



2008

Generating Keys for TimePassages



Author: Nacho-Dj

Graphic Editor: Shub-Nigurrath

ARTeam

July 2008



DISCLAIMER

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this tutorial have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched by other fellows, and cracked versions were available since a lot of time. ARTeam or the authors of the papers shouldn't be considered responsible for damages to the companies holding rights on those programs. The scope of this document as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>





TABLE OF CONTENTS

Disclaimer	2
Verification	2
1 FOREWORDS	5
Abstract	6
1.1 Target	6
2 EXAMINING OUR TARGET	7
2.1 Starting.....	7
2.2 Searching for any clue	9
2.3 Using the software	10
3 ANALYZING	11
3.1 Facing encryption	11
3.2 Learning about Commercial License	11
3.3 Processing a predefined string.....	14
3.4 Encrypting a text	16
3.5 Operating with the User Name and the Serial Number	17
3.6 Operating with the Enabling Code	18
3.6.1 Managing numbers.....	18
3.6.2 Managing characters	19
3.6.3 Managing the remaining characters.....	20
3.7 First XOR	21
3.8 Second XOR.....	22
3.9 Last entry in encryption routine	23
3.10 Final Tracing of the program.....	25
4 REVERSING THE ALGORITHM	26
4.1 Some notes	26



4.2	Taking advantage of XOR properties	26
4.3	Example.....	28
4.4	Reversing "Operating with the Enabling Code"	28
4.4.1	Third group of characters	29
4.4.2	First group of characters.....	30
4.4.3	Second group of characters	31
4.5	Enabling all the possible registration options	32
4.5.1	Enabling Basic Edition and Standard Edition.....	33
4.5.2	Generating keys for other versions	33
4.6	Pseudocode.....	33
5	CONCLUSIONS	36
6	GREETINGS	37



1 FOREWORDS

In this document a deep analysis for getting valid keys to register the TimePassages commercial software is done.

The main feature which turned this application attractive is the protection used by the software. It follows a method that involves some encryption.

Another issue to get it more attractive is the no existence of any Key Generator for this target.

Patching some areas of the code gives the appearance to get it registered, but a further analysis reveals that is just an illusion, due to the existing encryption code.

Thus, we are facing a good protection, and by analyzing it you could get a new point of view about the way of reversing some kind of targets using similar methods.

Happy entertainment.

Nacho_dj



ABSTRACT

In this tutorial we are going to analyze a target protected by a strong method, and after that, we could deduce all of the necessary to build a Key Generator (keygen) and get registered with valid keys.

The way of finding any working key to get registered could be more or less difficult, as it happens in any other protected software, but there is a limitation: if we are not correctly registered, we cannot access some content of files used by the target. Which could be the reason?

Well, these files have been encrypted; in that way is useless trying to access their content. Only strange symbols could be seen, in appearance arbitrary ones of the binary values composing the content of the file. After using a valid key to register, we will be able to extract the content of any file as text by using the software.

Thus, we know something in advance: a good key activates, in some way, the decryption process for the protected files.

Let's start to face it...

1.1 TARGET

In this case, it is a neither protected nor packed executable, so it can be debugged without any problem.

Here is the link to the product site:

<http://www.astrograph.com/downloads/>

Here is the link to download the target:

<http://www.astrograph.com/downloads/Win50Base.exe>



2 EXAMINING OUR TARGET

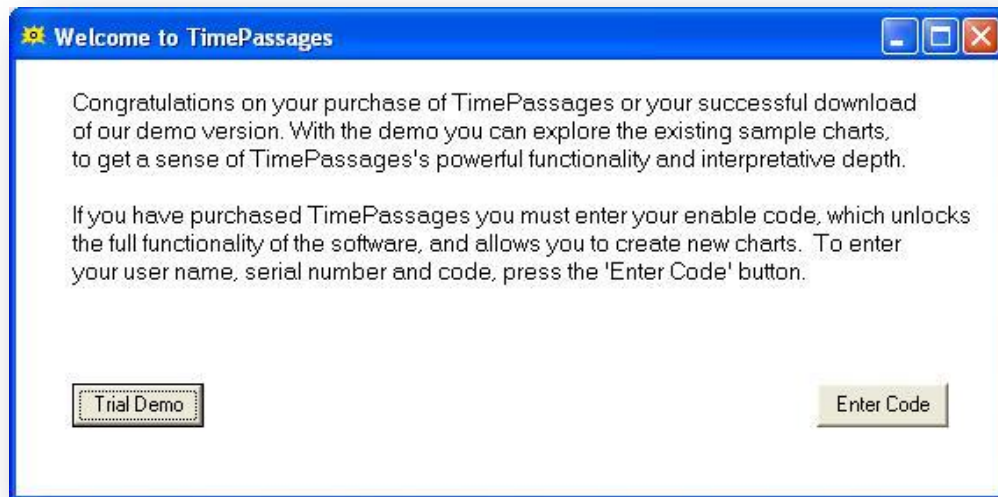
2.1 STARTING

There could be the possibility the target doesn't start normally, claiming there is a missing printer on your system. Alright, let's follow the steps shown by the OS provider to install it:

<http://support.microsoft.com/kb/141839/en-us/>

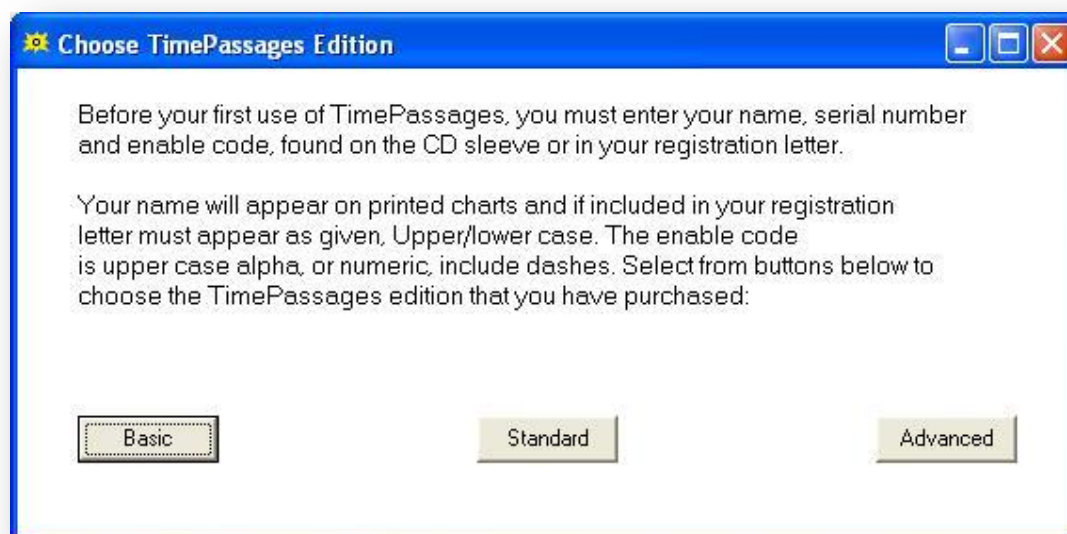
The first thing we could do about unprotecting any target is typing any key, to see what happens.

