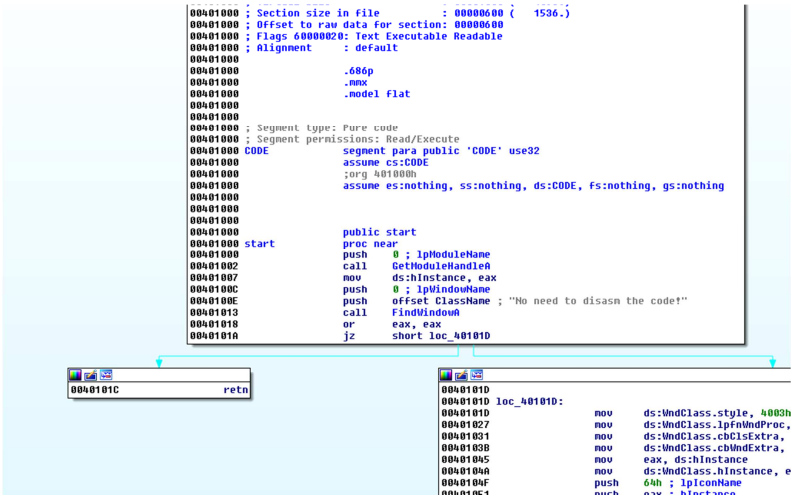
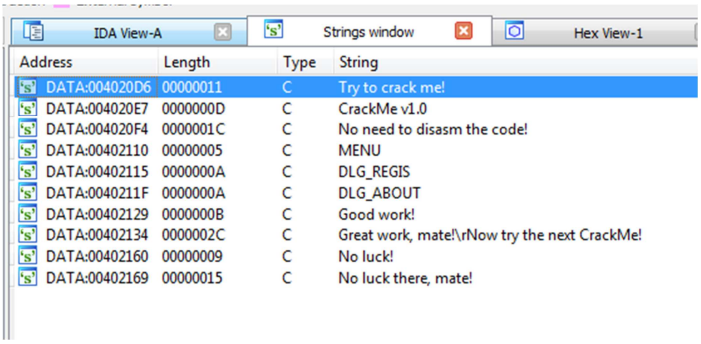
PART 19

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.



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.

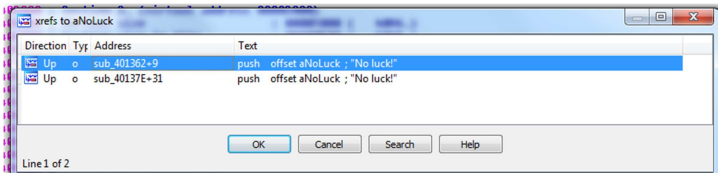
The first thing we do is seeing the strings. If that fails, we see the API's or functions used by the program. In this case, the strings are visible. So, let's follow this way.



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_40134D+2To
DATA:00402134 ; CHAR Text[]
DATA:00402134 Text            db 'Great work, mate!',0Dh,'Now try the next CrackMe!',0
DATA:00402134                                         ; DATA XREF: sub_40134D+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+EEto
DATA:00402169                                         ; sub_40137E+36To
DATA:0040217E ; CHAR byte_40217E[16]
DATA:0040217E byte_40217E     db 10h dup(0)           ; DATA XREF: WndProc+10BTo
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
.idata:00403000 ; Section 3. (virtual address 00003000)
.idata:00403000 ; Virtual size                : 00001000 (   4096.)
```

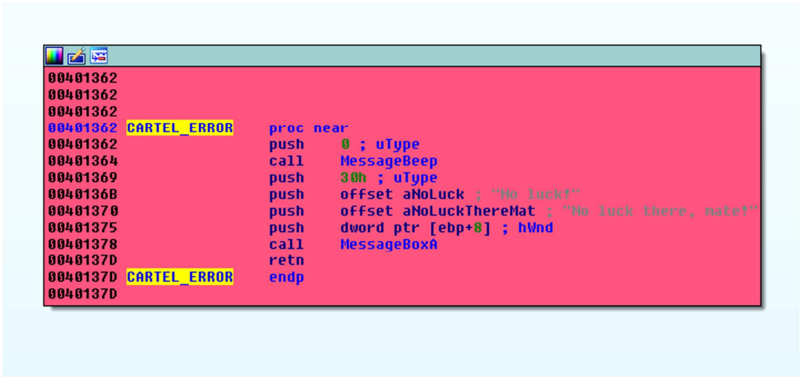
By pressing X or CTRL +X, we see two references to that string.



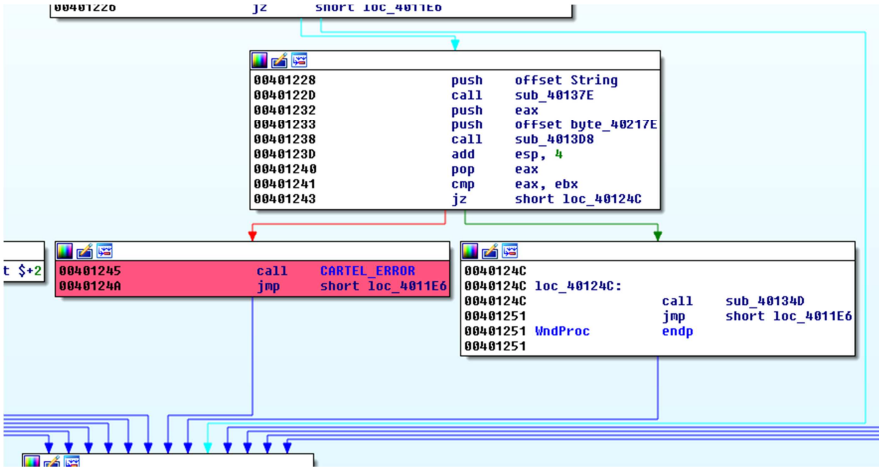
Let's see the first one.

