3 call    gets_s
002310B9 add    esp, 8
002310BC lea     edx, [ebp+Buf]
002310BF push    edx
002310C0 call    strlen
002310C5 add    esp, 4

```

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.

```

00231070 var_4= dword ptr -4
00231070
00231070 push    ebp
00231071 mov     ebp, esp
00231073 sub     esp, 94h
00231079 mov     eax, __security_cookie
0023107E xor     eax, ebp
00231080 mov     [ebp+var_4], eax
00231083 mov     [ebp+var_7D], 0

```

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

```

-00000015      db ? ; undefined
-00000014      db ? ; undefined
-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
+00000008 ; end of stack variables

SP+0000008C

```

EBP

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 is a non-argument function.

```

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
00000008 ; end of stack variables

R+0000008C

```

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.

xrefs to sub_231070		
Direction	Ty	Address
Down	p	start-7B
		call sub_231070

OK Cancel Save

Line 1 of 1

```

002313DB
002313DB loc_2313DB:
002313DB call    _p_Argv
002313E0 mov     edi, eax
002313E2 call    _p_Argc
002313E7 mov     esi, eax
002313E9 call    get_initial_narrow_environment
002313EE push   eax
002313EF push   dword ptr [edi]
002313F1 push   dword ptr [esi]
002313F3 call   sub_231070
002313F8 add    esp, 0Ch
002313FB mov    esi, eax
002313FD call   sub_23142F
00231402 test   al, al
00231404 jnz   short loc_23140C

```

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

```
00231070 ; Attributes: bp-based frame
00231070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00231070 main proc near
00231070     Size= dword ptr -94h
00231070     var_90= dword ptr -90h
00231070     var_8C= dword ptr -8Ch
00231070     var_88= dword ptr -88h
00231070     var_84= dword ptr -84h
00231070     var_70= byte ptr -7Dh
00231070     Buf= byte ptr -7Ch
00231070     var_4= dword ptr -4
00231070     argc= dword ptr 8
00231070     argv= dword ptr 0Ch
00231070     envp= dword ptr 10h
00231070
00231070     push    ebp
00231071     mov     ebp, esp
```

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

```
00231070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00231070 main proc near
00231070     Size= dword ptr -94h
00231070     var_90= dword ptr -90h
00231070     var_8C= dword ptr -8Ch
00231070     var_88= dword ptr -88h
00231070     var_84= dword ptr -84h
00231070     var_70= byte ptr -7Dh
00231070     Buf= byte ptr -7Ch
00231070     var_4= dword ptr -4
00231070     argc= dword ptr 8
00231070     argv= dword ptr 0Ch
00231070     envp= dword ptr 10h
00231070
00231070     push    ebp
00231071     mov     ebp, esp
00231073     sub    esp, 94h
00231079     mnu    eax, security cookie
```

Warning
There are no xrefs to argc
OK
 Don't display this message again (for this session only)

They are not used, they're not important.

3000000E	db ? ; undefined
3000000D	db ? ; undefined
3000000C	db ? ; undefined
3000000B	db ? ; undefined
3000000A	db ? ; undefined
30000009	db ? ; undefined
30000008	db ? ; undefined
30000007	db ? ; undefined
30000006	db ? ; undefined
30000005	db ? ; undefined
30000004 var_4	dd ?
30000000 S	db 4 dup(?)
30000004 R	db 4 dup(?)
30000008 argc	dd ?
3000000C argv	dd ? ; offset
30000010 envp	dd ? ; offset
30000014	
30000014 ; end of stack variables	

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.

xrefs to var_4			
Direction	Ty	Address	Text
Down	w	main+10	mov [ebp+var_4], eax
Down	r	main+16B	mov ecx, [ebp+var_4]
OK Cancel Search			

Line 1 of 2

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

```

00231070 var_84= dword ptr -84h
00231070 var_7D= byte ptr -7Dh
00231070 Buf= byte ptr -7Ch
00231070 var_4= dword ptr -4
00231070 argc= dword ptr 8
00231070 argv= dword ptr 0Ch
00231070 envp= dword ptr 10h
00231070
00231070 push    ebp
00231071 mov     ebp, esp
00231073 sub    esp, 94h
00231079 mov     eax, __security_cookie
0023107E xor     eax, ebp
00231080 mov     [ebp+var_4], eax
00231080 ...

```

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

Cla es aquí.