To get registered, let's launch the target. A nag is shown; it asks to choose among Trial Demo or Enter Code:



Since we are quite brave, let's choose Enter Code option.

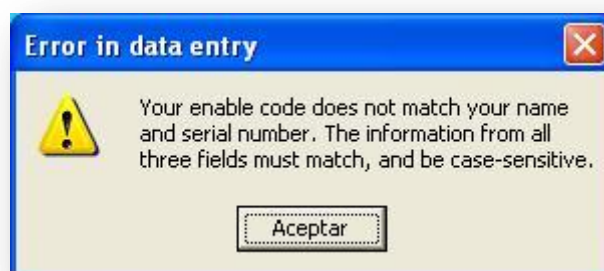
After having pushed the button, another nag is shown. This time we are asked to choose among three different features:



Well, our pockets are empty, consequently it is the same for us first, second or third option. So, let's choose Advanced option.

OK, in this case we are prompted to fill three fields: User name, Serial Number and Enabling Code.

Let's try by filling with any data. Then we push OK and we are getting a nice ErrorMessageBox (it's a pity we did not get registered in our first try):



In any case, we have information enough to start our work.



2.2 SEARCHING FOR ANY CLUE

Let's turn to our good friend OllyDbg for first analyzing this target from its inside.

Where we could start from? A good starting point is trying to find out the error messagebox after registering with any key, that is, the bad boy message.

So, let's close the target and load it in OllyDbg.

We could search in OllyDbg all strings starting by "Your enable code". Let's do *Search for* → *All referenced text strings*. Then use *Search for text* option. Then type *Your enable code* and check *Entire scope* checkbox. Press OK.

Only one match appears. When pushing enter on it, we land here:

004AC46C	55	PUSH EBP	
004AC46D	8BEC	MOV EBP,ESP	
004AC46F	83C4 D8	ADD ESP,-28	
004AC472	53	PUSH EBX	
004AC473	8BD8	MOV EBX,EAX	
004AC475	B8 7C856200	MOV EAX,TimePass.0062857C	
004AC47A	E8 DDA81300	CALL TimePass.005E6D5C	
004AC47F	66:C745 E8 0C00	MOV WORD PTR SS:[EBP-18],0C	
004AC485	BA 6D5C6200	MOV EDX,TimePass.00625C6D	ASCII "Your enable code does not mat
004AC48A	8D45 FC	LEA EAX,[LOCAL.1]	
004AC48D	E8 36A61400	CALL TimePass.005F6AC8	
004AC492	FF45 F4	INC [LOCAL.3]	
004AC495	8B10	MOV EDX,DWORD PTR DS:[EAX]	
004AC497	8BC3	MOV EAX,EBX	
004AC499	E8 1A020000	CALL TimePass.004AC6B8	
004AC49E	FF4D F4	DEC [LOCAL.3]	
004AC4A1	8D45 FC	LEA EAX,[LOCAL.1]	
004AC4A4	BA 02000000	MOV EDX,2	
004AC4A9	E8 32A81400	CALL TimePass.005F6CE0	
004AC4AE	8B4D D8	MOV ECX,[LOCAL.10]	
004AC4B1	64:890D 00000000	MOV DWORD PTR FS:[0],ECX	
004AC4B8	5B	POP EBX	00C6F240
004AC4B9	8BE5	MOV ESP,EBP	
004AC4BB	5D	POP EBP	00C6F240
004AC4BC	C3	RETN	

Let's set a breakpoint (by pushing F2 on the line selected), then if we try to register we reach this line. Press F9 to run, and repeat all steps to try to register.

We can trace now to find where lies the condition for getting the execution of this code. It won't be detailed how to find it, but it's not much difficult to check that a patch of the conditional jump at 0x520324 to turn it unconditional would be enough for us to be able of getting registered in any case with all features.



We could check it by having a look at the information provided by the About window: *Menu bar -> Help -> About TimePassages*:



But, do you think it is actually unprotected?

2.3 USING THE SOFTWARE

To test it, let's launch the target again from the start using that patch. It gets open right. Now, go to Menu bar *File -> New...*

We can open a new form, this was not possible previously to our patch. Alright, let's fill the name of city and choose a country. Now push Lookup, and... strange values are appearing. What are we missing?

It seems there should be something more apart from only patching a condition. In fact, the information provided by the files comes encrypted, and we need it decrypted. That's the obstacle we will try to surpass.



3 ANALYZING

3.1 FACING ENCRYPTION

Patching the target as the only fix to get registered has not been enough. We should analyze further to find a better way to solve it. So, a good point of starting is by using our handy OllyDbg again.

Let's launch the target from Olly, choosing *Trial Demo* when asked. When it gets totally open, we could go up to the Menu bar: *Help* → *Enable Features* → *Commercial Edition*. Then, let's fill all the fields.

Before pushing OK, we should search for a suitable breakpoint. In many encryption codes **XOR** instruction is normally used.

After selecting in Olly *Search for* → *All commands* and type XOR EAX,EBX we get a list with some lines found, with the particularity of being all of them close to the other. We could select and push enter on the first line in the list (0x49B91B) and we land on a typical encryption/decryption code.

So we could set a breakpoint with F2 after having selected the line, at the address 0x49B2EC, that is, the beginning of that routine.

3.2 LEARNING ABOUT COMMERCIAL LICENSE

Now, after having all the fields filled, we could push OK, and immediately we reach the breakpoint chosen.

Tracing with F8 key (step-by-step), when reaching 0x49B2F8 address, let's select *Follow in dump* → *Memory Address*.

We can see in the address 0x12F970 the value 0x67452301, in 0x12F974 → 0xEFCDA889, in 0x12F978 → 0x98BADCEF and in 0x12F97C → 0x10325476 (These and the subsequent addresses in the document could be different on your machine). They are initial values typically used in encryption:

```
0012F970  01234567 89ABCDEF EFCDA889 67452301
```

Let's keep tracing with F8, and we will notice that all of these values are being used.

After some operations with the previous values, we find in the address 0x49B330 how is assigned the value from 0x12F8D8, and you could easily see its content with a *Follow in dump* a string contained in the executable:



```

0049B32C . 23C8      AND ECX,EAX
0049B32E . 0BD1      OR EDX,ECX
0049B330 . 8B4D BC   MOV ECX,[LOCAL.17]
0049B333 . 03D1      ADD EDX,ECX
0049B335 . 81C2 78A46AD7 ADD EDX,076AA478
0049B33B . 03DA      ADD EBX,EDX
0049B33D . 8BD3      MOV EDX,EBX

```

Stack SS:[0012F8D8]=656D6954
ECX=10325476

Address	Hex dump	ASCII
0012F8D8	54 69 6D 65 50 61 73 73 61 67 65 73 28 74 6D 29	TimePassages(tm)
0012F8E8	20 43 6F 6D 6D 65 72 63 69 61 6C 20 4C 69 63 65	Commercial Lice
0012F8F8	6E 73 65 20 56 61 6C 69 64 61 74 69 6F 6E 80 00	nse ValidationC.
0012F908	00 00 00 00 00 00 00 00 70 01 00 00 00 00 00 00	p0.....
0012F918	76 54 32 10 38 F9 12 00 2F B2 49 00 70 F9 12 00	vT2p8-#.L.p#.