```
00401362
00401362
00401362
00401362 sub_401362      proc near
00401362                 push    0 ; uType
00401364                 call    MessageBeep
00401369                 push    30h ; uType
0040136B                 push    offset aNoLuck ; "No luck!"
00401370                 push    offset aNoLuckThereMat ; "No luck there, mate!"
00401375                 push    dword ptr [ebp+8] ; hWnd
00401378                 call    MessageBoxA
0040137D                 retn
0040137D sub_401362      endp
0040137D
```

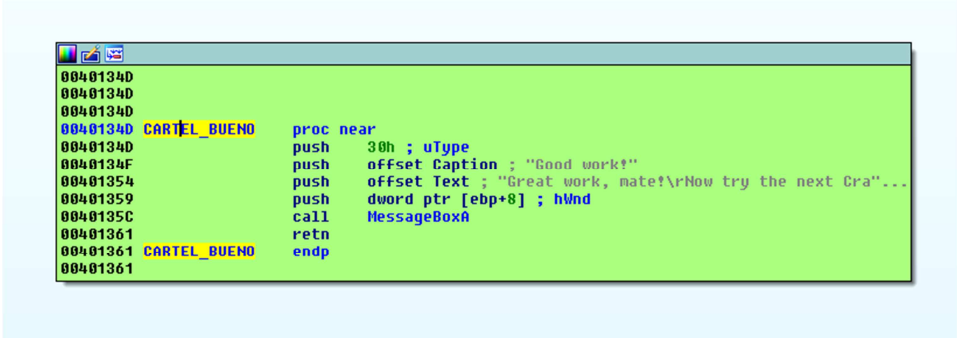
I painted the block in red because it belongs to the bad boy.



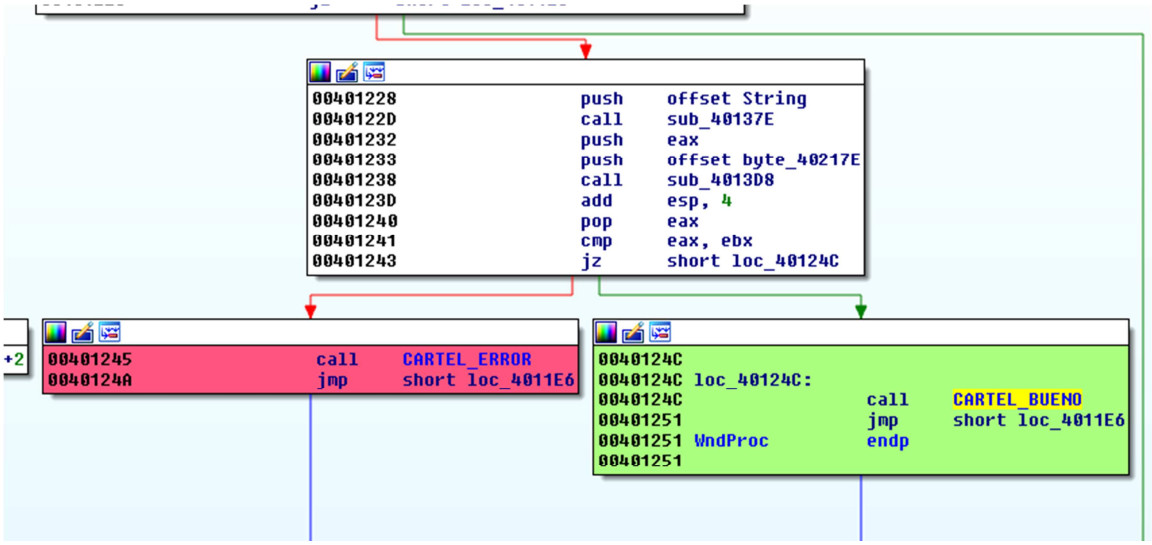
Let’s see where it comes from.



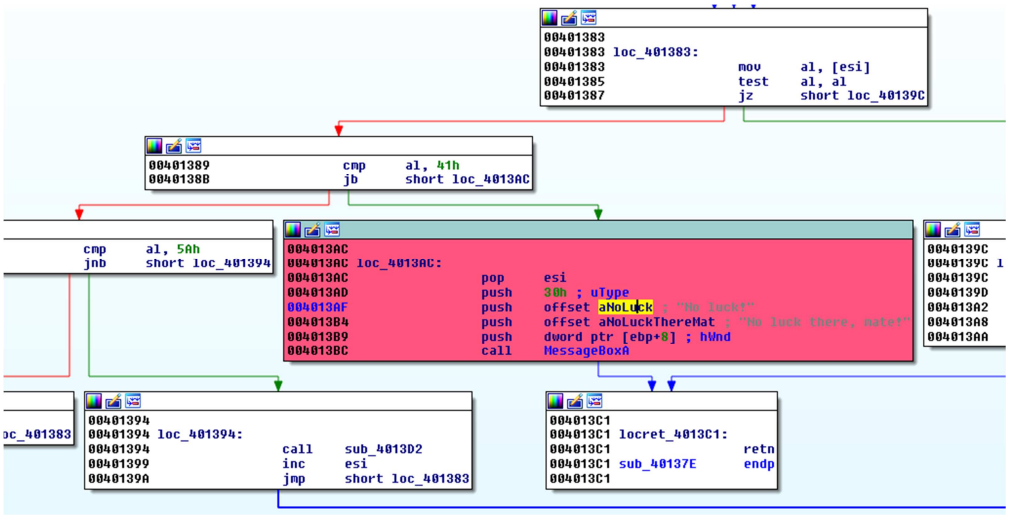
.So, the other way should be the good boy. Let’s see inside the 0x40134D function.



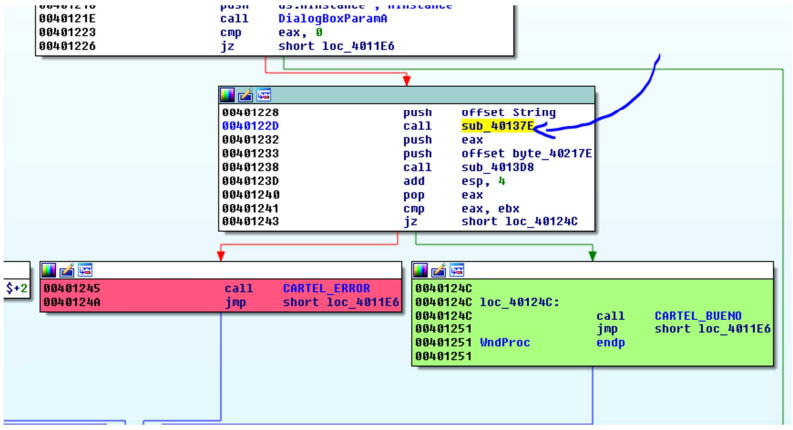
So, in the reference…

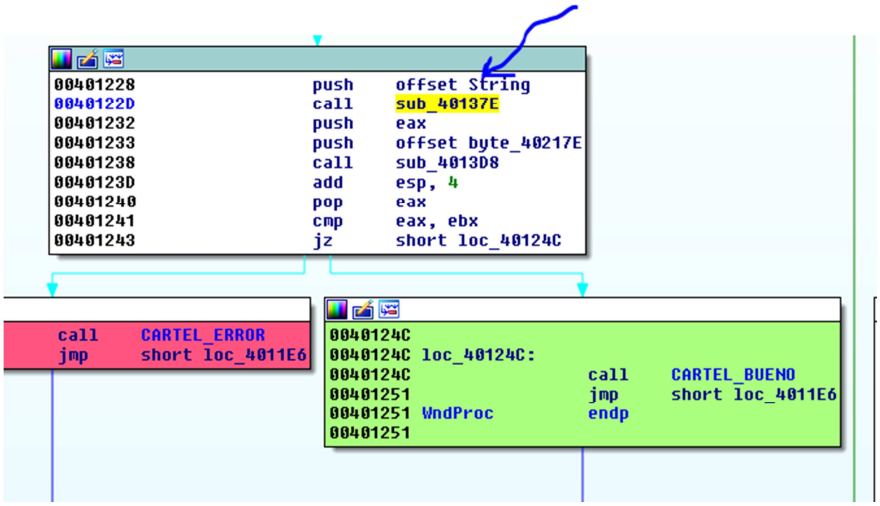


The other reference to the NO LUCK string comes from here.

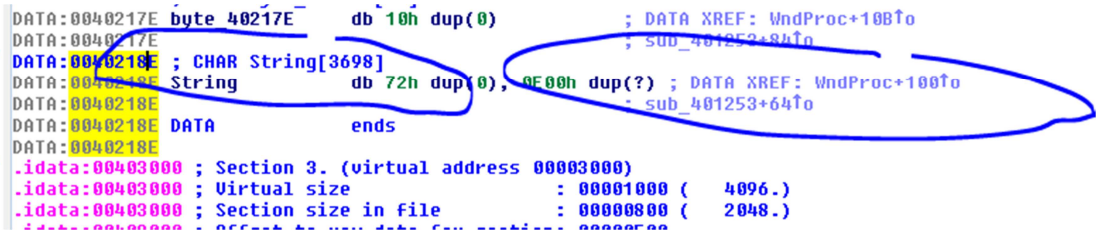


And the other error message reference comes from here.



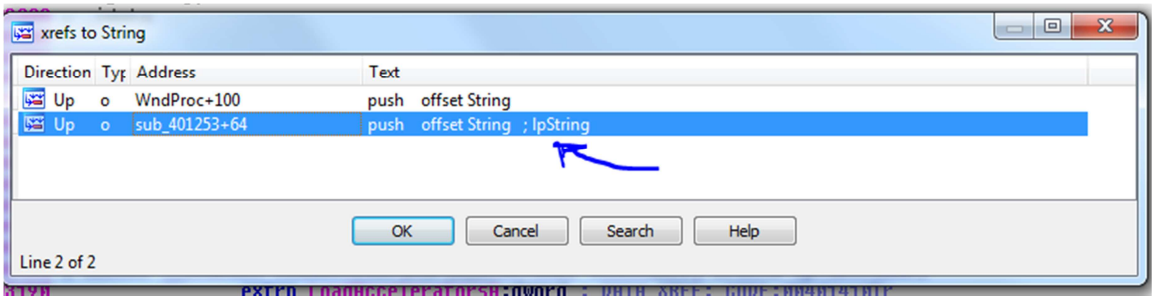


We see the argument of that function is a variable global address (OFFSET) called string. If we click on it…

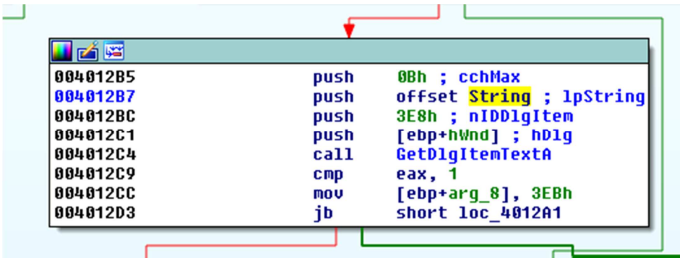


It's a 3698-byte buffer in the 0x40218E address of the DATA section.

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.

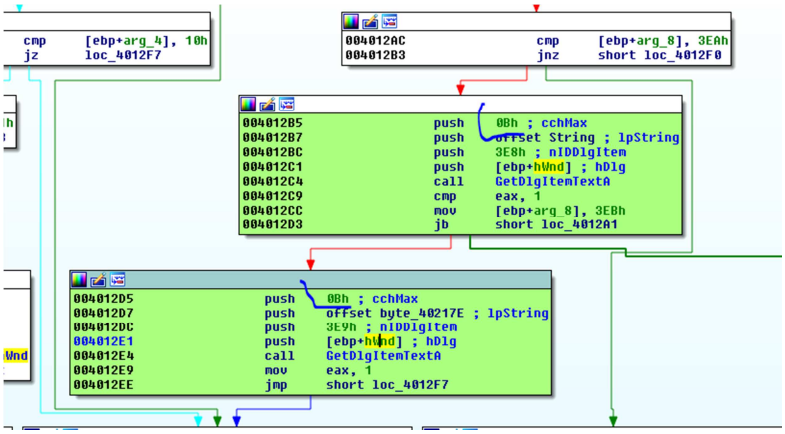
Return value

Type: **UINT**

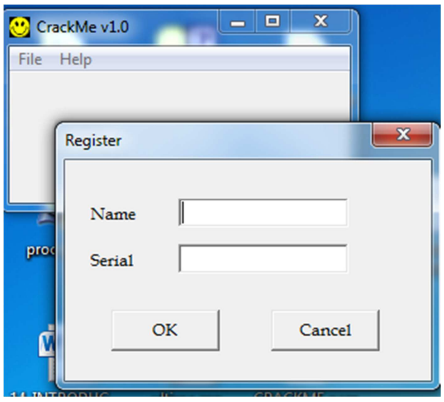
If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

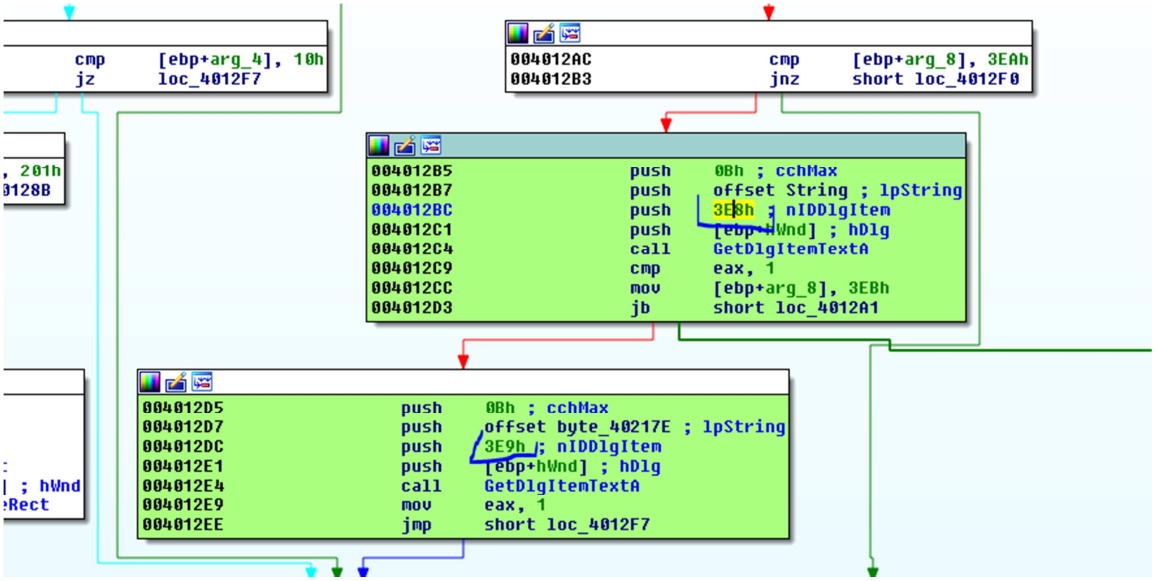
There, it receives some text through the keyboard.



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.

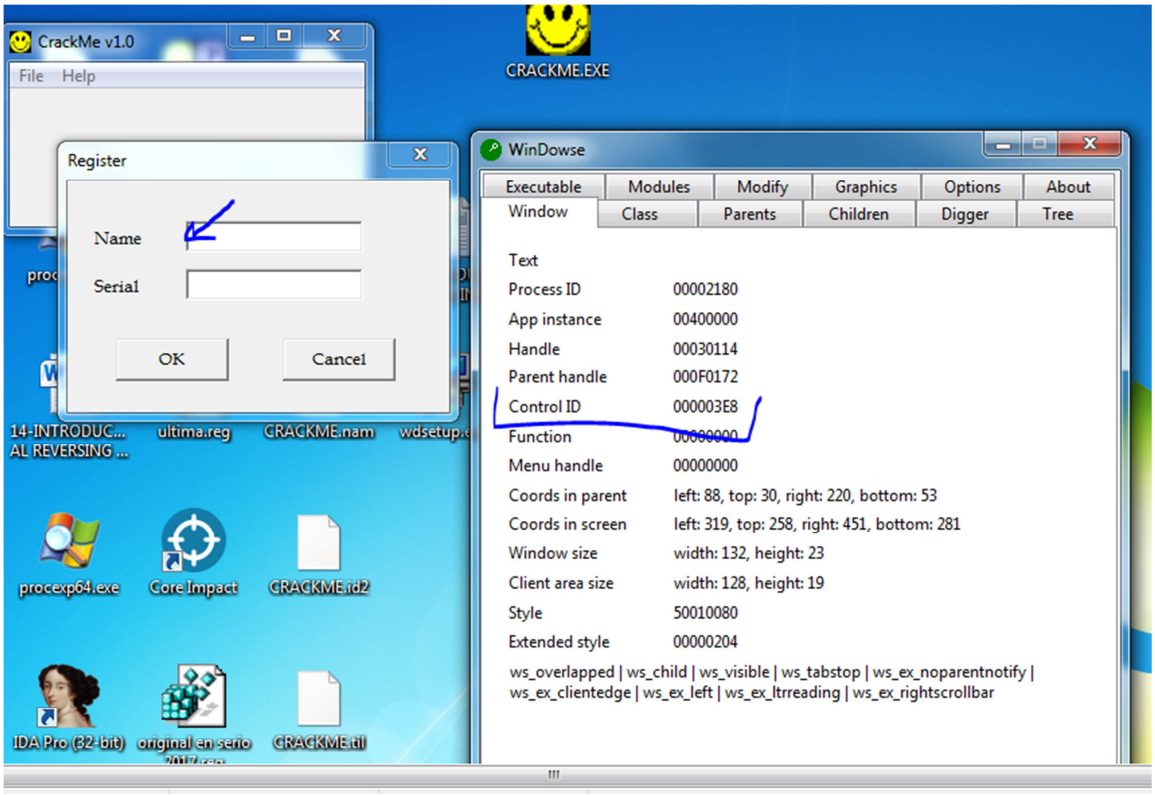


Both of them have the nIDDlgItem control number of each one: 0x3E8 and 0x3E9.



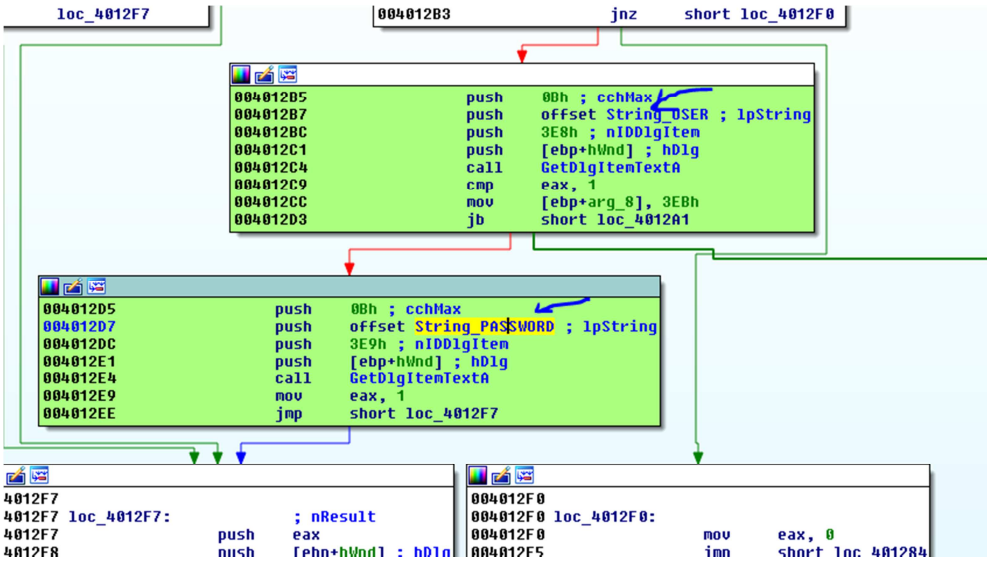
Using the GREATIS WINDOWSE program, it gives us info about the windows where we place the cursor.

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