```

002311D3
002311D3 loc_2311D3:
002311D3 call    getch
002311D9 xor     eax, eax
002311DB mov     ecx, ([ebp+var_4]
002311DE xor     [ecx], ebp
002311E0 call    sub_231230
002311E5 mov     esp, ebp
002311E7 pop    ebp
002311E8 retn
002311E8 main endp
002311E8

```

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.

```

00231230
00231230
00231230
00231230 sub_231230 proc near
00231230 cmp    ecx, _security_cookie
00231236 repne jnz short loc_23123B
00231239 repne ret
0023123B
0023123B loc_23123B:
0023123B repne jmp sub_2314A0
0023123B sub_231230 endp
0023123B

```

D0000630 00231230: sub_231230 (Synchronized with Hex View-1)

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.

```

002311D3
002311D3 loc_2311D3:
002311D3 call   getch
002311D9 xor    eax, eax
002311DB mov    ecx, [ebp+CANARY]
002311DE xor    ecx, ebp
002311E0 call   CHECK_CANARY
002311E5 mov    esp, ebp
002311E7 pop    ebp
002311E8 retn
002311E8 main endp
002311E8

```

Now, it looks better.

```

0023107E xor    eax, ebp
00231080 mov    [ebp+CANARY], eax
00231083 mov    [ebp+var_70], 0
00231087 mov    [ebp+var_90], 0
00231091 mov    [ebp+Size], 8
00231098 push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4
002310A8 mnll   eax [ebp+Size]

```

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.

```

00231198 mov [ebp+var_8C], edx
00231196 mov eax, [ebp+var_8C]
0023119C push eax
0023119D mov ecx, [ebp+var_90]
002311A3 push ecx
002311A4 call sub_231010
002311A9 add esp, 8
002311AC mov [ebp+var_7D], al
002311AF movzx edx, [ebp+var_7D]
002311B3 test edx, edx
002311B5 jz short loc_2311C6

```

```

GoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
F0
c_2311D3

```

```

002311C6 loc_2311C6: ; "Bad reverser JI
002311C6 push offset aBadReverserJua
002311CB call sub_2311F0
002311D0 add esp, 4

```

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.

```

-00000094 ;
-00000094
-00000094 Size dd ?
-00000090 var_98 dd ?
-0000008C var_8C 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

```

Rename it pressing N.

```

002311A4 call sub_231010
002311A9 add esp, 8
002311AC mov [ebp+FLAG_EXITO], al
002311AF movzx edx, [ebp+FLAG_EXITO]
002311B3 test edx, edx
002311B5 jz short loc_2311C6

```

```

set aGoodReverserCl ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
_2311F0
,4
rt loc_2311D3

```

```

002311C6 loc_2311C6: ; "Bad reverser JI
002311C6 push offset aBadReverserJua
002311CB call sub_2311F0
002311D0 add esp, 4

```

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.

```
00231103 mov     ecx, [ebp+var_84]
00231109 mousx  edx, [ebp+ecx+Buf]
0023110E add    edx, [ebp+var_90]
00231114 mov    [ebp+var_90], edx
0023111A jmp    short loc_2310EB
```

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.