Learning how this routine operates could be a little complicated, anyway our goal is not to understand the entire behaviour of the code, but what is happening to the data involved. What we can see is all data are being used in a buffer of 0x40 bytes along the routine. Let's go down many lines in the code to get the end of the routine, and set a breakpoint in 0x49BD0F. At this address, the initial values are being modified by adding them certain quantities:

[12F970]=67452301, EBX=A57CDC79 → After F8, the value gets modified to 0CC1FF7A.

The next consecutive addresses are being modified too, giving this:

0012F970 7AFFC10C 43B4CFB9 7AA6D14E 7548FD50

Now continue tracing with F7, to find where are being used those values. We reach these instructions (Figure 1):

```

0049BD59 > 8A18 MOV BL, BYTE PTR DS:[EAX]
0049BD5B . 80E3 FF AND BL, 0FF
0049BD5D . 8B1A MOV BYTE PTR DS:[EDX], BL
0049BD5F . 8B18 MOV EBX, DWORD PTR DS:[EAX]
0049BD61 . C1EB 08 SHR EBX, 8
0049BD63 . 80E3 FF AND BL, 0FF
0049BD65 . 8B5A 01 MOV BYTE PTR DS:[EDX+1], BL
0049BD67 . 8B18 MOV EBX, DWORD PTR DS:[EAX]
0049BD69 . C1EB 10 SHR EBX, 10
0049BD6B . 80E3 FF AND BL, 0FF
0049BD6D . 8B5A 02 MOV BYTE PTR DS:[EDX+2], BL
0049BD6F . 8B18 MOV EBX, DWORD PTR DS:[EAX]
0049BD71 . C1EB 18 SHR EBX, 18
0049BD73 . 80E3 FF AND BL, 0FF
0049BD75 . 8B5A 03 MOV BYTE PTR DS:[EDX+3], BL
0049BD77 . 83C0 04 ADD EAX, 4
0049BD79 . 46 INC ESI
0049BD7B . 83C2 04 ADD EDX, 4
0049BD7D . 83C1 04 ADD ECX, 4
0049BD7F . 3BF9 CMP EDI, ECX
0049BD81 . 77 CA JA SHORT TimePass.0049BD59
0049BD83 > 5F POP EDI

```

Stack DS:[0012F970]=7A ('z')
BL=70 ('p')
Jump from 0049BD8D

Address	Hex dump
0012F970	7A FF C1 0C 43 B4 CF B9 7A A6 D1 4E 75 48 FD 50
0012F980	00 02 00 00 00 00 00 00 F4 68 6D 65 50 61 73 73

Figure 1

There is transference of the 4 values to 12F9C8, 12F9CC, 12F9D0 and 12F9D4 addresses.

Let's trace with F8, and we are reaching this area of code, where the first two dwords are assigned to newer addresses (Figure 2):

```

00520620 > 8A18 MOV BL, BYTE PTR DS:[EAX]
00520622 . 8B1A MOV BYTE PTR DS:[EDX], BL
00520624 . 41 INC ECX
00520626 . 42 INC EDX
00520628 . 40 INC EAX
0052062A . 3BF1 CMP ESI, ECX
0052062C . 7F F5 JG SHORT TimePass.00520620
0052062E > 5E POP ESI

```

Stack DS:[0012F9C8]=7A ('z')
BL=34 ('4')
Jump from 00520629

Address	Hex dump
0012F9C8	7A FF C1 0C 43 B4 CF B9 7A A6 D1 4E 75 48 FD 50
0012F9D0	00 02 00 00 00 00 00 00 F4 68 6D 65 50 61 73 73

Figure 2



When leaving the loop, we have got in the addresses 12F9FC and 12FA10 this:

0012F9FC 7AFFC10C 43B4CFB9

These two dwords are going to be used later, let's keep in mind.

3.3 PROCESSING A PREDEFINED STRING

Pushing F9 key we reach the encryption routine again. What is it being processed now? Let's trace until 0x49B330 and we can see this:

0049B32C	. 23C8	AND ECX, EHX	
0049B32E	. 0BD1	OR EDX, ECX	
0049B330	. 8B40 BC	MOV ECX, [LOCAL.17]	
0049B333	. 03D1	ADD EDX, ECX	
0049B335	. 81C2 78A46AD7	ADD EDX, 076AA478	
0049B33B	. 03DA	ADD EBX, EDX	

Stack SS:[0012F900]=20776F4E
ECX=10325476

Address	Hex dump	ASCII
0012F900	4E 6F 77 20 69 73 20 74 68 65 20 74 69 6D 65 20	Now is the time
0012F910	66 6F 72 20 61 6C 6C 20 67 6F 6F 64 20 6D 65 6E	for all good men
0012F920	20 74 6F 20 63 6F 6D 65 20 74 6F 20 74 68 65 20	to come to the
0012F930	61 69 64 20 6F 66 20 74 68 65 69 72 20 63 6F 75	aid of their cow
0012F940	76 54 32 10 60 F9 12 00 2F B2 49 00 80 F9 12 00	uT2'-'-.@I.C-.

It is a hardcoded string, located at the address 0x641730. We could find it with an Hex-Editor by opening the executable file, at the offset 0x240130.

Alright, at the end of the routine these values are got:

0012F980 61867A19 2132052B 5E280D5C D5AA3EED

There is a second entering in the routine, now the string comes from the same piece of text of before, but the next words. The 4 values taken when entering the routine are the ones got at the end of the routine in the previous execution of it:

Address	Hex dump	ASCII
0012F900	6E 74 72 79 2E 20 46 6F 75 72 20 71 75 69 63 6B	ntry. Four quick
0012F910	20 73 68 65 65 70 20 6F 75 74 77 69 74 20 6D 61	sheep outwit ma
0012F920	6E 79 20 7A 69 6E 67 79 20 77 69 74 6C 65 73 73	ny zingy witless
0012F930	20 66 6F 78 65 73 2E 20 54 69 6D 65 50 61 73 73	foxes. TimePass
0012F940	10 23 0F C7 60 F9 12 00 40 B2 49 00 80 F9 12 00	uT2'-'-.@I.C-.

0012F980 A39C2C28 4EF984E4 C68A5D7C E5CD4CB4

Address	Hex dump	ASCII
0012F8E4	61 67 65 73 28 74 6D 29 20 69 73 20 79 6F 75 72	ages(tm) is your
0012F8F4	20 61 73 74 72 6F 6C 6F 67 69 63 61 6C 20 63 68	astrological ch
0012F904	6F 69 63 65 2E 80 00 00 00 00 00 00 00 00 00 00	oice.Ç.....
0012F914	00 00 00 00 00 00 00 00 28 05 00 00 00 00 00 00	(*).....

