

REVERSING WITH IDA PRO FROM SCRATCH

PART 19

In this chapter, we can reverse the original Cruehead's Crackme. Let's open it with the LOADER unchecking MANUAL LOAD. This is just reversing. The original file is not packed. So, it is not necessary to load it manually.

The screenshot shows the assembly view in IDA Pro. The code is as follows:

```
00401000 ; Section size in file : 00000600 ( 1536.)
00401000 ; Offset to raw data for section: 00000600
00401000 ; Flags 60000020: Text Executable Readable
00401000 ; Alignment : default
00401000
00401000     .486p
00401000     .model flat
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 CODE    segment para public 'CODE' use32
00401000     assume cs:CODE
00401000     org 401000
00401000     assume es:nothing, ss:nothing, ds:CODE, fs:nothing
00401000
00401000
00401000     public start
00401000 start   proc near
00401000     push    ebp
00401000     mov     ebp,esp
00401000     call    GetModuleHandle
00401000     mov     eax,esi
00401000     mov     ds:instance, eax
00401000     push    0 ; lpWindowName
00401000     push    offset ClassName ; "No need to disasm the code!"
00401000     call    FindWindow
00401000     or     eax, eax
00401000     jz     short loc_401010
0040101C     ret
```

Below the main assembly window, there is a smaller window showing the assembly for the `loc_401010` label:

```
0040101D loc_401010:
0040101D     nov    ds:UmdClass.style, 4003h
0040101D     nov    ds:UmdClass.lpFindProc,
0040101D     nov    ds:UmdClass.lpFindExtra,
0040101D     nov    ds:UmdClass.cbFindExtra,
0040101D     nov    eax, ds:instance
0040104A     nov    ds:UmdClass.hInstance, e
0040104F     push   64h ; lpIconName
00401054     nopl
```

It stopped there. In this case, as it is not a console application, it's not just analyzing the main function. We know that, in window applications, there is not the message LOOP that processes the user's window interactions, e.g. the clicks. And it is programmed to execute different functions with its code.

The first thing we do is seeing the strings. If that fails, we see the API's or functions used by the program. In this case, the strings are visible. So, let's follow this way.

Address	Length	Type	String
DATA:004020D6	00000011	C	Try to crack me!
DATA:004020E7	0000000D	C	CrackMe v1.0
DATA:004020F4	0000001C	C	No need to disasm the code!
DATA:00402110	00000005	C	MENU
DATA:00402115	0000000A	C	DLG_REGIS
DATA:0040211F	0000000A	C	DLG_ABOUT
DATA:00402129	00000008	C	Good work!
DATA:00402134	0000002C	C	Great work, mate!\rNow try the next CrackMe!
DATA:00402160	00000009	C	No luck!
DATA:00402169	00000015	C	No luck there, mate!

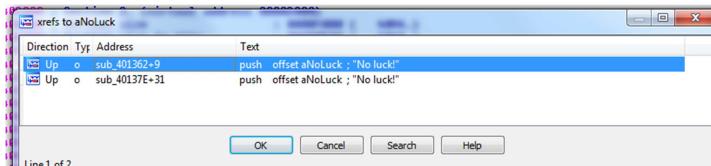
Following the NO LUCK string, we had reached the zone where it makes the decision because when we double click on that string, we go to...

```

DATA:00402129 Caption      db 'Good work!',0      ; DATA XREF: sub_401340+2To
DATA:00402134 ; CHAR Text[]
DATA:00402134 Text        db 'Great work, mate!',0Dh,'Now try the next CrackMe!',0
DATA:00402134             ; DATA XREF: sub_401340+7To
DATA:00402160 ; CHAR aNoLuck[]
DATA:00402160 aNoLuck     db 'No luck!',0      ; DATA XREF: sub_401362+9To
DATA:00402160             ; sub_40137E+31To
DATA:00402169 ; CHAR aNoLuckThereMat[]
DATA:00402169 aNoLuckThereMat db 'No luck there, mate!',0 ; DATA XREF: sub_401362+ETo
DATA:00402169             ; sub_40137E+36To
DATA:0040217E ; CHAR byte_40217E[16]
DATA:0040217E byte_40217E  db 10h dup(0)       ; DATA XREF: WndProc+100To
DATA:0040217E             ; sub_401253+84To
DATA:0040218E ; CHAR String[3698]
DATA:0040218E String      db 72h dup(0), 0E00h dup(?) ; DATA XREF: WndProc+100To
DATA:0040218E             ; sub_401253+64To
DATA:0040218E DATA        ends
DATA:0040218E             ; sub_401253+64To
.DATA:00403000 ; Section 3. (virtual address 00003000)
.IDATA:00403000 : Virtual size           : 00001000 ( 4096.)

```

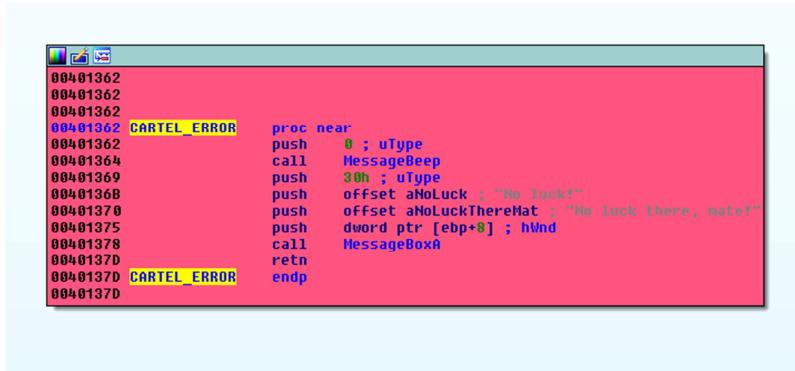
By pressing X or CTRL +X, we see two references to that string.



Let's see the first one.

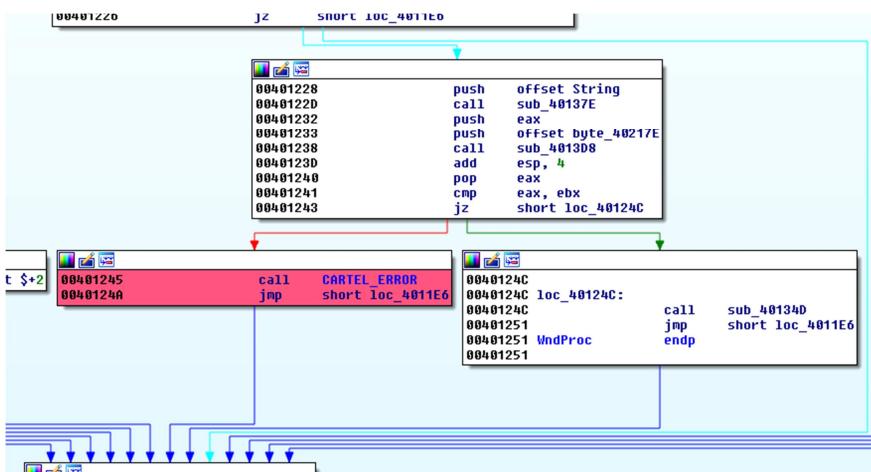


I painted the block in red because it belongs to the bad boy.

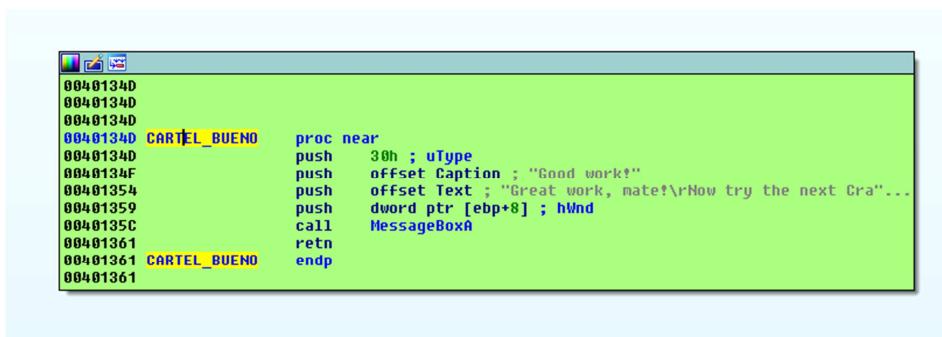


```
00401362
00401362
00401362
00401362 CARTEL_ERROR proc near
00401362     push    0 ; uType
00401364     call    MessageBeep
00401369     push    30h ; uType
0040136B     push    offset aNoLuck ; "No luck?""
00401370     push    offset aNoLuckThereHAt ; "No luck there, mate!""
00401375     push    dword ptr [ebp+8] ; hWnd
00401378     call    MessageBoxA
0040137D     retn
0040137D CARTEL_ERROR endp
0040137D
```

Let's see where it comes from.