```
002310DF loc_2310DF:
002310F0 mov    [ebp+var_84], 0
002310E9 jmp    short loc_2310FA
```

```
002310FA loc_2310FA:
002310F0 cmp    [ebp+var_84], 4
00231101 jge    short loc_23111C
```

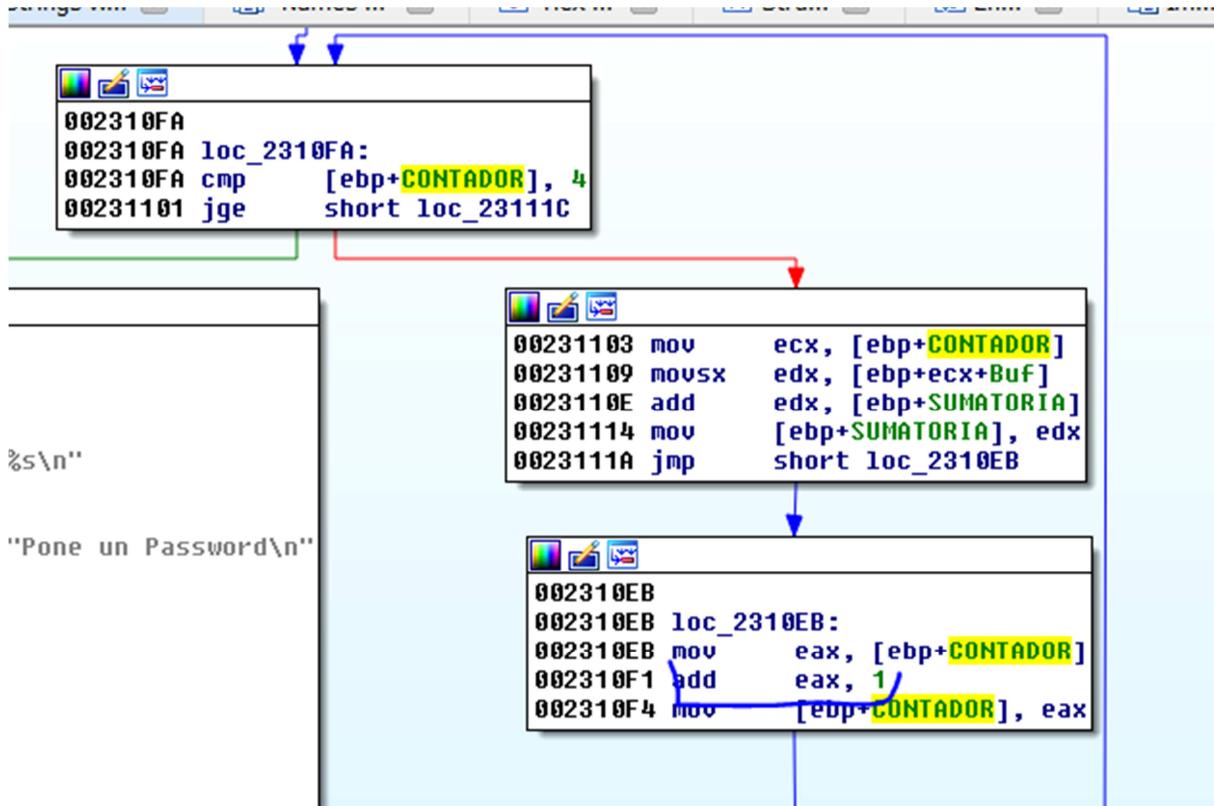
```
00231103 mov    ecx, [ebp+var_84]
00231109 mousx  edx, [ebp+ecx+Buf]
0023110E add    edx, [ebp+SUMATORIA]
00231114 mov    [ebp+SUMATORIA], edx
0023111A jmp    short loc_2310EB
```

```
002310EB loc_2310EB:
002310F0 mov    eax, [ebp+var_84]
002310F1 add    eax, 1
002310F4 mov    [ebp+var_84], eax
```

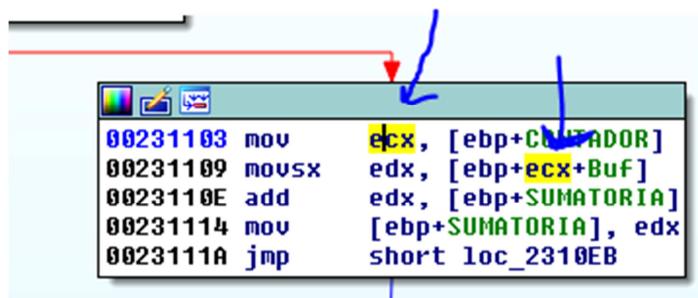
0EB: main:loc 2310EB (Synchronized with Hex View-1)

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
00231098 push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4
002310A8 mov    eax, [ebp+Size]
002310AE push   eax, [ebp+Buf] ; Size
002310AF lea    ecx, [ebp+Buf]
002310B2 push   ecx, [ebp+Buf] ; Buf
002310B3 call   gets_s
002310B9 add    esp, 8
002310BC lea    edx, [ebp+Buf]
002310BF push   edx, [ebp+Buf] ; Str
002310C0 call   strlen
002310C5 add    esp, 4