0012F980 BC1BFC15 2F8708E9 4D84DBBF D1420FC4

By tracing with F7, we find another translation of values in the code of [Figure1](#), giving:

```
00C8B9A0  BC1BFC15  2F8708E9  4D84DBBF  D1420FC4
```



3.4 ENCRYPTING A TEXT

When tracing, we could see that an operation is done among the two double dwords got in the chapter [Learning about Commercial License](#) and the text of the last chapter:

```

005206C9 > 8BC1 MOV EAX,ECX
005206CB . 99 CDQ
005206CC . F77D 14 IDIV [ARG.4]
005206CF . 8B45 10 MOV EAX,[ARG.3]
005206D2 . 8A1410 MOV DL,BYTE PTR DS:[EAX+EDX]
005206D5 . 3016 XOR BYTE PTR DS:[ESI],DL
005206D7 . 41 INC ESI
005206D8 . 46 INC ESI
005206D9 > 3BD9 CMP EBX,ECX
005206DB . 7F EC JG SHORT TimePass.005206C9
005206DD . 8BC7 MOV EAX,EDI

```

Stack DS:[0012F9FC]=7A ('z')
DL=00

Address	Hex dump	ASCII
00C8A7B4	4E 6F 77 20 69 73 20 74 68 65 20 74 69 6D 65 20	Now is the time
00C8A7C4	66 6F 72 20 61 6C 6C 20 67 6F 6F 64 20 6D 65 6E	for all good men
00C8A7D4	20 74 6F 20 63 6F 6D 65 20 74 6F 20 74 68 65 20	to come to the
00C8A7E4	61 69 64 20 6F 66 20 74 68 65 69 72 20 63 6F 75	aid of their cou
00C8A7F4	6E 74 72 79 2E 20 46 6F 75 72 20 71 75 69 63 6B	ntry. Four quick
00C8A804	20 73 68 65 65 70 20 6F 75 74 77 69 74 20 6D 61	sheep outwit ma
00C8A814	6E 79 20 7A 69 6E 67 79 20 77 69 74 6C 65 73 73	ny zingy witless
00C8A824	20 66 6F 78 65 73 2E 20 54 69 6D 65 50 61 73 73	foxes. TimePass
00C8A834	61 67 65 73 28 74 6D 29 20 69 73 20 79 6F 75 72	ages(tn) is your
00C8A844	20 61 73 74 72 6F 6C 6F 67 69 63 61 6C 20 63 68	astrological ch
00C8A854	6F 69 63 65 2E 00 00 00 00 00 00 00 00 00 00	oice.....
00C8A864	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

When finished the loop, we get this:

```

005206C9 > 8BC1 MOV EAX,ECX
005206CB . 99 CDQ
005206CC . F77D 14 IDIV [ARG.4]
005206CF . 8B45 10 MOV EAX,[ARG.3]
005206D2 . 8A1410 MOV DL,BYTE PTR DS:[EAX+EDX]
005206D5 . 3016 XOR BYTE PTR DS:[ESI],DL
005206D7 . 41 INC ESI
005206D8 . 46 INC ESI
005206D9 > 3BD9 CMP EBX,ECX
005206DB . 7F EC JG SHORT TimePass.005206C9
005206DD . 8BC7 MOV EAX,EDI

```

EDI=00C8A7A0
EAX=0012F9FC

Address	Hex dump	ASCII
00C8A7B4	34 90 B6 2C 2A C7 EF CD 12 9A E1 78 2A D9 AA 99	4E4, *A' = 00B6 * 1 - 0
00C8A7C4	1C 90 B3 2C 22 D8 A3 99 10 90 AE 68 63 D9 AA D7	LEI, "iü0#Ehc" - i
00C8A7D4	5A 8B AE 2C 20 D8 A2 DC 5A 8B AE 2C 37 DC AA 99	Zi<, 00Zi<, 7 - 0
00C8A7E4	18 96 A5 2C 2C D2 EF CD 12 9A A8 7E 63 D7 A0 CC	+üN,,E' = 00c"ciälf
00C8A7F4	14 8B B3 75 6D 94 89 D6 0F 8D E1 7D 36 D0 AC D2	¶i umöëi*ip>6!%E
00C8A804	5A 8C A9 69 26 C4 EF D6 0F 8B B6 65 37 94 A2 D8	Zi0i8-'i*iae7ööi
00C8A814	14 86 E1 76 2A DA A8 C0 5A 8B A8 78 2F D1 BC CA	¶äBv*rc'ZëçK/0#
00C8A824	5A 99 AE 74 26 C7 E1 99 2E 96 AC 69 13 D5 BC CA	Zö<t&äB0.ü%i!!'0#
00C8A834	18 98 A4 7F 6B C0 A2 90 5A 96 B2 2C 3A D8 BA CB	+üä0k'0eZüü, : tr
00C8A844	5A 9E B2 78 31 D8 A3 D6 10 96 A2 6D 2F 94 AC D1	Z>üx1üüi#ü0n/0%0
00C8A854	15 96 A2 69 6D 00 00 00 00 00 00 00 00 00 00	Sü0lm.....
00C8A864	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	



3.5 OPERATING WITH THE USER NAME AND THE SERIAL NUMBER

If we keep activated the breakpoint in the encryption routine, after pushing F9 key we reach that routine again at 0x49B2EC address.

Once again, at 0x49B2F8 the well-known starting values are assigned:

```
0012F904  01234567 89ABCDEF FEDCBA98 76543210
```

This time a string with the User Name and the Serial Number is going to be processed. There are another parameters in the string added by the target (you could use *Follow in dump* when reaching 0x49B330 address):

Address	Hex dump	ASCII
0012F86C	4E 61 63 68 6F 5F 64 6A 0D 40 35 2E 30 7C 0D 31	Nacho_dj.@5.0!.1
0012F87C	32 33 34 35 36 37 38 80 00 00 00 00 00 00 00 00	23456780.....
0012F88C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	@.....
0012F89C	00 00 00 00 00 00 00 00 B8 00 00 00 00 00 00 00	@.....
0012F8AC	76 54 32 10 CC F8 12 00 2F B2 49 00 04 F9 12 00	vT2!F0+./@I.+0.

At the end of the routine, and taking into account what we have seen previously, we got these values for the 4 dwords:

```
0012F904  8C9C7FFB 8CEE612D 639379A7 AE40993C
```

Now let's keep tracing with F7, and we reach the instructions shown in [Figure 1](#) again.

When leaving the loop, a transference of previous 4 dwords has been done, to the addresses 12F95C, 12F960, 12F964 and 12F968.

By keeping tracing with F8, we reach the instructions in [Figure 2](#), getting this:

```
0012F9C4  8C9C7FFB 8CEE612D
```

Pay attention to this result, it is something similar to what was obtained at the end of the chapter [Learning about Commercial License](#).

So, let's keep in mind these two dwords, they are going to be used soon.



3.6 OPERATING WITH THE ENABLING CODE

Let's keep tracing with F8 key.

At the address 0x5203B0 there is an interesting call. In fact, it is very important for our goals.

We are going to dig inside it. Here is where the enabling code is being processed.

Let's enter the call using F7 key. We are only focusing on the instructions useful for us.