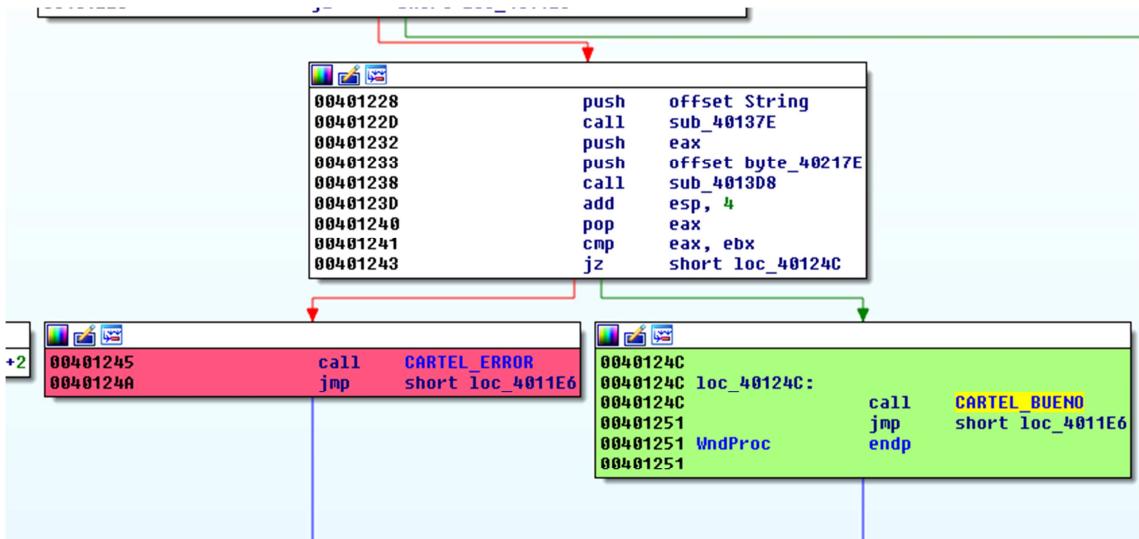


.So, the other way should be the good boy. Let's see inside the 0x40134D function.

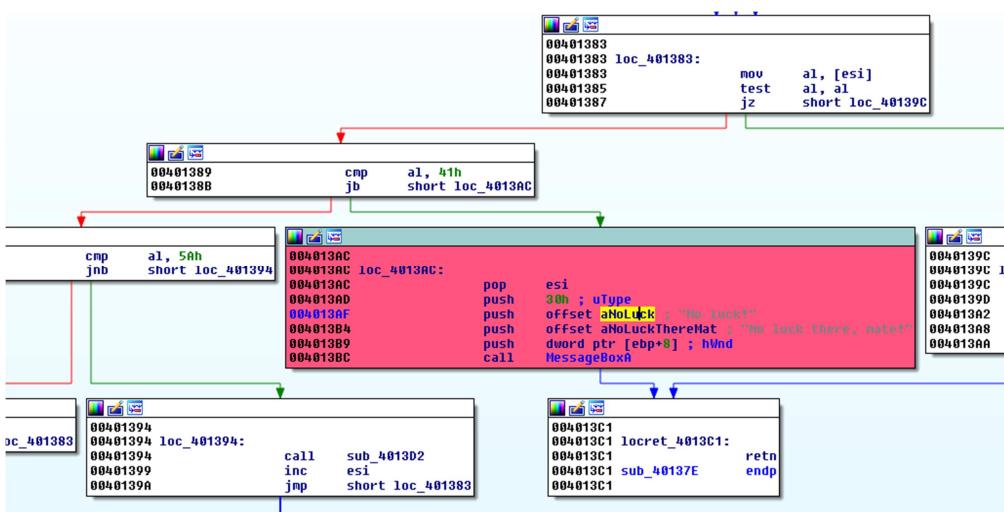


```
0040134D
0040134D
0040134D
0040134D CARTEL_BUENO proc near
0040134D     push    30h ; uType
0040134F     push    offset Caption ; "Good work!"
00401354     push    offset Text ; "Great work, mate!\rNow try the next Cra"...
00401359     push    dword ptr [ebp+8] ; hWnd
0040135C     call    MessageBoxA
00401361     retn
00401361 CARTEL_BUENO endp
00401361
```

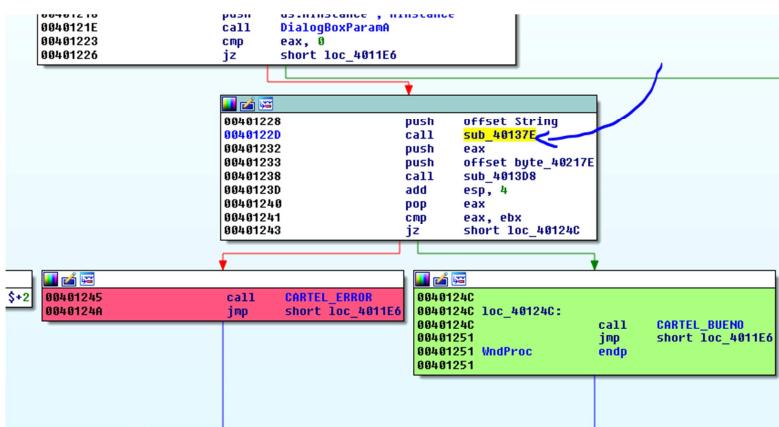
So, in the reference...

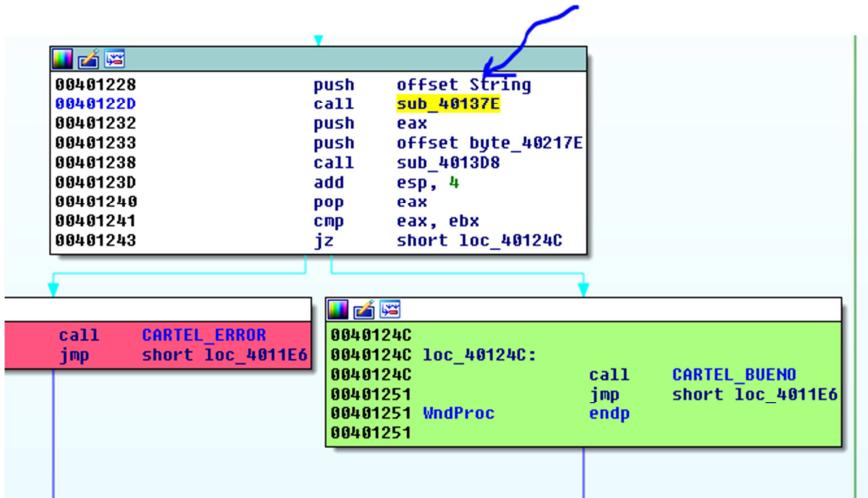


The other reference to the NO LUCK string comes from here.



And the other error message reference comes from here.





We see the argument of that function is a variable global address (OFFSET) called string. If we click on it...

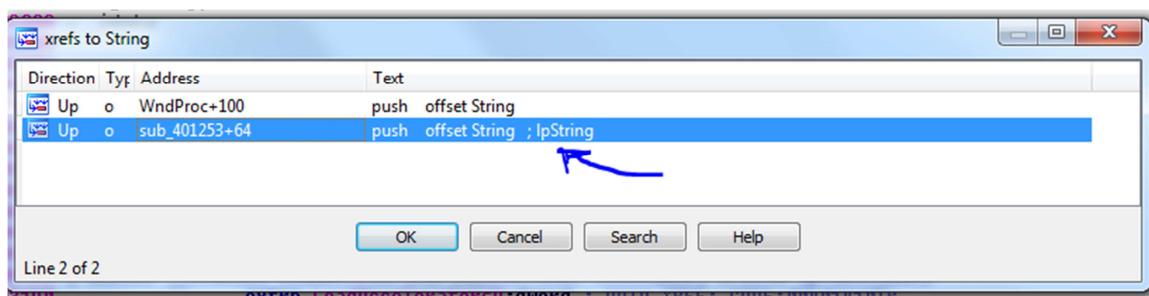
```

DATA:0040217E byte 40217E      db 10h dup(0)
DATA:0040217E
DATA:0040218E ; CHAR String[3698]
String      db 72h dup(0), 0F00h dup(?); DATA XREF: WndProc+100↑o
DATA:0040218E
DATA:0040218E DATA      ends
DATA:0040218E
DATA:0040218E .idata:00403000 ; Section 3. (virtual address 00003000)
idata:00403000 ; Virtual size           : 00001000 ( 4096.)
idata:00403000 ; Section size in file : 00000800 ( 2048.)
idata:00403000 ; Section size in file : 00000800 ( 2048.)

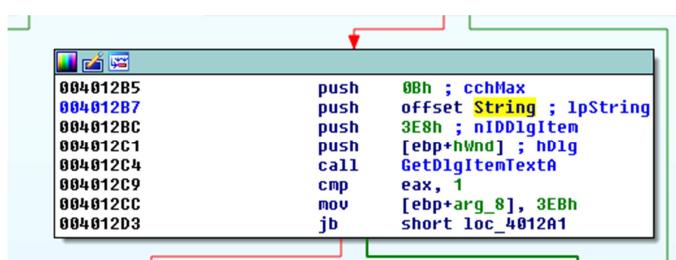
```

It's a 3698-byte buffer in the 0x40218E address of the DATA section.