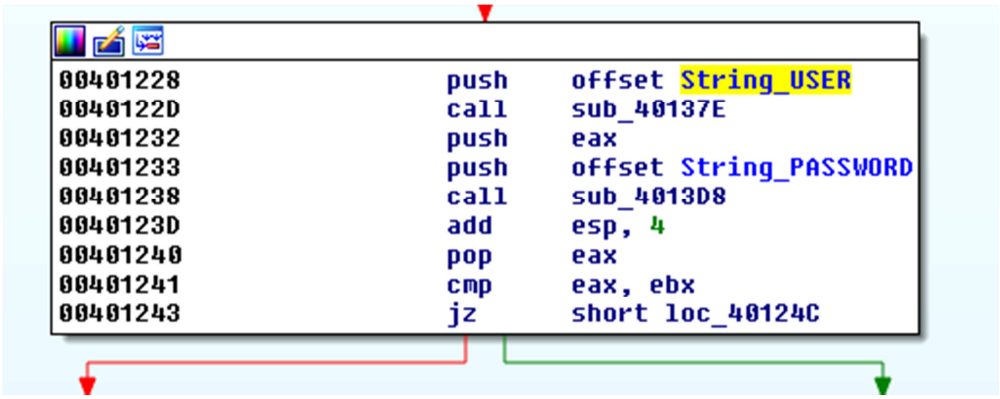
I verify that the first edit box control ID is 0x3E8 and the second one is 0x3E9.

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.

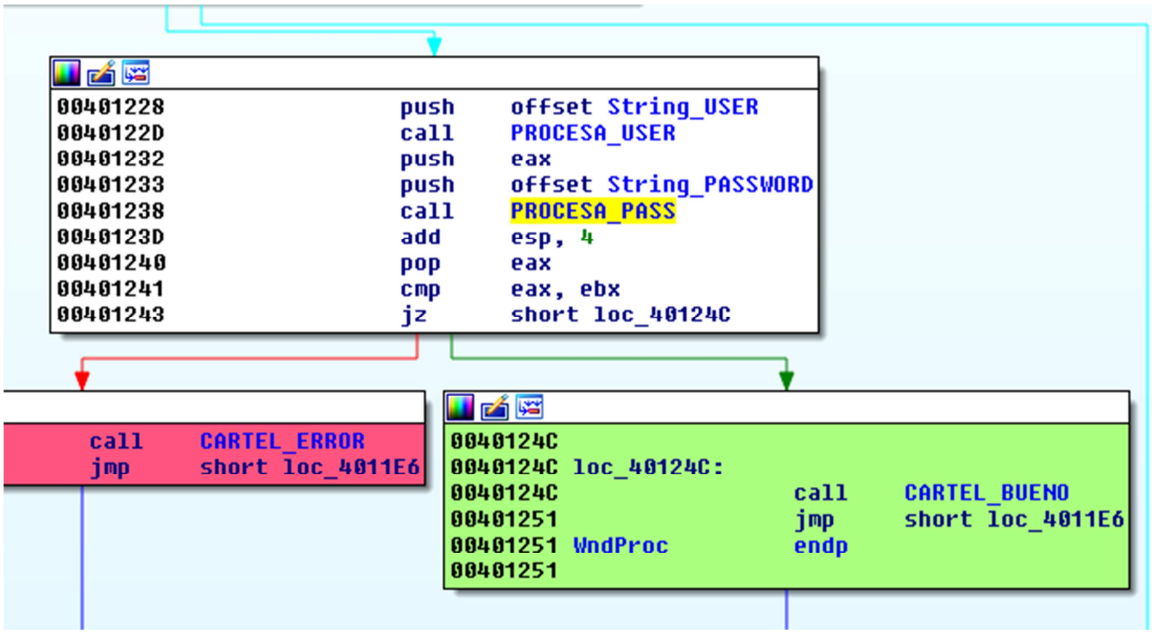


We know where it saves the user and password we enter. Let's see what it does with them.

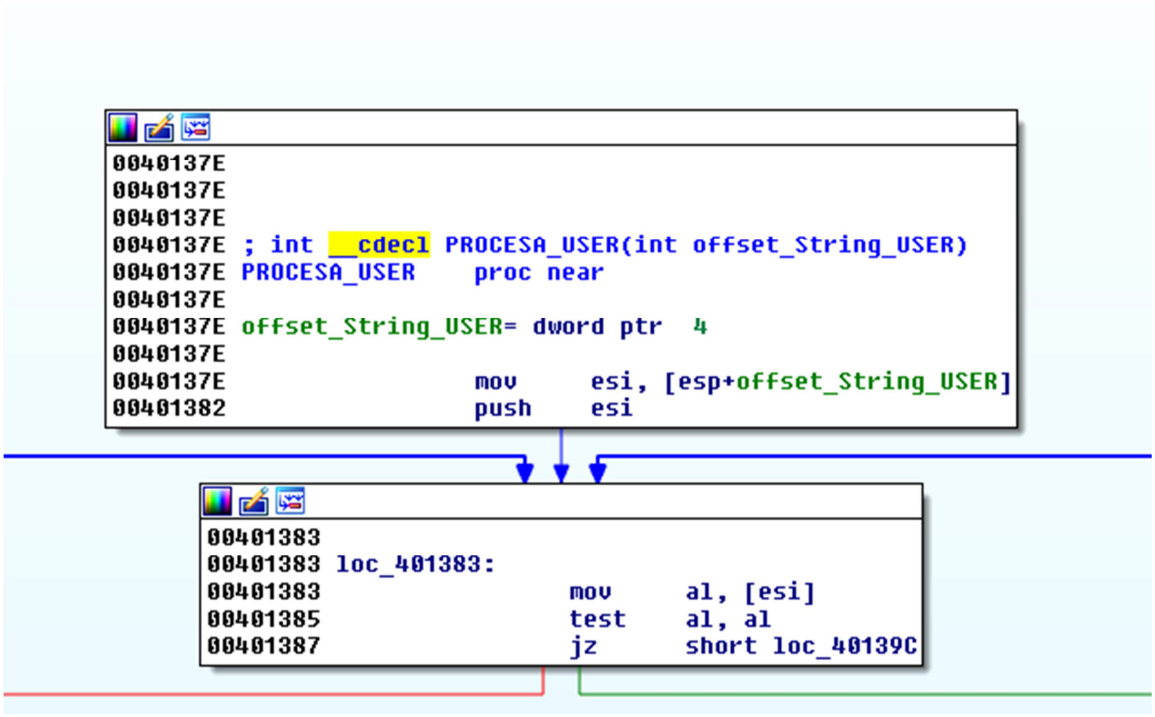
The String_USER is processed in here.



Let’s rename the functions.

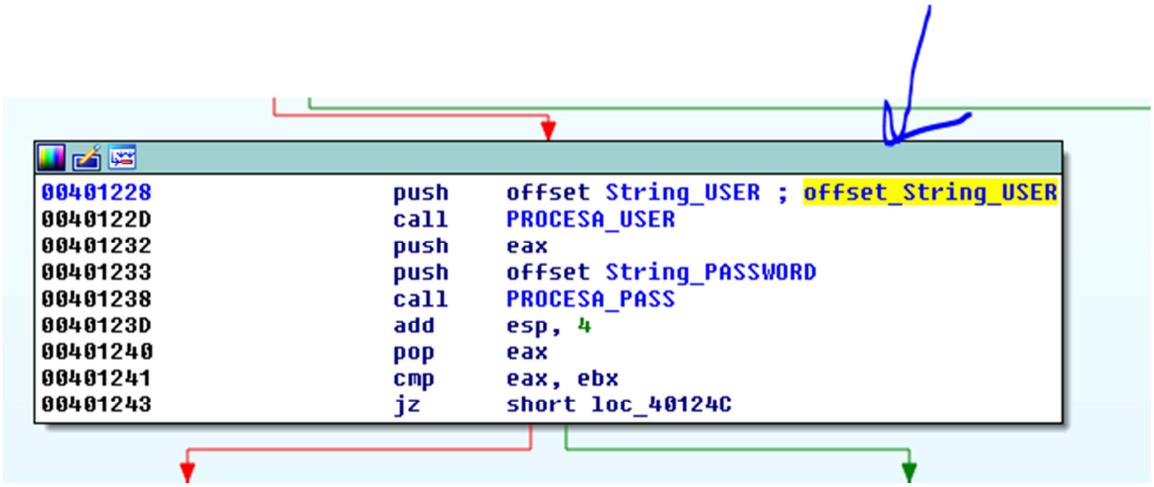


Let’s analyze PROCESA_USER now.



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.

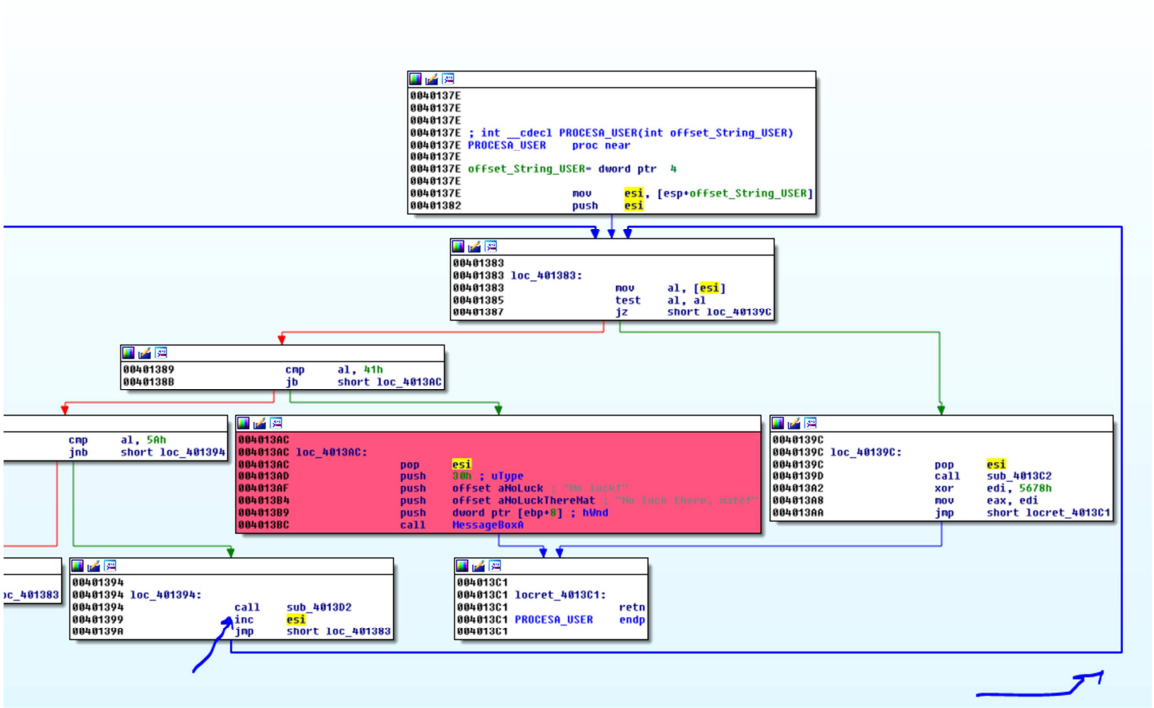
I propagate the function with SET TYPE and I see if it matches the reference argument.

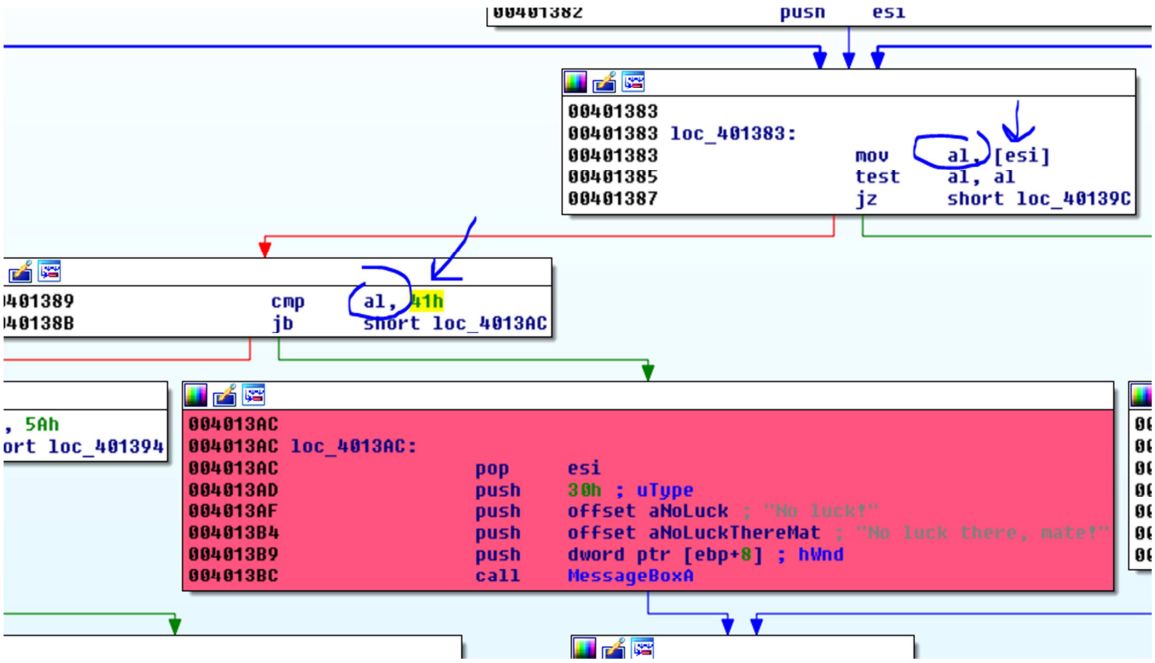


It matches with the arg name in the clarification it added.

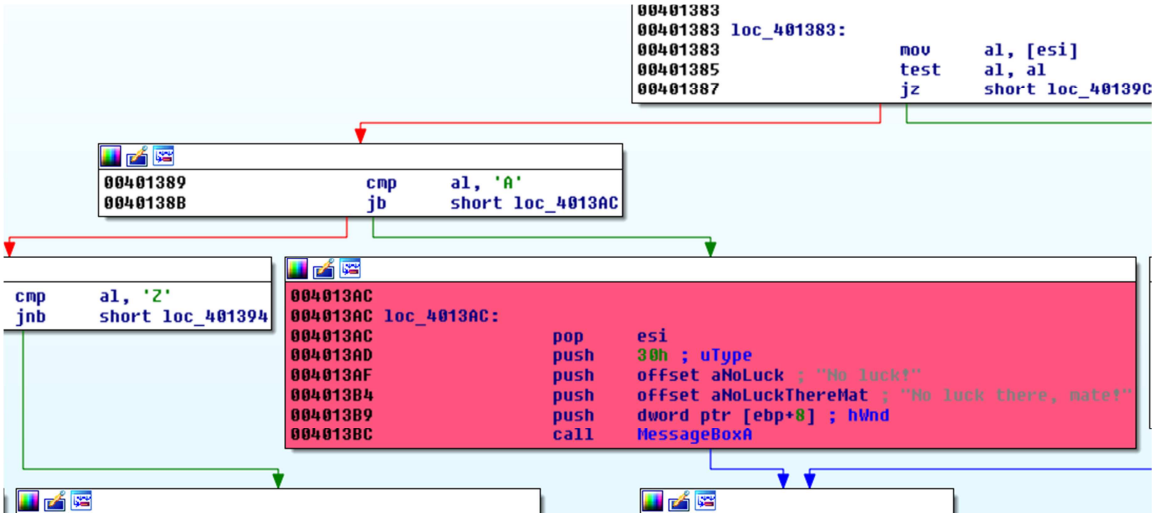
There is a LOOP that reads the String_USER bytes. This loop is repeated while it is not 0 or the end of the string and it will follow the red arrow way.

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





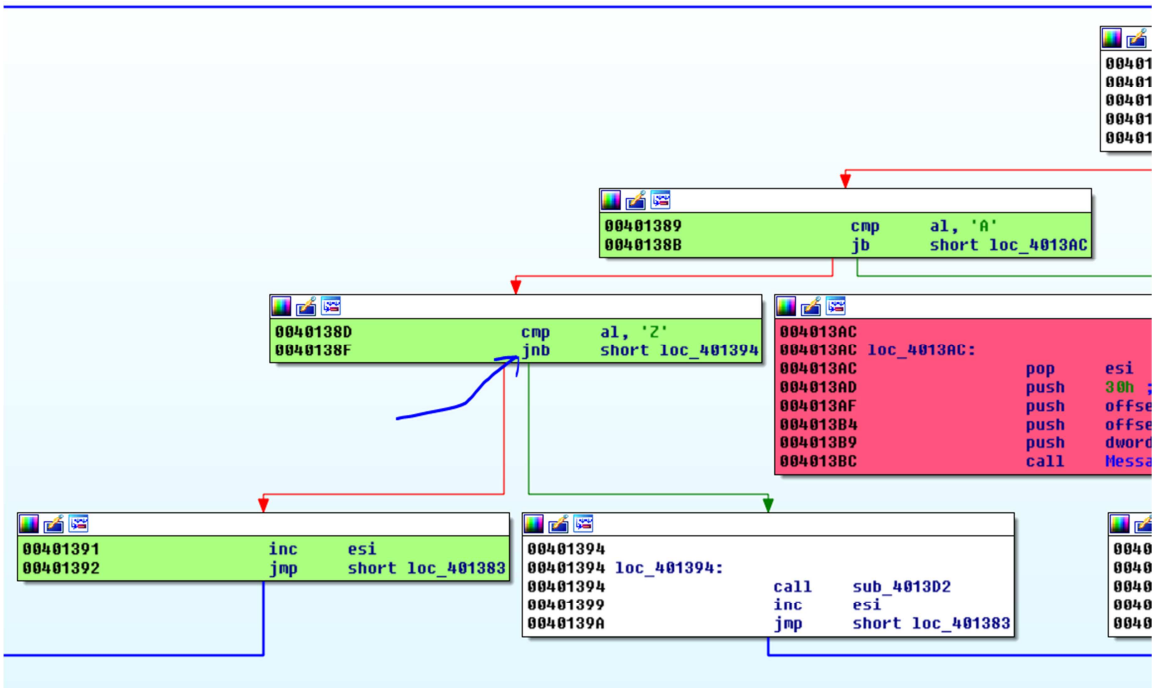
We can right click on the 0x41 and change it by A that is the ASCII character of that value.



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**.

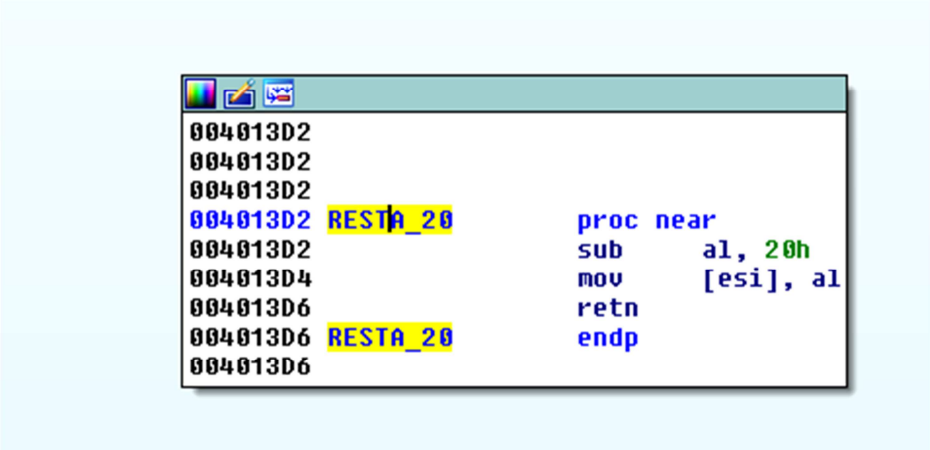
DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
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

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

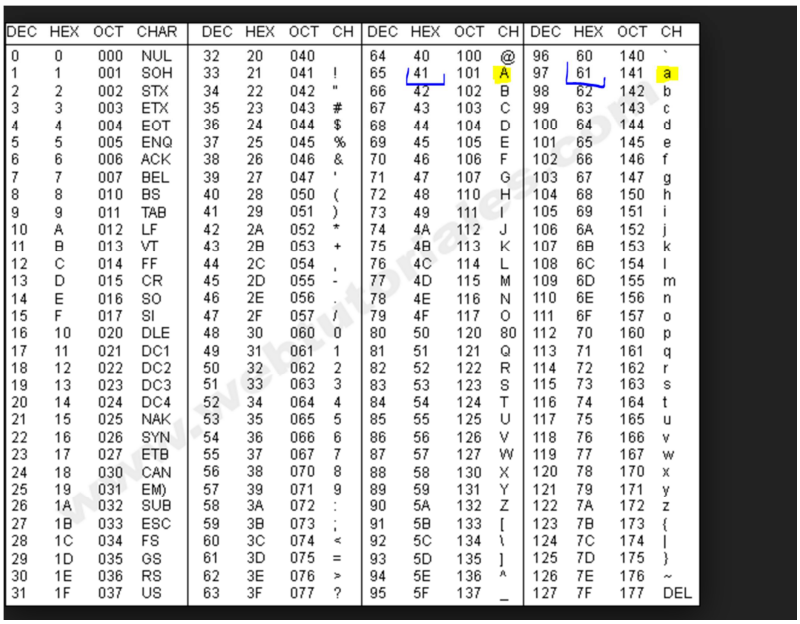


It also checks if it is not below Z, with JNB, sends it to the 0x401394 block. If not, it leaves it like that and continues with the next char for the green blocks way.

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



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 “**:**”.

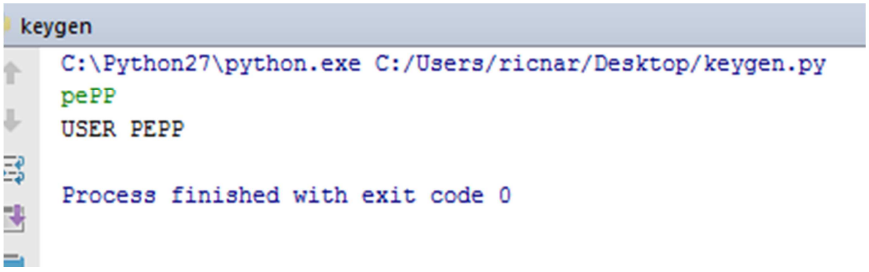
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	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

We can create a Python script to make the keygen.

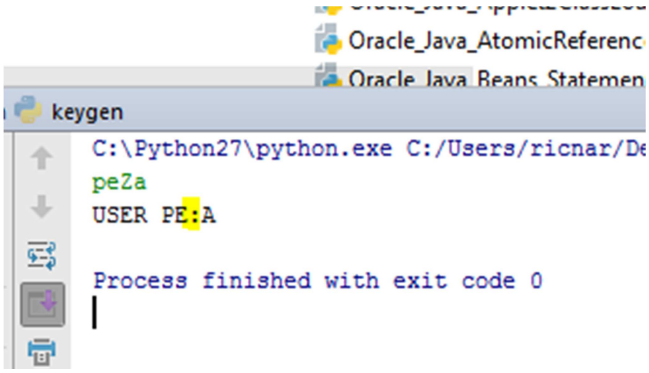
```
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