At 0x5203F4 we could see the enabling code typed by us. So we are in the good way :)

```

005203F0 | . 85D2      TEST EDX,EDX
005203F2 | . 74 04      JE SHORT TimePass.005203F8
005203F4 | . 8B0E      MOV ECX,DWORD PTR DS:[ESI]
005203F6 | . EB 05      JMP SHORT TimePass.005203FD
005203F8 | . B8 30F2C400 MOV ECX,TimePass.005203F8
Stack DS:[0012FAF4]=00CDE578, (ASCII "Idonotknowwhichoneshoulditbe")
ECX=0012FAF4

```

3.6.1 MANAGING NUMBERS

Let's trace a little. We reach a loop where some operations are being done using our enabling code.

```

00520485 | . EB 13      JMP SHORT TimePass.0052049A
00520487 | . 8BC8      MOV ECX,EAX
00520489 | . 03C9      ADD ECX,ECX
0052048B | . 8D0C89    LEA ECX,DWORD PTR DS:[ECX+ECX*4]
0052048E | . 89C8      MOV EAX,ECX
00520490 | . 0FBF0A    MOVSX ECX,BYTE PTR DS:[EDX]
00520493 | . 83C1 D0   ADD ECX,-30
00520496 | . 03C1      ADD EAX,ECX
00520498 | . 43        INC EBX
00520499 | . 42        INC EDX
0052049A | . 0FBF0A    MOVSX ECX,BYTE PTR DS:[EDX]
0052049D | . 83F9 2D   CMP ECX,2D
005204A0 | . 74 13      JE SHORT TimePass.005204B5
005204A2 | . 8B0E      MOV ECX,DWORD PTR DS:[ESI]
005204A4 | . 85C9      TEST ECX,ECX
005204A6 | . 74 07      JE SHORT TimePass.005204AF
005204A8 | . 8B0E      MOV ECX,DWORD PTR DS:[ESI]
005204AA | . 8B49 FC   MOV ECX,DWORD PTR DS:[ECX-4]
005204AD | . EB 02      JMP SHORT TimePass.005204B1
005204AF | . 33C9      XOR ECX,ECX
005204B1 | . 3BD9      CMP EBX,ECX
005204B3 | . 7C D2      JL SHORT TimePass.005204B7
005204B5 | . 43        INC EBX

```

By tracing carefully, we can find that first step in the loop is checking if the current byte is a '-' (0x2D). Since our enabling code in the example of above doesn't have one, we are fired from the loop.

So, let's use an enabling code with a '-' in the middle.

Let's set a breakpoint in 0x52049A, then use ALT+B keys to see all breakpoints. Disable all but the last one, then push F9. We are getting the [error message](#), and we can try to register again.

Let's type an Enabling Code containing a '-'. Then push OK and we reach 0x52049A.



Alright, now we get in 0x52049A the string typed next to the '-'. Good.

Let's go tracing to see what happens. In the first go nothing seems to happen, but after the first char is assigned in 0x520490 we can start to know what is being done. At the address 0x520493 a 0x30 is subtracted from the byte. What this could be meaning?

We could easily imagine what is happening. It seems that a number is being expected, so by subtracting from it a 0x30 (ASCII value in hexa from zero) we are getting the numeric quantity performed by the char.

Alright, push F9 several times, until getting the error message again and let's type a new enabling code, this time typing numbers next to the '-'. Then push OK.

We reach the loop again. The instruction at the address 0x520489 doubles ECX value, and next instruction multiplies it by 5, so we are getting a multiplying by 10 to the quantity stored, and then the number corresponding to the char is added. Alright, we have got the algorithm to process the numbers in our enabling code.

Keep in mind that EAX stores all the sums done. With F8 key the end of the loop is got after some iteration.

3.6.2 MANAGING CHARACTERS

The next loop we are reaching is some kind of the previous:

0052048B	EB 14	JMP SHORT TimePass.005204D1
0052048D	8D3C40	LEA EDI,DWORD PTR DS:[EAX+EAX*2]
005204C0	8D3CB8	LEA EDI,DWORD PTR DS:[EAX+EDI*4]
005204C3	03FF	ADD EDI,EDI
005204C5	89F8	MOV EAX,EDI
005204C7	0FBF3A	MOVSX EDI,BYTE PTR DS:[EDX]
005204CA	83C7 BF	ADD EDI,-41
005204CD	03C7	ADD EAX,EDI
005204CF	41	INC ECX
005204D0	42	INC EDX
005204D1	0FBF3A	MOVSX EDI,BYTE PTR DS:[EDX]
005204D4	83FF 20	CMP EDI,20
005204D7	74 13	JE SHORT TimePass.005204EC
005204D9	8B3E	MOV EDI,DWORD PTR DS:[ESI]
005204DB	85FF	TEST EDI,EDI
005204DD	74 07	JE SHORT TimePass.005204E6
005204DF	8B3E	MOV EDI,DWORD PTR DS:[ESI]
005204E1	8B7F FC	MOV EDI,DWORD PTR DS:[EDI-4]
005204E4	EB 02	JMP SHORT TimePass.005204E8
005204E6	33FF	XOR EDI,EDI
005204E8	3BCF	CMP ECX,EDI
005204EA	7C D1	JL SHORT TimePass.005204BD
005204EC	8B4D 10	MOV ECX,[ARG_3]

However, we could see a subtraction of 0x41 instead of 0x30 like before.

What does 0x41 stand for? This is the ASCII value (hex) of 'A' char.

In this loop, we start by the first char of the enabling code.

Then, the instruction at the address 0x5204BD multiplies by 3 what is stored in EAX, and the following instruction multiplies by 4 and adds EAX. The next instruction multiplies by 2 EDI value.



Conclusion: what was stored in EAX has been multiplied by 26. This is exactly the number of alphabetic characters in the ASCII code, in this case as capital letters...

Keep tracing generates a new value in EAX when exiting the loop. Let's trace a little bit more and we will see how the results are being moved into the address 0x12F990 (instructions at 0x5204F4, 0x520502, 0x520510 y 0x52051B, but note that high bytes are being swapped to low bytes.

3.6.3 MANAGING THE REMAINING CHARACTERS

With F8 key we keep tracing and reach another loop:

00520525	EB 14	JMP SHORT TimePass.0052053B
00520527	800C40	LEA ECX,DWORD PTR DS:[EAX+EAX*2]
0052052A	800C88	LEA ECX,DWORD PTR DS:[EAX+ECX*4]
0052052D	03C9	ADD ECX,ECX
0052052F	89C8	MOV EAX,ECX
00520531	0FBF0A	MOVSX ECX,BYTE PTR DS:[EDX]
00520534	83C1 BF	ADD ECX,-41
00520537	03C1	ADD EAX,ECX
00520539	43	INC EBX
0052053A	42	INC EDX
0052053B	8B0E	MOV ECX,DWORD PTR DS:[ESI]
0052053D	85C9	TEST ECX,ECX
0052053F	74 07	JE SHORT TimePass.00520548
00520541	8B0E	MOV ECX,DWORD PTR DS:[ESI]
00520543	8B49 FC	MOV ECX,DWORD PTR DS:[ECX-4]
00520546	EB 02	JMP SHORT TimePass.0052054A
00520548	33C9	XOR ECX,ECX
0052054A	3BD9	CMP EBX,ECX
0052054C	7C D9	JL SHORT TimePass.00520527
0052054E	8B4D 10	MOV ECX,CARG.31

This is similar to the previous one. We could find that an enabling code like XXXXXX-000000 (X being any alphabetic character and 0 any number) we cannot get this loop being processed, because it doesn't find any byte to start. How would you solve this?