It has two references. Let's see what string there is.



There is a reference that comes from here.



The GetDlgItemTextA API is used to enter some data. Let's see.

GetDlgItemText function

Retrieves the title or text associated with a control in a dialog box.

Syntax

```
C++  
UINT WINAPI GetDlgItemText(  
    _In_    HWND   hDlg,  
    _In_    int     nIDDlgItem,  
    _Out_   LPTSTR lpString,  
    _In_    int     nMaxCount  
)
```

Parameters

hDlg [in]
Type: **HWND**

A handle to the dialog box that contains the control.

nIDDlgItem [in]
Type: **int**

The identifier of the control whose title or text is to be retrieved.

lpString [out]
Type: **LPTSTR**

The buffer to receive the title or text.

nMaxCount [in]
Type: **int**

The maximum length, in characters, of the string to be copied to the buffer pointed to by *lpString*. If the length of the string, including the null character, exceeds the limit, the string is truncated.

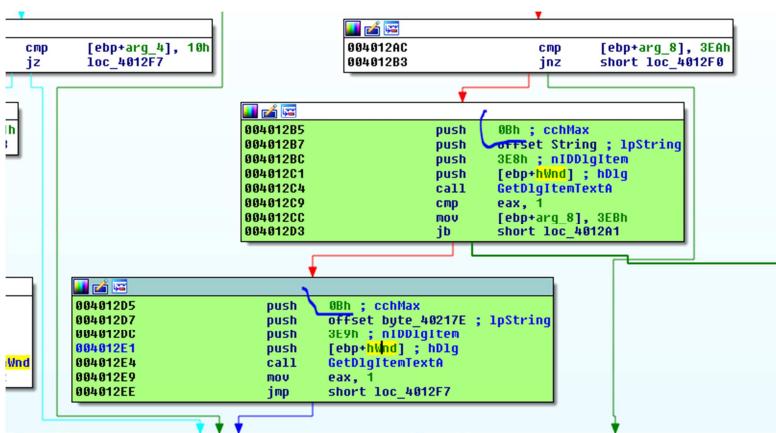
Return value

Type: **UINT**

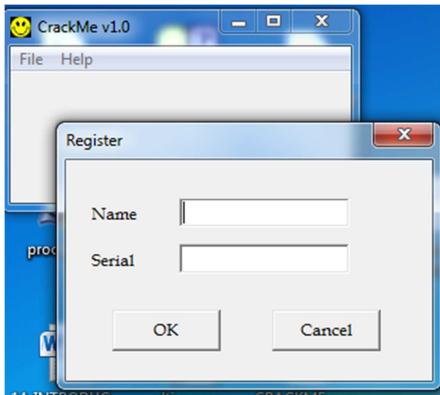
If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

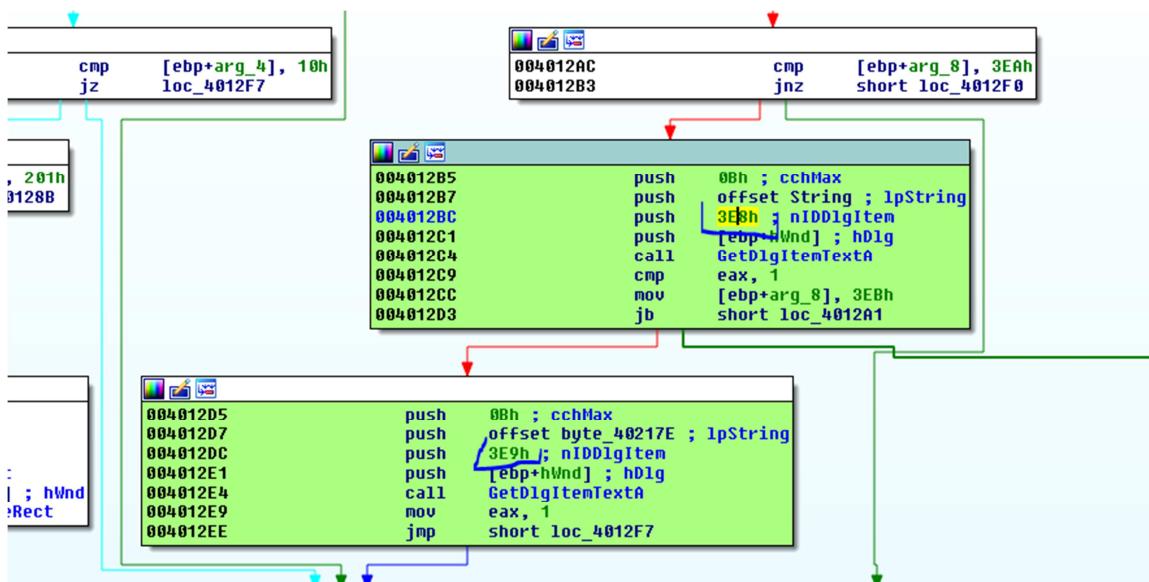
There, it receives some text through the keyboard.



There are two followed entries with the same handle *hWnd*. I suppose they are for the user and password in the Crackme when we click on REGISTER.

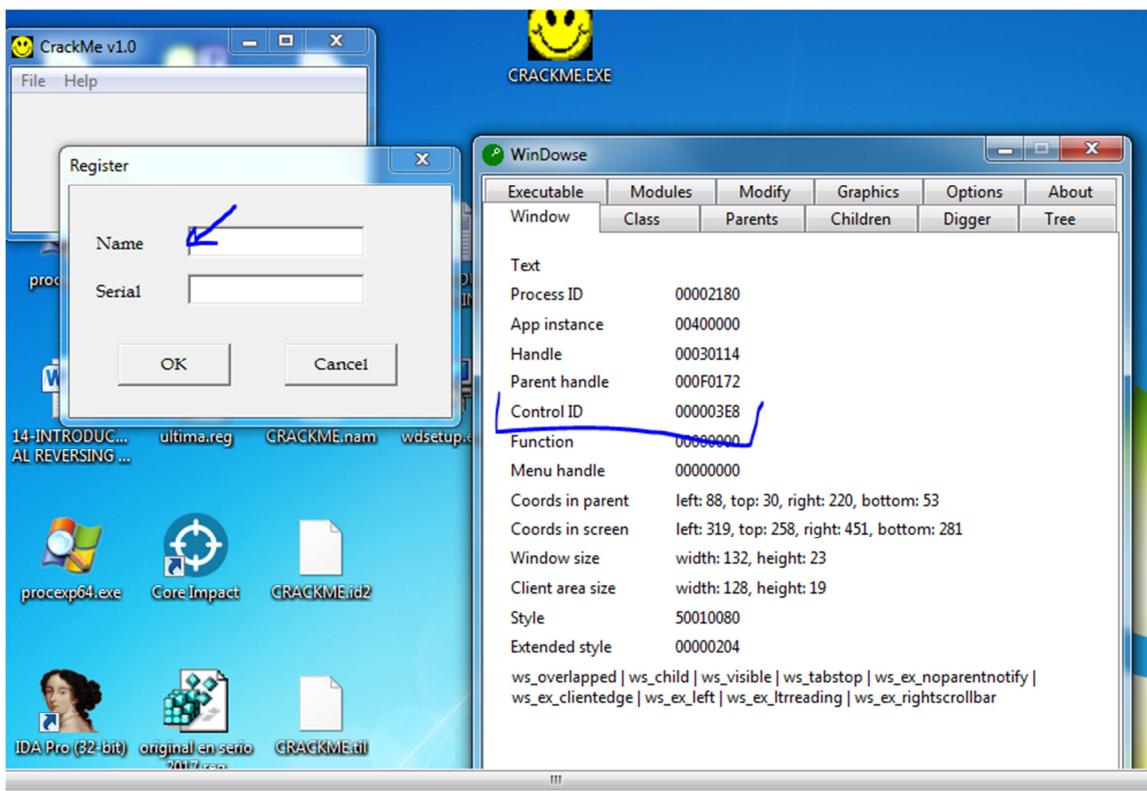


Both of them have the nIDDlItem control number of each one: 0x3E8 and 0x3E9.



Using the GREATIS WINDOWSE program, it gives us info about the windows where we place the cursor.

<http://www.greatis.com/wdsetup.exe>



I verify that the first edit box control ID is 0x3E8 and the second one is 0x3E9.