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.

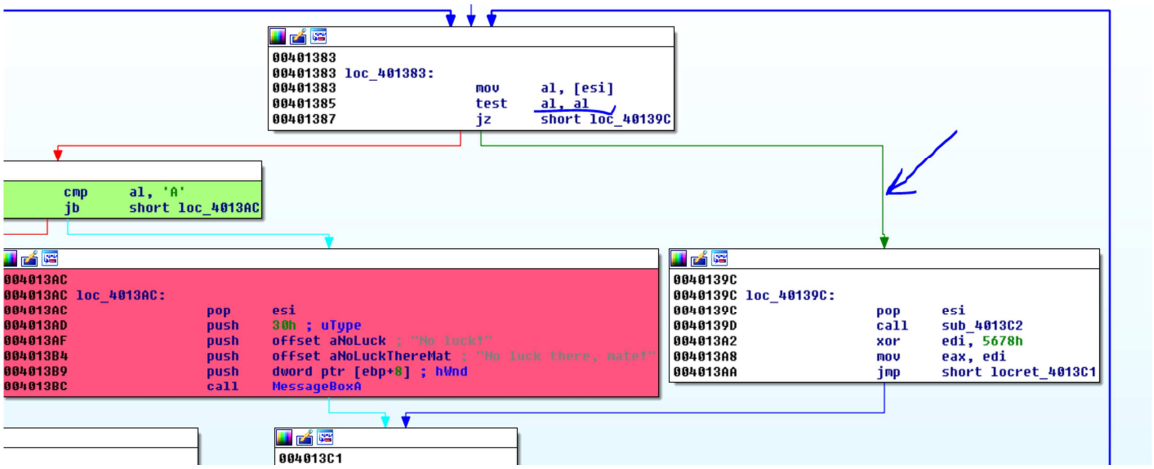
If I enter pePP, it changes to PEPP.