Well, just adding a '-' next to the last number in our enabling code and some characters will do the work. So, after pushing F9 and getting the error message, let's fill the enabling code again and push F9 key. It's expected to reach this loop again.

So, using an enabling code like the pattern XXXXXX-000000-XXXXXX we get the three loops to be processing the code.

In the last loop all instructions are similar to the second loop, but the difference is this loop has started with a value stored in EAX, from the second loop. The EAX value as result of the loop is going to fill the dword next to the first one, particularly to 0x12F994, and here we have again a swapping of high bytes to low.



After having used an enabling code from the pattern XXXXXX-000000-XXXXXX, we get these dwords (they depend on your enabling code used):

0012F990 **10F09611 10F09611**

3.7 FIRST XOR

Let's trace with F8, so exiting the routine. We reach immediately another interesting piece of code:

The screenshot shows a debugger window with the following assembly code and register values:

```

005203C1  > 7E 0B  JLE SHORT TimePass.005203C6
005203C3  > 8A18  MOV BL, BYTE PTR DS:[EAX]
005203C5  > 301A  XOR BYTE PTR DS:[EDX], BL
005203C7  > 41     INC ECX
005203C8  > 42     INC EDX
005203C9  > 40     INC EAX
005203CA  > 3BF1  CMP ESI, ECX
005203CC  > 7F F5  JG SHORT TimePass.005203C3
005203CE  > 5F     POP EDI
  
```

Below the assembly code, the register values are shown:

```

BL=38 ('8')
Stack DS:[0012F9C4]=8C ('I')
  
```

At the bottom, there is a hex dump table:

Address	Hex dump
0012F9C4	8C 9C 7F FB 8C EE 61 2D 4E 11 54 00 F5 29 54 00
0012F9C4	9C 9C 7F FB 8C EE 61 2D 4E 11 54 00 F5 29 54 00

Here the two dwords from the chapter [Operating with the User Name and the Serial Number](#) are going to be operated by **XOR** with two dwords of the last chapter, coming from enabling code. Interesting, isn't it?

The result is stored in 0x12F9C4 and 0x12F9C8.

0012F9C4 **9C6CE9EA 9C1EF73C**



3.8 SECOND XOR

Let's keep tracing with F8 till 0x520376. Enter the call with F7, and trace till 0x5202C4, entering again with F7. At 0x5E67FB it is being copied the entire block of bytes, that was encrypted previously in the chapter [Encrypting a text](#), to the address 0xD725B0.

After being tracing with F8 a few lines, we follow till this piece of code is reached (Figure 3):

```

005202D3 .> EB 00      JMP SHORT TimePass.005202E2
005202D5 .> 8BC1      MOV EAX, ECX
005202D7 .> 41        INC ECX
005202D8 .> 99        CDQ
005202D9 .> F77D 14   IDIV DWORD PTR SS:[EBP+14]
005202DC .> 8A417     MOV AL, BYTE PTR DS:[EDI+EDX]
005202DF .> 3003      XOR BYTE PTR DS:[EBX], AL
005202E1 .> 43        INC EBX
005202E2 .> 8B56 10   MOV EDX, DWORD PTR DS:[ESI+10]
005202E5 .> 3BD1      CMP EDX, ECX
005202E7 .> 77 EC     JA SHORT TimePass.005202D5

```

Stack DS:[0012F9C4]=9C ('E')
AL=00

Address	Hex dump	ASCII
00DC0070	34 90 B6 2C 2A C7 EF CD 12 9A E1 78 2A D9 AA 99	4E2, *A'='0Bx#-0
00DC0080	1C 90 B3 2C 22 D8 A3 99 1D 90 AE 68 63 D9 AA 07	LEI, "i00#E<hc-i
00DC0090	5A 88 AE 2C 20 D8 A2 DC 5A 88 AE 2C 37 DC AA 99	Zi<, 0Zi<, 7-0
00DC00A0	18 96 A5 2C 2C D2 EF CD 12 9A A8 7E 63 D7 A0 CC	+00,,E'='00<"oiaf
00DC00B0	14 8B B3 75 6D 94 89 D6 0F 8D E1 7D 36 D0 AC D2	0i um0ei*ib0!%E
00DC00C0	5A 8C A9 69 26 C4 EF D6 0F 8B B6 65 37 94 A2 D8	Zi0i&-i*iae700i
00DC00D0	14 86 E1 76 2A DA A8 C0 5A 88 A8 78 2F D1 BC CA	0Bp0*r0'Z00x/p0
00DC00E0	5A 99 AE 74 26 C7 E1 99 2E 96 AC 69 13 D5 BC CA	Z0<t&A00.0%!!'0
00DC00F0	18 98 A4 7F 68 C0 A2 90 5A 96 B2 2C 3A D8 BA CB	+000k'0eZ00,,0000
00DC0100	5A 9E B2 78 31 D8 A3 D6 1D 96 A2 6D 2F 94 AC D1	Z>00x100i#00m/0%0
00DC0110	15 96 A2 69 6D 00 00 00 00 00 00 00 00 00 00	0000000000000000
00DC0120	00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000

Figure 3

Now we could carefully examine what is being done.

The couple of dwords from the previous chapter are being processed with **XOR** against a block of bytes, which is shown in the Figure 3. When finished the loop, all bytes of the block are modified. What could we deduce from it? Let's be aware of the size of the block: 0xA5 bytes

In fact, this block is being **encrypted**. Or maybe decrypted?

We could remember that this block comes from a text block that was originally encrypted, as was shown in the chapter [Encrypting a text](#).

When the loop is finished, we have this (Figure 4):

Figure 4

3.9 LAST ENTRY IN ENCRYPTION ROUTINE

We land again in the encryption routine (see [Facing encryption](#)), where the four values are initialized as follows:

The block to be processed is this:

Let's see the [Figure 4](#). Yes, this is a piece of the block got at the end of the chapter [Second XOR](#).