I can also rename the buffer where it receives the strings. The first one belongs to the String_USER and the second one to the String_PASSWORD. Both only accept a 0x0B characters length. In spite of being greater buffers.

```

loc_4012F7
        push    ; nResult
        push    eax
        push    [ebp+hWnd1] ; hDlg
004012F8 loc_4012F0:   mov    eax, 0
004012F9 loc_4012F0:   mov    imm, short Inc 401284

004012F7 4012F7 loc_4012F7:   push    ; nResult
004012F7 4012F7 loc_4012F7:   push    eax
004012F7 4012F7 loc_4012F7:   push    [ebp+hWnd1] ; hDlg
004012F7 4012F7 loc_4012F7:   push    0Bh ; cchMax
004012F7 4012F7 loc_4012F7:   push    offset String_USER ; lpString
004012F7 4012F7 loc_4012F7:   push    3E8h ; nIDDItem
004012F7 4012F7 loc_4012F7:   push    [ebp+hWnd1] ; hDlg
004012F7 4012F7 loc_4012F7:   call    GetDlgItemTextA
004012F7 4012F7 loc_4012F7:   cmp    eax, 1
004012F7 4012F7 loc_4012F7:   mov    [ebp+arg_8], 3EBh
004012F7 4012F7 loc_4012F7:   jb     short loc_4012A1

004012B5 4012B5 loc_4012B3:   push    0Bh ; cchMax
004012B7 4012B7 loc_4012B3:   push    offset String_PASSWORD ; lpString
004012B9 4012B9 loc_4012B3:   push    3E9h ; nIDDItem
004012C1 4012C1 loc_4012B3:   push    [ebp+hWnd1] ; hDlg
004012C4 4012C4 loc_4012B3:   call    GetDlgItemTextA
004012C9 4012C9 loc_4012B3:   cmp    eax, 1
004012CC 4012CC loc_4012B3:   mov    [ebp+arg_8], 3EBh
004012D3 4012D3 loc_4012B3:   jb     short loc_4012A1

```

We know where it saves the user and password we enter. Let's see what it does with them.

The String_USER is processed in here.

```
00401228      push    offset String_USER
0040122D      call    sub_40137E
00401232      push    eax
00401233      push    offset String_PASSWORD
00401238      call    sub_4013D8
0040123D      add     esp, 4
00401240      pop    eax
00401241      cmp    eax, ebx
00401243      jz     short loc_40124C
```

Let's rename the functions.

```
00401228      push    offset String_USER
0040122D      call    PROCESA_USER
00401232      push    eax
00401233      push    offset String_PASSWORD
00401238      call    PROCESA_PASS
0040123D      add     esp, 4
00401240      pop    eax
00401241      cmp    eax, ebx
00401243      jz     short loc_40124C
```

```
call    CARTEL_ERROR
jmp    short loc_4011E6
```

```
0040124C      loc_40124C:
0040124C          call    CARTEL_BUENO
00401251          jmp    short loc_4011E6
00401251  WndProc
00401251          endp
```

Let's analyze PROCESA_USER now.

```

0040137E
0040137E
0040137E
0040137E ; int __cdecl PROCESA_USER(int offset_String_USER)
0040137E PROCESA_USER    proc near
0040137E
0040137E offset_String_USER= dword ptr  4
0040137E
0040137E             mov     esi, [esp+offset_String_USER]
00401382             push    esi

```

```

00401383
00401383 loc_401383:
00401383             mov     al, [esi]
00401385             test    al, al
00401387             jz      short loc_40139C

```

When renaming the variable, I use the same name IDA used to respect process. Although, I could have used **p_String_USER** because it is also the pointer to it.

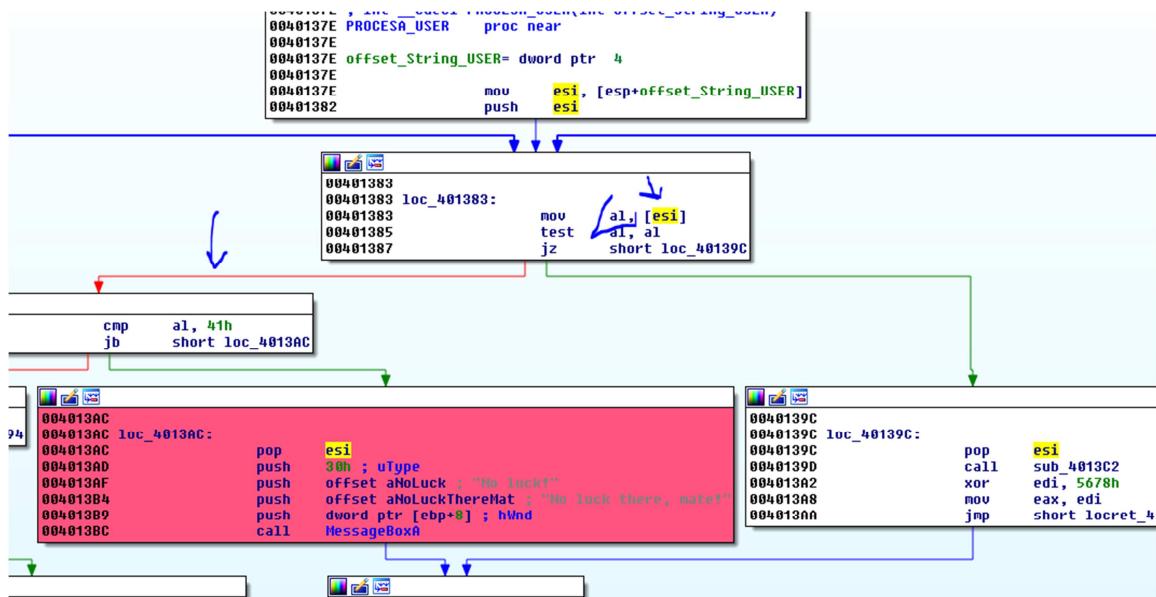
I propagate the function with SET TYPE and I see if it matches the reference argument.

```

00401228         push    offset String_USER ; offset_String_USER
0040122D         call    PROCESA_USER
00401232         push    eax
00401233         push    offset String_PASSWORD
00401238         call    PROCESA_PASS
0040123D         add     esp, 4
00401240         pop    eax
00401241         cmp     eax, ebx
00401243         jz      short loc_40124C

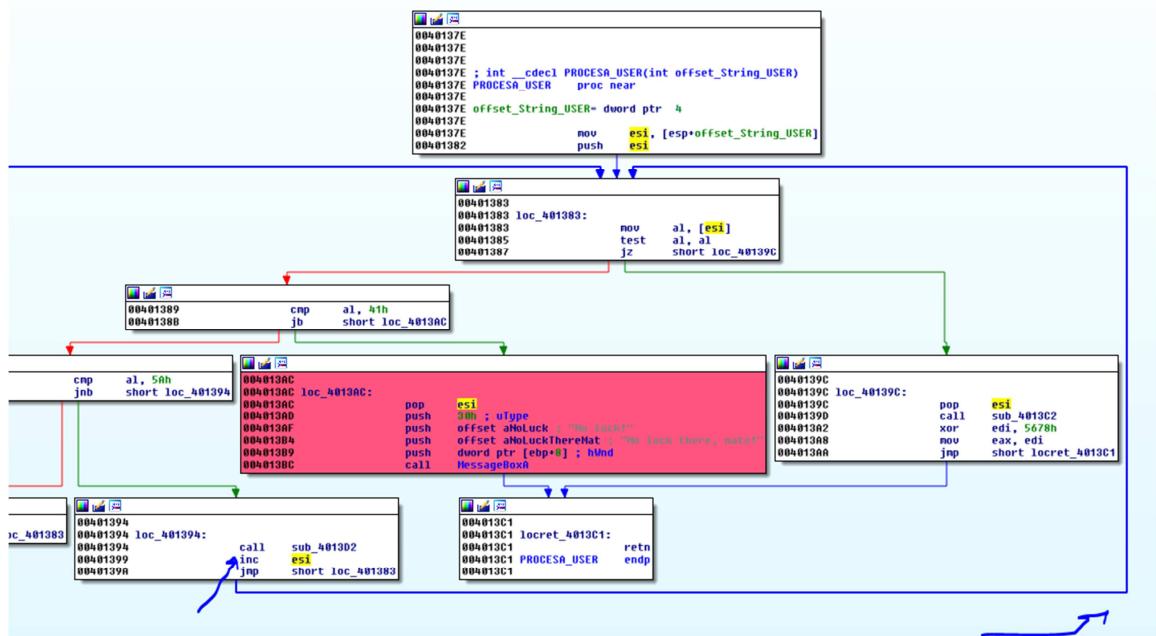
```