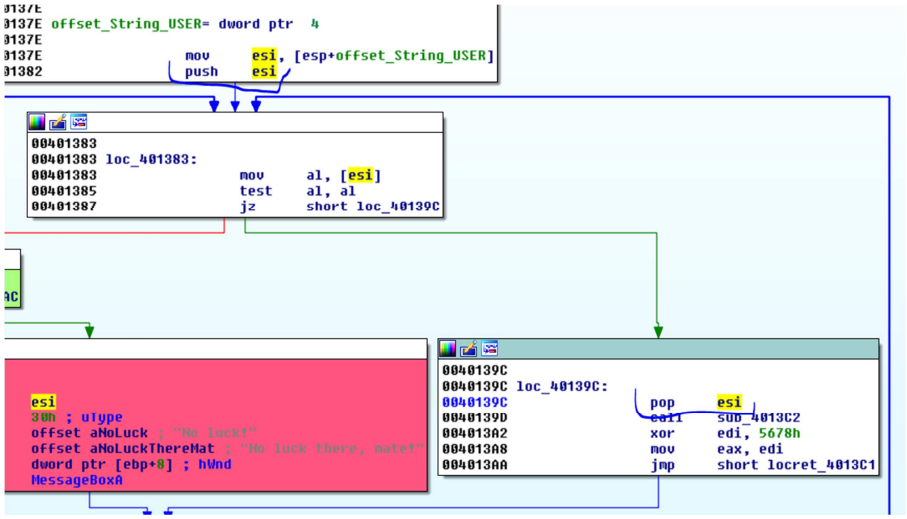
If I enter Z, it changes to "**:**".