When finished the routine, the 4 values have been modified to:

```
0012F944  49CAB6D4 4BDE6803 8A1DC07E D9536000
```

There is a second entry to the routine, with the last 4 values as input parameters. The block of bytes now is composed by the next bytes of the previous picture:

Address	Hex dump	ASCII
0012F8B4	88 E7 5A 9F F1 8A 7E EA 93 E1 08 97 AA C3 5B EE	èfZf±è~008w-[-
0012F8C4	C6 E0 40 83 BA DA 18 EA 93 E7 5F 8F AB 8A 55 E4	š00ë r†00ë_A%ëU%
0012F8D4	88 EA 08 9C B6 C4 5F FC C6 E4 41 92 B3 CF 4B F6	ëUëEA-?ššAE 0K÷
0012F8E4	C6 F5 47 9E BA D9 16 A5 B2 FA 45 83 8F CB 4B F6	ššG× -ššEšAfk÷
0012F8F4	D9 53 60 00 14 F9 12 00 4B B3 49 00 44 F9 12 00	šššššššššššššššš

At the end of the routine, we exit with these 4 values:

```
0012F944  E0405740 8035A60D 4636F1FE F34A0F74
```

A third entry is performed, with last 4 values as input parameters again. The block of bytes is next to the ones of previous picture:

Address	Hex dump	ASCII
0012F898	87 F4 4D 95 F7 DE 55 AC C6 FA 5B C6 A6 C5 4D F7	çqMö·iU%š·[šš±M·
0012F8A8	C6 F2 5B 92 AD C5 54 EA 81 FA 4B 87 B3 8A 5B E0	š=[šš±T0ü·Kçlêçÿ
0012F8B8	89 FA 4B 83 F1 80 00 00 00 00 00 00 00 00 00	ë·Kš±ç.....
0012F8C8	00 00 00 00 00 00 00 00 28 05 00 00 00 00 00	(.....
0012F8D8	F3 4A 0F 74 F8 F8 12 00 2F B2 49 00 44 F9 12 00	%šš±šššššššššš

We exit from the routine with these 4 values:

```
0012F944  D13C814E F9A923BC D716574E B0ABA423
```

Note again at the end of bytes the 0x80 byte and a quantity at the fifteenth dword.



3.10 FINAL TRACING OF THE PROGRAM

By using F8, we reach the piece of code of the [Figure 1](#); this time the values are copied to:

```
0012F934 D13C814E F9A923BC D716574E B0ABA423
```

Then we trace with F8, and we reach this code, where it is validated our enabling code:



At 0x52031E address the first byte of [EAX] is moved; let's see what does EAX contain:

```
EAX=C8BEE0
```

A *Follow in dump* shows us:

```
00C8BEE0 BC1BFC15 2F8708E9 4D84DBBF D1420FC4
```

...and these are the bytes got at the end of the chapter [Processing a predefined string](#).

At 0x520320 address the first byte of the following block of bytes is moved:

```
0012F934 D13C814E F9A923BC D716574E B0ABA423
```

that is, the block got in the last chapter.

Then, both bytes are compared. If they are not equal, we are not registered and by pushing F9, this [error message](#) is shown again. What a bad luck!



4 REVERSING THE ALGORITHM

4.1 SOME NOTES

Now, it's time of trying to reverse all what we have seen before.

In the last chapter a comparison has been done. When it is performed with the same bytes, we are getting registered.

There are two bytes being compared, extracted from different results. One of them keeps always invariable; this is the one that comes from what we saw in the chapter [Processing a predefined string](#).

The variable byte depends of the User Name, the Serial Number and the Enabling Code.

We have to get either result (byte) being the same.

How could we success? Let's go back with the good values got in the chapter [Last entry in encryption routine](#).

4.2 TAKING ADVANTAGE OF XOR PROPERTIES

To get succeeded in registering it is needed in the [Last entry in encryption routine](#) chapter entering with a unencrypted text, like the one used in the chapter [Processing a predefined string](#).

Note that the encrypted block has been got from operating a **XOR** on the input block, as was shown in [Figure 4](#). The trick is trying to get a decrypted block after performing that **XOR** on it. This could be possible because of the reversibility of **XOR** instruction.

For instance, let's try this: $10 \text{ XOR } 01 \rightarrow 11$. If we want now to get the first operator from the result (11) and the second operator (01), it could be done in this way: $11 \text{ XOR } 01 \rightarrow 10$. This reversibility is always true for any value processed by **XOR**.

Thus, let's take the text used when starting [Encrypting a text](#) chapter. We can see that a text is being encrypted by performing a **XOR** with two dwords and this is repeated until the end of the text is reached, so we only need to know the first **XOR** done with the first 8 bytes of the text:

First two dwords of the text:

4E6F7720 69732074

XOR

Two dwords got in [Learning about Commercial feature](#) chapter

7AFFC10C 43B4CFB9



As a result, we get:

3490B62C 2AC7EFCD

Set of dwords #1

Let's pay attention now to the beginning of the [Second XOR](#) chapter. The two dwords used should match the ones of *Set of dwords #1*. Where do these dwords come from? Let's follow going back.

We are now at [First XOR](#) chapter, where a XOR was performed in this way:

Dwords got as [Operating with the User Name and the Serial Number](#) chapter shows

XOR

Dwords got as [Operating with the Enabling Code](#) shows.

So, because of the reversibility of **XOR** instruction, we should operate in this way:

Set of dwords #1

XOR

Dwords got as [Operating with the User Name and the Serial Number](#) chapter shows

We would need to get the same values got at the end of the chapter [Operating with the Enabling Code](#) to be able of getting registered.



4.3 EXAMPLE

Let's try with the values used in this tut:

```
3490B62C 2AC7EFCD (Set of dwords #1)
```

XOR

```
8C9C7FFB 8CEE612D (Result of user name and serial name)
```

It gives this:

```
B80CC9D7 A6298EE0
```

Set of dwords #2

So, this is what it is needed at the end of the chapter [Operating with the Enabling Code](#).

4.4 REVERSING "OPERATING WITH THE ENABLING CODE"

In this chapter we had entered with an enabling code following the pattern XXXXXX-000000-XXXXXX, but it didn't work. However, we know that the enabling code could follow that pattern, or at least the code is ready to process a pattern like that.

Let's go back in the code.

The second dword of the *Set of dwords #2* must come from the operations done on the substring next to the second '-' of the enabling code.



4.4.1 THIRD GROUP OF CHARACTERS

This is the code that provides the second dword (see [Managing the remaining characters](#)):

```

00520525 |> EB 14 | JMP SHORT TimePass.0052053B
00520527 |> 800C40 | LEA ECX,DWORD PTR DS:[EAX+EAX*2]
0052052A |> 800C88 | LEA ECX,DWORD PTR DS:[EAX+ECX*4]
0052052D |> 03C9 | ADD ECX,ECX
0052052F |> 89C8 | MOV EAX,ECX
00520531 |> 0FBF0A | MOVSX ECX, BYTE PTR DS:[EDX]
00520534 |> 83C1 BF | ADD ECX,-41
00520537 |> 03C1 BF | ADD EAX,ECX
00520539 |> 43 | INC EBX
0052053A |> 42 | INC EDX
0052053B |> 8B0E | MOV ECX,DWORD PTR DS:[ESI]
0052053D |> 85C9 | TEST ECX,ECX
0052053F |> 74 07 | JE SHORT TimePass.00520548
00520541 |> 8B0E | MOV ECX,DWORD PTR DS:[ESI]
00520543 |> 8B49 FC | MOV ECX,DWORD PTR DS:[ECX-4]
00520546 |> EB 02 | JMP SHORT TimePass.0052054A
00520548 |> 33C9 | XOR ECX,ECX
0052054A |> 3BD9 | CMP EBX,ECX
0052054C |> 7C D9 | JL SHORT TimePass.00520527
0052054E |> 8B4D 10 | MOV ECX,CARG.31

```

The way of reversing it is as easy as this:

Aux ← Value of Second Dword

Loop

Remainder ← Aux **MOD** 26

Aux ← Aux - Remainder

Aux ← Aux **DIV** 26

ASCII code of Char ← Remainder + 0x41

Until Aux = 0

NB: 26 is the number of alphabetical characters available in the ASCII code. 41 is the hexadecimal ASCII value from the 'A' character.