```

Buf equals 8 at the beginning as maximum 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
00231098 push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   sub_2311F0
002310A5 add    esp, 4

```

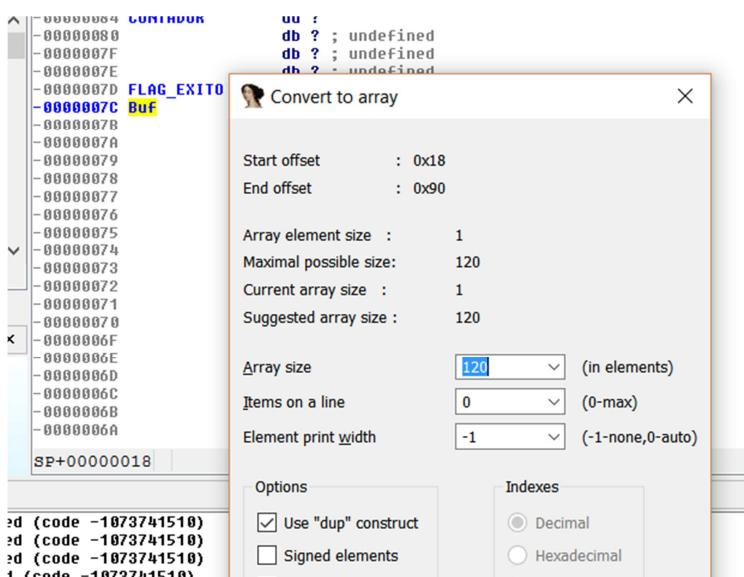
Ready.

```

00231087 mov    [ebp+SUMATORIA], 0
00231091 mov    [ebp+Size], 8
00231098 push   offset aPoneUnUser ; "Pone un User\n"
002310A0 call   printf
002310A5 add    esp, 4