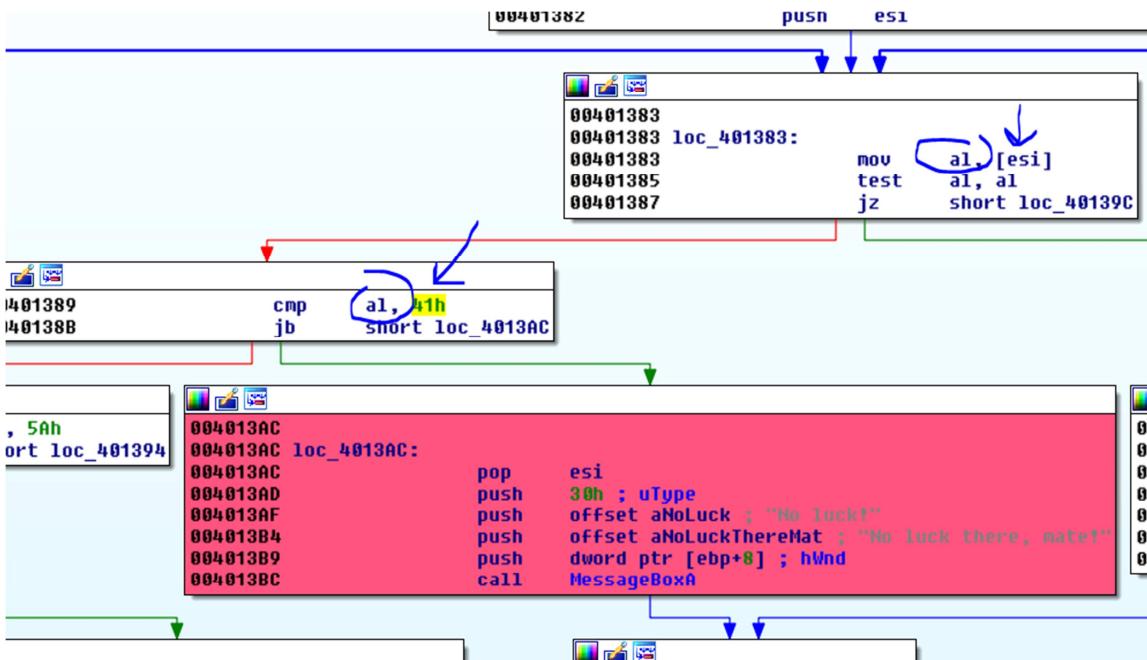
It matches with the arg name in the clarification it added.



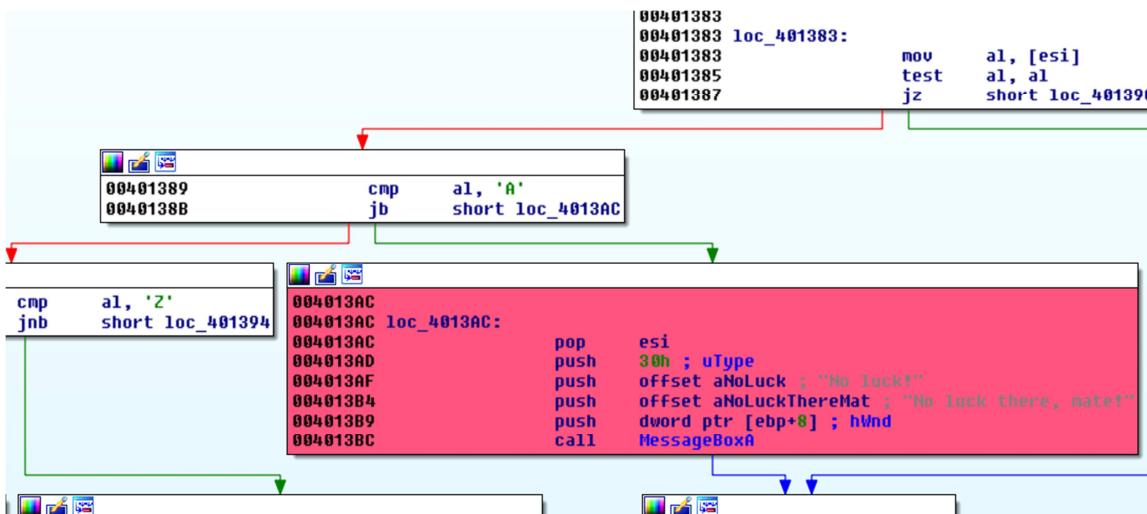
There is a LOOP that reads the String_USER bytes. This loop is repeated while it is not 0 or the end of the string and it will follow the red arrow way.

There, we have the complete loop. It increases ESI to read each byte of the String_USER characters and compares each one to 0x41.





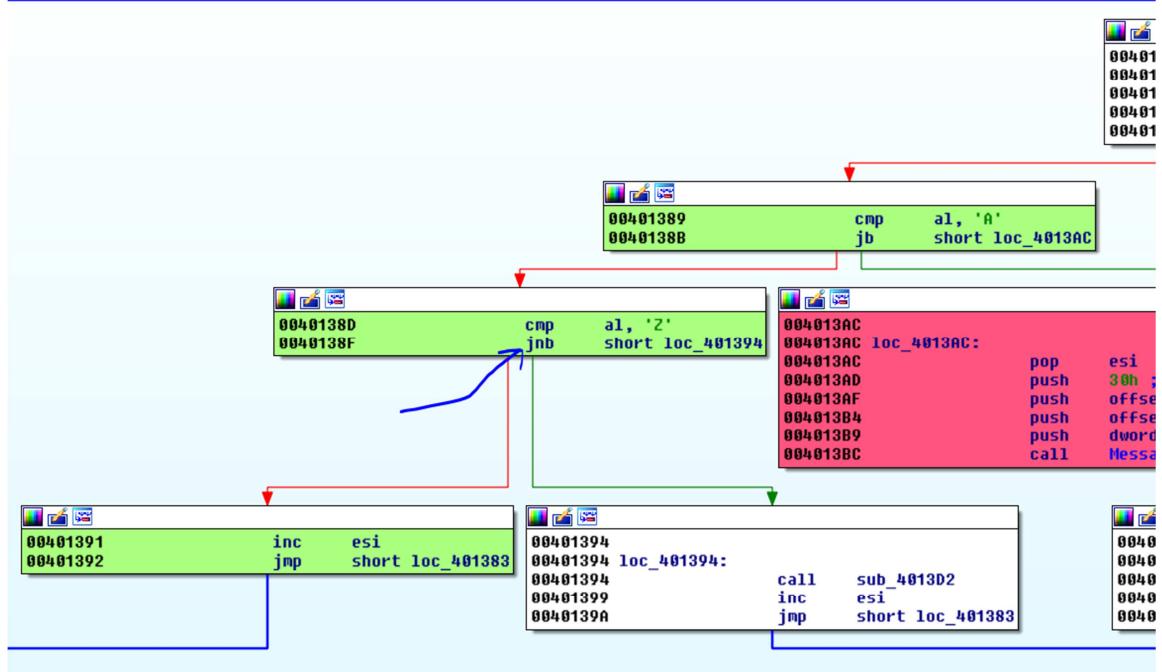
We can right click on the 0x41 and change it by A that is the ASCII character of that value.



If it is below A, it sends you to NO LUCK. So, if we see the ASCII table, it doesn't accept numbers in USER because they have to be greater or equal to A.