We have got the last character of the key. By performing a loop using the algo till Aux equal to zero, the following string is got:

LGUSNGU



This should be the third part of the good key:

XXXXXX-000000-LGUSNGU

4.4.2 FIRST GROUP OF CHARACTERS

The first dword from *Set of dwords #2* gives us the other part of the enabling code.

First, the starting string of the enabling code is processed; this will be done with the same algo from above:

```
Aux ← Value of First Dword  
i ← 0  
  
Loop  
  
Remainder ← Aux MOD 26  
  
Aux ← Aux - Remainder  
  
Aux ← Aux DIV 26  
  
ASCII code of Char ← Remainder + 0x41  
  
Inc (i)  
  
Until i = 4
```

When the loop is performed, we could set a limited number of iterations. If this not done, Aux value could reach zero value and there won't be the possibility of getting any numbers for the enabling code. So, we set 4 characters as maximum (4 iterations). If you configure another maximum number of characters, it would work, too.



After the loop, we have got the first string of the enabling code:

VWHR-000000-LGUSNGU

4.4.3 SECOND GROUP OF CHARACTERS

We haven't still finished. We have left the loop with a value of Dword not equal to zero, because of the limited number of iterations to 4. Now we have to reverse the piece of code that processes the numbers of the enabling code. That is, we should reverse the code seen in the chapter [Managing numbers](#):

Loop

Remainder $\leftarrow$ Aux **MOD** 10

Aux $\leftarrow$ Aux - Remainder

Aux $\leftarrow$ Aux **DIV** 10

ASCII code of Char $\leftarrow$ Remainder + 0x30

Until Aux = 0

The result of performing this loop until Aux is zero is what we have to type in the enabling code instead of the '000000' of the pattern. We are going to hide this piece of serial because of obvious reasons. :P

NB: 10 is the number of numerical characters available in the ASCII code. 30 is the hexadecimal ASCII value from the '0' character.



Now, we have got a valid enabling code, and thus, covered the goal of generating an enabling code that is dependent from the user name and the serial name.

Let's try to register with the enabling code got. Then, go to *Menu bar -> Help -> About*:



4.5 ENABLING ALL THE POSSIBLE REGISTRATION OPTIONS

There are more features for registering. Here is a list of all of them:

- Basic Edition
- Standard Edition
- Advanced Edition
- Compatibility Add-on
- Relationship Add-on
- Commercial License

To be able to register any of them, we only need the equivalent reference to *Set of Dwords #1* for each one of the features.



We would need for operating with **XOR** a couple of values equivalent to [Learning about Commercial feature](#) but in this case for the feature in which we want to register, they will be generated from a different text.

A handy way of being ready for every feature of registration is building a table composed by pairs of double dwords, each of them for a certain feature, since all of them should be constant.

4.5.1 ENABLING BASIC EDITION AND STANDARD EDITION

Since we have started analyzing the more difficult feature to register, this case is not complicated in any way. The only change is done when processing the code of [Operating with the User Name and the Serial Number](#) chapter, because for this case you only have to insert 'BE' or 'SE' string next to the user name.

This could be seen by performing a dump to memory, as the chapter [Operating with the User Name and the Serial Number](#) shows.

4.5.2 GENERATING KEYS FOR OTHER VERSIONS

The way of getting it is the same of the previous chapter. You probably would have been aware of the string '5.0' in the middle of the composed string. You only have to replace that string by the number of version to be reversed, for instance: '4.2'.

4.6 PSEUDOCODE

Codify: Routine starting at 0x49B2EC

2 D. W. from Text $\leftarrow$ Constant

2 D.W. User_Serial $\leftarrow$ *Codify* (User Name + Version + Serial Number)

2 D.W. from Type of Feature $\leftarrow$ *Codify* (Text of Type of Feature)

Set of Dwords #1 (1) $\leftarrow$ D.W. from Text (1) **XOR** D.W. from Type of Feature (1)

Set of Dwords #1 (2) $\leftarrow$ D.W. from Text (2) **XOR** D.W. from Type of Feature (2)

D.W. 1 $\leftarrow$ Set of Dwords #1 (1) **XOR** D.W. User_Serial (1)



D.W. 2 $\leftarrow$ Set of Dwords #1 (2) **XOR** D.W. User_Serial (2)

Aux $\leftarrow$ D. W. 2

Serial3 $\leftarrow$ ''

Loop

Remainder $\leftarrow$ Aux **MOD** 26

Aux $\leftarrow$ Aux - Remainder

Aux $\leftarrow$ Aux **DIV** 26

Char $\leftarrow$ Remainder + 0x41

Serial3 $\leftarrow$ Char + Serial3

Until Aux = 0

Aux $\leftarrow$ D. W. 1

Serial1 $\leftarrow$ ''

i $\leftarrow$ 0

Loop

Remainder $\leftarrow$ Aux **MOD** 26

Aux $\leftarrow$ Aux - Remainder

Aux $\leftarrow$ Aux **DIV** 26

Char $\leftarrow$ Remainder + 0x41

Serial1 $\leftarrow$ Char + Serial1

Inc (i)

Until i = 4

Serial2 $\leftarrow$ ''

Loop

Remainder $\leftarrow$ Aux **MOD** 10

Aux $\leftarrow$ Aux - Remainder



Aux $\leftarrow$ Aux **DIV** 10

Char $\leftarrow$ Remainder + 0x30

Serial2 $\leftarrow$ Char + Serial2

Until Aux = 0

Enabling Code $\leftarrow$ Serial1 + '-' + Serial2 + '-' + Serial3

NB: For the full code see included Delphi sources



5 CONCLUSIONS

All the stuff explained in this tut has been tested and a Key Generator working for any user name and serial number. Even, there are valid enabling codes not containing any '-'. You are who choose the way of coding your algorithm for the keygen, and in that way will be generated every valid key.

So, as expected this document has provided some new ways of reversing knowledge, and when we face similar disassembled codes it wouldn't intimidate you in any way and you could assume the challenge of researching up, up & away...



6 GREETINGS

This tutorial is dedicated to all the components of ARTeam and Shub for editing it. Big thanks to Baxter, from TeamICU, for raising the challenge that was the origin of this work. To all friends of TeamICU, Seek n Destroy, RCE Forums, The Reverse Code Engineering Community, Exetools, BiW, UnPackcN, UIC, CRK, CTB and special greetings to CrackSlatinoS.

And of course, to you, who decided to read this document, because without your support and contribution this task wouldn't be worth to be done.

Nacho_dj / ARTeam – June, 24th 2008