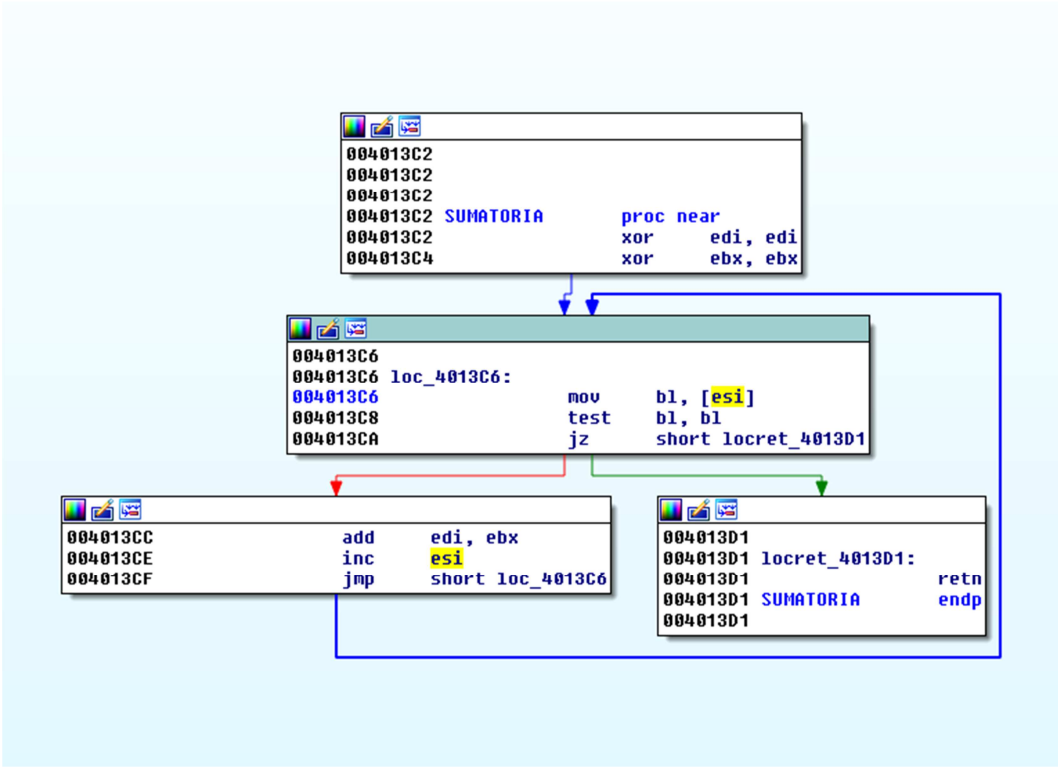
Until this point, it does the same that the program. Let's see what it does after leaving the LOOP. It continues this way.