	DEC	HEX	OCT	CHAR		DEC	HEX	OCT	CH		DEC	HEX	OCT	CH		DEC	HEX	OCT	CH
0	0	000	NUL		32	20	040		@	64	40	100	@	96	60	140	`		
1	1	001	SOH		33	21	041	"	A	65	41	101	A	97	61	141	a		
2	2	002	STX		34	22	042	#	B	66	42	102	B	98	62	142	b		
3	3	003	ETX		35	23	043	\$	C	67	43	103	C	99	63	143	c		
4	4	004	EOT		36	24	044	%	D	68	44	104	D	100	64	144	d		
5	5	005	ENQ		37	25	045	&	E	69	45	105	E	101	65	145	e		
6	6	006	ACK		38	26	046	*	F	70	46	106	F	102	66	146	f		
7	7	007	BEL		39	27	047	'	G	71	47	107	G	103	67	147	g		
8	8	010	BS		40	28	050	(H	72	48	110	H	104	68	150	h		
9	9	011	TAB		41	29	051)	I	73	49	111	I	105	69	151	i		
10	A	012	LF		42	2A	052	*	J	74	4A	112	J	106	6A	152	j		
11	B	013	VT		43	2B	053	+	K	75	4B	113	K	107	6B	153	k		
12	C	014	FF		44	2C	054	,	L	76	4C	114	L	108	6C	154	l		
13	D	015	OR		45	2D	055	-	M	77	4D	115	M	109	6D	155	m		
14	E	016	SO		46	2E	056	.	N	78	4E	116	N	110	6E	156	n		
15	F	017	SI		47	2F	057	/	O	79	4F	117	O	111	6F	157	o		
16	10	020	DLE		48	30	060	0	P	80	50	120	80	112	70	160	p		
17	11	021	DC1		49	31	061	1	Q	81	51	121	Q	113	71	161	q		
18	12	022	DC2		50	32	062	2	R	82	52	122	R	114	72	162	r		
19	13	023	DC3		51	33	063	3	S	83	53	123	S	115	73	163	s		
20	14	024	DC4		52	34	064	4	T	84	54	124	T	116	74	164	t		
21	15	025	NAK		53	35	065	5	U	85	55	125	U	117	75	165	u		
22	16	026	SYN		54	36	066	6	V	86	56	126	V	118	76	166	v		
23	17	027	ETB		55	37	067	7	W	87	57	127	W	119	77	167	w		
24	18	030	CAN		56	38	070	8	X	88	58	130	X	120	78	170	x		
25	19	031	EM)		57	39	071	9	Y	89	59	131	Y	121	79	171	y		
26	1A	032	SUB		58	3A	072	:	Z	90	5A	132	Z	122	7A	172	z		
27	1B	033	ESC		59	3B	073	;	{	91	5B	133	{	123	7B	173	{		
28	1C	034	FS		60	3C	074	<	}	92	5C	134	}	124	7C	174	}		
29	1D	035	GS		61	3D	075	=		93	5D	135)	125	7D	175)		
30	1E	036	RS		62	3E	076	>		94	5E	136	^	126	7E	176	~		
31	1F	037	US		63	3F	077	?		95	5F	137	_	127	7F	177	DEL		

It checks if all String_USER characters are greater than 0x41 (A).



It also checks if it is not below Z, with JNB, sends it to the 0x401394 block. If not, it leaves it like that and continues with the next char for the green blocks way.

So, it accepts capital letters except Z. The chars greater than Z or equal go to that block. Let's see what happens there.

```

004013D2
004013D2
004013D2
004013D2 RESTA_20 proc near
004013D2     sub    al, 20h
004013D4     mov    [esi], al
004013D6     retn
004013D6 RESTA_20 endp
004013D6
    
```

I renamed it RESTA (subtraction in English) because that's what it does. If it is greater than Z, it subtracts 20 and saves it.

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	135]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

If you enter 0x61 that is the “a” upper letter, when subtracting 0x20, it will be 0x41 that is the “A” capital letter. It does the same the thing with all chars greater than Z or equal.

If it is **Z - 0x20**, the result is **0x3A** that is “：“.

21	15	U25	NAK	55	55	U65	5	85	55	125	U	117	75	165	U
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	:	91	5B	133	[123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS	61	3D	075	=	93	5D	135]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

We can create a Python script to make the keygen.

```
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY
```

This script does the same that the program does. It takes each user string char and compares them to 0x41. If it is below, it tells you it is an invalid char and exits. If not below 0x41, it compares it to 0x5A. If above or equal to 0x5A, it subtracts 0x20 and adds it to the userMAY string.

If I enter pePP, it changes to PEPP.

```
keygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pePP
USER PEPP

Process finished with exit code 0
```

If I enter Z, it changes to ":".

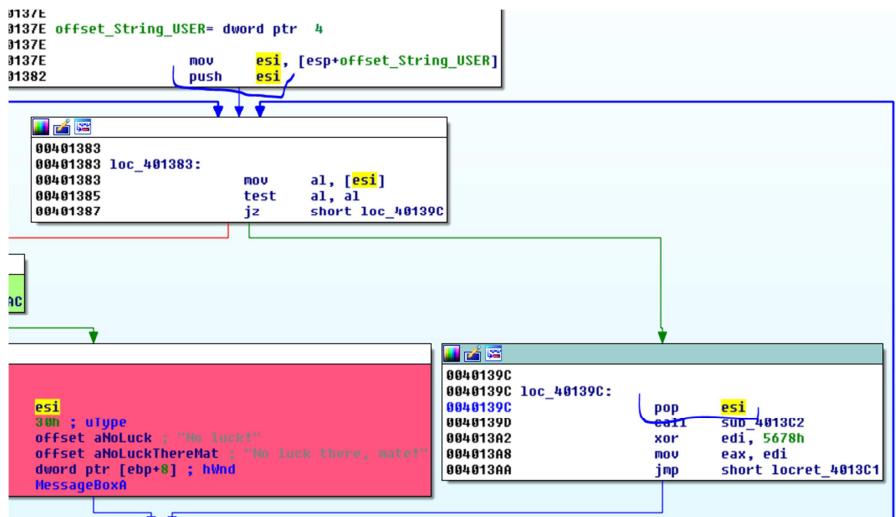
```
keygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
peza
USER PE:A

Process finished with exit code 0
```

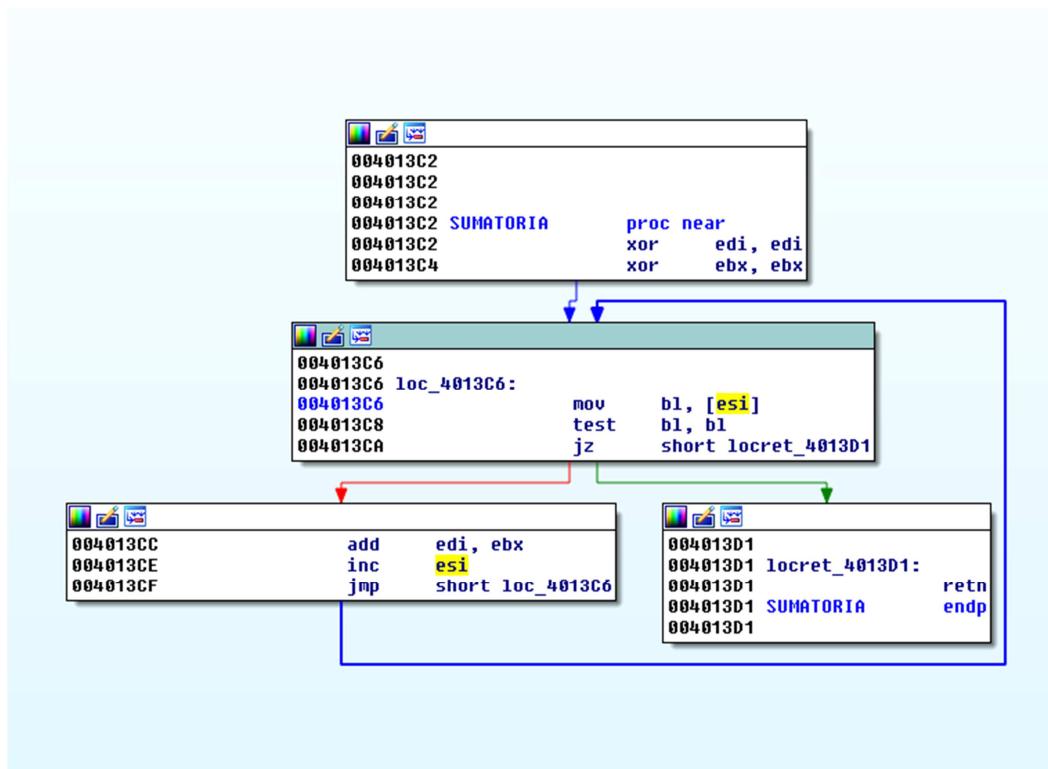
Until this point, it does the same that the program does. Let's see what it does after leaving the LOOP. It continues this way.



When it finds a 0 char, it will leave the LOOP and go to the 0x40139C block.



Before increasing ESI, it had PUSHed it into the stack to save the original value that pointed to the string start. With POP ESI, it recovers it before entering the 0x4013C2 CALL.



It is a LOOP that adds all the bytes. That's why I renamed it as SUMATORIA or ADDITION. So, we can add that in our new script.

```

sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord_(userMAY[i])

print "SUMATORIA", hex(sum)

```

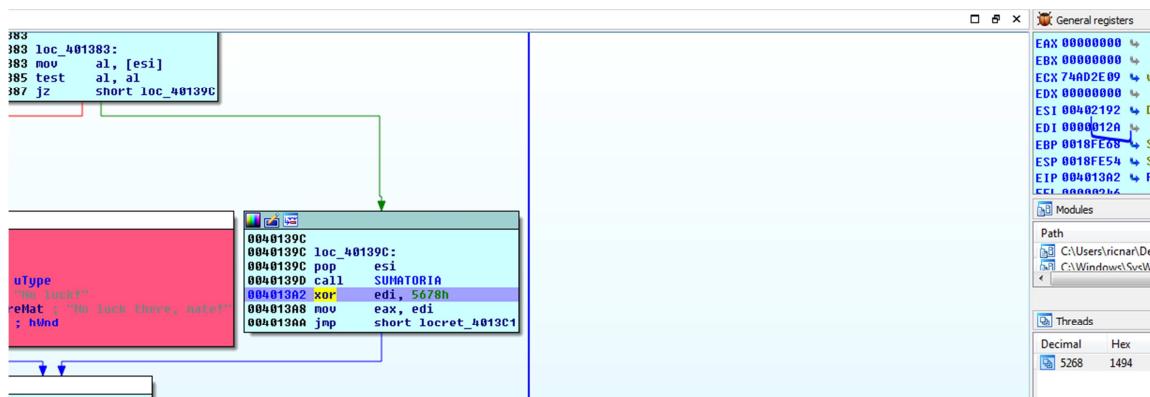
```

C:\Python27\python.exe C:/Users/ricnar/Des:
pepe
USER PEPE
SUMATORIA 0x12a
Process finished with exit code 0

```

It adds all the bytes and prints the addition.