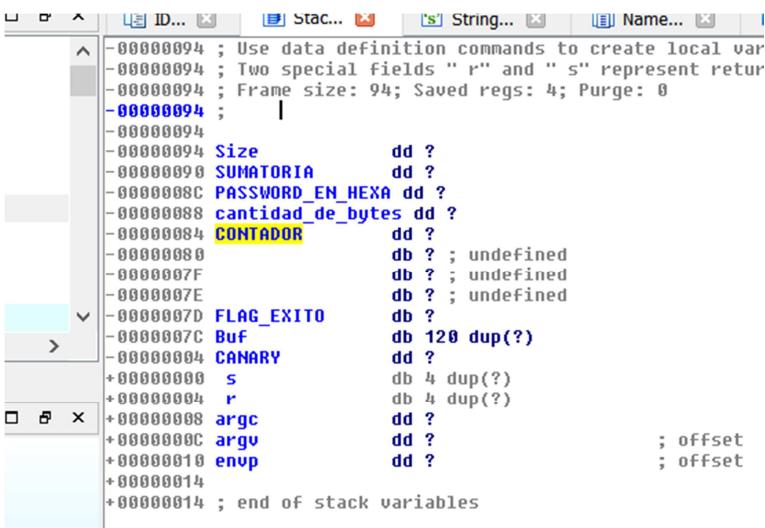
```

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



It matches the source code.

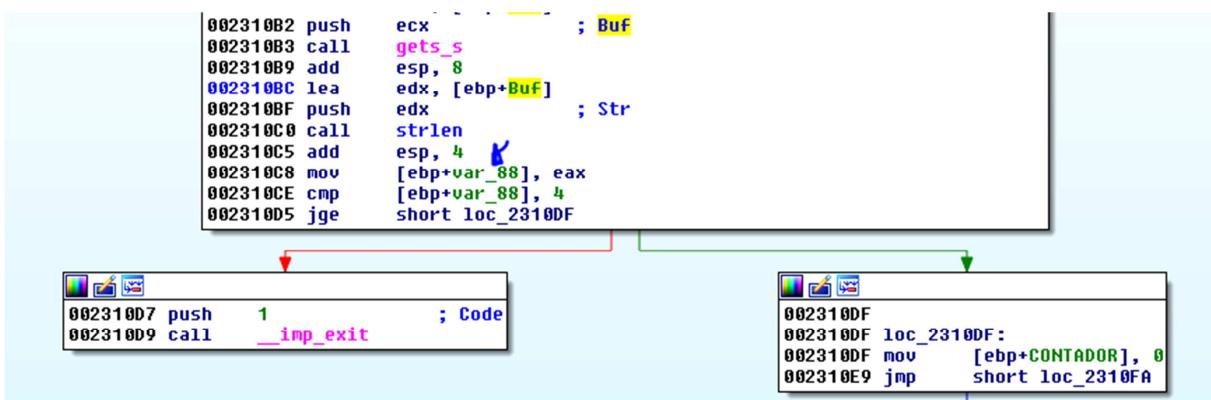
```
int main(int argc, char *argv[])
{
    bool resul = false;
    int pass;
    //int user;
    int sum=0;
    char buf[120];
    int max = 8;
    printf("Pone un User\n");
```



ID...	Stac...	String...	Name...
-00000094	Use data definition commands to create local var		
-00000094	Two special fields " r" and " s" represent return		
-00000094	Frame size: 94; Saved regs: 4; Purge: 0		
-00000094			
-00000094	Size dd ?		
-00000090	SUMATORIA dd ?		
-0000008C	PASSWORD_EN_HEXA dd ?		
-00000088	cantidad_de_bytes dd ?		
-00000084	CONTADOR dd ?		
-00000080	db ? ; undefined		
-0000007F	db ? ; undefined		
-0000007E	db ? ; undefined		
-0000007D	FLAG_EXITO db ?		
-0000007C	Buf db 120 dup(?)		
-00000094	CANARY dd ?		
+00000000	s db 4 dup(?)		
+00000004	r db 4 dup(?)		
+00000008	argc dd ?		
+0000000C	argv dd ?		; offset
+00000010	envp dd ?		; offset
+00000014	+00000014 ; end of stack variables		

The stack representation is clearer.

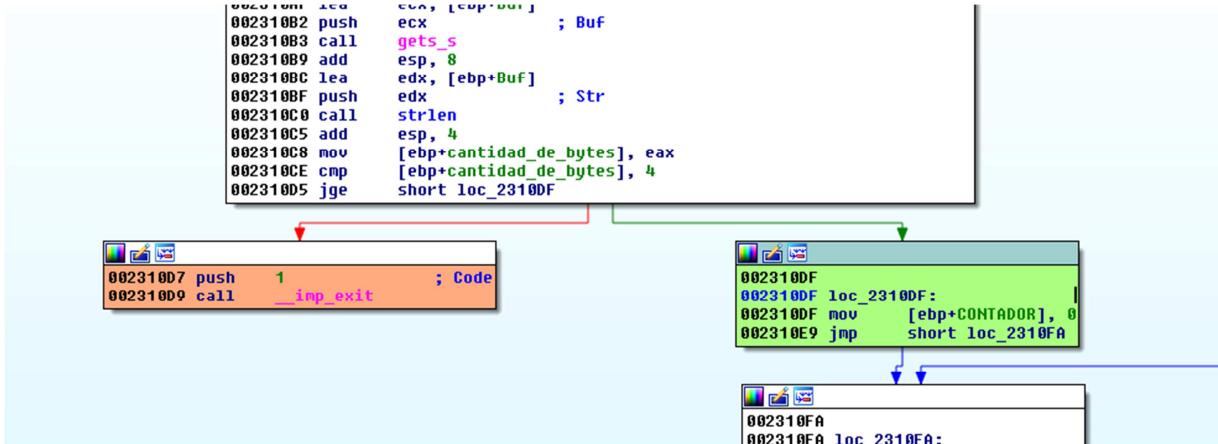
After receiving in the Buf, it calls **strlen** to know the size of what you typed.