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.



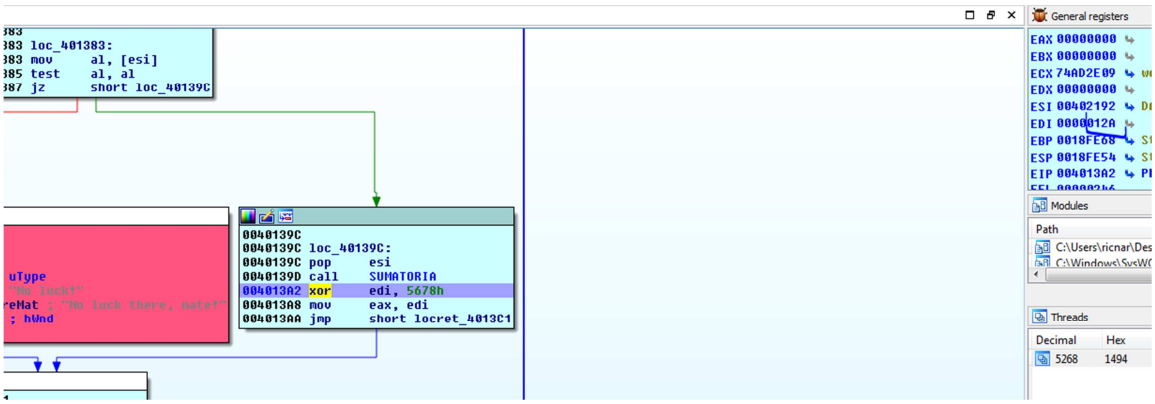
It is a LOOP that adds all the bytes. That's why I renamed it as SUMATORIA or ADDITION. So, we can add that in our new script.

```
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.

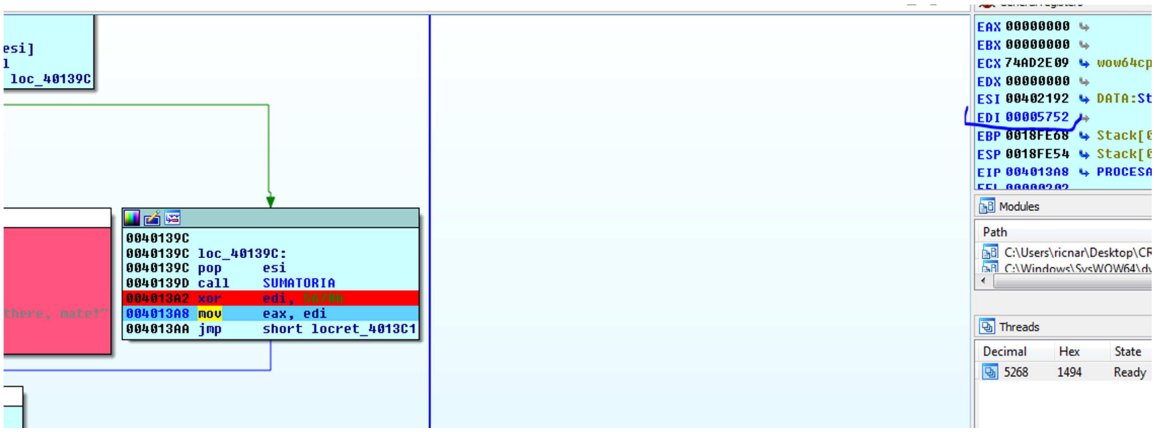


The addition result is 0x12A. So, we are OK.

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

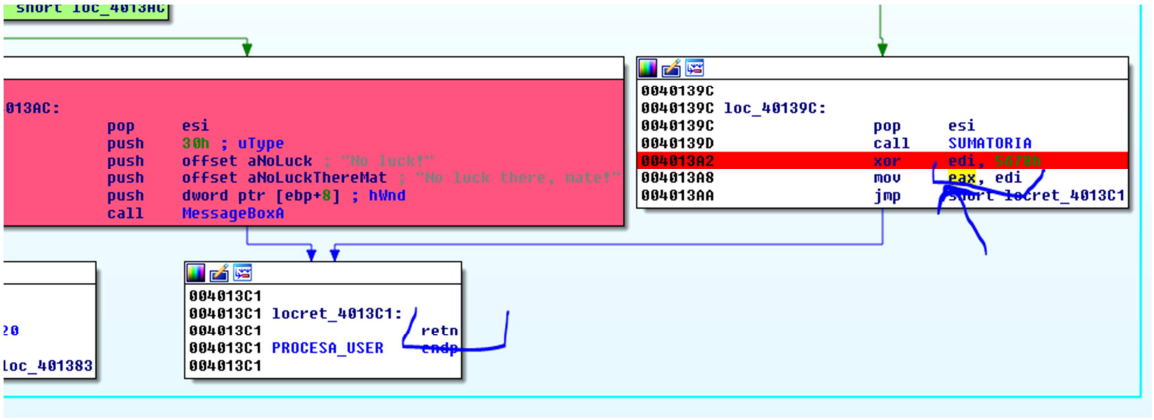
```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

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)

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

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

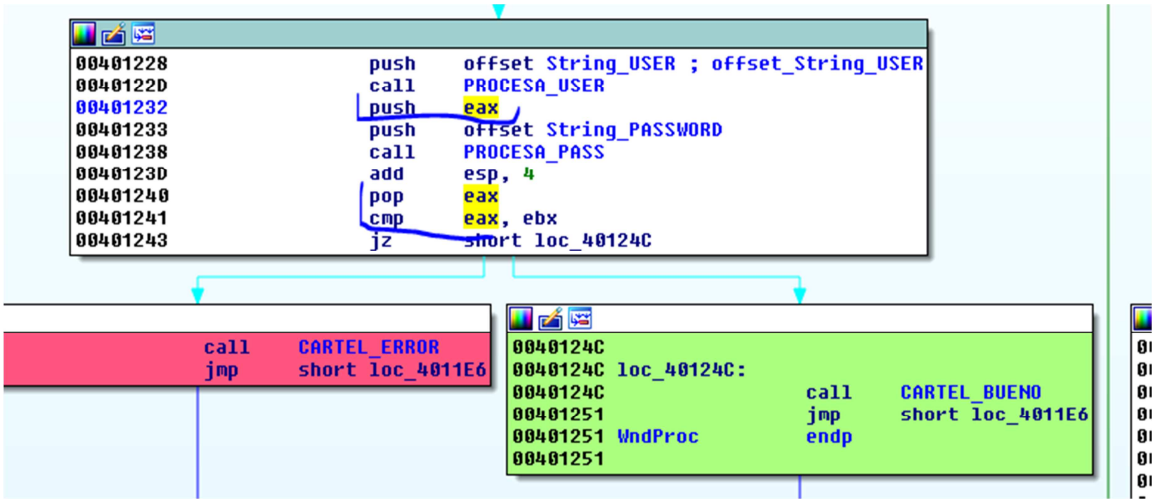



```
C:\Python27\python.exe C:/Users/ricnar/Desktop/keygen.py
pepe
USER PEPE
SUMATORIA 0x12a
XOREADO 0x5752

Process finished with exit code 0
```