To check if I'm OK, I set a BP at the end of the addition and add pepe in the user edit and enter the 989898 password.



The addition result is 0x12A. So, we are OK.

In that line, it XORs it with 0x5678. So, I add that to the script.

```
sum=0
user=raw_input()
largo=len(user)
if (largo>_0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

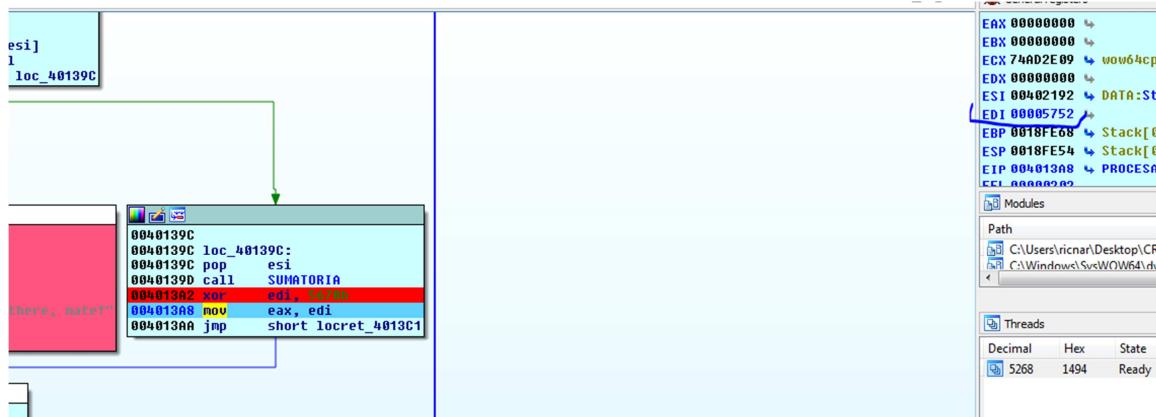
print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord_(userMAY[i])

print "SUMATORIA", hex(sum)

xoreando= sum ^ 0x5678
print "XOREADO", hex(xoreando)
```

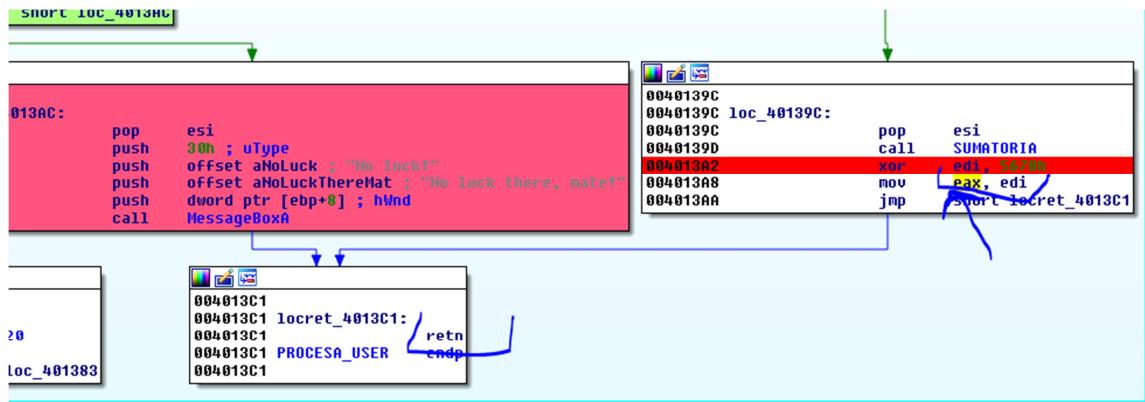
If I execute that line, it XORs and the result is the same as in the script.



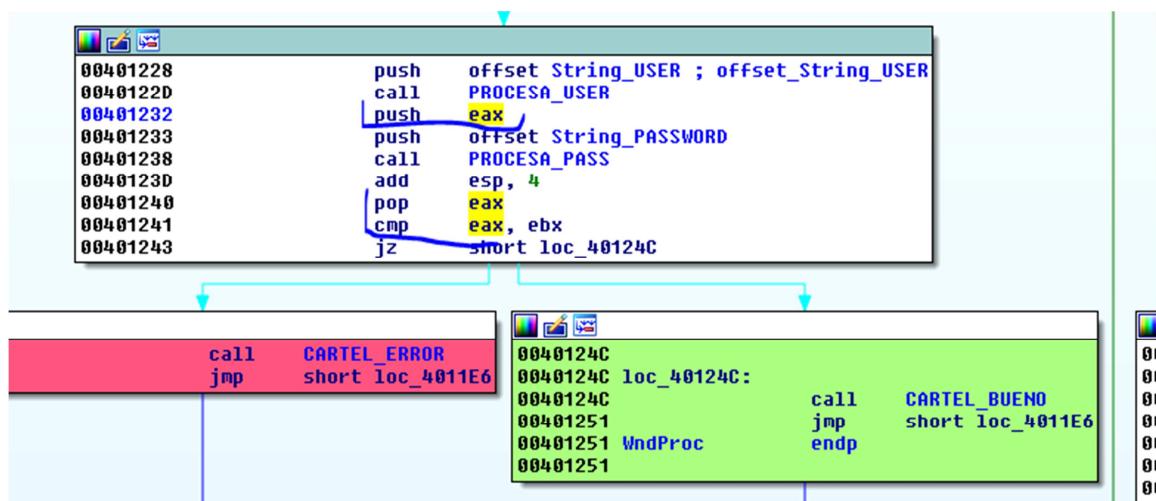
```
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752

Process finished with exit code 0
```

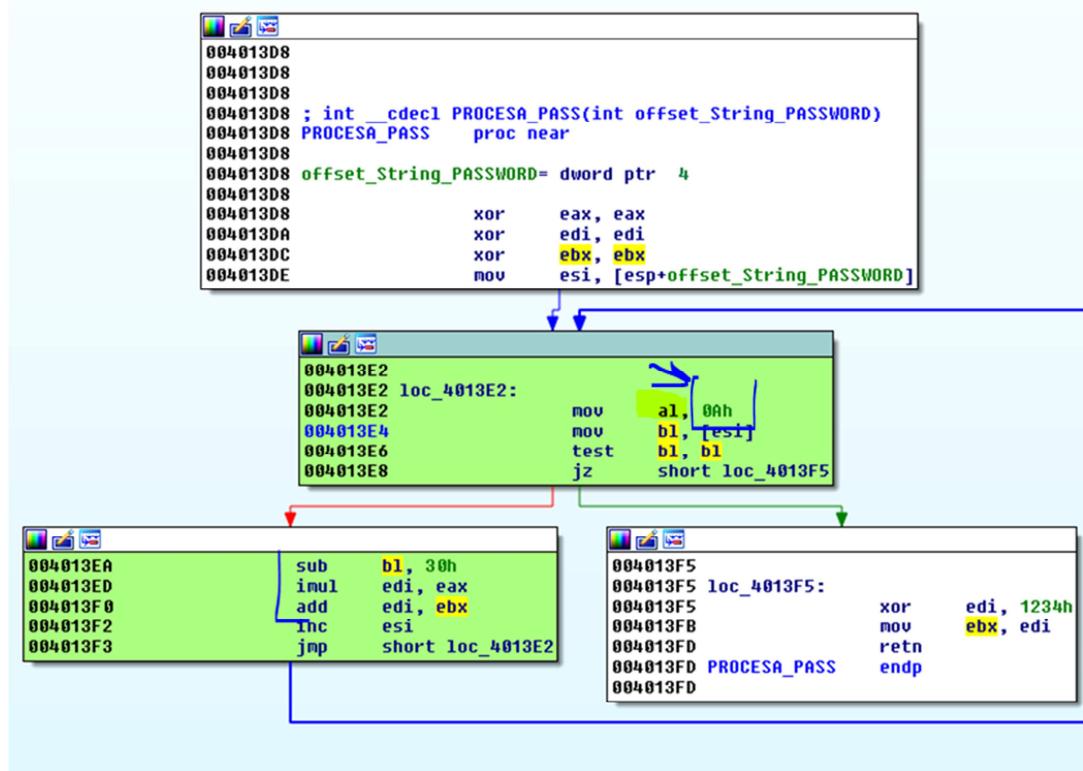
Then, it moves the result to EAX and leaves.