```
002310B2 push    ecx          ; Buf
002310B3 call    gets_s
002310B9 add     esp, 8
002310BC lea     edx, [ebp+BUF]
002310BF push    edx          ; Str
002310C0 call    strlen
002310C5 add     esp, 4
002310C8 mov     [ebp+var_88], eax
002310CE cmp     [ebp+var_88], 4
002310D5 jge     short loc_2310DF

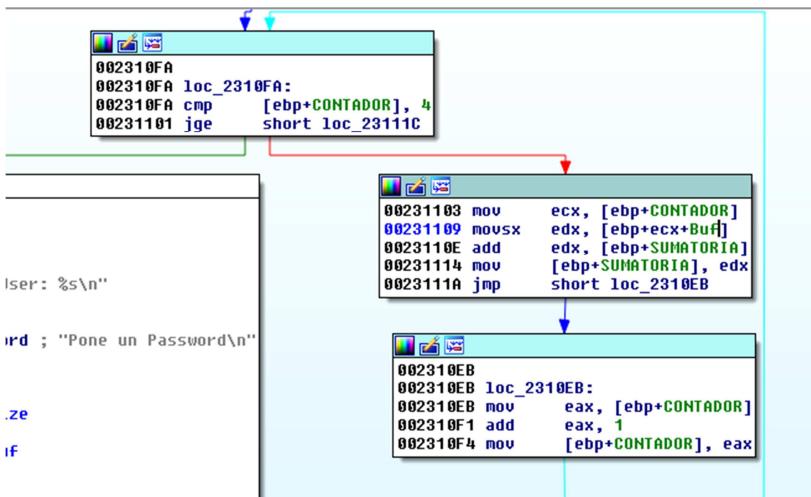
002310D7 push    1             ; Code
002310D9 call    _imp_exit
002310DF loc_2310DF:
002310DF mov     [ebp+CONTADOR], 0
002310E9 jmp     short loc_2310FA
```

var_88 is the number of bytes you typed.

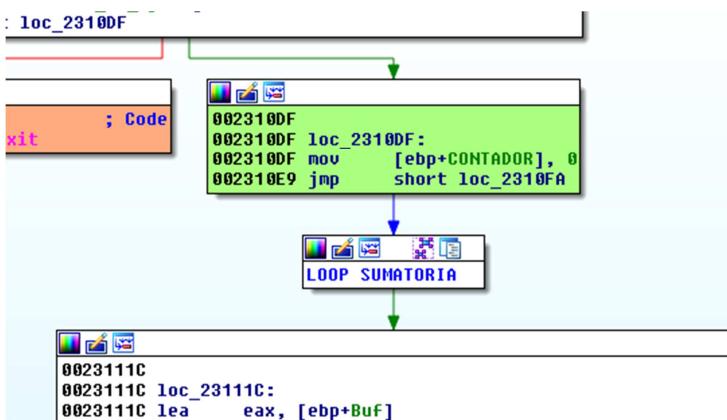


If it is less than 4, it goes to EXIT.

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.