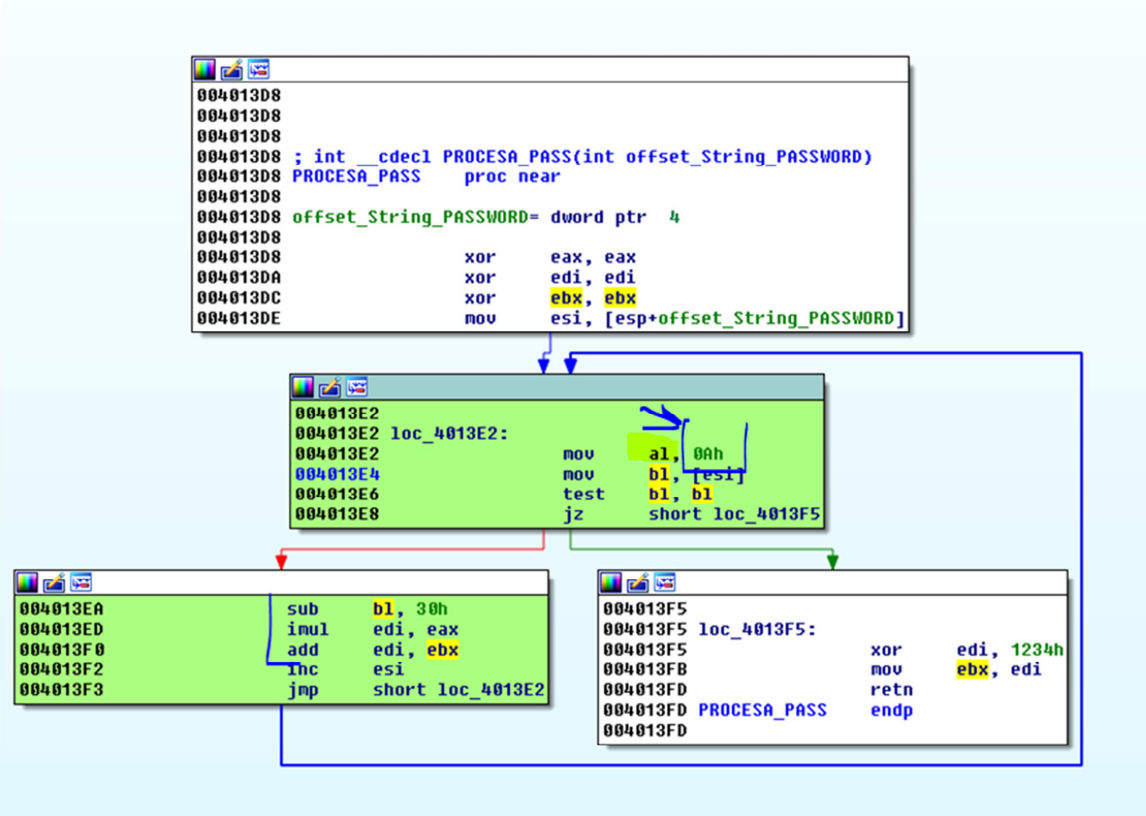
Then, it moves the result to EAX and leaves.



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 0x0A and it adds it EBX. I’ll do other part of the script with this.

```
nelder_Electric_SoMachine_HVAC_AxEditGrid_ActiveX_Exploit.py
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:\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
```

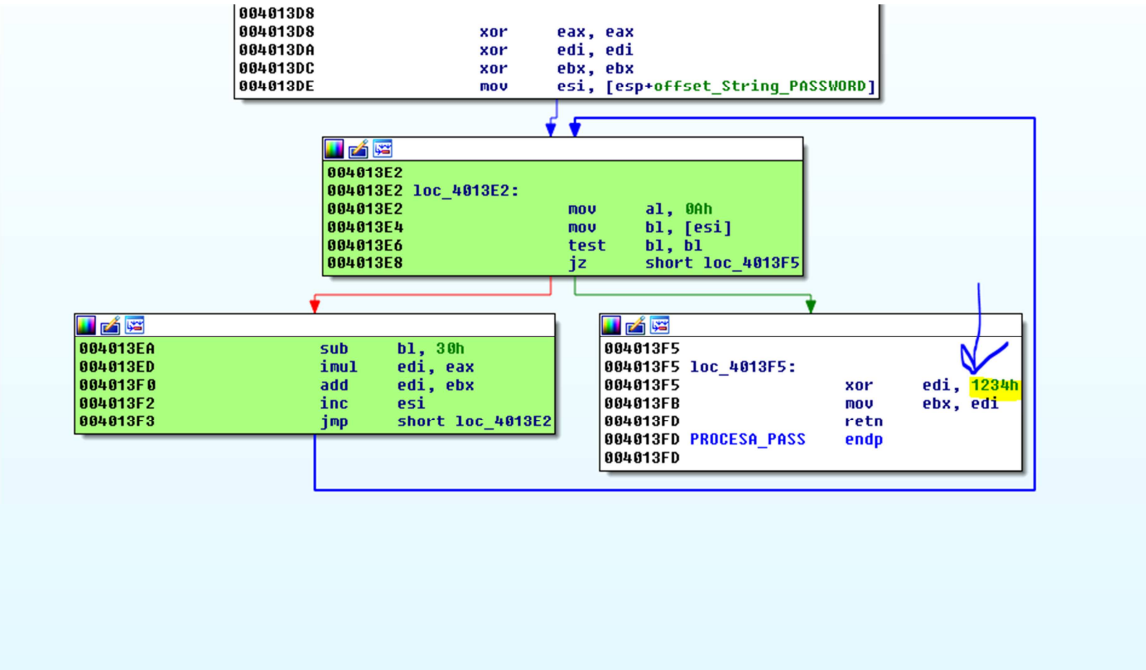
All that in the script can be sumarized as converting the string to hex with the hex() function.

```
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=int("989898")

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…

hex(password)^0x1234 = XORED.

Where XOREADO or XORED is the result that PROCESA_USER returned.

Let’s clear it.

hex(password) = XORED ^ 0x1234

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

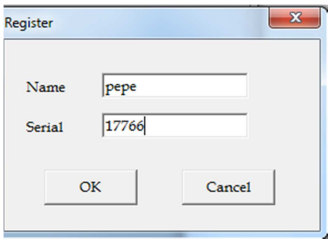
_Exploit.py

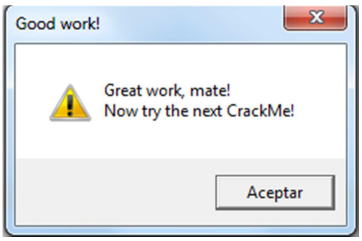
```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

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])
    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.

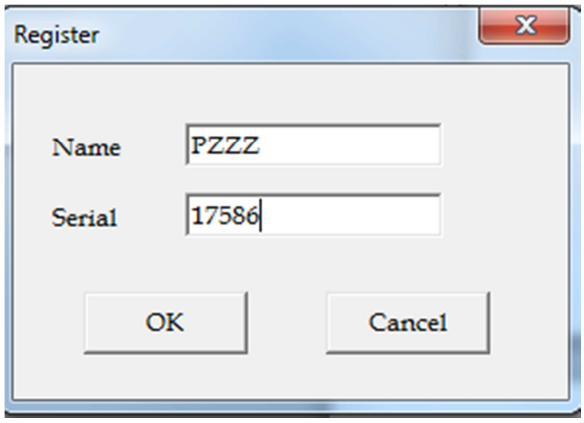
Here, I copy it to the keygen.

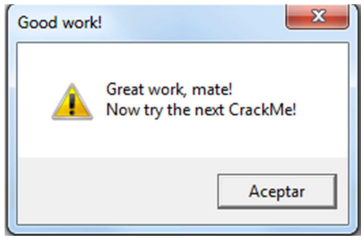
```
sum=0
user=raw_input()
largo=len(user)
if (largo> 0xb):
    exit()

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