Then, it PUSHes EAX and recovers it with POP EAX before the final comparison. In the CMP EAX, EBX, the first member will be this value that comes from PROCESA_USER.



Let's see what happens with the password in PROCESA_PASS.



It reads each byte and moves them to BL and subtracts 0x30 to each one leaving the result in EBX. Then, it multiplies EDI that has the addition by 0xA and it adds it EBX. I'll do other part of the script with this.

```
nieder_Electric_Somachine_HVAC_AxEditGrid_ActiveX_Exploit.py x [ ] 

sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord(userMAY[i])

print "SUMATORIA", hex(sum)

xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)

#-----

password="989898"

sum2=0

for i in range(len(password)):
    sum2 = sum2 * 0xa
    sum2+=(ord(password[i])-0x30)

print "RESUL", hex(sum2)
```

I don't enter the password with the keyboard. I'm just trying it because, in the keygen, we just enter the user, but if my password is, for example, 989898.

It multiplies sum2 (the saved addition) by 0xA, then it adds it the byte minus 0x30 as in the program.

```
c:\cygwin
C:\Python27\python.exe C:/Users/ricnar/Desktop
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
RESUL 0xf1aca

Process finished with exit code 0
```

When I execute it, the 989898 password gave **f1aca** that is the hex value of 989898 as the result of all that

```
|~~~~~
| Python>hex(989898)
| 0xf1aca
| Python |
```

All that in the script can be summarized as converting the string to hex with the `hex()` function.

```
1 sum=0
2 user=raw_input()
3 largo=len(user)
4 if (largo>_0xb):
5     exit()
6
7 userMAY=""
8
9 for i in range(largo):
10    if (ord(user[i])<0x41):
11        print "CARACTER INVALIDO"
12        exit()
13    if (ord(user[i]) >= 0x5a):
14        userMAY+= chr(ord(user[i])-0x20)
15    else:
16        userMAY+= chr(ord(user[i]))
17
18 print "USER",userMAY
19
20 for i in range(len(userMAY)):
21    sum+=ord_(userMAY[i])
22
23 print "SUMATORIA", hex(sum)
24
25 xoreado= sum ^ 0x5678
26 print "XOREADO", hex(xoreado)
27
28 #-----
29
30 password=int("989898")
31
32 print "RESUL", hex(password)
```

It gives me the same result.

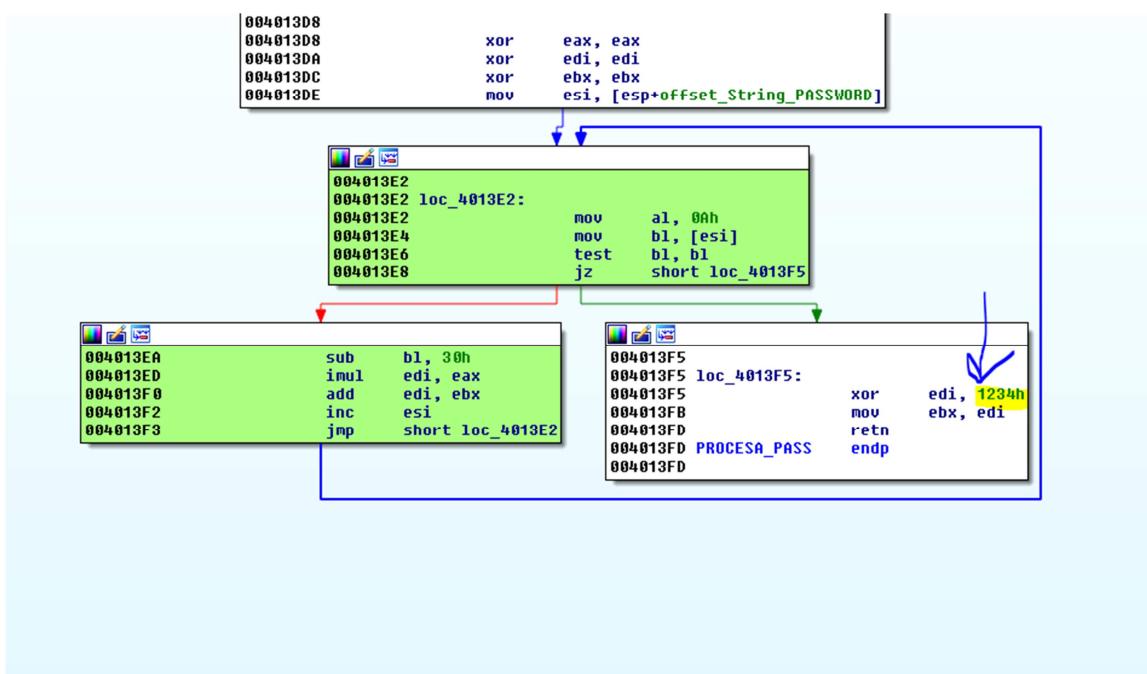
```

keygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
RESUL 0xf1aca

Process finished with exit code 0

```

Finally, it XORs that result with 0x1234 and leaves to compare it with the value returned by PROCESA_USER in EAX.



The generic formula would be...

$\text{hex(password)} \wedge 0x1234 = \text{XORED}$.

Where XOREADO or XORED is the result that PROCESA_USER returned.

Let's clear it.

$\text{hex(password)} = \text{XORED} \wedge 0x1234$

If I XOR the result I had with 0x1234, I almost have it.

```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

Exploit.py

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i]))
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord_(userMAY[i])

print "SUMATORIA", hex(sum)

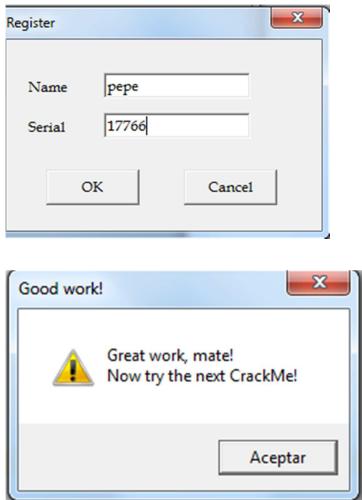
xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)

TOTAL= xoreado ^ 0x1234

print "TOTAL", TOTAL
```

If I run it with the pepe string...

```
ygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752
TOTAL 17766
Process finished with exit code 0
```



So, we already have the keygen. It is not necessary to convert the result into decimal because Python does the conversion.

Here, I copy it to the keygen.

```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

userMAY=""

for i in range(largo):
    if (ord(user[i])<0x41):
        print "CARACTER INVALIDO"
        exit()
    if (ord(user[i]) >= 0x5a):
        userMAY+= chr(ord(user[i])-0x20)
    else:
        userMAY+= chr(ord(user[i]))

print "USER",userMAY

for i in range(len(userMAY)):
    sum+=ord (userMAY[i])

print "SUMATORIA", hex(sum)
```

```
xoreado= sum ^ 0x5678
print "XOREADO", hex(xoreado)
```

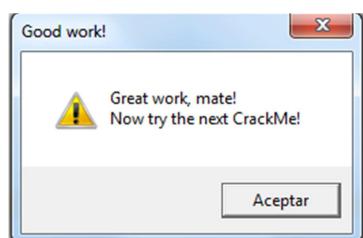
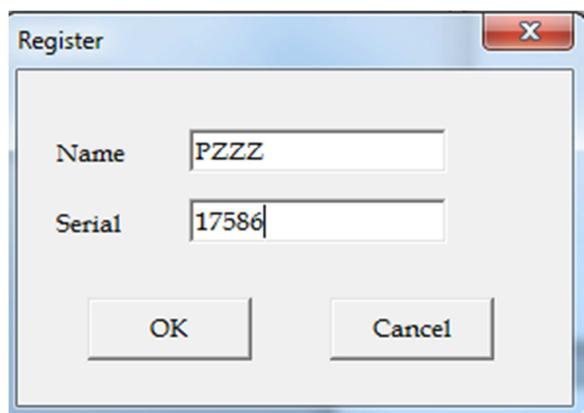
```
TOTAL= xoreado ^ 0x1234
```

```
print "PASSWORD", TOTAL
```

Even in rare cases with the Z.

```
eygen
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
PZZZ
USER P:::
SUMATORIA 0xfe
XOREADO 0x5686
PASSWORD 17586

Process finished with exit code 0
```



So, we reversed and made a keygen to the Cruehead's Crackme.

Ricardo Narvaja
Translated by: @IvinsonCLS