```
0023112A add    esp, 8
0023112D push   offset aPoneUnPassword ; "Pone un Password\n"
00231132 call   printf
00231137 add    esp, 4
0023113A mov    ecx, [ebp+Size]
00231140 push   ecx      ; Size
00231141 lea    edx, [ebp+Buf]
00231144 push   edx      ; Buf
00231145 call   gets_s
00231148 add    esp, 8
```

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

```
0023114B add    esp, 8
0023114E lea    eax, [ebp+Buf]
00231151 push   eax      ; Str
00231152 call   strlen
00231157 add    esp, 4
0023115A mov    [ebp+cantidad_de_bytes], eax
00231160 cmp    [ebp+cantidad_de_bytes], 4
00231167 jge    short loc_231171
```

```
00231169 push   1          ; Code
0023116B call   __imp_exit
```

```
00231171
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx      ; Str
00231175 call   atoi
00231178 add    esp, 4
0023117F mnv   [ehn+var_8C1], eax
```

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

```
00231171
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx      ; Str
00231175 call   atoi
00231178 add    esp, 4
0023117F mnv   [ehn+var_8C1], eax
```

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

```
PDBSRC: loading sym
Python>hex(989898)
0xf1aca
Python
```

```

00231171
00231171 loc_231171:
00231171 lea    ecx, [ebp+Buf]
00231174 push   ecx          ; Str
00231175 call   atoi
00231178 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]
002311A0 .....

```

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    eax, 1234h
00231196 mov    edx, [ebp+PASSWORD_EN_HEXA], edx
0023119C push   eax
0023119D mov    ecx, [ebp+SUMATORIA]
002311A3 push   ecx
002311A4 call   CHEQUEO_EXITO
002311A9 add    esp, 8
002311AC mnw   [ehn+FLAG_EXIT01.. al]

```

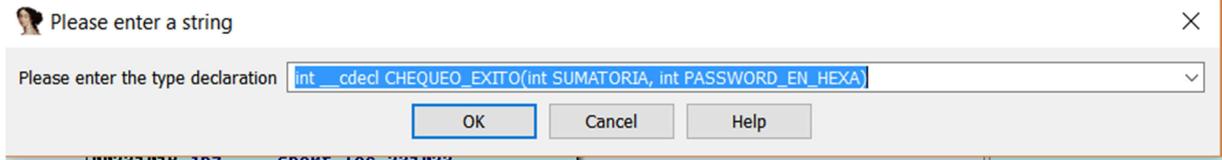
Let's rename both arguments inside the function.

```

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.

```
00231010 ; Attributes: bp-based frame
00231010
00231010 ; int __cdecl CHEQUEO_EXITO(int SUMATORIA, int PASSWORD_EN_HEXA)
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
00231018     jnz    short loc_231023
```

We'll see if the references match.

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

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

```
00231010 ; Attributes: bp-based frame
00231010 ; int __cdecl CHEQUEO_EXITO(int SUMATORIA, int PASSWORD_EN_HEXA)
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
00231018     jnz    short loc_231023
```

Below the main assembly window, there are two smaller windows showing the propagation of values:

- The first window shows a green box around the instruction "mov al, 1" and a blue box around the label "short loc_231025".
- The second window shows a blue box around the instruction "jmp short loc_231025" and a red box around the label "loc_231023".
- The third window shows a blue box around the instruction "xor al, al" and a red box around the label "loc_231023".

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

$(X \wedge 0x1234) * 2 = \text{SUMATORIA}$

If we clear X

$X \wedge 0x1234 = (\text{SUMATORIA}/2)$

X= (SUMATORIA/2) ^ 0x1234

The XOR function is invertible.

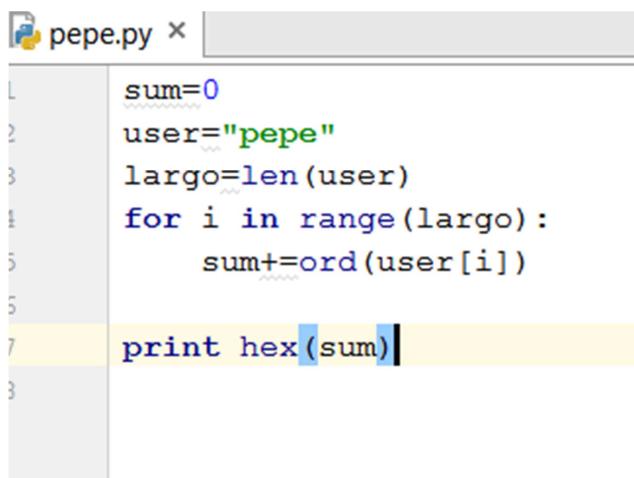
$A \wedge B = C$

$A = B \wedge C$

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 x
1 sum=0
2 user="pepe"
3 largo=len(user)
4 for i in range(largo):
5     sum+=ord(user[i])
6
7 print hex(sum)|
```

There, I got the addition for my user pepe.

```
1 sum=0
2 user="pepe"
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])
6
7 print hex(sum)
8
```

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

```
1 sum=0
2 user="pepe"
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])

6 if (largo>=4):
7     print hex(sum)
```

debug pepe

Debugger Console →

pydev debugger: process 6116 is connecting

Connected to pydev debugger (build 162.1967.10)

0x1aa

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.

The screenshot shows the PyCharm IDE interface. The top bar displays 'untitled > pepe.py >'. Below it is a tab bar with 'pepe.py x'. The code editor contains the following Python script:

```
1 sum=0
2 user=raw_input()
3 largo=len(user)
4 for i in range(4):
5     sum+=ord(user[i])
6
7 if (largo>=4):
8     print hex(sum)
```

The run console below shows the output of running the script 'pepe':

```
Run pepe
C:\Python27\python.exe C:/Users/ricna/PycharmPro
pepe
0x1aa
Process finished with exit code 0
```

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.

The screenshot shows the PyCharm run console for the script 'pepe'. The command is 'C:\Python27\python.exe C:/Users/ricna/PycharmPro pepe'. The output shows the input 'fiaca' and the resulting hex value '0x193'.

```
n pepe
C:\Python27\python.exe C:/Users/ricna/PycharmPro
pepe
fiaca
0x193
Process finished with exit code 0
```

We had the formula.

$$X = (\text{SUMATORIA}/2) \wedge 0x1234$$

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

```
pepe.py x

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.

```
C:\Users\ricna\Desktop\PACKED_PRACTICA_1
Pone un User
pepe
User: pepe
Pone un Password
4833
Good reverser CLAP CLAP
ENTER PARA IRTE
R pepe
ATORI
SWORD
```

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.

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

Process finished with exit code 0
```

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.

```
C:\Users\fiaca\Desktop\PRUEBAS
Pone un User
pepepep
User: pepepep
Pone un Password
4833
Good reverser CLAP CLAP
ENTER PARA IRTE
```

Until 7 it works well.

There is just a problema when the addition is negative.

```
pepe.py x C:\...\pepe.py x
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
```

```
un pepe
fiaca
USER fiaca
SUMATORIA 0x193
PASSWORD 4861
```

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.

The screenshot shows a Python code editor with a file named 'pepe.py'. The code reads a user input 'user' and calculates its sum. It then checks if the sum is even (sum % 2 == 0). If it is, it prints 'PAR', calculates the password as (sum / 2) ^ 0x1234, and prints 'SUMATORIA' and 'PASSWORD'. If the sum is odd (else), it prints 'IMPAR SIN SOLUCION'. The output window below shows the execution of the script with input 'pepe', resulting in 'USER pepe', 'PAR', 'SUMATORIA 0x1aa', and 'PASSWORD 4833'.

```
for i in range(4):
    sum+=ord(user[i])

print "USER",user
if (sum%2==0):
    print "PAR"
    if (largo >= 4):
        print "SUMATORIA", hex(sum)
        password = (sum / 2) ^ 0x1234
        print "PASSWORD", password
else:
    print "IMPAR SIN SOLUCION"

USER pepe
PAR
SUMATORIA 0x1aa
PASSWORD 4833
```

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

The screenshot shows a terminal window titled 'pepe (1)' running on Python 2.7. It executes the script 'pepe.py' with the argument 'fiaca'. The output shows 'USER fiaca' and 'IMPAR SIN SOLUCION'.

```
C:\Python27\python.exe C:/Users/ricna/Desktop/pepe.py
fiaca
USER fiaca
IMPAR SIN SOLUCION
```

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

Ricardo Narvaja
Translated by: @IvinsonCLS